# ПРИМЕНЕНИЕ UML И ШАБЛОНОВ ПРОЕКТИРОВАНИЯ

**Введение в объектно-ориентированный анализ, проектирование и унифицированный процесс UP**

**ВТОРОЕ ИЗДАНИЕ**

"Люди часто спрашивают меня о том, с помощью какой книги лучше всего познакомиться с миром объектно-ориентированного проектирования. С тех пор, как я увидел книгу *Применение UML и шаблонов проектирования*, я рекомендую именно ее."
*– Мартин Фовлер, автор книг* UML Distilled *и* Refactoring

# Крэг Ларман

*Предисловие Филиппа Крачтена*

# APPLYING UML AND PATTERNS

## An Introduction to Object-Oriented Analysis and Design and the Unified Process

**Second Edition**

**Craig Larman**

**Prentice Hall PTR**
Upper Saddle River, NJ 07458
www.phptr.com

# ПРИМЕНЕНИЕ UML
# И ШАБЛОНОВ ПРОЕКТИРОВАНИЯ

Введение в объектно-ориентированный анализ,
проектирование и унифицированный процесс UP

**Второе издание**

**Крэг Ларман**

Издательский дом "Вильямс"
Москва • Санкт-Петербург • Киев
2004

ББК 32.973.26-018.2. 75
Л25
УДК 681.3.07

Издательский дом "Вильямс"

Перевод с английского и редакция канд.техн.наук *А.Ю. Шелестова*

По общим вопросам обращайтесь в Издательский дом "Вильямс"
по адресу: info@williamspublishing.com, http://www.williamspublishing.com

**Ларман, Крэг.**

Л25 Применение UML и шаблонов проектирования. 2-е издание. : Пер. с англ. — М. : Издательский дом "Вильямс", 2004. — 624 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0250-9 (рус.)

Книга поможет освоить основные принципы и самые современные приемы объектно-ориентированного анализа и проектирования (ООА/П). В ней вы найдете новые сведения о шаблонах проектирования, прецедентах, архитектурном анализе и многих других вопросах, которые рассматриваются в рамках одного из самых популярных итеративных процессов проектирования UP.

На протяжении всей книги рассматривается один реальный пример, модифицированный по сравнению с первым изданием книги. Для построения моделей используется унифицированный язык моделирования UML, ставший фактическим стандартом объектно-ориентированного анализа и проектирования.

Данная книга будет хорошим путеводителем для всех, кто интересуется вопросами ООА/П, как для новичков, так и для специалистов.

**ББК 32.973.26-018.2.75**

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Prentice Hall, Inc..

Authorized translation from the English language edition published by Prentice Hall, Inc., Copyright © 2002

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2002

ISBN 5-8459-0250-9 (рус.)
ISBN 0-13-092569-1 (англ.)

© Издательский дом "Вильямс", 2002
© Graig Larman, 2002

# Оглавление


Часть I. Введение 31

Глава 1. Объектно-ориентированный анализ и проектирование 33

Глава 2. Итеративная разработка и унифицированный процесс 43

Глава 3. Конкретный пример: система автоматизации торговли NextGen 59

Часть II. Начальная фаза 63

Глава 4. Начало 65

Глава 5. Осмысление требований 69

Глава 6. Описание требований в контексте модели прецедентов 73

Глава 7. Определение остальных требований 107

Глава 8. От начальной фазы к стадии развития 129

Часть III. Первая итерация фазы развития 137

Глава 9. Модель прецедентов: диаграммы последовательностей 139

Глава 10. Модель предметной области: визуализация понятий 147

Глава 11. Модель предметной области: добавление ассоциаций 169

Глава 12. Модель предметной области: добавление атрибутов 181

Глава 13. Модель прецедентов: детализация с помощью описаний операций 191

Глава 14. От анализа требований к проектированию на данной итерации 205

Глава 15. Система обозначений для диаграмм взаимодействия 209

Глава 16. GRASP: шаблоны для распределения обязанностей 223

Глава 17. Модель проектирования: реализация прецедентов на основе шаблонов GRASP 253

Глава 18. Модель проектирования: области видимости 285

Глава 19. Модель проектирования: создание диаграммы классов 291

Глава 20. Модель реализации: преобразование результатов проектирования в программный код 307

Часть IV. Вторая итерация фазы развития 323

Глава 21. Вторая итерация и требования к ней 325

Глава 22. Дополнительные шаблоны GRASP для распределения обязанностей 331

Глава 23. Реализация прецедентов с использованием шаблонов GoF 347

Часть V. Третья итерация фазы развития 383

Глава 24. Третья итерация и ее требования 385

Глава 25. Взаимосвязь прецедентов 387

Глава 26. Обобщение модели 395

Глава 27. Усовершенствование модели предметной области 409

Глава 28. Поведение системы 427
Глава 29. Моделирование поведения на диаграммах состояний 431
Глава 30. Проектирование систем на основе шаблонов 439
Глава 31. Создание модели проектирования и реализации на основе пакетов 467
Глава 32. Введение в архитектурный анализ 475
Глава 33. Реализация новых прецедентов на основе объектов и шаблонов 495
Глава 34. Проектирование контура взаимодействия с базой данных на основе шаблонов 523

ЧАСТЬ VI. СПЕЦИАЛЬНЫЕ ВОПРОСЫ 551

Глава 35. Средства создания диаграмм 553
Глава 36. Знакомство с итеративным планированием и проектированием 559
Глава 37. Комментарии к итеративной разработке и UP 571
Глава 38. Дополнительные обозначения UML 583
Приложение. Артефакты унифицированного процесса, шаблоны GRASP и условные обозначения языка UML 589
Литература 597
Словарь терминов 603
Предметный указатель 609

# Содержание

**Введение** 26

ЧАСТЬ I. ВВЕДЕНИЕ 31

**Глава 1. Объектно-ориентированный анализ и проектирование** 33

1.1. Применение UML и шаблонов в процессе объектно-ориентированного анализа и проектирования 33
Другие навыки 35
1.2. Распределение обязанностей 35
1.3. Что такое анализ и проектирование 36
1.4. Объектно-ориентированный анализ и проектирование 36
1.5. Пример объектно-ориентированного анализа и проектирования 37
Определение прецедентов 37
Определение модели предметной области 38
Диаграммы взаимодействий 39
Диаграммы классов 39
Еще несколько слов об игре в кости 40
1.6. Унифицированный язык моделирования UML 40
Почему в нескольких следующих главах речь не идет об UML 41
1.7. Дополнительная литература 41

**Глава 2. Итеративная разработка и унифицированный процесс** 43

Введение 43
Если меня не интересует унифицированный процесс 44
2.1. Главная идея унифицированного процесса — итеративная разработка 44
Допустимость изменений: обратная связь и адаптация 46
Преимущества итеративной разработки 47
Фиксированная длительность итерации 48
2.2. Дополнительные рекомендации и концепции унифицированного процесса 48
2.3. Фазы унифицированного процесса и соответствующие термины 49
2.4. Дисциплины унифицированного процесса (в прошлом — последовательности выполняемых действий) 50
Дисциплины и фазы 51
Фазы UP, дисциплины и структура книги 52
2.5. Настройка процесса и набор документов для проекта 53
Необязательные артефакты 53
Набор документов 53
2.6. Гибкость унифицированного процесса 54
2.7. Последовательный жизненный цикл или жизненный цикл типа “водопад” 55
2.8. Вы не поняли, что такое унифицированный процесс, если... 56
2.9. Дополнительная литература 56

**Глава 3. Конкретный пример: система автоматизации торговли NextGen** 59

Введение 59
3.1. Система автоматизации торговли NextGen 59
3.2. Архитектурные уровни и основные моменты 60
3.3. Стратегия данной книги: итеративное изучение и разработка 61

Часть II. Начальная фаза 63

Глава 4. Начало 65

Введение 65

4.1. Аналогия 66

4.2. Начало может быть очень коротким 66

4.3. Какие артефакты относятся к начальной фазе 67

Не слишком ли много документации 67

4.4. Вы не поняли, что такое начальная фаза, если... 68

Глава 5. Осмысление требований 69

Введение 69

5.1. Типы требований 70

5.2. Дополнительная литература 71

Глава 6. Описание требований в контексте модели прецедентов 73

Введение 73

6.1. Задачи и описания 74

6.2. Предыстория 74

6.3. Прецеденты и ощутимый результат 75

6.4. Прецеденты и функциональные требования 76

6.5. Типы и форматы прецедентов 76

Прецеденты типа "черный ящик" и системные обязанности 76

Степень формализации 77

6.6. Пример развернутого описания прецедента Оформление продажи 77

Формат usecases.org 77

Представление в виде двух колонок 80

Какой формат лучше? 81

6.7. Пояснения 81

Вводные элементы 81

Заинтересованные лица и их потребности 81

Предусловия и постусловия 82

Основной успешный сценарий (или основной процесс) 82

Расширения (или альтернативные потоки) 83

Специальные требования 84

Список технологий и типов данных 84

6.8. Задачи и рамки прецедента 85

Прецеденты для элементарных бизнес-процессов 85

Прецеденты и задачи 87

Пример: использование рекомендаций в соответствии с EBF 87

Вспомогательные задачи и прецеденты 88

Составные задачи и прецеденты 89

6.9. Определение основных исполнителей, задач и прецедентов 89

Шаг 1. Определение рамок системы 89

Шаги 2 и 3. Определение основных исполнителей и задач 89

Шаг 4. Определение прецедентов 92

6.10. Поздравляем: прецеденты описаны плохо 92

Необходимость обсуждений и совместных усилий 92

6.11. Описание прецедентов, относящихся к интерфейсу пользователя, в свободном стиле 93

Новая и важная информация! Отпечатки пальцев 93

Базовый стиль описания 93

Контрпримеры 94

6.12. Исполнители 95

6.13. Диаграммы прецедентов 95

Система обозначений для диаграммы прецедентов 97

Предупреждение 97
6.14. Требования и списки низкоуровневых свойств 98
Списки высокоуровневых свойств системы вполне допустимы 99
Когда нужен подробный список свойств? 99
6.15. Объектно-ориентированный подход и прецеденты 99
6.16. Прецеденты в рамках унифицированного процесса 100
Прецеденты и спецификация требований в рамках итеративного процесса 100
Временные рамки создания артефактов UP 101
Прецеденты начальной фазы 102
Прецеденты на стадии развития 103
Прецеденты на стадии конструирования 103
6.17. Пример: прецеденты начальной фазы проекта NextGen 104
6.18. Дополнительная литература 104
6.19. Артефакты UP в контексте процесса 104

**Глава 7. Определение остальных требований 107**

Введение 107
7.1. Примеры для системы NextGen 108
7.2. Пример NextGen: фрагмент дополнительной спецификации 108
7.3. Дополнительная спецификация (комментарии) 111
Атрибуты качества 112
Бизнес-правила 113
Информация из предметной области 113
7.4. Пример для системы NextGen: в*и*дение (фрагмент) 114
7.5. Видение (комментарий) 117
Ту ли проблему мы решаем? 117
Системные свойства — функциональные требования 117
Другие требования в документе "Видение" 120
Видение, свойства или прецеденты — что раньше? 120
7.6. Пример системы NextGen: словарь терминов (фрагмент) 120
7.7. Комментарии: словарь терминов 121
Словарь терминов в роли словаря данных 121
Единицы измерения 122
Составные термины 122
7.8. Надежные спецификации — нет ли здесь противоречия? 122
7.9. Размещение артефактов на Web-узле проекта 123
7.10. Не слишком ли много UML на начальной стадии проекта? 123
7.11. Остальные артефакты дисциплины определения требований в рамках UP 123
Начальная фаза 123
Фаза развития 124
Фаза конструирования 125
7.12. Дополнительная литература 125
7.13. Артефакты UP в контексте процесса 125

**Глава 8. От начальной фазы к стадии развития 129**

Введение 129
8.1. Контрольная точка: что сделано на начальной стадии? 129
8.2. Фаза развития 130
Что является архитектурно важным на стадии развития? 131
8.3. Планирование следующей итерации 132
8.4. Требования и акценты первой итерации: основные вопросы ООА/П 133
Первая итерация: требования 133

Инкрементальная разработка одного и того же прецедента в течение нескольких итераций 134
8.5. Разработка каких артефактов начинается на стадии развития? 134
8.6. Вы не поняли, что такое фаза развития, если... 135

ЧАСТЬ III. ПЕРВАЯ ИТЕРАЦИЯ ФАЗЫ РАЗВИТИЯ 137

**Глава 9. Модель прецедентов: диаграммы последовательностей 139**
Переход к первой итерации 139
Введение 140
9.1. Поведение системы 140
9.2. Диаграммы последовательностей системы 140
9.3. Пример диаграммы последовательностей 141
9.4. Межсистемные диаграммы последовательностей 142
9.5. Системные события и прецеденты 142
9.6. Системные события и границы системы 142
9.7. Имена системных событий и операций 143
9.8. Отображение текста из описания прецедента 144
9.9. Диаграммы последовательностей и словарь терминов 144
9.10. Диаграммы последовательностей в контексте UP 144
Фазы 145
9.11. Дополнительная литература 145
9.12. Артефакты UP 146

**Глава 10. Модель предметной области: визуализация понятий 147**
Введение 147
10.1. Модели предметной области 148
Основная идея: модель предметной области — это визуальный словарь абстракций 149
Концептуальная модель — это не модель программных компонентов 149
Концептуальные классы 150
Модели предметной области и декомпозиция 151
Концептуальные классы предметной области торговли 151
10.2. Идентификация концептуальных классов 152
Стратегии идентификации концептуальных классов 152
Использование списка категорий концептуальных классов 153
Определение концептуальных классов с помощью выявления существительных 154
10.3. Кандидатуры на роль концептуальных классов для предметной области торговли 155
Объекты отчета: включать ли понятие “товарный чек” в модель 155
10.4. Принципы создания модели предметной области 155
Как создать модель предметной области 155
Имена и модели: стратегия построения карт 156
Типичная ошибка при выделении концептуальных классов 157
10.5. Разрешение конфликта сходных концептуальных классов: Register или POST 157
10.6. Моделирование “нереального” мира 158
10.7. Спецификация или описание концептуальных классов 159
Необходимость спецификаций или описание концептуальных классов 159
Когда требуются понятия-спецификации 160
Еще один пример спецификации 160
Описания услуг 161
10.8. Термины языка UML, модели и методы: различные ракурсы 161

Согласование терминологии, принятой в различных методах и UML 163
10.9. Сокращение разрыва в представлениях 164
10.10. Пример: модель предметной области POS-системы NextGen 165
10.11. Модели предметной области в рамках UP 166
Начальная фаза 166
Фаза развития 166
Объектная бизнес-модель и модель предметной области в рамках UP 166
10.12. Дополнительная литература 167
10.13. Артефакты UP 167

**Глава 11. Модель предметной области: добавление ассоциаций 169**

Введение 169
11.1. Ассоциации 169
Поиск ассоциаций 170
11.2. Система обозначений для ассоциаций языка UML 170
11.3. Поиск ассоциаций: список стандартных ассоциаций 171
Ассоциации с высоким приоритетом 172
11.4. Рекомендации по назначению ассоциаций 172
11.5. Роли 172
Кратность 173
11.6. Насколько важны ассоциации 174
11.7. Имена ассоциаций 175
11.8. Несколько ассоциаций между двумя типами 175
11.9. Ассоциации и их реализация 176
11.10. Ассоциации для предметной области POS-системы NextGen 177
Отношения в магазине, которые должны быть учтены 177
Использование списка категорий ассоциаций 177
11.11. Модель предметной области POS-системы NextGen 178
Сохранение только важных ассоциаций 178
Важные и второстепенные ассоциации 179

**Глава 12. Модель предметной области: добавление атрибутов 181**

Введение 181
12.1. Атрибуты 181
12.2. Система обозначений атрибутов в языке UML 182
12.3. Корректные типы атрибутов 182
Атрибуты должны быть простыми 182
Концептуальный аспект и аспект реализации: реализация атрибутов в программном коде 183
Типы данных 183
12.4. Непримитивные типы классов 184
Как иллюстрировать классы типов данных 185
12.5. Совет разработчикам: не используйте атрибуты в качестве внешних ключей 186
12.6. Моделирование атрибутов `Quantity` и `Unit` 186
12.7. Атрибуты модели предметной области системы NextGen 186
11.8. Кратная ассоциация между понятиями SalesLineItem и `Item` 188
12.9. Заключительные замечания о модели предметной области 188

**Глава 13. Модель прецедентов: детализация с помощью описаний операций 191**

Введение 191
13.1. Описания 191
Системные операции и системный интерфейс 191

13.2. Пример описания системной операции: enterItem 192
13.3. Разделы описания 193
13.4. Постусловия 193
Связь постусловий с моделью предметной области 194
Преимущества описания постусловий 194
Дух постусловий: сцена и занавес 194
Насколько детальными должны быть постусловия описаний 195
13.5. Обсуждение: постусловия описания операции enterItem 195
Создание и удаление экземпляра 195
Модификация атрибута 195
Формирование и разрыв ассоциации 196
13.6. Описание операций приводит к изменению предметной области 196
13.7. Когда нужны описания операций? Что лучше: описания операций или прецедентов? 196
13.8. Составление описания 197
Советы по составлению описаний системных операций 197
Наиболее типичная ошибка при создании описаний системных операций 197
13.9. Пример POS-системы NextGen: описания 198
Системные операции для прецедента `Оформление продажи` 198
13.10. Изменение модели предметной области 199
13.11. Описания, операции и UML 199
Описание операций на языке OCL 199
Проектирование на основе описаний 200
Языки программирования и описания операций 200
13.12. Описания операций в рамках UP 200
Фазы 200
Взаимосвязь артефактов 201
13.13. Дополнительная литература 202

**Глава 14. От анализа требований к проектированию на данной итерации 205**

Введение 205
14.1. Занимаясь итеративной разработкой, делайте это правильно 205
14.2. Возможно, на это потребуются недели? Вовсе нет 206
14.3. Переход к проектированию 206
Важнее иметь навыки объектного проектирования, чем знать систему обозначений UML 206

**Глава 15. Система обозначений для диаграмм взаимодействия 209**

Введение 209
Рекомендации по проектированию приводятся в следующих главах 209
15.1. Диаграммы последовательностей и кооперации 210
15.2. Пример диаграммы кооперации: `makePayment` 211
15.3. Пример диаграммы последовательностей 211
15.4. Диаграммы взаимодействия — это важный артефакт 211
15.5. Основные обозначения для диаграмм взаимодействия 213
Отображение классов и экземпляров объектов 213
Синтаксис для отображения сообщений 213
15.6. Основные обозначения диаграммы кооперации 213
Отображение связей 213
Отображение сообщений 214
Сообщения, передаваемые самому объекту 214
Создание экземпляров объектов 214
Представление порядка передачи сообщений 215

Представление условных сообщений 216
Представление взаимоисключающих условных маршрутов 216
Представление итерационного процесса или циклов 217
Итерационный процесс для коллекций объектов 217
Сообщения, передаваемые классу 217
15.7. Основные обозначения диаграммы последовательностей 218
Связи 218
Сообщения 218
Фокус управления и блоки активации 219
Возвращаемые значения 219
Создание зкземпляров объектов 219
Сообщения, передаваемые самому объекту 219
Линии жизни объектов и уничтожение объектов 220
Представление условных сообщений 220
Представление взаимоисключающих условных маршрутов 220
Представление итерационного процесса для одного сообщения 221
Итерационный процесс для последовательности сообщений 221
Итерационный процесс для коллекции (сложного объекта) 222
Сообщения, передаваемые классу 222
**Глава 16. GRASP: шаблоны для распределения обязанностей 223**
Введение 223
GRASP — это методический подход к объектному проектированию 223
16.1. Обязанности и методы 224
16.2. Обязанности и диаграммы взаимодействий 224
16.3. Шаблоны 225
Шаблоны не содержат новых идей 226
Шаблоны имеют имена 226
16.4. Шаблоны GRASP: общие принципы распределения обязанностей 227
Как применять шаблоны GRASP 227
16.5. Система обозначений диаграммы классов в языке UML 228
16.6. Шаблон Information Expert 228
16.7. Шаблон Creator 233
16.8. Шаблон Low Coupling 236
16.9. Шаблон High Cohesion 239
16.10. Шаблон Controller 243
16.11. Проектирование объектов и карты CRC 252
16.12. Дополнительная литература 252
**Глава 17. Модель проектирования: реализация прецедентов на основе шаблонов GRASP 253**
Введение 253
17.1. Реализация прецедентов 254
17.2. Комментарии к артефактам 254
Диаграммы взаимодействия и реализация прецедентов 254
Описания системных операций и реализация прецедентов 255
Предупреждение: требования могут измениться 257
Модель предметной области и реализация прецедентов 257
Концептуальные классы и классы модели проектирования 257
17.3. Реализация прецедентов для данной итерации разработки системы NextGen 258
17.4. Проектное решение: makeNewSale 258
Выбор класса-контроллера 259
Создание нового экземпляра объекта Sale 260

Заключение 261
17.5. Проектное решение: enterItem 261
Выбор класса-контроллера 261
Отображение описания товара и его цены 261
Создание экземпляров SalesLineItem 261
Нахождение ProductSpecification 262
Обеспечение видимости класса ProductCatalog 263
Получение значения ProductSpecification из базы данных 263
Диаграмма взаимодействия для системной операции enterItem 263
Сообщения сложным объектам 263
17.6. Проектное решение: endSale 264
Выбор класса-контроллера 265
17.6.1. Установка значения атрибута Sale.isComplete 265
Обозначения UML для ограничений, комментариев и описаний алгоритмов 265
Вычисление общей стоимости покупки 266
Проектное решение Sale--getTotal 268
17.7. Проектное решение: makePayment 268
Создание экземпляра Payment 269
Регистрация покупки 271
Вычисление причитающейся сдачи 272
17.8. Проектное решение: startUp 273
Когда создавать диаграмму взаимодействия для системной операции startUp 273
Как запускаются приложения 274
Интерпретация системной операции startUp 275
Операция startUp POS-приложения 275
Выбор исходного объекта предметной области 275
Объекты из базы данных: ProductSpecification 276
Проектное решение: Store--create() 276
17.9. Подключение уровня интерфейса пользователя к уровню предметной области 277
17.10. Реализация прецедентов в рамках UP 280
Фазы 281
Артефакты UP в контексте унифицированного процесса 281
17.11. Резюме 284
**Глава 18. Модель проектирования: области видимости 285**
Введение 285
18.1. Видимость объектов 285
18.2. Области видимости 286
Обеспечение видимости посредством атрибутов 287
Обеспечение видимости посредством параметров 287
Локальная видимость 288
Глобальная видимость 290
18.3. Иллюстрация видимости между объектами средствами языка UML 290
**Глава 19. Модель проектирования: создание диаграммы классов 291**
Введение 291
19.1. Когда следует создавать диаграммы классов 291
19.2. Пример диаграммы классов 292
19.3. Диаграммы классов и терминология UP 292
19.4. Классы из модели предметной области и модели проектирования 293
19.5. Создание диаграммы классов для POS-системы NextGen 293

Идентификация программных классов и их отображение 293
Добавление имен методов 294
Выбор имен методов 295
Имена методов: create 295
Имена методов: методы доступа 295
Имена методов: сложные объекты 296
Имена методов: синтаксис с учетом языка 296
Добавление дополнительной информации о типах 296
Добавление ассоциаций и информации о навигации 297
Добавление зависимостей 300
19.6. Обозначения для детальной информации о членах класса 301
Параметры видимости, используемые по умолчанию 302
Отображение тела метода на диаграмме классов 303
19.7. Построение диаграмм классов и CASE-средства 304
19.8. Диаграммы классов в контексте UP 304
Фазы 304
19.9. Артефакты в контексте унифицированного процесса 305

**Глава 20. Модель реализации: преобразование результатов проектирования в программный код** **307**

Введение 307
Выбор языка программирования 307
20.1. Программирование и процесс разработки 308
Внесение изменений на стадии реализации 308
Модификация кода и итеративный процесс 309
Изменение кода, CASE-средства и обратное проектирование 309
20.2. Преобразование результатов проектирования в программный код 310
20.3. Создание определений классов на основе диаграмм классов 310
Определение класса с методами и простыми атрибутами 310
Добавление атрибутов-ссылок 310
Атрибуты-ссылки и имена ролей 312
Отображение атрибутов 312
20.3. Создание методов на основе диаграмм взаимодействия 312
Метод Register--enterItem 313
20.5. Классы-контейнеры в программном коде 315
20.6. Исключения и обработка ошибок 315
20.7. Определение метода Sale--makeLineItem 316
20.8. Порядок реализации 316
20.9. Программирование на основе тестирования 316
20.10. Заключительные замечания по поводу преобразования проектного решения в код 318
20.11. Основное программное решение 318

ЧАСТЬ IV. ВТОРАЯ ИТЕРАЦИЯ ФАЗЫ РАЗВИТИЯ 323

**Глава 21. Вторая итерация и требования к ней** **325**

21.1. Вторая итерация: объектное проектирование и шаблоны 325
21.2. От первой ко второй итерации 325
Упрощения, принятые при рассмотрении примера 326
21.3. Требования для второй итерации 327
Инкрементальная разработка одного и того же прецедента в течение нескольких итераций 328
21.4. Уточнение артефактов стадии анализа 328
Модель прецедентов: прецеденты 328
Модель прецедентов: диаграммы последовательностей 328

Модель предметной области 329
Модель прецедентов: описание системных операций 330

**Глава 22. Дополнительные шаблоны GRASP для распределения обязанностей 331**

Введение 331
22.1. Шаблон Polymorphism 332
22.2. Шаблон Pure Fabrication 335
22.3. Шаблон Indirection 338
22.4. Шаблон Protected Variations 339

**Глава 23. Реализация прецедентов с использованием шаблонов GoF 347**

Введение 347
Шаблоны Gang-of-Four 347
Словарь терминов 348
23.1. Шаблон Adapter (GoF) 348
Шаблоны Polymorphism, Indirection и Protected Variations (GRASP) 350
Соглашение об именовании: включать ли имя шаблона в имя типа 350
23.2. Анализ на этапе проектирования: модель предметной области 350
Поддержка модели предметной области 351
23.3. Шаблон Factory (GoF) 352
23.4. Шаблон Singleton (GoF) 353
Обозначение UML для доступа к единственному экземпляру на диаграмме взаимодействия 355
Вопросы реализации и проектирования 355
23.5. Еще несколько слов о внешних службах с разными интерфейсами 357
23.6. Шаблон Strategy (GoF) 357
Создание объекта стратегии на основе шаблона Factory 359
Вывод 362
23.7. Шаблон Composite (GoF) и другие принципы проектирования 362
Создание нескольких стратегий `SalePricingStrategy` 367
Резюме 370
23.8. Шаблон Facade (GoF) 371
Резюме 373
23.9. Шаблон Observer/Publish-Subscribe/ Delegation Event Model (GoF) 374
Почему этот шаблон называется Observer, Publish-Subscribe или Delegation Event Model 378
Шаблон Observer не только связывает интерфейс пользователя с объектами модели 379
На одно событие могут подписаться несколько пользователей 380
Реализация 381
Резюме 382
23.10. Заключение 382
23.11. Дополнительная литература 382

ЧАСТЬ V. ТРЕТЬЯ ИТЕРАЦИЯ ФАЗЫ РАЗВИТИЯ 383

**Глава 24. Третья итерация и ее требования 385**

24.1. Требования третьей итерации 385
24.2. Наиболее важные вопросы на стадии третьей итерации 385

**Глава 25. Взаимосвязь прецедентов 387**

Введение 387
Предостережение 387

25.1. Отношение включает 388
Отношение включает и обработка асинхронных событий 389
Заключительные замечания 390
25.2. Новые термины: конкретный, абстрактный, основной и дополнительный прецеденты 390
25.3. Отношение расширяет 391
25.4. Отношение обобщает 392
25.5. Диаграммы прецедентов 392
**Глава 26. Обобщение модели** **395**
Введение 395
26.1. Модель предметной области: новые понятия 395
Список категорий понятий 395
Определение понятий из текстовых описаний 396
Транзакции со службами авторизации 397
26.2. Что такое обобщение 397
26.3. Определение концептуальных суперклассов и подклассов 398
Обобщение и определение классов 399
Обобщение и множества классов 399
Совместимость определений подклассов 399
Совместимость множества подкласса 400
Что такое корректный подкласс 401
26.4. Когда нужно определять концептуальный подкласс 401
В каких случаях класс нужно разделять на подклассы 402
26.5. Когда нужно определять концептуальный суперкласс 403
26.6. Иерархия классов POS-системы NextGen 403
Классы Payment 403
Классы служб авторизации 403
Классы транзакций авторизации 404
26.7. Абстрактные классы 405
Обозначение абстрактных классов в UML 406
26.7. Моделирование изменения состояний 406
26.9. Иерархия классов и наследование 407
**Глава 27. Усовершенствование модели предметной области** **409**
Введение 409
27.1. Классы ассоциаций 409
Рекомендации 411
27.2. Агрегация и объединение 412
Агрегация в языке UML 412
Композитная агрегация: затененный ромб 413
Совместная агрегация: полый ромб 413
Как идентифицировать отношение агрегации 414
Преимущества отображения отношений агрегации 414
Отношение агрегации в модели предметной области POS-системы 415
27.3. Временные интервалы и цены товаров: устранение ошибки первой итерации 415
27.4. Имена ролей ассоциаций 416
27.5. Роли в форме понятий и роли, представленные в ассоциации 417
27.6. Производные элементы 418
27.7. Составные ассоциации 419
27.8. Рефлексивные ассоциации 420
27.9. Упорядоченные элементы 420
27.10. Использование пакетов для организации элементов модели предметной области 420

Обозначение пакетов в языке UML 420
Как разделить модель предметной области 422
Пакеты модели предметной области POS-системы 422
Пакет `Core/Misc` 422
Пакет `Payments` 422
Пакет `Products` 423
Пакет `Sales` 425
Пакет `Authorization Transactions` 426
**Глава 28. Поведение системы** **427**
28.1. Новые диаграммы последовательностей 427
Стандартное начало сценария прецедента `Оформление продажи` 427
Оплата с использованием кредитной карточки 427
Оплата чеком 429
28.2. Новые системные операции 429
28.3. Описания новых системных операций 429
**Глава 29. Моделирование поведения на диаграммах состояний** **431**
Введение 431
29.1. События, состояния и переходы 431
29.2. Диаграммы состояний 432
Предмет диаграммы состояний 432
29.3. Диаграммы состояний и UP 433
29.4. Диаграммы состояний для прецедентов 433
Преимущества диаграмм состояний прецедентов 433
29.5. Диаграммы состояний прецедентов для POS-приложения 434
Прецедент `Оформление продажи` 434
29.6. Для каких классов необходимы диаграммы состояний 434
Объекты, независимые и зависимые от состояний 435
Зависимые от состояния стандартные классы 435
29.7. Изображение внешних и внутренних событий 436
Типы событий 436
Диаграммы состояний для внутренних событий 436
29.8. Дополнительные обозначения для диаграмм состояний 437
Действия и условия, связанные с переходами 437
Вложенные состояния 437
29.9. Дополнительная литература 438
**Глава 30. Проектирование систем на основе шаблонов** **439**
Введение 439
30.1. Архитектура программной системы 439
Архитектурные представления в унифицированном процессе 440
Архитектурные шаблоны и категории шаблонов 440
30.2. Архитектурный шаблон Layers 441
30.3. Принцип Model-View Separation 463
Принцип Model-View Separation и взаимодействие “снизу вверх” 465
30.4. Дополнительная литература 466
**Глава 31. Создание модели проектирования и реализации на основе пакетов** **467**
Введение 467
Исходный код физической архитектуры в модели реализации 468
31.1. Рекомендации по организации пакетов 468
Вертикальное и горизонтальное зацепление функциональности пакетов 468
Пакет, представляющий собой семейство интерфейсов 469

Формирование пакетов на базе групп неустойчивых классов 469
Базовые пакеты должны быть более устойчивыми 470
Факторизация независимых типов 471
Факторизация для снижения зависимости от конкретных пакетов 471
Не используйте циклы в пакетах 472
31.2. Дополнительные обозначения UML для пакетов 473
31.3 Дополнительная литература 473

**Глава 32. Введение в архитектурный анализ 475**

Введение 475
32.1. Архитектурный анализ 476
Общие методы архитектурного анализа 477
32.2. Типы представлений архитектуры 478
32.3. Определение и анализ архитектурных факторов 478
Архитектурные факторы 478
Сценарии реализации качественных требований 479
Описание архитектурных факторов 479
Факторы и артефакты UP 480
32.4. Пример: фрагмент таблицы архитектурных факторов POS-системы NextGen 481
32.5. Определение архитектурных факторов 483
Альтернативы, решения и мотивировка 483
Приоритеты 485
Основные принципы проектирования архитектуры 486
Архитектурные шаблоны 488
32.6. Выводы 489
32.7. Архитектурный анализ в UP 489
Предупреждение: каскадный архитектурный анализ 489
Архитектурная информация в артефактах UP 489
Документ SAD и архитектурные представления 489
Фазы разработки 492
32.8. Дополнительная литература 493

**Глава 33. Реализация новых прецедентов на основе объектов и шаблонов 495**

Введение 495
33.1. Отказ локальных служб и обеспечение локальной буферизации 495
33.2. Обработка отказов 500
Генерирование исключений 502
Исключения в UML 503
Обработка ошибок 505
33.3. Взаимодействие с локальными службами на основе шаблона Proxy (GoF) 507
33.4. Реализация нефункциональных или качественных требований 511
33.5. Доступ к внешним физическим устройствам с помощью шаблона Adapter: купить или разработать 511
33.6. Шаблон Abstract Factory (GoF) для семейства взаимосвязанных объектов 512
Абстрактный класс абстрактной фабрики 513
33.7. Обработка платежей на основе шаблонов Polymorphism и Do It Myself 516
"Мелкие" классы 517
Авторизация платежей по кредитной карточке 518
33.8. Заключение 521
Предупреждение: "злоупотребление" шаблонами 521

**Глава 34. Проектирование контура взаимодействия с базой данных на основе шаблонов** 523
Введение 523
34.1. Проблема: объекты, подлежащие постоянному хранению 524
Механизмы хранения и постоянно хранимые объекты 524
34.2. Решение: контур интерфейса с базой данных 524
34.3. Контуры 525
Контуры предназначены для повторного использования 525
34.4. Требования к контуру интерфейса с базой данных 526
34.5. Ключевые идеи 526
34.6. Шаблон представления объектов в виде таблиц 527
34.7. Профиль моделирования данных UML 527
34.8. Шаблон Object Identifier 528
34.9. Доступ к службе взаимодействия с базой данных на основе шаблона Facade 529
34.10. Объекты-преобразователи: шаблон Database Mapper или Database Broker 529
Преобразователи на основе метаданных 531
34.11. Разработка контура на основе шаблона Template Method 532
34.12. Материализация на основе шаблона Template Method 532
Унифицированный процесс и описание программной архитектуры 537
Синхронизированные методы в UML 538
34.13. Настройка преобразователей с помощью объекта MapperFactory 538
34.14. Шаблон Cache Management 539
34.15. Объединение и сокрытие операторов SQL в одном классе 539
34.16. Состояние транзакции и шаблон State 540
34.17. Обработка транзакций на основе шаблона Command 543
34.18. Пассивная материализация на основе шаблона Virtual Proxy 545
Реализация виртуального объекта-посредника 547
Кто создает виртуальные объекты-посредники 548
34.19. Представление отношений в таблицах 549
34.20. Суперкласс `PersistentObject` 549
34.21. Нерешенные вопросы 550

ЧАСТЬ VI. СПЕЦИАЛЬНЫЕ ВОПРОСЫ 551

**Глава 35. Средства создания диаграмм** 553
Введение 553
35.1. Умозрительное проектирование и визуальное мышление 553
35.2. Предложения по построению диаграмм UML в рамках процесса разработки 554
Трудозатраты 554
Другие рекомендации 554
35.3. CASE-средства (примеры) 556
Неполная поддержка обозначений UML 557
Первый пример 557
Второй пример 557
Критерии выбора CASE-средства разработки 558

**Глава 36. Знакомство с итеративным планированием и проектированием** 559
Введение 559
36.1. Ранжирование требований 560
Основные критерии для начальных итераций: риск, охват, критичность, выработка навыков 560

Что ранжировать 560
Качественные методы ранжирования 560
Количественные методы ранжирования 561
Ранжирование требований к POS-системе NextGen 561
Прецеденты Запуск системы и Завершение работы 562
Предостережение: планирование проекта и задачи обучения 562
36.2. Ранжирование рисков проекта 562
36.3. Адаптивное и предиктивное планирование 563
36.4. Планы для фазы и итерации 564
36.5. План итерации: что делать на следующей итерации 564
36.6. Отслеживание требований в процессе итераций 566
Пример работы средства управления требованиями 566
36.7. Приблизительность начальных оценок 566
36.8. Организация артефактов проекта 567
36.9. Некоторые вопросы составления графика работ 568
Скорость групповой работы и постепенная настройка процесса 569
36.10. Вы не поняли принципов планирования в UP, если... 569
36.11. Дополнительная литература 570
**Глава 37. Комментарии к итеративной разработке и UP 571**
37.1. Дополнительные приемы и понятия UP 571
37.2. Фазы конструирования и передачи 573
Фаза конструирования 573
Стадия передачи 574
37.3. Другие интересные приемы 574
37.4. Планирование длительности итерации 574
37.5. Последовательный цикл каскадной разработки 575
Некоторые проблемы каскадного жизненного цикла 576
"Смягчение" некоторых проблем в рамках каскадного жизненного цикла 576
37.6. Проектирование интерфейса пользователя с учетом удобства использования 580
37.7. Модель анализа UP 580
37.8. Продукт RUP 580
37.9. Несколько слов о повторном использовании 581
**Глава 38. Дополнительные обозначения UML 583**
38.1. Символы общего назначения 583
Зависимости 583
Стереотипы и обозначения свойств с помощью тегированных значений 584
Интерфейсы пакетов 584
38.2. Диаграммы реализации 585
Диаграммы компонентов 585
Диаграммы развертывания 585
38.3. Классы шаблонов (параметризированные или родовые) 586
38.4. Диаграммы видов деятельности 587
**Приложение. Артефакты унифицированного процесса, шаблоны GRASP и условные обозначения языка UML 589**
Примерный набор артефактов унифицированного процесса и время их создания (н — начало, р — развитие) 589
Пример взаимосвязи артефактов унифицированного процесса 590
Основные шаблоны распределения обязанностей (шаблоны GRASP) 591
Диаграммы последовательностей 592

Диаграммы кооперации 592
Дополнительные обозначения диаграмм классов 593
Диаграммы прецедентов 595
**Литература** **597**
**Словарь терминов** **603**
**Предметный указатель** **609**

*Джулии.*

*Без твоей поддержки написать эту книгу*
*было бы невозможно.*

*Халли и Анне.*

*Еще раз благодарю за понимание обезумевшего отца.*

# Предисловие

Программирование — это развлечение, но разработка качественных программ — тяжелый труд. Между хорошей идеей, формулировкой требований, составлением “видения” и готовым программным продуктом лежит гораздо больше, чем просто программирование. Анализ и проектирование, определение способов решения задачи, выбор программных средств, удобное представление проектного решения, реализация и сопровождение программы — это основные вопросы, которые рассматриваются в данной книге. Это то, о чем вам предстоит из нее узнать.

Унифицированный язык моделирования UML стал общепринятым языком разработки программных систем. Это визуальный язык, используемый для изложения основных идей. В книге основное внимание уделяется применению элементов языка UML, а не особенностям самого языка.

При разработке сложных систем важную роль играют шаблоны. Шаблоны проектирования программ позволяют описать фрагменты проектного решения, повторно использовать хорошие идеи и перенять полезный опыт других разработчиков. Шаблоны имеют имена и определяются в форме абстрактных эвристик, правил и лучших наработок объектно-ориентированной технологии. Ни один здравомыслящий разработчик не захочет начинать создание системы с чистого листа. Эта книга как раз и предоставляет целую палитру готовых проектных решений.

Однако проектные решения выглядят слишком сухо и неестественно в отрыве от контекста процесса разработки. В этой связи стоит отметить, что во втором издании книги автор, Крэг Ларман (Craig Larman), положил в основу разработки унифицированный процесс проектирования UP, показал, как создавать системы в рамках этого процесса. В книге достаточно реалистично показан пример разработки системы в рамках итеративного процесса с учетом рисков и архитектурных особенностей. Разработка программной системы показана в динамике. Виды деятельности процесса разработки увязаны с другими задачами, поэтому они больше не выглядят оторванными от жизни. Автор книги и я оценили реальные преимущества итеративной разработки и постараемся донести их до читателя.

На мой взгляд, в книге описано множество взаимосвязанных компонентов. Здесь систематически изложены основы объектно-ориентированного анализа и проектирования. Автор книги — талантливый преподаватель, замечательный методист и признанный гуру объектно-ориентированного подхода, имеющий тысячи учеников во всем мире. Крэг Ларман описал этот метод в контексте унифицированного процесса. Он постепенно вводит все более сложные шаблоны, что делает книгу особенно полезной “при столкновении” лицом к лицу с реальными жизненными сложностями. Кроме того, здесь использована наиболее широко применяемая система обозначений.

Я очень рад, что мне представилась возможность работать напрямую с автором этой книги. Я с удовольствием прочел ее первое издание и был польщен предложением рецензировать второе. В процессе работы над книгой мы несколько раз встречались и интенсивно обменивались электронными сообщениями. Я многому научился у Крэга, в том числе в контексте унифицированного процесса разработки и его реализации в конкретной организации. Надеюсь, книга станет очень полезной и для вас, даже если вы уже знакомы с принципами объектно-

ориентированного анализа и проектирования. Как и я, вы сможете освежить в памяти его основные принципы и углубить свои знания с помощью доступных и грамотных пояснений Крэга.

В контексте итеративного процесса разработка в рамках второй итерации опирается на результаты первой. Это же относится и к книгам. Я надеюсь, что даже прочитав первое издание, вы с удовольствием познакомитесь и со вторым.

Приятного чтения!

Филипп Крачтен (Philippe Kruchten),
сотрудник компании Rational Software,
Ванкувер (Vancouver), Канада

# Введение

Спасибо за проявленное внимание к этой книге! Перед вами — практическое введение в область объектно-ориентированного анализа и проектирования (ООА/П), а также в смежные области итеративного процесса разработки. Мне очень приятно, что первое издание этой книги имело большой успех во всем мире и было переведено на многие языки. Второе издание расширяет и обновляет (а не заменяет) первое. Хотелось бы особо поблагодарить всех читателей первого издания.

Эта книга будет полезна читателям по следующим причинам.

**Во-первых**, использование объектной технологии предполагает разработку программного обеспечения на основе принципов ООА/П. Владение этими вопросами обеспечивает создание робастных и простых в поддержке объектно-ориентированных систем.

**Во-вторых**, тот, кто еще не знаком с вопросами объектно-ориентированного анализа и проектирования, наверняка планирует освоить эту область знаний. Данная книга станет хорошим путеводителем, поскольку в ней описан унифицированный процесс проектирования (unified process), понимание которого позволит шаг за шагом освоить путь от определения требований к системе до создания кода.

**В-третьих**, унифицированный язык моделирования UML является признанным стандартом для описания моделей, который обеспечивает возможность общения между разработчиками. В данной книге вопросы ООА/П освещаются с использованием системы обозначений UML.

**В-четвертых**, идиомы и удачные проектные решения при создании объектно-ориентированных систем были сформулированы в виде шаблонов, которые эксперты предлагают применять при создании систем. Из этой книги вы узнаете, как применять шаблоны проектирования, включая популярную “четверку”, а также шаблоны GRASP, в которых сконцентрированы фундаментальные принципы распределения обязанностей при объектно-ориентированном проектировании. Изучив и применив шаблоны, можно повысить уровень своего мастерства в области анализа и проектирования.

**В-пятых**, структура и содержание этой книги проверены годами практического опыта обучения специалистов искусству объектно-ориентированного анализа и проектирования. Этот опыт позволил выработать четкий, мотивированный и эффективный подход к изучению данного предмета, обеспечивающий оптимальность процесса чтения и обучения.

**В-шестых**, для иллюстрации всего процесса объектно-ориентированного анализа и проектирования в книге приводится исчерпывающее описание одного реального примера. Это достаточно реалистичное упражнение.

**В-седьмых**, здесь показано, как отобразить артефакты объектно-ориентированного проектирования в коде на языке Java.

**В-восьмых**, в книге рассказывается о том, как разработать многослойную архитектуру и связать уровень графического интерфейса пользователя со слоями реализации и технических служб.

И наконец, здесь рассказывается, как разработать объектно-ориентированный контур и использовать его для работы с базой данных.

## Основные задачи

Основная задача этой книги сводится к следующему.

| Помочь студентам и разработчикам в создании удачных объектно-ориентированных решений в результате применения набора обоснованных принципов и эвристик. |
|---|

В процессе изучения и применения приведенной здесь информации и предлагаемых методов можно глубоко изучить проблему в терминах ее процессов и понятий, а также разработать органичное программное решение с использованием объектного подхода.

## Для кого предназначена эта книга

В этой книге освещены вводные вопросы из области ООА/П, связанные с ними проблемы анализа требований и этапы итеративной разработки на примере унифицированного процесса. Она не предназначена для особо опытных специалистов, а рассчитана на следующую аудиторию.

- Разработчики с опытом создания программ на объектно-ориентированных языках, которые пока не являются экспертами в области объектно-ориентированного анализа и проектирования.
- Студенты компьютерных специальностей, изучающие объектную технологию.
- Специалисты по объектно-ориентированному анализу и проектированию, которые хотят изучить систему обозначений унифицированного языка моделирования UML (Unified Modeling Language), ознакомиться с шаблонами или углубить свои знания в области объектно-ориентированного анализа и проектирования.

## Что необходимо знать

Для успешного овладения предлагаемым материалом необходимо иметь следующие знания.

- Навыки и опыт программирования на объектно-ориентированном языке типа Java, C#, C++ или Smalltalk.
- Основные понятия объектной технологии, такие как класс, экземпляр, интерфейс, полиморфизм, инкапсуляция и наследование.

  Определения основных понятий объектной технологии в книге не приводятся.

## Примеры на языке Java

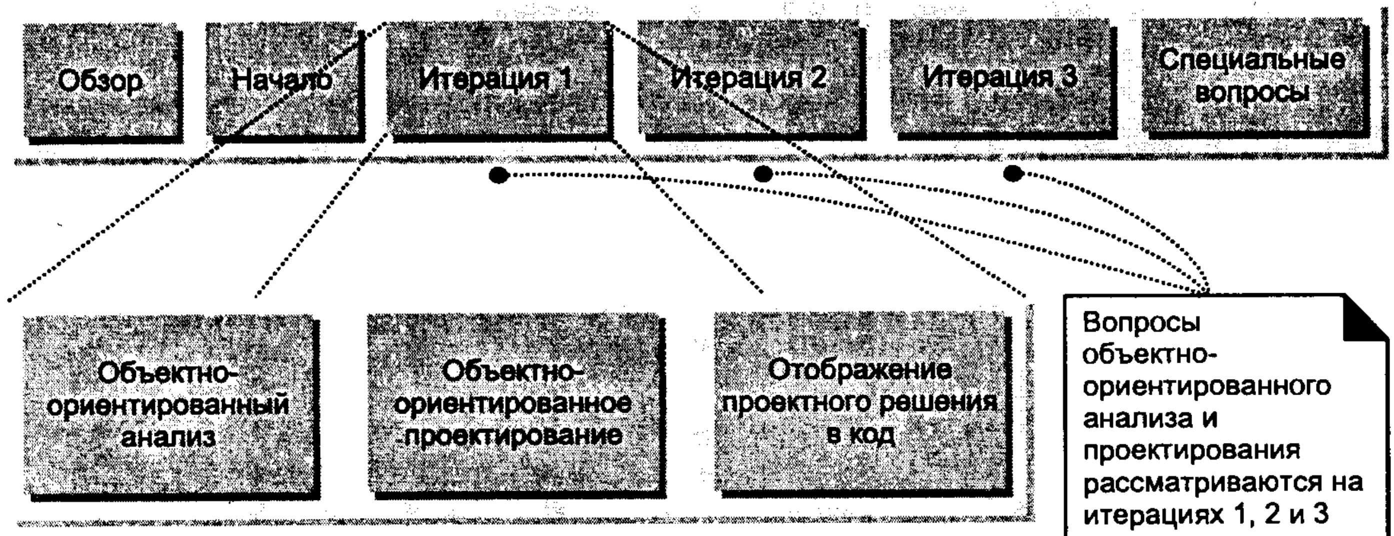
В книге приводятся примеры кода и обсуждаются вопросы реализации принципов ООА/П на языке Java, поскольку в последнее время он приобрел очень широкую популярность. Однако представленные здесь идеи можно применить к большинству (если не ко всем) объектно-ориентированных языков программирования.

## Структура книги

Общий принцип организации книги сводится к следующему. Вопросы объектно-ориентированного анализа и проектирования рассматриваются в той последовательности, в которой они возникают в процессе разработки системы в течение "начальной" фазы (термин из описания унифицированного процесса) и трех последовательных итераций.

1. В главах, посвященных "начальной" фазе, рассматриваются вопросы анализа требований.
2. При описании первой итерации ООА/П вводятся основные понятия анализа и проектирования, а также рассматриваются вопросы распределения обязанностей между объектами.
3. При переходе ко второй итерации основное внимание уделяется проектированию объектов, особенно некоторым популярным шаблонам проектирования.
4. При рассмотрении третьей итерации затрагивается множество вопросов, в том числе связанных с анализом архитектуры и проектированием контуров.

*Структура книги*




*Организация книги соответствует процессу разработки проекта*

## Ресурсы Web

- Статьи по вопросам объектной технологии, применению шаблонов и процессу разработки программных систем можно найти по адресу www.craiglarman.com.
- Некоторые ресурсы, связанные с вопросами обучения, содержатся также по адресу www.phptr.com/larman.

## Дополнения к первому изданию

Хотя в основу этой книги положено ее первое издание, она существенно обновлена.

- Модифицировано описание прецедентов в соответствии с очень популярным подходом, предложенным в [33].
- Для иллюстрации принципов ООА/П в качестве примера итеративного процесса разработки рассматривается широко известный унифицированный процесс UP (Unified Process). Поэтому все артефакты именуются в соответствии с терминологией UP. В частности, рассматривается модель предметной области.
- При рассмотрении примера конкретной системы вводятся новые требования, что влечет за собой необходимость третьей итерации проектирования.
- Обновлено описание шаблонов проектирования.
- Добавлено введение в архитектурный анализ.
- Для шаблонов GRASP вводятся защищенные вариации (Protected Variations).
- Выдержана пропорция 50/50 между диаграммами последовательностей и сотрудничества.
- Используется новейшая система обозначений языка UML.
- Обсуждаются некоторые практические аспекты применения CASE-средств.

## Благодарности

Во-первых, большое спасибо моим друзьям и коллегам из компании Valtech, которые внесли свой вклад в создание и рецензирование этой книги, в том числе Крису Тарру (Chris Tarr), Мишелю Эзрану (Michel Ezran), Тиму Снайдеру (Tim Snyder), Куртису Хайту (Curtis Hite), Челсо Гонзалесу (Celso Gonzalez), Паскалю Рокусу (Pascal Roques), Кену Делонгу (Ken DeLong), Бретту Шухерту (Brett Schuchert), Эшли Джонсону (Ashley Johnson), Крису Джонсу (Chris Jones), Томасу Лиоу (Thomas Liou), Дериллу Геберту (Darryl Gebert), Фрэнку Родориго (Frank Rodorigo), Джину-Ивсу Харди (Jean-Yves Hardy) и многим другим, кого я не в состоянии перечислить.

Спасибо Филиппу Крачтену (Philippe Kruchten) за написание предисловия, рецензирование книги и другую помощь.

Отдельное спасибо Мартину Фовлеру (Martin Fowler) и Алистеру Кокбурну (Alistair Cockburn) за плодотворное обсуждение процесса разработки и вопросов проектирования, а также за предоставленные материалы и рецензирование книги.

Благодарю Джона Влиссидеса (John Vlissides) и Криса Кобрина (Cris Kobryn) за удачные цитаты.

Большое спасибо сотрудникам компании Chelsea Systems и лично Джону Грею (John Gray) за помощь в формулировке требований к их системе ChelseaStore POS, разработанной с помощью технологии Java.

Благодарю Пита Коада (Pete Coad) и Дейва Астелса (Dave Astels) из компании TogetherSoft за поддержку.

Выражаю признательность другим рецензентам, в том числе Стиву Адольфу (Steve Adolph), Брюсу Андерсону (Bruce Anderson), Лену Бассу (Len Bass), Гари

К. Эвансу (Gary K. Evans), Элу Горнеру (Al Goerner), Люку Хохманну (Luke Hohmann), Эрику Лефебвру (Eric Lefebvre), Дэвиду Нанну (David Nunn) и Роберту Дж. Вайту (Robert J. White).

Благодарю Поля Беккера (Paul Becker) из издательства Prentice-Hall за его веру в успех первого издания, а также Поля Петралиа (Paul Petralia) и Патти Гурьери (Patti Guerrieri) за сопровождение второго.

И наконец, отдельное спасибо Грэхем Гласс (Graham Glass) за открытую дверь.

## Об авторе

Крэг Ларман (Craig Larman) работает техническим директором по вопросам процесса проектирования в международной консалтинговой компании Valtech, имеющей свои представительства в Европе, Азии и Северной Америке. Эта компания специализируется на разработке систем электронной коммерции, применении объектных технологий и итеративной разработке программных систем в рамках унифицированного процесса.

Начиная с середины 1980-х годов, Крэг помогал тысячам разработчиков освоить объектно-ориентированную технологию программирования, анализа и проектирования, а также консультировал организации по вопросам реализации итеративного процесса.

После неудачной попытки сделать карьеру странствующего уличного музыканта в 1970-е годы он занялся разработкой программных систем на языках APL, PL/1 и CISC. В начале 1980-х годов он стал "поклонником" искусственного интеллекта, методов проектирования баз знаний и способов обработки естественных языков и начал разрабатывать базы знаний на языках Lisp, Prolog и Smalltalk. Он солирует на гитаре в своей группе *Changing Requirements* (Изменчивые требования) (вообще-то, группа называется просто *Requirements*, но ее состав периодически изменяется).

Крэг Ларман имеет степени бакалавра и магистра компьютерных наук Ванкуверского университета имени Симона Фрейзера (Simon Fraser University), Канада.

С Крэгом Ларманом можно связаться по адресу `clarman@acm.org`, а также `www.craiglarman.com`.

## Соглашения, принятые в книге

*Новые термины* выделяются курсивом.

Имена классов и методов выделяются следующим образом: `Класс`, `метод`.

Ссылки на литературные источники заключаются в квадратные скобки, например [1].

Для обозначения принадлежности метода некоторому классу используется независимый от языка оператор разрешения контекста `--`, например `ИмяКласса--имяМетода`.

# Часть I

# Введение

Глава

# Объектно-ориентированный анализ и проектирование

*Смещение акцентов в сторону шаблонов окажет сильное и продолжительное влияние на способ написания программ.*

*Вард Каннингхем (Ward Cunningham), Ральф Джонсон (Ralph Johnson)*

Основные задачи

- Сопоставить и сравнить анализ и проектирование.
- Определить понятия объектно-ориентированного анализа и проектирования.
- Проиллюстрировать материал на простом примере.

## 1.1. Применение UML и шаблонов в процессе объектно-ориентированного анализа и проектирования

Какой смысл вкладывается в понятие "хорощо спроектированная объектно-ориентированная система"? Эта книга поможет разработчикам и студентам получить практические навыки объектно-ориентированного анализа и проектирования (ООА/П). Такие навыки необходимы для разработки профессиональных, робастных программных систем, созданных на основе объектной технологии проектирования и с помощью объектно-ориентированных языков программирования типа Java, C++, Smalltalk и C#.

Английская поговорка гласит: "Если у вас есть молоток, это еще не значит, что вы архитектор". Это особенно справедливо по отношению к объектной технологии. Владение объектно-ориентированным языком программирования (например, Java) — это необходимое, но не достаточное условие для создания объектной системы. Очень важно также уметь "мыслить объектами".

В этой книге содержится вводная информация о процессе объектно-ориентированного анализа и проектирования с помощью унифицированного языка моделирования UML (*Unified Modeling Language*), шаблонах и унифицированном процессе проектирования (рис. 1.1). Книга не рассчитана на опытных специалистов. Основное внимание уделяется основам проектирования, способам распределения обязанностей между объектами, часто используемой системе обозначений языка UML и типичным шаблонам проектирования. В то же время, особенно в последних главах, рассматриваемые темы излагаются более подробно, и речь идет о разработке контуров.

Однако эта книга не только о языке UML, представляющем собой систему обозначений, основанную на диаграммах. Нужно не только освоить эту систему обозначений; гораздо важнее изучить принципы объектно-ориентированного проектирования, научиться "мыслить объектами". Язык UML — это не принцип или метод ООА/П, это всего лишь система обозначений. Можно в совершенстве освоить синтаксис диаграмм UML и специализированные CASE-средства, но не научиться при этом разрабатывать новые эффективные программы или модифицировать существующие. Получить навыки разработки гораздо важнее и сложнее. И данная книга посвящена именно этой проблеме.

Однако для объектно-ориентированного анализа и проектирования нужен свой язык, позволяющий формулировать мысли и общаться с другими разработчиками. Поэтому в данной книге рассказывается о том, как *применить* язык UML в процессе ООА/П. Тем не менее, основное внимание здесь уделяется науке и искусству построения объектных систем, а не системе обозначений.

Как распределить обязанности между классами и объектами? Как должны взаимодействовать объекты? Какие функции выполняют конкретные классы? Эти вопросы являются определяющими при разработке системы. Некоторые проверенные временем решения проблем, возникающих в процессе разработки, могут быть (и были) сформулированы в виде набора принципов, эвристик или шаблонов (patterns) — именованных формул решения проблем, позволяющих систематизировать процесс разработки конкретных систем. Эта книга позволяет быстро научиться использовать такие фундаментальные идиомы объектно-ориентированного проектирования.

Изложение материала проиллюстрировано на едином общем примере, который рассматривается в книге. На этом примере достаточно глубоко изучены вопросы анализа и проектирования, при этом затронуто множество проблем, неизбежно возникающих при разработке реальной системы.

Процесс ООА/П (как и разработка программной системы в целом) тесно связан с подготовительным этапом или *анализом требований*, включающим в себя описание *прецедентов* (use case). Поэтому рассматриваемый в книге пример начинается с описания прецедентов, хотя это в строгом смысле не относится к объектно-ориентированному программированию.

Что должен делать разработчик или группа разработчиков, чтобы обеспечить реализацию требований к системе? Результаты анализа требований и ООА/П необходимо представить в контексте некоторого процесса разработки. В качестве примера *итеративного процесса разработки* выбран широко известный *унифицированный процесс*. Однако рассматриваемые методики анализа и проектирования применимы ко многим подходам, поэтому изучение их в контексте унифицированного процесса не ограничивает общности этих методов.

*Рис. 1.1. Рассмотренные вопросы и виды деятельности*

И наконец, эта книга поможет разработчикам:

- применять предложенные принципы и шаблоны для создания более совершенных систем;
- распределять стандартные виды деятельности в процессе анализа и проектирования в контексте унифицированного процесса;
- создавать типичные диаграммы в системе обозначений UML.

Все эти приемы проиллюстрированы на примере единой конкретной системы.

### Другие навыки

Для построения программных систем необходимо задействовать многие другие навыки, помимо рассматриваемых в книге анализа требований, ООА/П и объектно-ориентированного программирования. Например, на успех проекта в значительной степени влияют разработка пользовательского интерфейса или базы данных, а также тестирование системы.

Однако в книге основное внимание уделяется вопросам ООА/П, а не всем возможным проблемам создания программных систем. Нас интересует лишь один фрагмент общей картины.

## 1.2. Распределение обязанностей

На этапе анализа и проектирования могут использоваться различные виды деятельности и артефакты, а также принципы и рекомендации. Как же выделить среди всех рассмотренных здесь вопросов единственный, самый важный с практической точки зрения?

Наиболее важным моментом объектно-ориентированного анализа и проектирования является квалифицированное распределение обязанностей между программными компонентами.

На каком основании это можно утверждать? Дело в том, что это единственный вид деятельности, без которого невозможно обойтись. Кроме того, он оказывает определяющее влияние на робастность, масштабируемость, расширяемость и возможность повторного использования компонентов.

Конечно же, в процессе ООА/П решаются и другие важные вопросы, однако распределению обязанностей уделяется особое внимание, поскольку это самая сложная и жизненно важная задача. В реальном проекте разработчик может не иметь возможности выполнения любых других видов деятельности анализа или проектирования — зачастую участники проектов стараются немедленно приступить к написанию кода. Но даже в этой ситуации приходится распределять обязанности между разработчиками.

Поэтому при описании этапа проектирования основное внимание уделяется распределению обязанностей.

Для овладения навыками распределения обязанностей предлагается девять фундаментальных принципов, систематизированных в виде шаблонов GRASP.

## 1.3. Что такое анализ и проектирование

Этап *анализа* (analysis) состоит в исследовании системных требований и проблемы, а не в поисках путей ее решения. Например, при разработке новой информационной системы для компьютерной библиотеки необходимо описать экономические процессы, связанные с ее использованием.

Анализ — это достаточно широкое понятие. Его содержание более точно отражают термины *анализ требований* (requirements analysis) (т.е. исследование требований к системе) и *объектный анализ* (object analysis) (исследование объектов предметной области).

В процессе *проектирования* (design) основное внимание уделяется концептуальному решению, обеспечивающему выполнение основных требований, но не вопросам его реализации. Например, на этапе проектирования описываются программные объекты или схема базы данных. Конечно же, проектные решения могут быть реализованы в программном коде.

Как и понятие анализа, этот термин тоже стоит уточнить и говорить об *объектном проектировании* (object design) или *проектировании базы данных* (database design).

Подытоживая вышесказанное, ООА/П можно определить как "выявление правильных действий (анализ) и обеспечение их правильности (проектирование)".

## 1.4. Объектно-ориентированный анализ и проектирование

В процессе объектно-ориентированного анализа основное внимание уделяется определению и описанию объектов (или понятий) в терминах предметной области. Например, в случае библиотечной информационной системы среди понятий должны присутствовать `Book` (книга), `Library` (библиотека) и `Patron` (клиент).

В процессе объектно-ориентированного проектирования определяются программные объекты и способы их взаимодействия с целью выполнения системных требований. Например, в библиотечной системе программный объект `Book` может содержать атрибут `title` (название) и метод `getChapter` (определить номер главы) (рис.1.2).

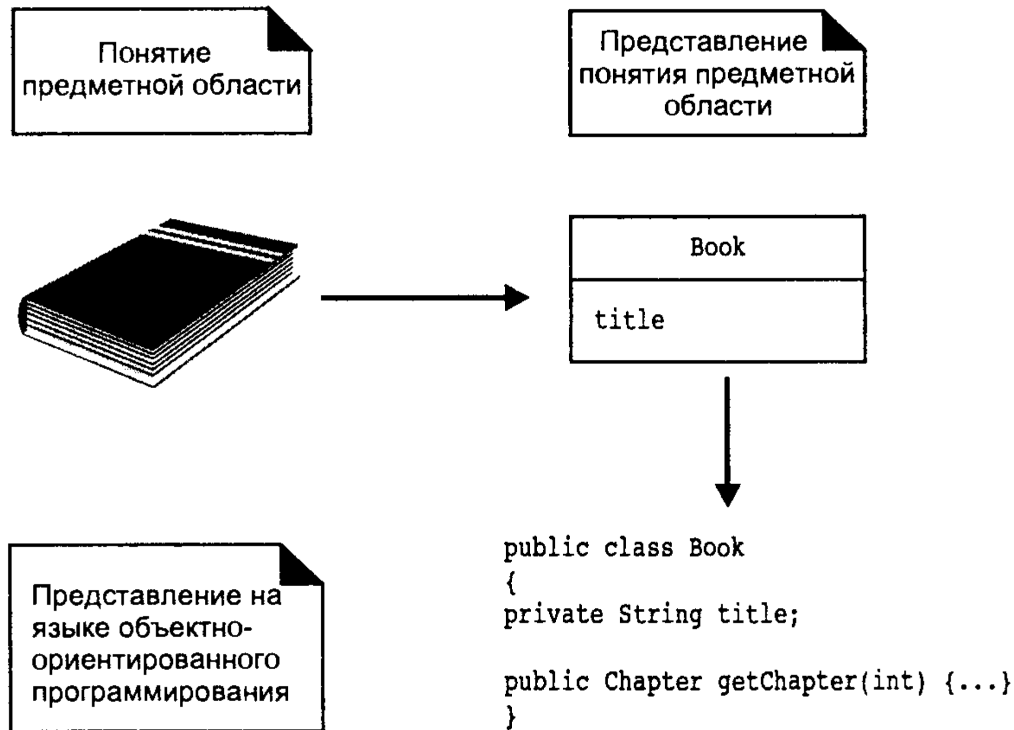
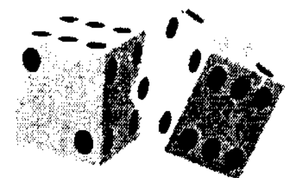



*Рис. 1.2. Представление объектов с использованием объектно-ориентированного подхода*

И наконец, на этапе *реализации* (implementation) или *объектно-ориентированного программирования* (object-oriented programming) обеспечивается реализация разработанных компонентов, таких как класс `Book`, на языке Java.

## 1.5. Пример объектно-ориентированного анализа и проектирования

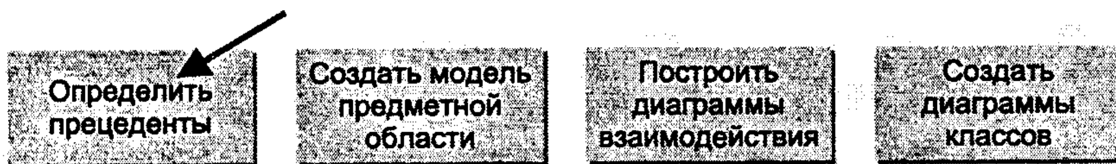


Прежде чем вдаваться в детали анализа требований и ООА/П, рассмотрим основные действия, выполняемые в процессе разработки программной системы, и создаваемые при этом диаграммы. В качестве примера возьмем игру в кости, в которой игрок бросает два кубика. Если сумма очков равна семи, участник считается победителем, в противном случае — проигравшим.

### *Определение прецедентов*

Анализ требований может включать описание процессов предметной области, представленное в форме прецедентов.

Прецеденты не являются артефактами объектно-ориентированного анализа — это просто повествовательные истории. Однако они составляют важный этап анализа требований и унифицированного процесса в целом.

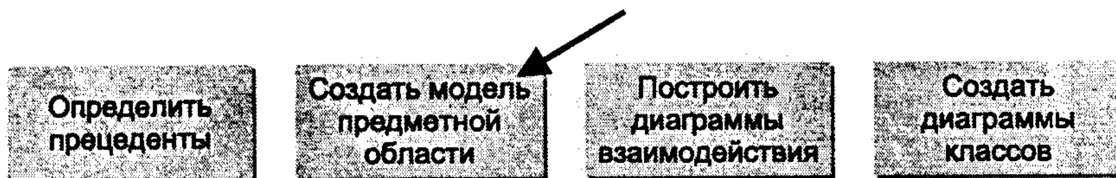



Например, в процессе игры в кости может присутствовать прецедент Игра в кости (Play a Dice Game) со следующим кратким описанием.

**Игра в кости.** Игрок берет и бросает кости. Если сумма очков составляет 7, игрок считается победителем, в противном случае — проигравшим.

## Определение модели предметной области

Объектно-ориентированный анализ связан с описанием предметной области с точки зрения классификации объектов. Декомпозиция предметной области задачи состоит в идентификации понятий, атрибутов и ассоциаций из предметной области, имеющих важное значение для решения задачи. Результат анализа выражается в *модели предметной области* (domain model), которая иллюстрируется с помощью набора диаграмм с изображенными на них понятиями или объектами предметной области




Например, модель предметной области игры в кости показана на рис. 1.3. Она иллюстрирует понятия Player (игрок), Die (кость) и DiceGame (игра в кости), а также их связи и атрибуты.

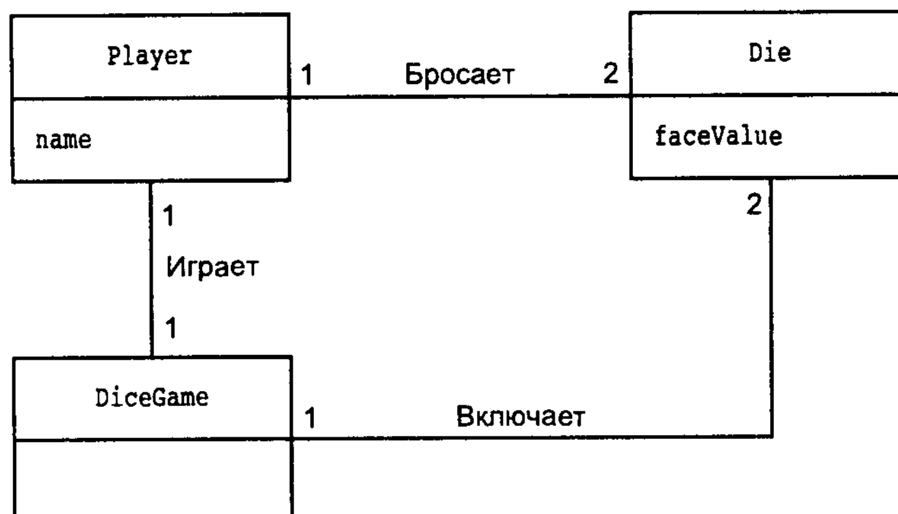
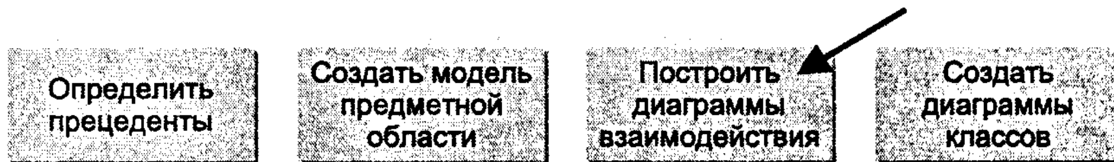



*Рис. 1.3. Модель предметной области игры в кости*

Модель предметной области — это не описание программных компонентов. Это представление понятий, выраженных в терминах предметной области задачи.

### Диаграммы взаимодействий

Объектно-ориентированное проектирование связано с определением программных компонентов и способов их взаимодействия. Для иллюстрации взаимосвязей между объектами используется *диаграмма взаимодействий* (collaboration diagram), отображающая потоки передачи сообщений между программными объектами и вызовы методов.




Например, рассмотрим задачу программного моделирования игры в кости. Диаграмма взаимодействий объектов этой системы иллюстрирует важные этапы игры и передачу сообщений между экземплярами классов `DiceGame` и `Die` (рис. 1.4).

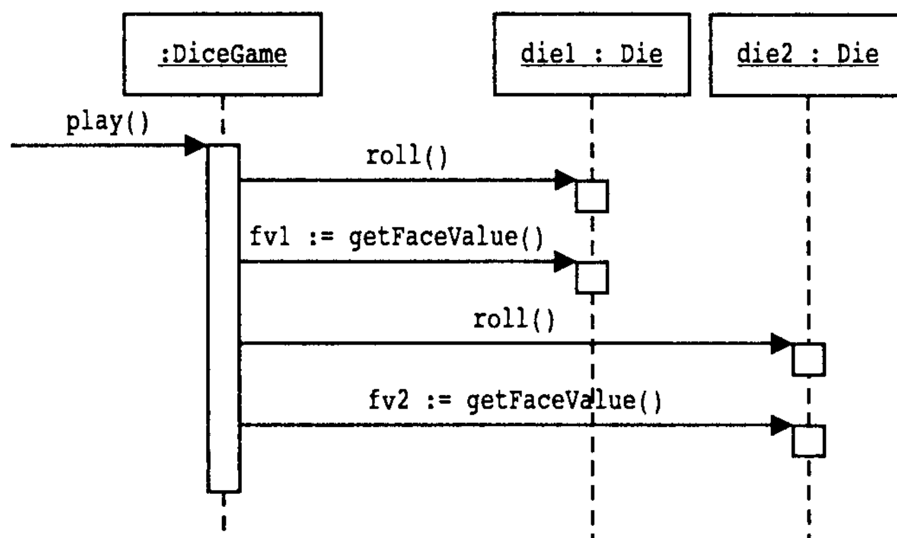



*Рис. 1.4. Диаграмма взаимодействий, иллюстрирующая передачу сообщений между программными объектами*

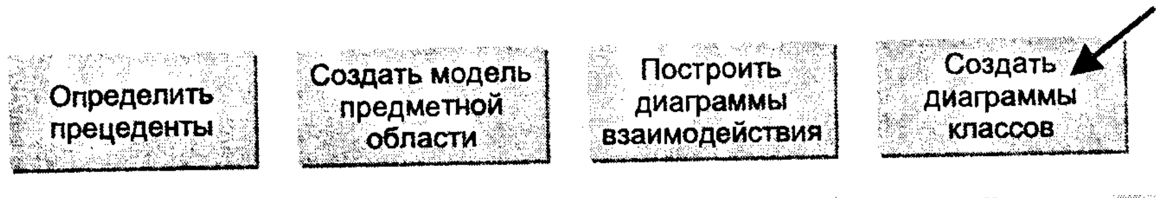
Обратите внимание на следующую особенность. Хотя в реальном мире кости подбрасывает игрок, в программной системе эту функцию выполняет объект `DiceGame` (т.е. передает сообщения объектам `Die`). Дело в том, что программные объекты в некотором смысле соответствуют объектам реального мира, но не являются их точными моделями или копиями.

### Диаграммы классов

Помимо *динамического представления* (dynamic view) взаимосвязи объектов, отображаемой на диаграмме взаимодействий, очень полезно строить *статическое представление* (static view) системы в виде *диаграммы классов проектирования* (design class diagram). На такой диаграмме отображаются атрибуты и методы классов.

Например, анализ диаграммы взаимодействия игры в кости приводит к диаграмме классов, показанной на рис. 1.5. Поскольку сообщение `play` переда-

ется объекту DiceGame, класс DiceGame должен содержать метод play, а класс Die — методы roll и getFaceValue.




В отличие от модели предметной области, эта диаграмма не иллюстрирует понятия реального мира, а описывает программные классы.

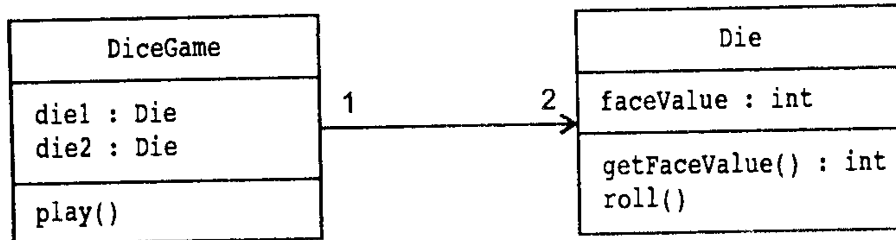



*Рис. 1.5. Диаграмма классов программных компонентов*

### Еще несколько слов об игре в кости

Программное моделирование игры в кости — очень простая задача, позволяющая сконцентрировать внимание на некоторых шагах и артефактах объектно-ориентированного анализа и проектирования. Для простоты изложения в тексте главы были прокомментированы не все используемые на диаграммах обозначения. Эти артефакты будут более подробно рассмотрены в последующих главах, посвященных вопросам анализа и проектирования.

## 1.6. Унифицированный язык моделирования UML

> "Унифицированный язык моделирования UML — это язык для определения, визуализации, конструирования и документирования артефактов программных систем, а также для моделирования экономических процессов и других не программных систем" [86].

UML — это важный фактический и юридический стандарт для объектно-ориентированного моделирования. Он появился в 1994 году в результате совместных усилий Гради Буча (Grady Booch) и Джима Румбаха (Jim Rumbaugh) по объединению их популярных методов — метода Буча и OMT (Object Modeling Technique). Затем эти две методологии были объединены Айваром Якобсоном (Ivar Jacobson), создателем метода OOSE (Object-Oriented Software Engineering). Впоследствии эту группу разработчиков назвали "три товарища". В разработку UML внесли свой вклад и другие специалисты, среди которых наиболее заметным является, пожалуй, Крис Кобрин (Cris Kobryn), руководитель процесса его последней модификации.

В 1997 году язык UML был принят в качестве стандартного языка моделирования группой промышленных стандартов OMG (Object Management Group). С тех пор он продолжает развиваться в новой редакции OMG UML.

В данной книге не приводится исчерпывающее описание UML, поскольку его система обозначений является весьма объемной (многие считают, что даже слишком)[1]. Здесь внимание сконцентрировано на чаще всего используемых диаграммах и их элементах, а также на основных обозначениях, которые вряд ли изменятся в ближайшее время.

### *Почему в нескольких следующих главах речь не идет об UML*

Эта книга посвящена не просто системе обозначений языка UML, а гораздо более широкой проблеме применения UML, шаблонов и итеративного процесса для разработки программных систем. Язык UML применяется преимущественно на этапе ООА/П, которому обычно предшествует этап анализа требований. Поэтому в первых главах рассматриваются важные вопросы описания прецедентов и анализа требований, а уже в последующих — подробности процесса ООА/П и специфика применения языка UML.

## 1.7. Дополнительная литература

Базовая система обозначений очень популярно и доступно изложена в книге Мартина Фовлера (Martin Fowler) [49].

Краткое и популярное введение в проблемы унифицированного процесса (и его модификации в виде Rational Unified Process) содержится в книге Филиппа Крачтена (Philippe Kruchten) [72].

Подробное описание системы обозначений версии 1.3 языка UML приводится в руководствах "трех товарищей" Буча, Якобсона и Румбаха [93] и [21]. Заметим, что эти тексты не предназначены для изучения вопросов объектного моделирования или ООА/П, а лишь для ознакомления с системой обозначения элементов диаграмм UML.

Описание текущей версии языка UML и его интерактивную спецификацию [86] можно найти по адресу www.omg.org. Процесс модификации языка UML и новейшие версии, которые только должны увидеть свет, приводятся на узле www.celigent.com/uml.

Шаблонам проектирования посвящено множество книг, однако классическим учебником считается книга Гамма (Gamma), Хелма (Helm), Джонсона (Johnson) и Влиссидеса (Vlissides) [52]. Эта книга действительно необходима всем, кто планирует изучать объектное проектирование. Однако она рассчитана не на новичков, а на специалистов, уже знакомых с основами объектного проектирования и программирования.

---

[1] В версии UML 2.0 планируется упростить и сократить систему обозначений.

Глава

# Итеративная разработка и унифицированный процесс

*Люди гораздо важнее любого процесса.*

*Хорошая команда с хорошим процессом всегда превосходит хорошую команду при отсутствии процесса.*

*Гради Буч (Grady Booch)*

---

### Основные задачи

- Обосновать содержание и порядок следования глав.
- Определить итеративный и адаптивный процесс.
- Определить фундаментальные понятия унифицированного процесса.

---

## Введение

Итеративная разработка — это технический подход к созданию программных систем, положенный в основу описания объектно-ориентированного анализа и проектирования в данной книге. Унифицированный процесс — это пример итеративного процесса разработки проектов на базе ООА/П. Именно на его примере рассматривается создание объектно-ориентированных систем в этой книге. Поэтому данная глава позволит ознакомиться с основными понятиями и связать их со структурой книги.

В этой главе приводятся некоторые ключевые идеи. Более подробная информация по итеративному процессу вообще и унифицированному процессу в частности содержится в главе 37.

По существу, *процесс разработки программного обеспечения* (software development process) включает построение, развертывание и, возможно, поддержку системы. *Унифицированный процесс UP* (Unified Process) [67] — это широко используемый процесс разработки объектно-ориентированных систем. В частности, широкую популярность приобрел унифицированный процесс *RUP* (*Rational Unified Process*) [72], обеспечивающий более детальную проработку всех этапов.

В унифицированном процессе нашел отражение и получил детальное описание наиболее удачный опыт разработки систем, в частности итеративный жизненный цикл и оценка рисков. Поэтому унифицированный процесс используется в этой книге в качестве примера реализации объектно-ориентированного подхода.

Эта книга начинается с описания унифицированного процесса по двум причинам.

1. UP — это *итеративный* процесс. Принципы ООА/П описываются в этой книге в контексте итеративной разработки, которая зарекомендовала себя наилучшим образом при создании множества систем.
2. В рамках унифицированного процесса очень удобно описывать принципы ООА/П.

> В данной главе приводятся лишь начальные сведения об унифицированном процессе, а не его полное описание. Основное внимание уделяется общим идеям и артефактам, связанным с анализом требований и ООА/П.

### *Если меня не интересует унифицированный процесс*

Унифицированный процесс рассматривается в качестве примера, в рамках которого проводится анализ требований и ООА/П. Анализ и проектирование всегда выполняются в контексте некоторого процесса, а унифицированный процесс или его модификация RUP распространены достаточно широко. Кроме того, в рамках UP представлены основные виды деятельности и богатый опыт разработок. И все же основные идеи этой книги (такие как описание прецедентов и шаблонов проектирования) не зависят от конкретного процесса и применимы к любому из них.

## 2.1. Главная идея унифицированного процесса — итеративная разработка

В основу унифицированного процесса положено несколько важных идей, но одна является действительно определяющей — это *итеративная разработка* (iterative development). В рамках этого подхода разработка выполняется в виде нескольких краткосрочных мини-проектов фиксированной длительности (например, по 4 недели), называемых *итерациями* (iteration). Каждая итерация включает свои собственные этапы анализа требований, проектирования, реализации и завершается тестированием, интеграцией и созданием работающей системы.

Итеративный жизненный цикл основывается на постоянном расширении и дополнении системы в процессе нескольких итераций с периодической обратной связью и адаптацией добавляемых модулей к существующему ядру. Система постепенно разрастается шаг за шагом, поэтому такой подход иногда называют *итеративной и инкрементальной разработкой* (iterative and incremental development) (рис. 2.1).

Первые идеи итеративного процесса назывались "проектированием по спирали и эволюционной разработкой" [16, 53].

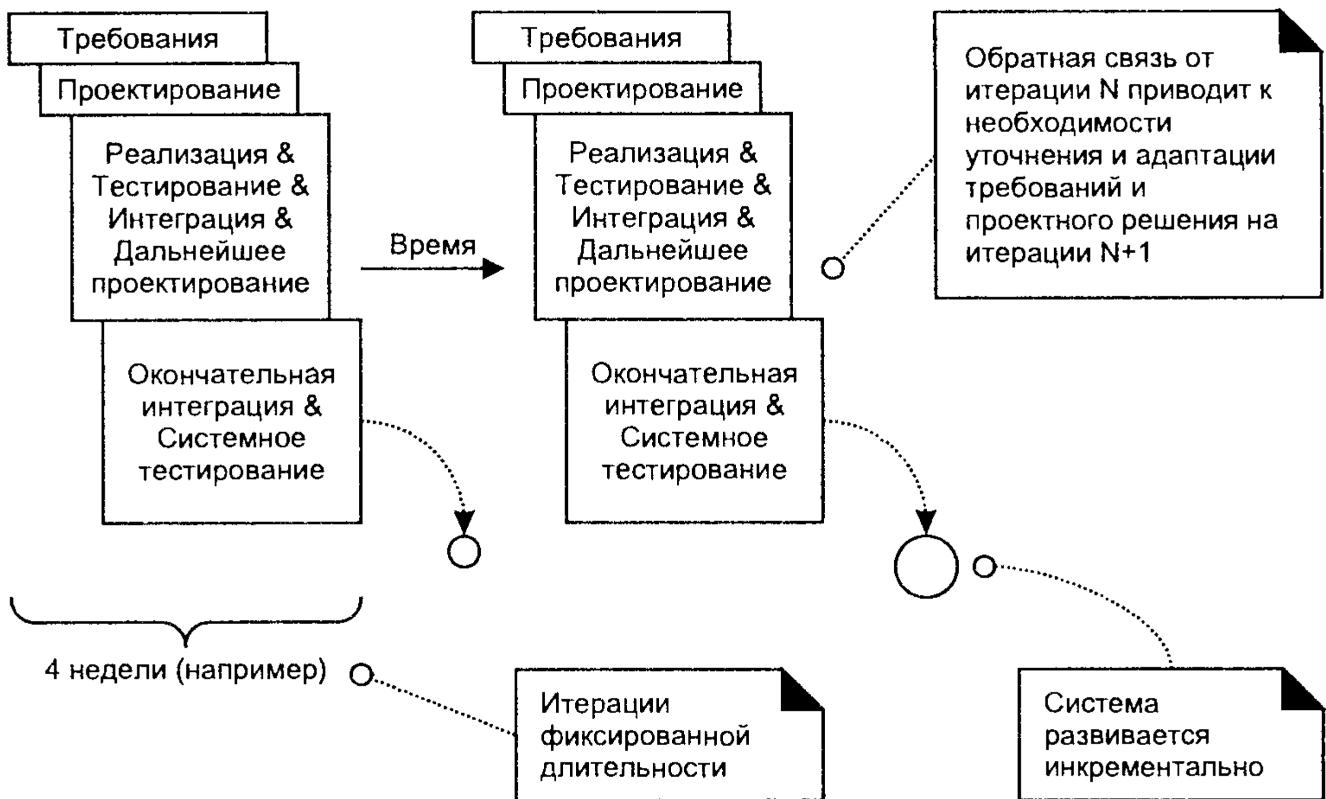



*Рис. 2.1. Итеративная и инкрементальная разработки*

*Пример*

В качестве примера (но не рецепта на все случаи жизни) рассмотрим двухнедельную итерацию процесса разработки. Понедельник уйдет на распределение и осмысление задач и требований этой итерации. Один из участников проекта в этот день может выполнить обратное проектирование и отобразить код, полученный на предыдущей итерации, в диаграммах UML (с помощью CASE-средств), а также распечатать основные диаграммы. Вторник предстоит провести у доски за проектированием и рисованием диаграмм UML, которые затем можно снять на цифровой фотоаппарат и ввести в компьютер. В этот же день пишется необходимый псевдокод и замечания по проектированию. Остальные восемь рабочих дней отводятся на реализацию, тестирование (модулей, применимости, удобства и т.д.), доработку, интеграцию, системное тестирование и завершение разработки части системы. Кроме того, в этот период результаты демонстрируются руководителям проекта, обсуждаются с ними и вырабатывается план на последующие итерации.

Обратите внимание, что при таком подходе исключено и слишком быстрое написание кода (без детальной проработки) и чрезмерно длительный этап детального проектирования и построения диаграмм без обратной связи. Проектирование и визуальное моделирование с помощью UML осуществляется в течение одного дня, причем для выполнения этого вида деятельности разработчики обычно разбиваются на пары.

В результате каждой итерации получается работающая, но не полнофункциональная система, которую еще нельзя "выставлять на продажу". Система может приобрести товарный вид только после 10 или 15 итераций.

Результатом каждой итерации является *не* экспериментальный прототип, и итеративная разработка не сводится к прототипированию. Скорее, результатом итерации является окончательная версия некоторой части всей системы.

И хотя, как правило, на каждой итерации определяются новые требования и система постепенно расширяется, некоторые итерации могут быть полностью посвящены редактированию существующей программы и ее усовершенствованию. Например, одна итерация может потребоваться для повышения производительности подсистемы, а не добавления новой функции.

### Допустимость изменений: обратная связь и адаптация

Подзаголовком одной книги, посвященной итеративной разработке, является фраза "Допускайте изменения" [11]. Эта фраза может служить девизом итеративного процесса разработки. Вместо того чтобы бороться за создание окончательного и неизменного варианта модели (как правило, безуспешно) и пытаться корректно сформулировать, зафиксировать, подписать и "заморозить" набор требований и модель до начала ее программной реализации, гораздо лучше осознать важность постоянной модификации и адаптации модели в процессе разработки системы и смириться с необходимостью регулярного внесения изменений.

Это не означает, что итеративная разработка и унифицированный процесс развиваются спонтанно и бесконтрольно. В последующих главах будет рассказано о том, как обеспечить баланс между стабильностью набора требований, с одной стороны, и его реальной модификацией в процессе все более глубокого осознания задач и специфики проекта его руководителями (или изменением рыночной ситуации), — с другой.

На каждой итерации разработки рассматривается небольшое подмножество требований, для удовлетворения которых быстро разрабатывается, реализуется и тестируется небольшая часть системы. На начальных итерациях требования и модель проекта могут быть очень далеки от идеала. Однако достаточно сжатые временные рамки каждой итерации, на которой выполняется весь процесс проектирования и реализации, обеспечивают быструю и своевременную обратную связь и общение между пользователями, разработчиками и специалистами по тестированию.

Ранняя обратная связь имеет неоценимое значение. При этом не тратится драгоценное время на длительные размышления и формулировку корректных требований или на проектирование. Благодаря обратной связи у специалистов по тестированию в процессе разработки постоянно появляются новые идеи и углубляется понимание требований и задач проекта. Конечные пользователи получают возможность быстро увидеть часть системы и сказать: "Да, это именно то, что мне нужно. Но когда я попробовал поработать с этой программой, у меня появились новые пожелания".[1] И это "да..., но" не является сигналом тревоги. Это нормальный путь развития и движения к совершенству, благодаря которому руководители проекта смогут лучше осознать основные задачи проекта и требования к системе. И это, естественно, не хаотическое и спонтанное движение, когда разработчики постоянно изменяют направление своей деятельности. При таком способе разработки достигается компромисс между устойчивостью и изменчивостью.

Помимо прояснения системных требований, ранняя обратная связь и тестирование каждой небольшой части системы (в частности, теста загрузки) позво-

---

[1] Правда, скорее всего, он скажет: "Вы абсолютно не поняли, чего я хотел".

ляют оценить преимущества выбранной стратегии или ее недостатки. В последнем случае на следующей итерации можно внести изменения в базовую архитектуру системы. Гораздо лучше оценить возможные риски и критические моменты на начальной стадии проекта, чем на конечном этапе реализации; и итеративная разработка обеспечивает именно такой механизм.

Таким образом, проект выполняется в виде последовательности циклов, каждый из которых включает этап построения системы, получения обратной связи и адаптации. Не удивительно, что на начальных итерациях отклонения от "пути истинного" в развитии системы (с точки зрения окончательной формулировки требований и проекта) гораздо существеннее, чем на последних итерациях. Со временем процесс развития системы идет по этому пути (рис. 2.2).

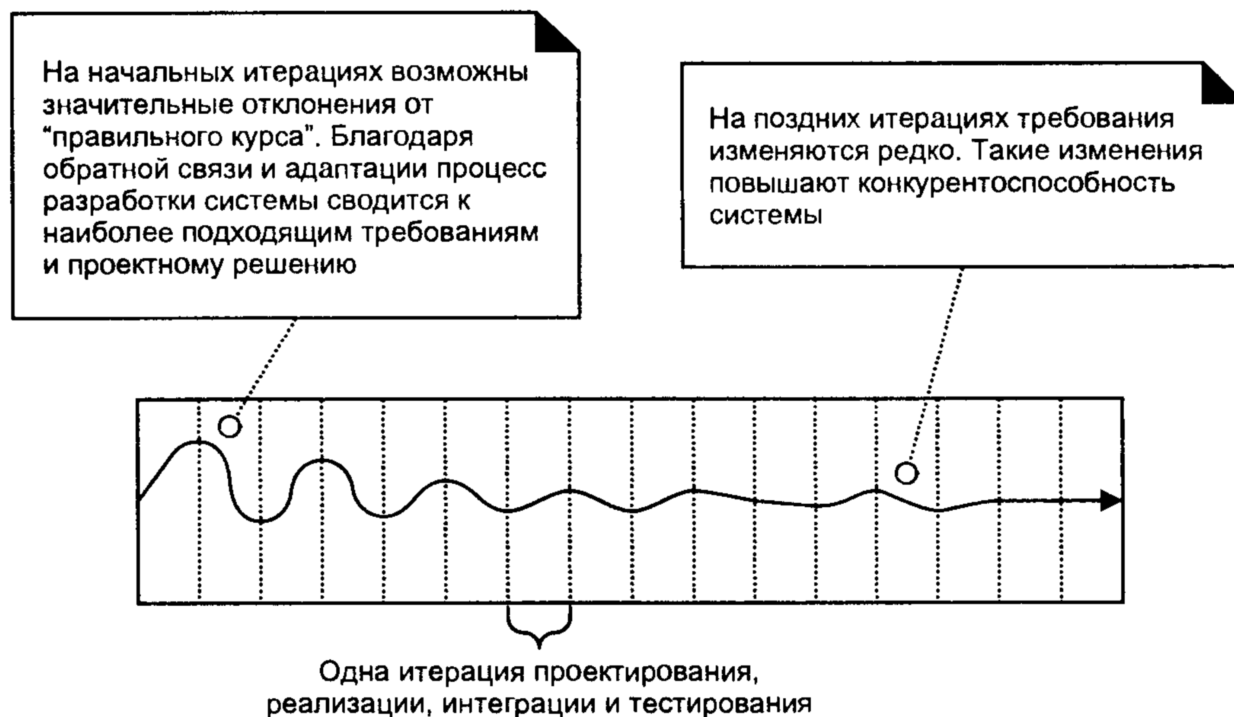



*Рис. 2.2. В процессе итеративной разработки, благодаря обратной связи и адаптации, система приобретает должный вид. Со временем требования и проект претерпевают все менее серьезные изменения*

## Преимущества итеративной разработки

Итеративная разработка обладает следующими преимуществами.

- Своевременное осознание возможных технических рисков, осмысление требований, задач проекта и удобства использования системы.
- Быстрый и заметный прогресс.
- Ранняя обратная связь, возможность учета пожеланий пользователей и адаптации системы. В результате система более точно удовлетворяет реальные требования руководителей и потребителей.
- Управляемая сложность. Команда разработчиков не перегружена лишней работой в рамках этапа анализа и проектирования и не "парализована" слишком сложными и долгосрочными задачами.
- Полученный при реализации каждой итерации опыт можно методически использовать для улучшения самого процесса разработки.

### *Фиксированная длительность итерации*

В рамках унифицированного процесса рекомендуется устанавливать длительность каждой итерации в пределах от двух до шести недель (опытные разработчики согласны с таким графиком работ). Небольшие шаги, быстрая обратная связь и адаптация — это основные принципы итеративной разработки. Большая длительность итераций сводит на нет саму идею итеративной разработки и повышает риск проекта. Если длительность итерации составляет менее двух недель, то разработчикам не остается времени для выполнения хоть сколько-нибудь существенной части работ. Если же итерация занимает более шести или восьми недель, то поставленные для нее задачи слишком усложняются, и откладывается момент получения обратной связи. При слишком длительных итерациях теряется сама идея итеративности. Поэтому чем короче, тем лучше.

Очень важно, чтобы длительность итераций была *фиксированной* (timeboxed). Например, если длительность следующей итерации должна составлять 4 недели, то за это время нужно успеть не только разработать и реализовать намеченную часть системы, но и провести ее интеграцию, тестирование и адаптацию. Если за указанный период выполнить все эти задачи нереально, нужно сократить список требований для данной итерации и перенести их реализацию на следующий цикл, но ни в коем случае не сдвигать дату завершения итерации. Эта стратегия будет более подробно обоснована в главе 37.

Для больших команд разработчиков (например, включающих несколько сотен человек) длительность одной итерации может превышать шесть недель, поскольку в таких коллективах очень много времени уходит на координацию и согласование работ между всеми участниками проекта. Однако даже в этом случае длительность итерации не должна превышать трех или шести месяцев. Например, система управления воздушными перевозками, разработанная в Канаде в 90-х годах, создавалась в рамках унифицированного процесса с итеративным жизненным циклом. Для успешной реализации проекта было привлечено 150 программистов, которые работали в рамках шестимесячных итераций.[2] Заметим, однако, что даже в этом случае всех участников проекта можно разбить на группы разработчиков подсистем по 10–20 человек, для каждой из которых одна общая итерация будет состоять из шести "внутренних" итераций длительностью 1 месяц.

Шестимесячные итерации — это исключение для очень больших коллективов разработчиков, а не общее правило. Напомним, что в рамках унифицированного процесса рекомендуется устанавливать длительность итерации в пределах от двух до шести недель.

## 2.2. Дополнительные рекомендации и концепции унифицированного процесса

Таким образом, основной идеей унифицированного процесса является разбиение разработки на короткие итерации фиксированной длительности и обеспечение адаптивности.

[2] Главным архитектором этого проекта был Филипп Крачтен (Philippe Kruchten), который возглавил также разработку процесса RUP.

Еще одна неявная, но ключевая идея процесса UP — использование объектных технологий, в том числе ООА/П и объектно-ориентированного программирования.

К числу других важных принципов и концепций унифицированного процесса относятся следующие.

- Оценка рисков и ключевых моментов проекта на ранних итерациях.
- Постоянный отклик пользователей, обратная связь и модификация требований.
- Построение базовой архитектуры на ранних итерациях.
- Постоянный контроль качества, раннее и частое тестирование в реальных условиях.
- Применение прецедентов.
- Визуализация программной модели (как правило, с помощью языка UML).
- Внимательное отношение к требованиям.
- Управление конфигурацией.

Эти принципы более подробно описаны в главе 37.

## 2.3. Фазы унифицированного процесса и соответствующие термины

В рамках унифицированного процесса работа над проектом включает четыре основные фазы.

1. **Начало** (inception) — определение начального в*и*дения проблемы, прецедентов, оценка сложности задачи.
2. **Развитие** (elaboration) — формирование более полного в*и*дения проблемы, итеративная реализация базовой архитектуры, создание наиболее критичных компонентов (разрешение высоких рисков), идентификация основных требований, получение более реалистичных оценок.
3. **Конструирование** (construction) — итеративная реализация менее критичных и простых элементов, подготовка к развертыванию.
4. **Передача** (transition) — бета-тестирование и развертывание.

Эти фазы более детально обсуждаются в последующих главах.

Это не последовательный жизненный цикл, при котором сначала определяются все требования, а только затем начинается вся разработка системы.

Начальная фаза — это не стадия формулировки требований в старом понимании. Это скорее этап оценивания ситуации, на котором принимается решение о целесообразности или нецелесообразности дальнейшей разработки.

Развитие — это не стадия проектирования, а фаза итеративной реализации базовой архитектуры и разрешения высоких рисков.

На рис. 2.3 показана взаимосвязь терминов, принятых при описании графика выполнения работ в рамках унифицированного процесса. Заметим, что один цикл разработки (который завершается созданием коммерческой версии программного продукта) состоит из многих итераций.

Цикл разработки

Итерация

Фаза

Начало | Развитие | Конструирование | Передача

Конец — Конец итерации, на которой получено важное решение

Релиз — Стабильная работающая часть программного продукта. В некотором смысле, релизом является окончание каждой итерации

Инкремент — Промежуток между двумя последовательными релизами

Окончательный релиз — Теперь система пригодна для коммерческого использования

*Рис. 2.3. Термины, используемые при описании графика выполнения работ в рамках унифицированного процесса*

## 2.4. Дисциплины унифицированного процесса (в прошлом — последовательности выполняемых действий)

Унифицированный процесс предполагает выполнение различных видов деятельности, например, описание прецедентов в рамках определенных *дисциплин* (disciplines), которые ранее назывались *последовательностями выполняемых действий* или *технологическими процессами* (workflow). По существу, дисциплина — это набор видов деятельности (и связанных с ними артефактов) в рамках одного этапа, например в рамках анализа требований. В контексте унифицированного процесса *артефактом* (artifact) называется любой результат работы: код, графическое изображение для Web, схема базы данных, текстовые документы, диаграммы, модели и т.д.

В рамках UP существует несколько дисциплин. В этой книге внимание сосредоточено на некоторых артефактах из следующих трех дисциплин.

- **Бизнес-моделирование** (business modeling). При разработке отдельного приложения эта дисциплина подразумевает моделирование объектов предметной области. Если же речь идет о крупномасштабном анализе или перестройке производственного процесса, то сюда включается динамическое моделирование производственных процессов всего предприятия.
- **Требования** (requirements). Это анализ требований для данного приложения, в том числе описание прецедентов и определение нефункциональных требований.
- **Проектирование** (design). Сюда относятся все аспекты проектирования, включая разработку общей архитектуры, объектов, баз данных, сетевой конфигурации и т.д.

Более полный перечень дисциплин UP показан на рис. 2.4.

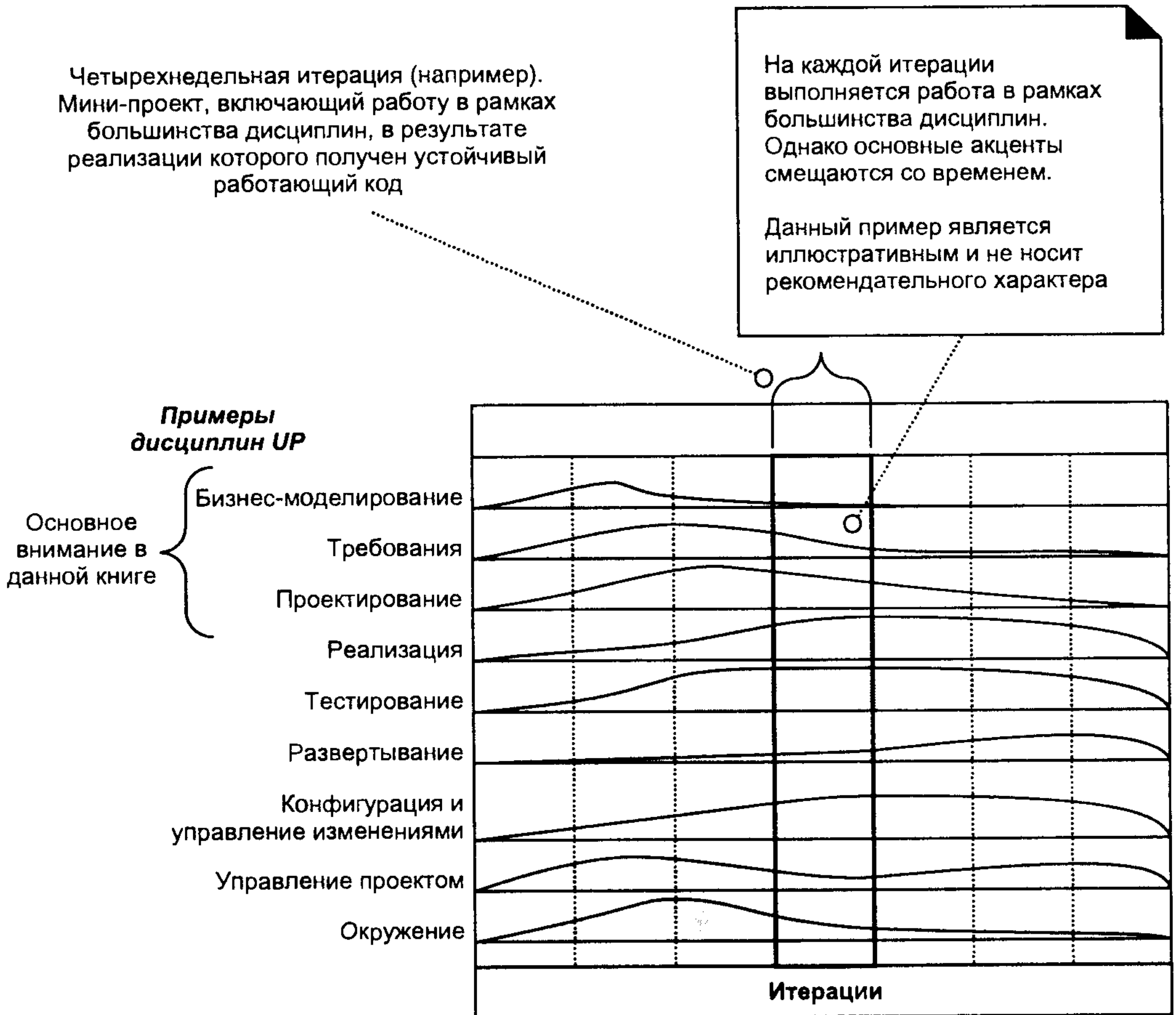



*Рис. 2.4. Дисциплины унифицированного процесса*[3]

В контексте UP *реализация* (implementation) означает программирование и построение системы, но не ее развертывание. Дисциплина "*окружение*" (environment) предполагает установку необходимых средств и настройку процесса для данного проекта.

## Дисциплины и фазы

Как видно из рис. 2.4, в течение одной итерации работа выполняется в рамках всех или большинства дисциплин. Однако удельный вес каждой дисциплины в общем объеме работ изменяется со временем. На ранних итерациях, естественно, основное внимание уделяется требованиям и проектированию, а на более поздних, когда требования и ядро системы уже "устоялись" благодаря обратной связи и адаптации, деятельность в рамках этих дисциплин сходит на нет.

---

[3] Эта диаграмма построена на основе процесса RUP.

На рис. 2.5 показан удельный вес различных дисциплин в контексте фаз унифицированного процесса (начало, развитие и т.д.). Заметим, что это соотношение следует рассматривать лишь как рекомендацию, а не догму. Например, фазе развития присущ относительно большой удельный вес работ по определению требований и проектированию. Однако при этом выполняется и программная реализация. В процессе конструирования основное внимание переключается с анализа требований на реализацию.

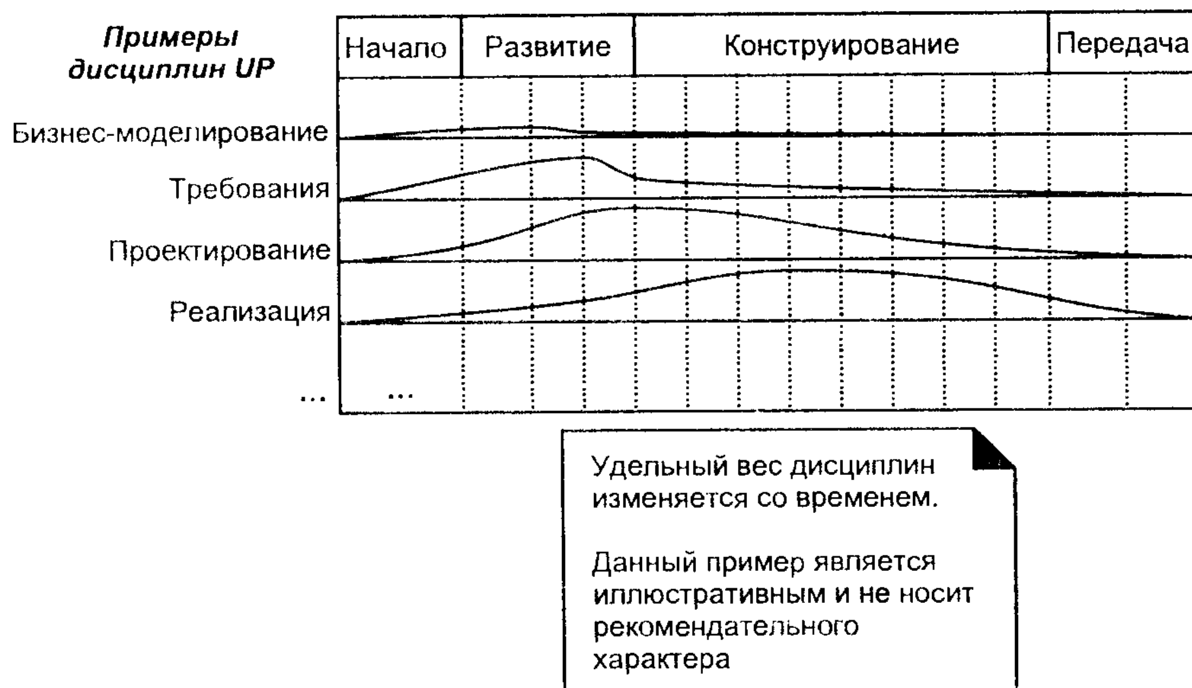



*Рис. 2.5. Дисциплины и фазы*

## Фазы UP, дисциплины и структура книги

На каких фазах и дисциплинах будет сосредоточено внимание при рассмотрении реального примера? Ответ сводится к следующему.

При описании реального примера основное внимание будет уделяться фазам начала и развития. При этом будут рассмотрены некоторые артефакты дисциплин бизнес-моделирования, требований и проектирования, поскольку именно с этими дисциплинами наиболее тесно связаны анализ требований, ООА/П, шаблоны и UML.

В начальных главах книги рассматриваются виды деятельности начальной фазы реализации проекта. Последующие главы посвящены нескольким итерациям фазы развития. В следующем списке и на рис. 2.6 представлена структура книги в контексте фаз UP.

1. В главах, посвященных "начальной" фазе, рассматриваются вопросы анализа требований.
2. При описании первой итерации вводятся основные понятия анализа и проектирования, а также рассматриваются вопросы распределения обязанностей между объектами.

3. При переходе ко второй итерации основное внимание уделяется проектированию объектов, особенно некоторым наиболее часто используемым шаблонам проектирования.
4. При рассмотрении третьей итерации затрагивается множество вопросов, в том числе связанных с анализом архитектуры и проектированием контуров.

*Структура книги*

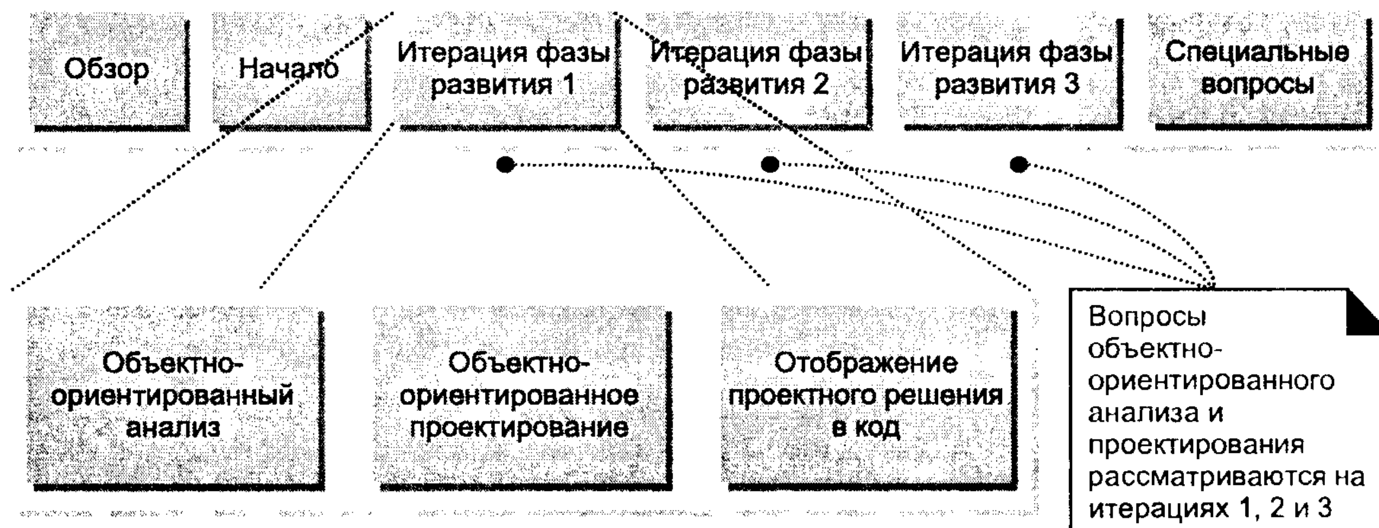



*Рис. 2.6. Структура книги в контексте фаз и итераций UP*

## 2.5. Настройка процесса и набор документов для проекта

### *Необязательные артефакты*

Некоторые принципы и концепции унифицированного процесса являются основными и неотъемлемыми. К ним относятся итеративная разработка на основе учета рисков, а также непрерывный контроль качества.

Однако при более пристальном взгляде на UP оказывается, что все виды деятельности и артефакты (модели, диаграммы, документы и т.д.) являются необязательными (за исключением, возможно, кода). Набор возможных артефактов, описанных в рамках UP, можно рассматривать как ассортимент медикаментов в аптеке. Покупатель вряд ли купит в аптеке все возможные медикаменты; он ограничится некоторым небольшим набором нужных ему лекарств. Так и разработчики проекта в рамках UP должны выбрать для себя небольшой набор артефактов, наиболее точно удовлетворяющий их потребности. В целом, нужно сосредоточиться на *небольшом* подмножестве артефактов, имеющих решающее практическое значение.

### *Набор документов*

Выбранные для данного проекта артефакты можно описать в кратком перечне под названием “Набор документов” (Development Case) (артефакт дисциплины “Окружение”). Например, в табл. 2.1 указан набор документов, описывающий артефакты для проекта NextGen, рассматриваемого в этой книге в качестве примера.

В последующих главах рассказывается о создании некоторых из этих артефактов, включая модель предметной области, модель прецедентов и модель проектирования.

Выбранный для этого проекта набор артефактов не обязательно подойдет для всех проектов. Например, для системы управления автомобилем более полезными окажутся различные диаграммы состояний. При разработке системы электронной коммерции для Web основное внимание нужно уделить прототипам интерфейса пользователя. Артефакты проекта по озеленению должны значительно отличаться от документации проекта системной интеграции.

**Таблица 2.1. Примерный набор документов, описывающих артефакты унифицированного процесса (н — начало, р — развитие)**

| Дисциплина | Артефакт<br>Итерация→ | Начало<br>I1 | Развитие<br>E1..En | Конструирование<br>C1..Cn | Передача<br>T1..Tn |
|---|---|---|---|---|---|
| Бизнес-моделирование | Модель предметной области | | н | | |
| Требования | Модель прецедентов | н | р | | |
| | Видение системы | н | р | | |
| | Дополнительная спецификация | н | р | | |
| | Словарь терминов | н | р | | |
| Проектирование | Модель проектирования | | н | р | |
| | Описание архитектуры | | н | | |
| | Модель данных | | н | р | |
| Реализация | Модель реализации | | н | р | р |
| Управление проектом | План разработки | н | р | р | р |
| Тестирование | Модель тестирования | | н | р | |
| Окружение | Набор документов | н | р | | |

## 2.6. Гибкость унифицированного процесса

Методологически процессы делятся на тяжелые и легкие, а также на детерминированные и адаптивные. "*Тяжелый процесс*" (heavy process) обладает следующими особенностями [47].

- Множество артефактов, создаваемых в бюрократической атмосфере.
- Отсутствие гибкости и управляемость.
- Долгосрочное детальное планирование.
- Детерминированность, а не адаптивность.

*Детерминированный процесс* (predictive process) предполагает планирование и предсказуемость всех видов деятельности, а также распределение человеческих ресурсов на длительный срок, охватывающий большую часть проекта. Детерминированным процессам обычно присущ последовательный жизненный цикл (каскадный цикл или цикл типа "водопада"), при котором сначала определяются все требования, затем выполняется детальное проектирование, и, наконец, наступает стадия реали-

зации. В отличие от этого, адаптивный процесс допускает возможность изменений, которые являются "двигателем" прогресса и обеспечивают гибкую адаптацию. Такие процессы отличаются итеративным жизненным циклом. *Гибкий процесс* (agile process) включает преимущества легкого и адаптивного процесса, благодаря которым обеспечивается выполнение изменчивых требований.

Унифицированный процесс задумывался авторами не как тяжелый или детерминированный, хотя обширный набор видов деятельности и артефактов иногда наводит разработчиков на такую мысль. Он разрабатывался как адаптивный и подвижный — *гибкий UP* (agile UP). Это обеспечивается следующим образом.

- Выбирается небольшой набор видов деятельности и артефактов UP. Для каждого проекта этот набор может быть своим, но всегда он должен оставаться небольшим.
- Поскольку унифицированный процесс является итеративным, то анализ требований и проектирование не завершаются к моменту начала реализации. Требования и модель проектирования адаптивно настраиваются в течение нескольких итераций на основе обратной связи.
- Не существует *подробного* плана всего проекта. Разрабатывается лишь план высокого уровня или так называемый *укрупненный план* (phase plan), в котором указывается дата завершения проекта и даты выполнения основных этапов, однако этот план не детализируется. Подробный план или так называемый *план итерации* (iteration plan) строится для каждой итерации в отдельности. Детальное планирование выполняется адаптивно от итерации к итерации. Более подробная информация о планировании итеративных проектов содержится в главе 36.

В рассматриваемом примере внимание будет сосредоточено на небольшом количестве артефактов и итеративной разработке, как и положено в рамках UP.

## 2.7. Последовательный жизненный цикл или жизненный цикл типа "водопад"

В отличие от итеративного жизненного цикла UP, ранее разработка программных систем выполнялась в рамках последовательного или линейного жизненного цикла (каскадного цикла) [95]. Обычно он включал следующие шаги.

1. Определение, запись и утверждение полного и неизменного набора требований.
2. Проектирование системы на основе этих требований.
3. Реализация системы на базе разработанного проекта.

В результате двухгодичного исследования успешных программных проектов были определены четыре определяющих фактора успеха, и первым в списке значилась итеративная, а не последовательная разработка [78].[4]

---

[4] Остальные факторы это: 2) ежедневная интеграция новых фрагментов кода с основной системой и быстрая обратная связь (посредством тестирования), необходимая для внесения изменений; 3) опытная команда; 4) заблаговременное решение вопросов построения и обоснования цельной архитектуры. Три из этих четырех факторов в явном виде предусмотрены в UP.

Краткое описание этих проблем и способов их преодоления в рамках итеративной разработки приводится в главе 37.

## 2.8. Вы не поняли, что такое унифицированный процесс, если...

Вот несколько признаков, свидетельствующих о непонимании сущности унифицированного процесса и итеративной разработки.

- Вы считаете, что начальная фаза эквивалентна определению требований, развитие означает разработку, а построение — реализацию (т.е. пытаетесь проектировать линейный жизненный цикл на UP).
- Вы думаете, что задачей фазы развития является полное и точное определение моделей, которые на этапе конструирования будут отображены в коде.
- Вы пытаетесь определить большую часть требований до начала проектирования и реализации.
- Вы стараетесь выполнить большую часть разработки до начала реализации; пытаетесь полностью определить и утвердить архитектуру до начала итеративного программирования и тестирования.
- Много времени тратится на определение требований или проектирование до начала программной реализации.
- Вам кажется, что продолжительность итерации должна составлять четыре месяца, а не четыре недели (за исключением проектов, в которых задействованы сотни разработчиков).
- Вы думаете, что в процессе проектирования и создания диаграмм UML нужно точно и подробно определить проектное решение, а этап программирования — это лишь механическое преобразование модели в код.
- Вы считаете, что унифицированный процесс предполагает выполнение множества операций и создание многочисленных документов и рассматриваете UP как формальный нервный процесс, включающий множество этапов.
- Вы хотите детально спланировать проект от начала до конца; стараетесь заранее предугадать все необходимые итерации и предсказать возможное развитие событий на каждой из них.
- Вы надеетесь получить правдоподобный план реализации проекта до завершения стадии развития.

## 2.9. Дополнительная литература

Краткое и доступное введение в проблемы унифицированного процесса (и его модификации в виде Rational Unified Process) содержится в книге главного архитектора RUP Филиппа Крачтена (Philippe Kruchten) [72].

Описание исходной версии UP содержится в книге Якобсона, Буча и Румбаха [67]. Это очень ценная книга, но книга Филиппа Крачтена (Philippe Kruchten) названа первой, поскольку она меньше по объему и проще. Кроме того, процесс RUP является более новой и расширенной версией UP.

Компания Rational Software продает интерактивный программный продукт для документирования процесса RUP с использованием Web-технологии. Там подробно описаны артефакты и виды деятельности RUP, а также приводятся шаблоны для большинства артефактов. Краткое описание этого продукта содержится в главе 37. Освоить унифицированный процесс можно и по книгам, но многие считают продукт RUP очень полезным средством для обучения и реализации процесса UP.

Виды деятельности в рамках UP очень приблизительно описаны в серии книг под редакцией Амблера (Ambler) и Константина (Constantine) (к примеру, [3]). В этих книгах содержатся репринты статей, опубликованных в течение нескольких лет в журнале Software Development. Статьи систематизированы по категориям в соответствии с фазами разработки и видами деятельности в терминах UP. Заметим, что изначально в этих статьях не описывался унифицированный процесс, хотя и содержалась ценная информация. Кроме того, в указанную серию вкралась одна небольшая ошибка. Там фаза развития описывается как этап создания прототипов, при котором можно не обращать внимания на совершенство проектного решения или качество кода. Это неверно: в процессе развития создаются промышленные проектные и программные решения (хотя бы частично). Редактор серии понял свою оплошность и собирается исправить ее в следующем издании.[5]

Другие "гибкие" методы описаны в серии книг XP (Extreme Programming — экстремальное программирование) [11, 12, 66]. Некоторые приемы XP упоминаются в последующих главах. Большинство идей XP (таких как программирование с предварительным тестированием (test-first programming) и итеративная разработка) схожи с концепциями UP (или совпадают с ними), поэтому их можно применять в проектах UP. Заметим, что краткие итерации и адаптивная разработка не являются изобретением XP; эти идеи в течение нескольких лет развивались в рамках UP и других итеративных методов. Между UP и XP существуют и некоторые значительные отличия, в том числе следующие. Во-первых, в рамках UP рекомендуется постепенно расширять описания прецедентов и нефункциональных требований, а в XP — нет. Во-вторых, в контексте UP в начале каждой итерации большое внимание уделяется визуальному проектированию и построению диаграмм (полдня или день), а апологеты XP рекомендуют выполнять эту работу очень быстро, в течение 30 минут.

Важность адаптивной разработки обосновывается в книге Хайсмита (Highsmith) [62].

---

[5] Это выяснилось в личных беседах с Амблером.

Глава

# Конкретный пример: система автоматизации торговли NextGen

*Мало с чем так сложно смириться, как с хорошим примером.*

*Марк Твен (Mark Twain)*

Основная задача

- Ознакомиться с конкретным примером, рассматриваемым в этой книге.

## Введение

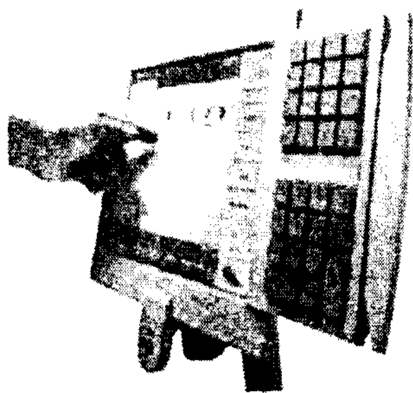
В этой главе кратко описывается основной пример, рассматриваемый во всей книге. Если читатель знаком с этой предметной областью, то данную главу можно пропустить. Эта задача была выбрана в качестве примера именно потому, что многие с ней знакомы. В книге рассматриваются интересные проектные и архитектурные решения этой задачи, а не сама постановка задачи и ее предметная область.

## 3.1. Система автоматизации торговли NextGen

Основным примером, рассматриваемым в этой книге, является система автоматизации торговой точки NextGen. Рассматриваемая задача очень интересна своими требованиями и проблемами, возникающими в процессе проектирования. Кроме того, она очень реалистична. Многие организации действительно разрабатывают POS-системы (point-of-sale system) на основе объектной технологии.

POS-система — это компьютеризированное приложение, предназначенное для организации товарооборота и обработки платежей в обычных магазинах. Система автоматизации торговли включает аппаратные компоненты (компьютер и устройство считывания штрих-кода), а также программное

обеспечение, выполняющее основные задачи системы. Это приложение связано с различными служебными программами, например с программой вычисления налогов, разработанной сторонними производителями, или с системой складского учета товаров. Подобные системы должны быть устойчивыми



к сбоям, т.е. работоспособными при временном выходе из строя удаленных служб (например, системы складского учета товаров). В критических ситуациях они должны обслуживать продажу товаров и обеспечивать обработку хотя бы платежей наличными (чтобы хозяин магазина не обанкротился).

Желательно, чтобы POS-система поддерживала различные типы клиентских терминалов и интерфейсов, в том числе клиентский терминал с Web-броузером ("тонкого" клиента), обычный персональный компьютер с графическим интерфейсом пользователя, сенсорный ввод информации и т.п.

Более того, коммерческие POS-системы должны уметь работать с различными категориями потребителей, для которых определены отдельные бизнес-правила. Для каждого потребителя может быть предусмотрена своя логика выполнения отдельных операций в рамках сценария использования системы, например, специфические действия при добавлении нового товара или создании новой продажи. Следовательно, необходимо предусмотреть механизм обеспечения этой гибкости и настройки системы.

На основе итеративной стратегии разработки мы выполним все необходимые этапы создания системы: формулировку требований, объектно-ориентированный анализ, проектирование и реализацию.

## 3.2. Архитектурные уровни и основные моменты

Архитектура типичной информационной системы, включающей графический интерфейс пользователя и взаимодействие с базой данных, обычно рассматривается в ракурсе нескольких уровней или подсистем (рис. 3.1). В качестве таких уровней могут выступать, например, следующие.

- **Интерфейс пользователя** (графические элементы и диалоговые окна).
- **Уровень логики приложения или объектов предметной области** (включает программное представление объектов предметной области (например, программный класс `Sale` (продажа)), реализующих требования к системе).
- **Технические службы** (объекты и подсистемы общего назначения, которые обеспечивают выполнение вспомогательных функций, например обмен информацией с базой данных или регистрацию событий).

> Объектно-ориентированный анализ и проектирование относятся, в основном, к уровням логики приложения и технических служб.

При рассмотрении данного примера мы, в основном, сосредоточимся на разработке объектов предметной области, распределении обязанностей между ними с целью выполнения требований к приложению. На основе объектно-ориентированного подхода будет также создана подсистема технической поддержки, обеспечивающая взаимодействие с базой данных.

При таком подходе уровень графического интерфейса выполняет весьма незначительные обязанности, поэтому его можно назвать "*тонким*" (thin). Диалоговые окна не имеют отношения к логике приложения или обработке информации. Основная нагрузка по выполнению системных требований приходится на объекты предметной области и служебные объекты.

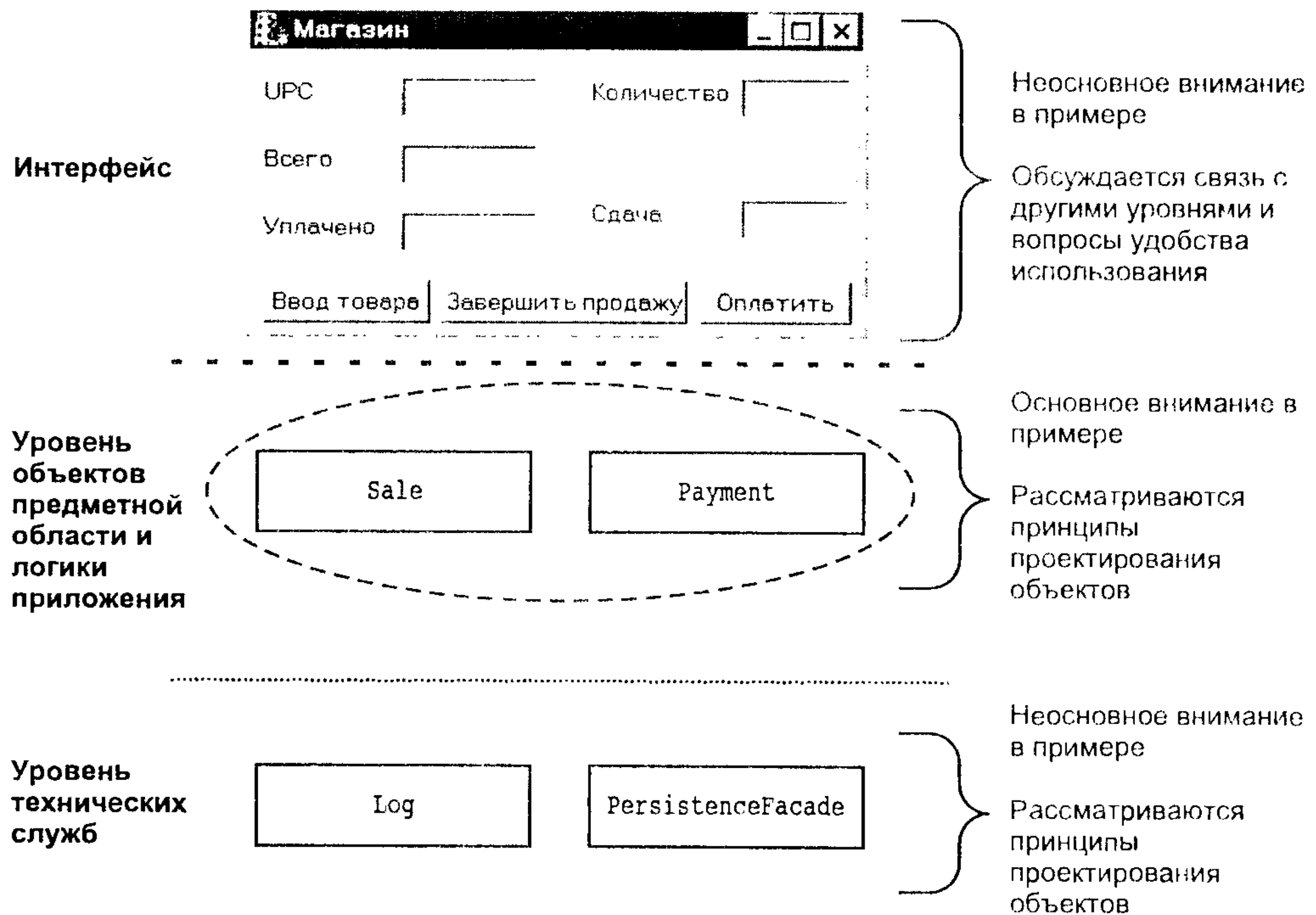



*Рис. 3.1. Уровни и объекты типичной объектно-ориентированной системы с указанием степени детализации их описания в данной книге*

## 3.3. Стратегия данной книги: итеративное изучение и разработка

Структура материала в этой книге основывается на итеративной стратегии разработки. Объектно-ориентированный анализ и проектирование применяются к POS-системе NextGen в процессе реализации нескольких итераций. На первой итерации обеспечивается выполнение некоторых базовых функций, а на последующих расширяется функциональность системы (рис. 3.2). В соответствии с итерационной стратегией разработки в книге будут последовательно представлены главы, посвященные объектно-ориентированному анализу и проектированию, системе обозначений языка UML и шаблонам. В главах, посвященных первой итерации, будут рассмотрены базовые вопросы анализа и проектирования, а также основные обозначения, принятые в UML. В главах, посвященных второй итерации, будут описаны новые идеи, новые обозначения UML, шаблоны и т.д.

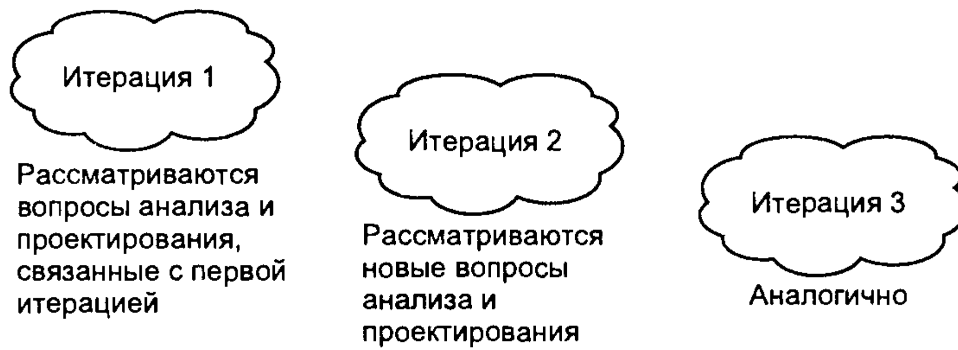



*Рис. 3.2. Последовательность изучения материала соответствует итеративному процессу разработки приложения*

# Часть II

# Начальная фаза

Глава

# Начало

*Le meiux est l'ennemi du bien (лучшее — враг хорошего).*

*Вольтер (Valtair)*

---

Основные задачи

- **Определить начальную фазу.**
- **Обосновать содержание глав этой части.**

---

## Введение

В этой главе определяется начальная фаза реализации проекта. Если вы не любите тратить время на осмысление идей или предпочитаете сначала сконцентрироваться на изучении основных видов практической деятельности этой фазы (а именно, моделировании прецедентов), то можете пропустить эту главу.

Для большинства проектов необходим небольшой начальный этап, на котором нужно сформулировать ответы на следующие вопросы.

- Каково ваше в*и*дение проекта?
- Реально ли осуществить задуманное?
- Что лучше: купить или разработать?
- В какую сумму (примерно) обойдется реализация проекта: 10–100 тысяч или миллионы долларов?
- Стоит ли браться за этот проект?

Определив свое в*и*дение задачи и приблизительно оценив необходимые затраты, можно приступать к изучению требований. Однако задачей начальной фазы не является определение всех требований, составление правдоподобного бюджета или плана реализации проекта. На этом этапе необходимо лишь осмыслить общие задачи проекта и его реалистичность, а также сформировать объективное мнение о целесообразности продолжения проекта (более глубокое изучение будет выполнено на стадии развития).

Поэтому для большинства проектов начальная фаза должна быть относительно короткой и занимать не более одной-двух недель. На самом деле, если этот этап занимает более одной недели, значит, начальная фаза была пропущена. На этой стадии нужно лишь решить, заслуживает ли данный проект серьезного исследования (на этапе его развития), а не выполнять его.

> В двух словах начальную стадию можно охарактеризовать так.
>
> *Представить масштаб продукта, сформировать свое видение и оценить затраты.*
>
> Основная проблема этой фазы.
>
> *Руководители проекта должны прийти к соглашению относительно сути проекта и целесообразности серьезных инвестиций.*

## 4.1. Аналогия

Когда в нефтяной промышленности принимается решение о начале разработки нового месторождения, выполняются следующие действия.

1. Определяется, достаточно ли информации и аргументов для обоснования необходимости бурения пробной скважины.
2. Если да, то проводятся измерения и выполняется бурение пробной скважины.
3. Оценивается нефтяной запас и целесообразность затрат на разработку.
4. Начинается разработка.

Начальная фаза разработки программного продукта соответствует первому этапу в этой аналогии, когда специалисты еще не знают, каков нефтяной запас в данном регионе или какой будет стоимость его разработки. Для подобных выводов еще недостаточно информации. Конечно, очень заманчиво сразу ответить на все вопросы "сколько" и "когда", не приступая к исследованию проблемы, но нефтяники понимают, что это нереально.

В терминах UP реальное исследование проблемы выполняется на стадии развития. Начальная фаза сродни анализу достижимости, на этой стадии требуется принять решение о целесообразности капиталовложений в бурение пробной скважины. И только после бурения пробной скважины и исследования нового месторождения (фазы развития) появляются основания для реалистичных оценок и планов. Так и в итеративной разработке: на начальной стадии нельзя рассчитывать на построение обоснованных планов и оценок. В этот период можно оценить лишь порядок величины, уровень необходимых затрат и принять решение о целесообразности дальнейших исследований.

## 4.2. Начало может быть очень коротким

Итак, начальная фаза предназначена для формирования общего видения задач проекта, выяснения достижимости поставленных целей и принятия решения о целесообразности дальнейших серьезных исследований на стадии развития. Если решение о перспективности и реалистичности проекта принято заранее (например, на том основании, что команда разработчиков уже успешно участвовала в подобных проектах),

то фаза начала будет особенно краткой. Она может включать лишь вводный семинар, на котором обсуждаются начальные требования и планирование первой итерации. После этого можно сразу перейти к фазе развития.

## 4.3. Какие артефакты относятся к начальной фазе

В табл. 4.1 приводится список типичных артефактов фазы начала (или первых шагов фазы развития) с краткими пояснениями. В последующих главах некоторые из этих артефактов будут рассмотрены подробнее, особенно модель прецедентов. В контексте итеративной разработки необходимо понимать, что на этой стадии не стоит выполнять серьезное исследование, все артефакты будут позднее доработаны. Поэтому на начальной стадии можно вообще обойтись без документирования, либо ограничиться краткими артефактами.

Например, в модели прецедентов (которая будет рассмотрена в последующих главах) могут содержаться имена наиболее важных прецедентов и их исполнителей. При этом лишь 10% этих прецедентов могут быть детализированы (чтобы сформировать общее *ви*дение масштаба проблемы, задач проекта и возможных рисков).

Начальная фаза может включать и программирование, т.е. создание прототипов (особенно относящихся к интерфейсу пользователя), обеспечивающих обоснование правильности идеи. Это поможет лучше осознать системные требования и выявить наиболее серьезные технические проблемы.

**Таблица 4.1. Приблизительный перечень артефактов начальной фазы**

| Артефакт | Пояснение |
|---|---|
| Видение проекта | Описываются общие задачи и ограничения, приводится заключение |
| Модель прецедентов | Описываются функциональные и нефункциональные требования |
| Дополнительная спецификация | Описываются другие требования |
| Словарь терминов | Содержит ключевую терминологию по данной предметной области |
| Перечень рисков и план управления ими | Описываются экономические, технические риски, риски, связанные с организацией планирования и ресурсами, а также идеи по их преодолению |
| Прототипы и обоснование идеи | Приводятся для лучшего осмысления проекта и оценки технических идей |
| План итерации | Описывается, что предстоит делать на первой итерации фазы развития |
| План на следующую фазу и план разработки | Приблизительный план фазы развития, описание средств, человеческих ресурсов, необходимых навыков и других ресурсов |
| Перечень документов | Описание этапов UP и артефактов данного проекта. Напомним, что в рамках UP набор артефактов определяется для каждого проекта в отдельности |

Перечисленные артефакты создаются на данной стадии не полностью. Они будут постоянно обновляться и совершенствоваться на последующих итерациях. Приведенные названия артефактов соответствуют терминологии, принятой в спецификации UP.

### *Не слишком ли много документации*

Напомним, что все артефакты являются не обязательными. Выберите только те из них, которые действительно важны для данного проекта, а на остальные не обращайте внимания.

Заметим, что артефакт — это не документ или диаграмма, а процесс осмысления, анализа и разработки (с последующей записью результатов во избежание

повторения или забывания). Генерал Эйзенхауэр говорил следующее: "При подготовке к битве я всегда убеждался в бесполезности планов, но планирование — процесс обязательный" [12, 85].

Артефакты лучше зафиксировать с помощью цифровой техники и в интерактивных документах, поместив их на Web-узле проекта, а не на бумаге.

Артефакты предыдущих проектов в рамках UP можно повторно использовать в последующих. Скорее всего, между проектами будет много общего, в частности риски, методы управления проектом, тестирование, артефакты окружения. Во всех проектах UP артефакты должны иметь одинаковые имена (например, "Перечень рисков", "Перечень документов" и т.д.). Это упрощает процесс поиска схожих артефактов и возможность их повторного использования в новых проектах.

## 4.4. Вы не поняли, что такое начальная фаза, если...

- Для большинства проектов она занимает более нескольких недель.
- На этом этапе вы стараетесь определить все требования.
- Надеетесь, что оценки и планы будут реалистичными.
- Занимаетесь определением архитектуры, хотя это надо делать на стадии развития.
- Считаете правильной следующую последовательность действий: 1) определение требований; 2) проектирование архитектуры; 3) реализация.
- У вас отсутствует артефакт "В*и*дение".
- Имена большинства прецедентов и исполнителей не определены.
- Все прецеденты описываются во всех подробностях.
- Ни один из прецедентов не описан в деталях (на самом деле, 10–20% прецедентов должны быть подробно описаны, чтобы оценить масштаб проблемы).

Глава

# Осмысление требований

*Быстро, дешево, хорошо — выберите любые два.*

*Автор неизвестен*

**Основные задачи**

- **Определить модель FURPS+.**
- **Связать типы требований с артефактами UP.**

## Введение

Не все требования равнозначны. В этой главе описываются категории требований в контексте модели FURPS+.

*Требования* (requirements) — это возможности или условия, которым должна соответствовать система или проект [67]. Основная задача этапа определения требований — найти, обсудить и зафиксировать, что действительно требуется от системы в форме, понятной и клиентам и членам команды разработчиков.

В рамках унифицированного процесса провозглашается ряд идей, к числу которых относится управление требованиями. Это не означает необходимости окончательного определения и стабилизации требований на первой фазе проекта. Наоборот, с учетом постоянно меняющихся требований заказчиков и руководителей проекта предлагается "систематизированный подход к поиску, документированию, организации и отслеживанию изменчивых требований к системе" [97]. То есть к формулировке требований нужно относиться очень ответственно. Обратите внимание на слово "изменчивые" в предыдущем предложении. В контексте UP изменение требований считается главным "двигателем" прогресса. Отметим также слово "поиск". Предполагается, что требования должны постоянно корректироваться в процессе описания прецедентов и специальных семинаров.

На рис. 5.1 показаны результаты исследования факторов риска для программных проектов. Согласно этим исследованиям, 37% риска связано с требованиями, следовательно, удельный вес проблем с требованиями среди общих проблем программных проектов очень высок [100]. Поэтому очень важно грамотно организовать управление требованиями. В рамках последовательного жиз-

ненного цикла эти результаты заставляют уделять более пристальное внимание вопросам формулировки, стабилизации и утверждения требований до начала проектирования и реализации. Но практика свидетельствует о том, что этот путь ведет к поражению. В рамках итеративного процесса разработки изменение требований и обратная связь рассматриваются как главные “двигатели” прогресса.

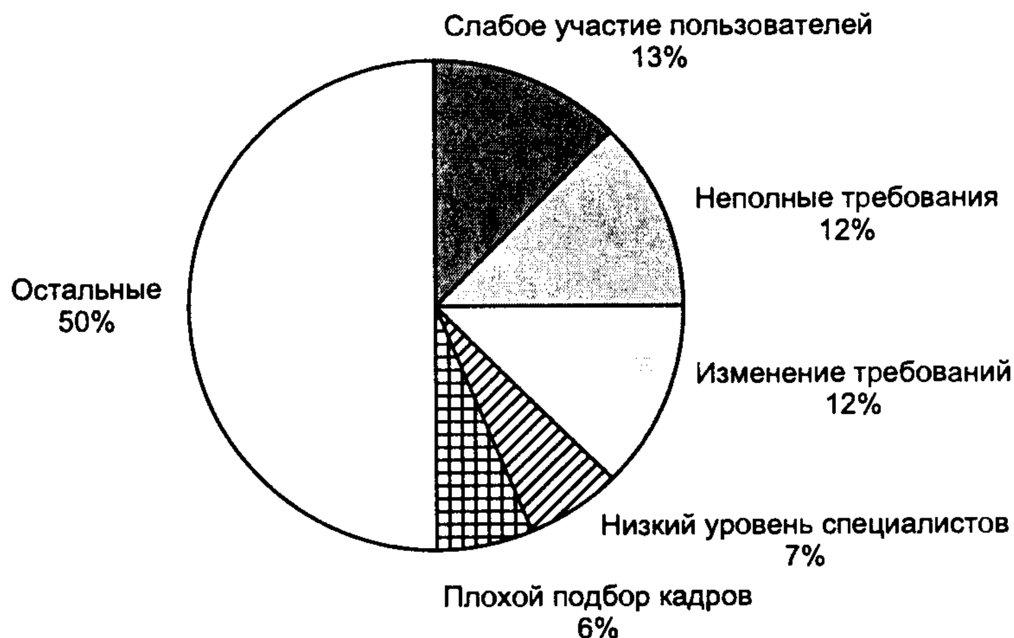



*Рис. 5.1. Факторы риска для программных проектов*

## 5.1. Типы требований

В рамках UP, согласно модели FURPS+ [57], требования делятся на следующие категории.[1]

- **Функциональные требования** — свойства, возможности, безопасность.
- **Удобство** — человеческий фактор, справочная система, документация.
- **Надежность** — частота сбоев, возможность восстановления и предсказуемость поведения.
- **Производительность** — время отклика, точность, доступность, использование ресурсов.
- **Возможность поддержки** — адаптивность, возможность поддержки, соответствие международным стандартам, возможность конфигурирования.

Символ “+” в названии FURPS+ означает дополнительные (не определяющие) факторы, к которым относятся следующие.

- **Реализация** — требования к ресурсам, языки и средства, аппаратное обеспечение.

[1] Существует несколько принципов систематизации требований и перечней атрибутов качества. Они опубликованы в книгах и документах организаций, занимающихся вопросами стандартизации, в том числе в стандарте ISO 9126 (список требований аналогичен модели FURPS+). Несколько списков требований были разработаны институтом SEI (System Engineering Institute). Все эти требования можно использовать в проектах UP.

- **Интерфейс** — ограничения, накладываемые необходимостью взаимодействия с внешними системами.
- **Операции** — управление системой и ее параметры.
- **Пакетирование.**
- **Юридические вопросы** — авторское право и т.п.

Категории FURPS+ полезно использовать при формулировке требований, чтобы не упустить важные аспекты жизнедеятельности системы.

Некоторые из этих требований (удобство, надежность, производительность и возможность поддержки) называются *атрибутами качества* (quality attributes). Обычно требования делят на две большие категории: *функциональные* (относящиеся к поведению) и *нефункциональные* (все остальные). Некоторые специалисты не любят такое обобщенное деление [9], но оно применяется довольно часто.

Функциональные требования исследуются и формулируются в процессе разработки модели прецедентов, которая будет описана в следующей главе, а также в процессе осмысления видения системы. Остальные требования формулируются при более детальном описании прецедентов или в дополнительной спецификации. Требования высокого уровня фиксируются в документе "Видение". Затем они уточняются и конкретизируются в последующих документах. В словаре терминов разъясняются термины, фигурирующие при описании требований. В рамках UP в этом словаре должно содержаться понятие "*словарь данных*", в котором описываются требования к данным, а именно правила верификации, допустимые значения и т.п. Механизм выяснения требований обеспечивает прототипирование.

Как будет видно дальше, требования к качеству оказывают существенное влияние на архитектуру системы. Например, требования высокой производительности или надежности влияют на выбор программного обеспечения и аппаратных средств, а также на конфигурацию системы. Наиболее важным "лейтмотивом" при разработке программных систем является необходимость адаптивности с целью отслеживания быстро изменяющихся функциональных требований.

## 5.2. Дополнительная литература

Ссылки на литературу, связанную с формулировкой требований в процессе описания прецедентов, приводятся в следующей главе. Отправной точкой для изучения требований в процессе описания прецедентов можно считать книгу [33].

Требования и многие другие вопросы, связанные с разработкой программных продуктов, обсуждаются на Web-узле `www.swebok.org` под флагом организации SWEBOK (Software Engineering Body of Knowledge).

Институт SEI (`www.sei.cmu.edu`) разработал несколько предложений по стандартизации требований к качеству. Стандарты ISO 9126, IEEE Std 830 и IEEE Std 1061, относящиеся к требованиям и атрибутам качества, приводятся на различных Web-узлах.

Однако книги, посвященные анализу требований, нужно изучать критически (даже те из них, где речь идет об описании прецедентов, итеративной разработке и требованиях в рамках UP). На то есть несколько причин.

1. Большинство авторов книг тяготеют к последовательному жизненному циклу и определению требований до начала фазы проектирования и реализации. Это не умаляет ценности их умозаключений или глубины изучения вопроса.

Однако следует понимать, что они не следуют идеям итеративной разработки. Это может быть связано с тем, что авторы книг имели опыт разработки систем в последовательном жизненном цикле и привыкли вырабатывать требования до начала проектирования. И даже если в подобных книгах упоминается итеративная разработка, то зачастую это просто дань моде. Поэтому книги и статьи, посвященные формулировке требований, нужно читать очень осмотрительно, иначе можно скатиться к убеждению о необходимости определения всех требований на начальной стадии проекта. А это не согласуется с идеологией итеративной разработки.

2. Во многих книгах, посвященных формулировке требований, прецеденты используются очень неумело. Судя по всему, авторы вовсе не понимают, что означает формулировка требований на основе прецедентов. Это может быть вызвано тем, что предыдущий опыт авторов был связан с традиционными методами определения требований, а теперь они пытаются притянуть за уши к своему опыту современную концепцию описания прецедентов. При этом они не понимают основную идею Айвара Якобсона (Ivar Jacobson) и унифицированного процесса о том, что в основе определения требований лежит описание прецедентов. Описание прецедентов — ключевой момент в процессе определения требований, а не дополнительный или промежуточный прием, добавляемый к традиционным документам или подходам по формулировке требований.

Таким образом, в книгах, посвященных анализу требований, приводятся полезные советы и приемы по сбору требований. Эти книги, как правило, написаны опытными авторами, но зачастую относятся к последовательному процессу разработки, не предполагающему глубокого осмысления прецедентов. Все рекомендации типа "старайтесь сначала определить большинство требований, а затем приступайте к проектированию и реализации" не имеют ничего общего с итеративной разработкой и UP.

Глава

# Описание требований в контексте модели прецедентов

*Необходимый первый шаг для достижения заветной цели — понять, чего же вы хотите.*

*Бен Стейн (Ben Stein)*

**Основные задачи**

- **Определить и описать прецеденты.**
- **Связать прецеденты с потребностями пользователя и элементарными экономическими процессами.**
- **Использовать сжатый, свободный и развернутый форматы описания идеальных прецедентов.**
- **Связать работу над прецедентами с итеративной разработкой.**

## Введение

Эту главу нужно обязательно изучить при первом чтении книги, поскольку прецеденты — это широко используемый механизм осмысления и формулировки требований (особенно функциональных). Они оказывают влияние на множество аспектов проекта, включая ООА/П. Поэтому очень важно уметь создавать прецеденты.

Описание прецедентов — повествовательных историй об использовании системы — отличный метод осмысления и формулировки требований. В этой главе изучаются основные понятия, связанные с описанием прецедентов, и приводятся примеры прецедентов для приложения NextGen.

В контексте UP *модель прецедентов* (Use-Case Model) относится к дисциплине "Требования". Действительно, требования — это весь набор прецедентов, т.е. модель функционирования системы и ее окружения.

## 6.1. Задачи и описания

У потребителей и конечных пользователей есть свои задачи (которые в контексте UP называют потребностями), решение которых должна обеспечить компьютерная система. Эти задачи могут варьироваться от регистрации торговых операций до оценивания объемов добычи нефти из перспективных месторождений. Существует несколько способов выделения этих задач или системных требований. Наилучшие из них достаточно просты и доступны, поскольку это облегчает участие конечных пользователей в определении требований к системе. А участие в этом процессе потребителей снижает риск провала проекта.

Прецеденты — это механизм упрощения этапа формулировки требований для всех заинтересованных лиц. По существу это рассказы об использовании системы в процессе решения поставленных задач. Вот пример сжатого формата описания прецедента.

> **Обработка продажи** (process sale). Покупатель подходит к кассе с выбранными товарами. Кассир с помощью POS-системы регистрирует каждый товар. Система отображает информацию о каждом наименовании товара и вычисляет общую сумму. Покупатель вводит требуемую информацию; система ее верифицирует и регистрирует. Система выполняет инвентаризацию. Покупатель получает товарный чек и покидает магазин с покупками.

Зачастую прецеденты нужно продумывать гораздо детальнее. Но основная идея состоит в исследовании и формулировке функциональных требований путем написания историй "из жизни системы". Эти истории помогают сформулировать различные задачи и представляют собой сценарии использования системы.[1] На первый взгляд, описать прецеденты не сложно, хотя зачастую достаточно трудно определить, что требуется от системы и описать это на нужном уровне детализации.

О прецедентах написано достаточно много. Но все равно существует опасность, что умные творческие люди могут слишком усложнить изначально простую идею. Зачастую начинающие специалисты по описанию прецедентов слишком увлекаются созданием диаграмм прецедентов, описанием их взаимосвязей, составлением пакетов прецедентов, рассмотрением необязательных атрибутов и т.д., а не написанием историй. Сила механизма прецедентов состоит в возможности масштабировать уровень сложности и формальности описания в зависимости от реальных потребностей.

## 6.2. Предыстория

Идея описания функциональных требований в виде прецедентов была сформулирована в 1986 году Айваром Якобсоном (Ivar Jacobson) [65] — главным разработчиком языка UML и UP. Эта идея была признана конструктивной и одобрена широкой общественностью. Основными ее преимуществами являлись простота и полезность. Многие авторы пытались внести свой вклад в развитие этого вопроса. Однако наиболее важный вклад в проблему определения сущности прецедентов и способа их описания внес Алистер Кокбурн (Alistair Cockburn).

---

[1] Термин "прецедент" (use case) появился в Швеции и в переводе означал "случай использования" (usage case).

Его в*и*дение проблемы изложено в очень популярной книге [33], основанной на его опыте работы и материалах предыдущих публикаций.

## 6.3. Прецеденты и ощутимый результат

Сначала введем некоторые неформальные определения. *Исполнителем* (actor) будем называть сущность, обладающую поведением, например, человека (идентифицируемого по роли), компьютерную систему или организацию, например кассира.

*Сценарий* (scenario) — это специальная последовательность действий или взаимодействий между исполнителями и системой. Его иногда также называют *экземпляром прецедента* (use case instance). Это один конкретный сценарий использования системы либо один проход прецедента, например, сценарий успешной покупки товаров за наличный расчет, либо сценарий неудачного завершения покупки из-за прерванной транзакции по обработке данных кредитной карточки.

Неформально, *прецедент* (use case) — это набор взаимосвязанных успешных и неудачных сценариев, описывающий использование системы исполнителем для решения одной из задач. Например, рассмотрим свободный формат прецедента, включающего некоторые альтернативные сценарии.

**Возврат товара (Handle Returns)**

*Основной успешный сценарий.* Покупатель подходит к кассе с товарами, подлежащими возврату. Кассир использует POS-систему для регистрации каждого возвращаемого товара...

*Альтернативные сценарии.* Если в авторизации кредитной карточки отказано, кассир информирует об этом покупателя и предлагает ему другой способ оплаты покупки.

Если идентификатор товара в системе не обнаружен, система уведомляет об этом кассира и предлагает ему вручную ввести идентификационный код (возможно, штрих-код поврежден и его сложно считать).

Если у системы возникают сложности при коммуникации с внешней системой вычисления налога, ...

Несколько другое, но подобное определение прецедента приводится в RUP.

Прецедент — это набор сценариев использования, в котором каждый экземпляр сценария представляет собой последовательность действий, выполняемых системой для достижения ощутимого для конкретного исполнителя результата [97].

Фраза "*ощутимый результат*" несколько туманна, но очень важна, поскольку она указывает на то, что поведение системы должно быть ощутимо для пользователя.

> Основное внимание при описании прецедента нужно сконцентрировать на вопросе: "Как использование системы обеспечивает ощутимый для пользователя результат или решает его задачу?", а не на обдумывании системных требований в терминах свойств или функций.

На первый взгляд, требование обеспечения ощутимого результата может показаться очевидным. Однако в сфере программного обеспечения известно множество неудачных проектов, которые провалились из-за невыполнения реальных задач пользователей. К такому отрицательному результату может привести подход к описанию системных требований в виде списка свойств и функций системы, поскольку он не заставляет заинтересованных лиц рассматривать требования в более широком контексте достижения некоторого ощутимого результата или некоторой цели. В отличие от этого подхода, в контексте прецеден-

тов свойства и функции системы ориентированы на достижение цели. Это и отражено в названии главы.[2]

В этом состоит основная идея, сформулированная Якобсоном относительно прецедентов: требования должны быть направлены на получение ощутимого результата и достижения целей.

## 6.4. Прецеденты и функциональные требования

Прецеденты — это требования. В основном, это функциональные требования, указывающие на то, что должна делать система. В контексте типов требований, определяемых моделью FURPS+, основное внимание уделяется функциональным требованиям. Однако остальные типы требований тоже могут быть связаны с прецедентами. В рамках UP и большинства других современных методов прецеденты являются основным механизмом, рекомендуемым для их определения и исследования. Прецеденты определяют пожелания или соглашения относительно поведения системы.

Итак, прецеденты — это требования (хотя и не все требования). Некоторые считают требованиями только список функций и свойств типа "система должна...". На самом деле это не так. Ключевая идея использования прецедентов как раз и состоит (обычно) в снижении роли списка требований в старом понимании этого слова. Теперь в качестве функциональных требований выступают сами прецеденты. Более подробно этот вопрос рассматривается ниже.

Описания прецедентов — это текстовые документы, а не диаграммы. Моделирование прецедентов — это процесс написания текста, а не рисования. Однако для иллюстрации имен прецедентов и исполнителей, а также их взаимоотношений в UML определены обозначения для диаграммы прецедентов.

## 6.5. Типы и форматы прецедентов

### Прецеденты типа "черный ящик" и системные обязанности

*Прецеденты типа "черный ящик"* (black-box use cases) — это самый типичный и рекомендуемый тип прецедентов. Они не описывают внутреннюю работу системы, ее компоненты или дизайн. Наоборот, системе вменяются некоторые *обязанности* (responsibilities). Этот метафорический термин широко применяется в объектно-ориентированном проектировании: программные элементы имеют обязанности и взаимодействуют с другими элементами со своими обязанностями.

Определяя обязанности системы через прецеденты типа "черный ящик", можно указать, *что* должна делать система (функциональные требования), не расписывая, *как* это делать (не выполняя проектирование). Вообще, термины "анализ" и "проектирование" зачастую сводятся к вопросам "что" и "как". Это важные вопросы в хорошей программной разработке. В процессе анализа требований нужно избегать принятия решений "как", а описывать лишь внешнее поведение системы как черного ящика. Позднее, на этапе проектирования, создается решение, удовлетворяющее разработанной спецификации.

---

[2] Эта глава получила свое название под влиянием статьи [54] (название главы согласовано с авторами статьи).

| Стиль черного ящика | Другой стиль |
|---|---|
| Система регистрирует покупку | Система записывает сведения о покупке в базу данных.<br>Или, еще хуже:<br>система генерирует оператор SQL `INSERT` для данной продажи... |

### *Степень формализации*

Прецеденты описываются в различных форматах, в зависимости от потребностей. Помимо типов "черного ящика" и "белого ящика", выделяют несколько степеней формализации описания прецедентов.

- *Сжатый* — аннотация в виде одного абзаца. Обычно она описывает только главный успешный сценарий. Пример такого описания приведен выше для прецедента `Оформление продажи` (Process Sale).
- *Свободный* — неформальный стиль описания. Описание прецедента занимает несколько абзацев и охватывает различные сценарии. Примером такого описания является рассмотренный выше прецедент `Возврат товара`.
- *Развернутый* — наиболее подробный стиль описания. При таком подходе детально описываются все шаги и варианты развития сценария, а также предусловия и результаты.

Рассмотрим пример развернутого описания прецедента для системы NextGen.

## 6.6. Пример развернутого описания прецедента `Оформление продажи`

Развернутые описания прецедентов структурированы и содержат большое количество деталей. Их полезно использовать для углубления понимания целей, задач и требований. Для примера POS-системы NextGen такие описания можно обсуждать на семинарах по определению требований на начальной стадии проекта вместе с системным аналитиком, экспертами предметной области и разработчиками.

### *Формат `usecases.org`*

Для развернутого описания прецедентов существуют различные шаблоны форматирования. Однако чаще всего используется шаблон, приведенный на Web-узле `www.usecases.org`. Этот стиль проиллюстрирован в следующем примере.

Этот пример детального описания прецедента относится к рассматриваемой в данной книге системе NextGen и отражает множество типичных элементов и вопросов.

#### Прецедент П1. Оформление продажи

**Основной исполнитель.** Кассир.

**Заинтересованные лица и их требования**

- Кассир. Хочет точно и быстро ввести данные, не допуская ошибок в платеже, поскольку недостача вычитается из его зарплаты.
- Продавец. Хочет получить свои комиссионные от продажи.
- Покупатель. Хочет купить товары и быстро оформить покупку с минимальными усилиями. Хочет получить подтверждение факта покупки для возможного возврата товара.
- Компания. Хочет аккуратно записать транзакцию и удовлетворить интересы покупателя. Хочет удостовериться, что служба авторизации платежей зафиксировала все данные о платеже. Заин-

заинтересована в обеспечении устойчивости к сбоям; хочет продолжать регистрировать продажи, даже если серверные компоненты (например, служба удаленной проверки кредитоспособности) недоступны. Хочет автоматически обновлять бухгалтерскую документацию и вести складской учет.
- Государственные налоговые службы. Хотят получать налог от каждой продажи. Таких служб может быть несколько, в том числе национальная и местная.

**Предусловия.** Кассир идентифицирован и аутентифицирован.

**Результаты (Постусловия).** Данные о продаже сохранены. Налоги корректно вычислены. Бухгалтерские и складские данные обновлены. Комиссионные начислены. Чек сгенерирован. Авторизация платежа выполнена.

**Основной успешный сценарий (или основной процесс)**

1. Покупатель подходит к кассовому аппарату POS-системы с выбранными товарами.
2. Кассир открывает новую продажу.
3. Кассир вводит идентификатор товара.
4. Система записывает наименование товара и выдает его описание, цену и общую стоимость. Цена вычисляется на основе набора правил.

*Кассир повторяет действия, описанные в пп. 3-4, для каждого наименования товара.*

5. Система вычисляет общую стоимость покупки с налогом.
6. Кассир сообщает покупателю общую стоимость и предлагает оплатить покупку.
7. Покупатель оплачивает покупку, система обрабатывает платеж.
8. Система регистрирует продажу и отправляет информацию о ней внешней бухгалтерской системе (для обновления бухгалтерских документов и начисления комиссионных) и системе складского учета (для обновления данных).
9. Система выдает товарный чек.
10. Покупатель покидает магазин с чеком и товарами (если он что-то купил).

**Расширения (или альтернативные потоки)**

*а. При каждом выходе системы из строя.
  Для ввода системы в строй и корректной обработки платежа нужно обеспечить восстановление всех транзакций и событий с любого шага сценария.
  1. Кассир перезапускает систему, регистрируется и предлагает восстановить предыдущее состояние.
  2. Система восстанавливает предыдущее состояние.
    2а. Система определяет аномалию, повлекшую сбой.
      1. Система уведомляет об ошибке кассира, регистрирует ошибку и переходит в начальное состояние.
      2. Кассир начинает новую продажу.

3а. Неправильный идентификатор.
  1. Система уведомляет об ошибке и отменяет ввод данного наименования товара.

3б. В рамках одной категории существует несколько различных наименований товара и идентифицировать конкретное наименование не нужно (например, 5 пакетов леденцов).
  1. Кассир может ввести идентификатор категории товара и количество единиц.

3-6а. Покупатель просит кассира отменить покупку одного из товаров.
  1. Кассир вводит идентификатор товара для удаления из продажи.
  2. Система отображает обновленную промежуточную стоимость.

3-6б. Покупатель просит кассира отменить продажу.
  1. Кассир отменяет продажу.

3-6в. Кассир приостанавливает продажу.
  1. Система записывает сведения о продаже таким образом, чтобы они были доступны с любого терминала POS-системы.

4а. Сгенерированная системой цена товара не устраивает покупателя (например, у него есть дисконтная карта и он рассчитывает на более низкую цену товара).
  1. Кассир вводит команду об изменении цены.
  2. Система вычисляет новую цену.

5а. Система выявляет сбой при коммуникации с внешней службой вычисления налога.
  1. Система перезапускает службу с данного узла POS-системы и продолжает работу.

1а. Система определяет, что служба не перезапускается.
1. Система сигнализирует об ошибке.
2. Кассир может вручную вычислить и ввести сумму налога либо отменить продажу.

5б. Покупатель сообщает о положенной ему скидке (например, для сотрудников данного предприятия или постоянных покупателей).
1. Кассир отправляет запрос на скидку.
2. Кассир вводит идентификационные данные покупателя.
3. Система представляет сумму скидки, вычисленную на основе дисконтных правил.

5в. Покупатель сообщает об открытом в данном магазине кредите на фиксированную сумму и просит оформить продажу с его помощью.
1. Кассир отправляет запрос на оформление платежа с использованием открытого кредита.
2. Кассир вводит идентификационную информацию о покупателе.
3. Система снижает стоимость покупки (вплоть до 0) и уменьшает оставшуюся сумму кредита.

6а. Покупатель сообщает, что хочет оплатить покупку наличными, но у него недостаточно денег.
1а. Покупатель использует альтернативный способ платежа.
1б. Покупатель просит кассира отменить продажу. Кассир отменяет продажу в системе.

7а. Оплата наличными.
1. Кассир вводит предложенную покупателем сумму.
2. Система вычисляет положенную сдачу и открывает кассу с наличностью.
3. Кассир складывает поученные деньги и выдает сдачу покупателю.
4. Система регистрирует платеж наличными.

7б. Оплата по кредитной карточке.
1. Покупатель вводит информацию о своей кредитной карточке.
2. Система отправляет запрос на авторизацию платежа внешней системе службы авторизации платежей и запрашивает подтверждение платежа.
   2а. Система определяет сбой при взаимодействии с внешней системой.
   1. Система сигнализирует об ошибке кассиру.
   2. Кассир просит покупателя изменить тип платежа.
3. Система получает информацию о подтверждении платежа и сообщает об этом кассиру.
   3а. Система получает информацию об отказе в выполнении платежа.
   1. Система сообщает об отказе кассиру.
   2. Кассир просит покупателя изменить тип платежа.
4. Система регистрирует платеж по кредитной карточке, включая информацию о подтверждении платежа.
5. Система предоставляет механизм ввода подписи для платежа по кредитной карточке.
6. Кассир просит покупателя подписать чек на оплату по кредитной карточке. Покупатель вводит подпись.

7в. Оплата чеком.

7г. Оплата по дебитной карточке.

7д. Покупатель предоставляет купоны.
1. До обработки платежа кассир вводит информацию о каждом купоне, в соответствии с которой система снижает стоимость покупки. Система регистрирует предъявленный купон для нужд бухгалтерии.
   1а. Купон действует не для всех выбранных товаров.
   1. Система сигнализирует об ошибке кассиру.

9а. Генерация чека.
1. Система выводит формы и чеки для каждого товара.

9б. Покупатель просит выдать ему подарочный чек (без указания цены).
1. Кассир вводит запрос на подарочный чек, и система выдает его.

**Специальные требования**

- Сенсорный экран с интерфейсом пользователя для большого плоского монитора. Текст должен быть виден с расстояния один метр.
- Отклик службы авторизации в 90% случаев приходит в течение 30 секунд.
- Каким-то образом нужно обеспечить робастное восстановление информации в случае сбоя при

доступе к удаленным службам, таким как система складского учета.
- Возможность локализации (представления на различных языках) отображаемого текста.
- Возможность добавления новых бизнес-правил на шагах 3 и 7 в процессе функционирования системы.
- ...

**Список технологий и типов данных**

3а. Идентификатор товара считывается со штрих-кода (при наличии последнего) лазерным сканером или вводится с клавиатуры.

3б. Идентификатор товара может определяться по схемам кодирования UPC, EAN, JAN или SKU.

7а. Информация об открытом кредите вводится с помощью считывающего устройства или с клавиатуры.

7б. Подпись при оплате чеком ставится на бумажном документе. Однако ожидается, что в течение двух лет большинство покупателей будут требовать цифровые устройства считывания подписи.

**Частота использования**: почти постоянно.

**Открытые вопросы**
- Изучить законодательство по налогообложению.
- Исследовать вопрос восстановления удаленных служб.
- Какая настройка потребуется для различных типов магазинов.
- Должен ли кассир снимать кассу при выходе из системы.
- Может ли пользователь сам использовать устройство считывания данных с карточки или это должен делать кассир.

Этот прецедент скорее является иллюстративным, чем исчерпывающим (хотя и основывается на реальных требованиях к POS-системе). Тем не менее, он описан достаточно подробно и позволяет составить реалистичное представление о развернутом формате описания прецедентов и их сложности. Этот пример может служить моделью для многих других прецедентов.

## *Представление в виде двух колонок*

Некоторые предпочитают оформлять прецеденты в виде двух колонок, обращая внимание на факт взаимодействия исполнителей и системы. Такой формат впервые был предложен Ребеккой Вирфс-Брок (Rebecca Wirfs-Brock) в работе [105] и поддержан Константином (Constantine) и Локвудом (Lockwood) в работе [28]. Вот как выглядит рассмотренное выше описание, представленное в виде двух колонок.

### Прецедент П1. Оформление продажи

**Основной исполнитель**: ...
... как и ранее...

**Основной успешный сценарий**

| Действие исполнителя | Отклик системы |
|---|---|
| 1. Покупатель подходит к кассовому аппарату POS-системы с выбранными товарами. | |
| 2. Кассир открывает новую продажу. | |
| 3. Кассир вводит идентификатор товара. | 4. Система записывает наименование товара и выдает его описание, цену и общую стоимость. Цена вычисляется на основе набора правил. |
| *Кассир повторяет действия, описанные в пп. 3-4 для каждого наименования товара.* | 5. Система вычисляет общую стоимость покупки с налогом. |

| | |
|---|---|
| 6. Кассир сообщает покупателю общую стоимость и предлагает оплатить покупку. | |
| 7. Покупатель оплачивает покупку. | 8. Система обрабатывает платеж. |
| | 9. Система регистрирует продажу и отправляет информацию о ней внешней бухгалтерской системе (для обновления бухгалтерских документов и начисления комиссионных) и системе складского учета (для обновления данных). Система выдает товарный чек. |
| ... | ... |

### Какой формат лучше?

Однозначного ответа на этот вопрос не существует. Одни предпочитают использовать колонки, другие — нет. В приведенное описание можно добавлять новые разделы или удалять лишние. Заголовки разделов тоже можно изменять. Все эти частности большой роли не играют. Главная задача — детально описать основной успешный сценарий и его расширения в некоторой стандартной форме. Множество полезных форматов описано в [33].

*Из личного опыта*

Это лишь опыт автора книги, а не рекомендация. В течение нескольких лет автор использовал формат описания в виде двух колонок, поскольку он обеспечивает наглядное представление общения с системой. Однако затем он перешел к обычному формату, поскольку он более компактен, и его легче форматировать. Эти два преимущества для автора книги с лихвой перекрывают преимущества формата с колонками. Кроме того, для облегчения восприятия можно описывать действия исполнителя и системы в отдельных абзацах со своими номерами.

## 6.7. Пояснения

### Вводные элементы

В описание прецедента можно добавлять различные вводные элементы. Те из них, которые важны для изложения последующего сценария, следует помещать в начало описания. Второстепенный материал можно располагать в конце сценария. Например, в описание можно добавить следующий вводный элемент.

**Главный исполнитель.** Основной исполнитель, вызывающий системные службы для достижения цели.

### Заинтересованные лица и их потребности

Этот список играет более важную роль, чем это кажется на первый взгляд. С его помощью можно понять, что должна делать система. Приведем цитату.

> "Система реализует соглашение между заинтересованными лицами. Поведение системы описывается с помощью прецедентов... Прецедент, как соглашение о поведении, включает все возможные аспекты поведения, связанные с удовлетворением запросов заинтересованных лиц" [33].

Эта цитата дает ответ на вопрос, что нужно описывать в прецеденте. Там нужно описывать все, что служит удовлетворению запросов заинтересованных лиц. Кроме того, начиная описание прецедента с перечня заинтересованных лиц

и их интересов, можно более точно определить функции системы. Например, можно ли было включить в перечень ее функций обработку комиссионных продавца, если до этого не определить его интересы? На самом деле, автор книги в процессе начального анализа упустил из виду этот аспект и вспомнил о нем только после составления списка заинтересованных лиц. Этот пример еще раз подтверждает важность составления списка.

**Заинтересованные лица и их потребности**

– Кассир. Хочет точно и быстро ввести данные, не допуская ошибок в платеже, поскольку недостача вычитается из его зарплаты.

– Продавец. Хочет получить свои комиссионные от продажи.

– ...

## Предусловия и постусловия

**Предусловия** (preconditions) — это перечень предпосылок, которые *всегда должны* выполняться до начала сценария прецедента. Предусловия *не* проверяются при реализации прецедента. То есть это условия, которые считаются истинными. Обычно в качестве предусловия выступает успешный результат выполнения другого сценария, например, загрузки или авторизации. Заметим, что в качестве предусловий перечисляются не все возможные истинные условия. Например, никто не упоминает в предусловиях наличие напряжения в электросети. Предусловия — это те предпосылки, на выполнение которых разработчик прецедента хочет обратить особое внимание.

**Результаты или постусловия** (postconditions). Описывают, какие условия обязательно должны выполняться в случае успешного завершения сценария. Эти результаты должны удовлетворять интересам всех заинтересованных лиц.

**Предусловия**. Кассир идентифицирован и аутентифицирован.
**Результаты (постусловия)**. Данные о продаже сохранены. Налоги корректно вычислены. Бухгалтерские и складские данные обновлены. Комиссионные начислены. Чек сгенерирован.

## Основной успешный сценарий (или основной процесс)

Это пункт можно также назвать "сценарием успеха" или, более прозаично, основным процессом. В нем описывается типичная последовательность действий, приводящая к успешному завершению сценария и удовлетворяющая потребности всех заинтересованных лиц. Заметим, что чаще всего в этом разделе *нет* никаких условий или ветвей. И хотя вводить какие-либо условия не запрещается, их обычно выносят в раздел расширений.

*Совет*

Выносите все условия или возможные варианты развития событий в раздел расширений.

В разделе основного сценария описываются три вида действий.

1. Взаимодействие между исполнителями.[3]
2. Верификация (обычно со стороны системы).

---

[3] Заметим, что описываемая система сама является исполнителем, если рассматривается аспект ее взаимодействия с другими системами.

3. Изменение состояния системы (например, запись или модификация некоторых сущностей).

Первый шаг прецедента не всегда подпадает под эту классификацию. Он служит триггером события начала сценария.

Имена исполнителей принято начинать с заглавной буквы для облегчения их идентификации. Повторяющиеся действия выделяются курсивом.

**Основной успешный сценарий**

1. Покупатель подходит к кассовому аппарату POS-системы с выбранными товарами.
2. Кассир открывает новую продажу.
3. Кассир вводит идентификатор товара.
4. ...

*Кассир повторяет действия, описанные в пп. 3-4, для каждого наименования товара.*

5. ...

## *Расширения (или альтернативные потоки)*

Этот раздел очень важен. Здесь указываются все остальные сценарии или ветви, приводящие к успешному или неудачному завершению прецедента. Обратите внимание, что этот раздел больше и сложнее предыдущего. Так и должно быть.

При описании прецедента основной успешный сценарий и его расширения должны охватывать *почти* все интересы заинтересованных лиц. Некоторые интересы лучше выразить в виде нефункциональных требований в дополнительной спецификации, а не в описании прецедента.

Расширения — это ответвления от основного сценария. Например, при реализации п. 3 основного сценария идентификатор товара может оказаться неправильным либо по причине некорректного ввода, либо из-за его отсутствия в системе. В расширении 3а сначала определяются условия, а затем реакция на них. Расширения для каждого пункта основного сценария обозначаются последовательностью, состоящей из номера этого пункта и буквы алфавита. Например, расширения для п. 3 будут нумероваться 3а, 3б и т.д.

**Расширения**

3а. Неправильный идентификатор.
  1. Система уведомляет об ошибке и отменяет ввод данного наименования товара.

3б. В рамках одной категории существует несколько различных наименований товара и идентифицировать конкретное наименование не нужно (например, 5 пакетов леденцов).
  1. Кассир может вести идентификатор категории товара и количество единиц.

Описание расширения состоит из двух частей: условия и способа его обработки.

*Совет.* Описывайте условия как факты, *выявляемые* системой или исполнителем. Сравните следующие два пункта.

5а. Система выявляет сбой при коммуникации с внешней службой вычисления налога.
5а. Внешняя система вычисления суммы налога не работает.

Первый пример предпочтительнее, поскольку в нем описан факт, который может быть выявлен системой.

Способ обработки возникшего условия можно описать в одном или нескольких пунктах (как приведено ниже).

3-6а. Покупатель просит кассира отменить покупку одного из товаров.
  1. Кассир вводит идентификатор товара для удаления из продажи.
  2. Система отображает обновленную промежуточную стоимость.

По окончании обработки расширения по умолчанию выполняется возврат к основному сценарию, если в расширении не предусмотрен другой ход событий (например, завершение работы системы).

Иногда некоторые расширения оказываются очень сложными, например, платеж по кредитной карточке. В этом случае расширение можно выделить в отдельный прецедент.

Вот пример, демонстрирующий описание сбоя при реализации расширения.

> 7б. Оплата по кредитной карточке.
> 1. Покупатель вводит информацию о своей кредитной карточке.
> 2. Система отправляет запрос на авторизацию платежа внешней системе службы авторизации платежей и запрашивает подтверждение платежа.
>
> 2а. Система определяет сбой при взаимодействии с внешней системой.
> 1. Система сигнализирует об ошибке кассиру.
> 2. Кассир просит покупателя изменить тип платежа.
>
> 3. ...

Если нужно описать условия, которые могут возникнуть в любой момент, то в обозначении пункта можно использовать символ * (см. ниже).

> *а. При каждом выходе системы из строя.
> Для ввода системы в строй и корректной обработки платежа нужно обеспечить восстановление всех транзакций и событий с любого шага сценария.
> 1. Кассир перезапускает систему, регистрируется и предлагает восстановить предыдущее состояние.
> 2. Система восстанавливает предыдущее состояние.

## Специальные требования

В этот раздел заносятся нефункциональные требования, атрибуты качества или ограничения, связанные с данным прецедентом. Сюда относятся характеристики производительности, надежности, удобства использования и конструктивные ограничения (например, на устройства ввода-вывода).

> **Специальные требования**
> - Сенсорный экран с интерфейсом пользователя для большого плоского монитора. Текст должен быть виден с расстояния один метр.
> - Отклик службы авторизации в 90% случаев приходит в течение 30 секунд.
> - Возможность локализации (представления на различных языках) отображаемого текста.
> - Возможность добавления новых бизнес-правил на шагах 3 и 7 в процессе функционирования системы.

Запись этих условий при описании прецедента — классический совет разработчиков унифицированного процесса. Однако на практике многие специалисты помещают эти требования в едином общем документе, например в дополнительной спецификации. Тогда их удобнее читать и осмысливать, поскольку обычно они рассматриваются в общем контексте в процессе анализа архитектуры.

## Список технологий и типов данных

Зачастую при реализации проекта важно не *что* сделать, а *как*. Перечень используемых технологий тоже приводится в описании прецедента. Типичным примером такой ситуации являются технические ограничения, выдвигаемые заинтересованными лицами для технологий ввода и вывода. Например, заказчик может потребовать, чтобы POS-система поддерживала ввод данных кредитной

карточки с клавиатуры и с помощью считывающего устройства. Заметим, что здесь приводятся лишь примеры проектных решений и ограничений, появляющихся на ранней стадии реализации проекта. В целом, такой конкретизации следует избегать, однако иногда она бывает неизбежна, особенно в отношении технологий ввода/вывода.

Нужно также указать возможные варианты типов данных, например, различные кодировки идентификаторов товаров. Их тоже указывают в этом разделе (см. ниже).

**Список технологий и типов данных**

3а. Идентификатор товара считывается со штрих-кода (при наличии последнего) лазерным сканером или вводится с клавиатуры.

3б. Идентификатор товара может определяться по схемам кодирования UPC, EAN, JAN или SKU.

7а. Информация об открытом кредите вводится с помощью считывающего устройства или с клавиатуры.

7б. Подпись при оплате чеком ставится на бумажном документе. Однако ожидается, что в течение двух лет большинство покупателей будут требовать цифровые устройства считывания подписи.

*Совет*

В этом разделе не должно быть много пунктов, описывающих различные действия для разных ситуаций. Если это необходимо, перенесите эти описания в раздел расширений.

## 6.8. Задачи и рамки прецедента

Как выделить прецедент? Зачастую определить правильный (а точнее, полезный) прецедент очень сложно. Каждую задачу можно рассматривать на разных уровнях детализации, начиная от конкретных простых действий и заканчивая деятельностью на уровне предприятия.

На каком же уровне детализации следует формулировать прецеденты?

В следующих разделах рассматриваются элементарные бизнес-процессы и задачи в аспекте идентификации прецедентов в приложении.

### *Прецеденты для элементарных бизнес-процессов*

Какой из следующих пунктов можно считать прецедентом?

- Переговоры с поставщиком
- Обработка возврата товара
- Регистрация

Можно сказать, что все это прецеденты на разных уровнях детализации, выделенные в зависимости от рамок системы, исполнителей и задач. Мы еще вернемся к этому вопросу после рассмотрения элементарных бизнес-процессов.

Вместо вопроса "Что такое корректный прецедент?" целесообразно задать себе вопрос: "На каком уровне следует рассматривать прецеденты в процессе анализа требований к приложению?".

*Совет*

В процессе анализа требований к компьютерному приложению следует сосредоточить внимание на уровне *элементарных бизнес-процессов* (EBP — Elementary Business Processes).

Термин EBP заимствован из области исследования бизнес-процессов[4] и определяется следующим образом.

"Элементарный бизнес-процесс — это задача, выполняемая одним человеком в одном месте в одно время в ответ на некоторое бизнес-событие, добавляющая измеряемое бизнес-значение и переводящая данные в некоторое устойчивое состояние, например, подтверждение платежа по кредитной карточке или распоряжение брокеру при изменении цен" (источник неизвестен).

Это определение можно воспринимать слишком буквально. Тогда возникает вопрос: могут ли в сценарии прецедента участвовать два человека? Скорее всего, да. Однако общая идея этого определения верна. Прецедент — это не один маленький шаг, такой, например, как удаление товара или печать документа. Основной сценарий прецедента обычно включает пять-десять шагов. Описываемый сценарий выполняется не в течение нескольких дней или сеансов, как, например, переговоры с поставщиками. Это задача, выполняемая в течение одного сеанса. Реализация прецедента может длиться от нескольких минут до нескольких дней. Как и при определении унифицированного процесса, основное внимание уделяется получению ощутимого (измеримого) результата и переходу системы и данных в устойчивое состояние.

Типичной ошибкой является рассмотрение прецедентов на слишком низком уровне, когда сценарий выполняется за один шаг и по существу является некоторой подфункцией или подзадачей в рамках элементарного бизнес-процесса.

### Обоснованные отклонения от элементарного бизнес-процесса

Несмотря на то, что основные прецеденты приложения должны соответствовать элементарным бизнес-процессам, зачастую полезно создавать отдельные прецеденты более низкого уровня, представляющие подзадачи или подпоследовательности действий в рамках основного прецедента. То есть могут существовать прецеденты, не соответствующие элементарным бизнес-процессам, а функционирующие на более низком уровне. Общая рекомендация позволяет выявить только основные прецеденты в процессе анализа требований к приложению и сконцентрировать внимание на их написании.

Например, подзадача или расширение "Оплата по кредитной карточке" может быть представлена в нескольких основных прецедентах. Поэтому ее желательно выделить в отдельный прецедент (не удовлетворяющий условиям элементарного бизнес-процесса) и связать с несколькими основными прецедентами, чтобы избежать дублирования информации.

Вопросы взаимосвязи прецедентов рассматриваются в главе 25.

---

[4] Термин EBF аналогичен термину *пользовательская задача* (user task), применяемому в области изучения удобства использования. Однако в той предметной области его значение является менее жестким.

### *Прецеденты и задачи*

Исполнители имеют свои задачи (или потребности), для решения которых они используют систему. Поэтому прецеденты уровня EBF еще называют прецедентами уровня *задач пользователя* (user goal). Это делается для того, чтобы обратить внимание на реализацию потребностей пользователей системы или основного исполнителя.

Отсюда следует алгоритм выделения прецедентов.

1. Выделить задачи (цели) пользователей.
2. Определить для каждой из них отдельный прецедент.

При таком подходе несколько смещаются акценты аналитиков. Вместо вопроса "Каковы прецеденты для данной системы?" возникает вопрос "Каковы задачи исполнителей?". Чтобы отобразить эту взаимосвязь, имя прецедента должно соответствовать названию задачи. Например, задаче электронного оформления продажи должен соответствовать прецедент Оформление продажи.

Заметим, что такая симметрия позволяет оценить адекватность уровня выделения прецедентов и задач пользователя в соответствии с правилом выделения элементарных бизнес-процессов.

Таким образом, ключевая идея состоит в том, чтобы для выделения прецедентов исследовать задачи исполнителей.

> Допустим, мы собрались на семинаре для формулировки требований. На этом семинаре основной вопрос можно формулировать двумя способами.
>
> - Что делает система?
> - Каковы задачи исполнителей?
>
> Ответы на первый вопрос, скорее, отражают текущие решения и процедуры, а также сложность этих процедур.
>
> Ответы на второй вопрос, особенно в сочетании с исследованием целей более высокого уровня ("в чем задача этой задачи?"), открывают в*и*дение новых и более эффективных решений, акцентируют внимание на получении ощутимого результата и позволяют глубже понять основные потребности заинтересованных лиц в контексте обсуждаемой системы.

### *Пример: использование рекомендаций в соответствии с EBF*

Допустим, вы являетесь системным аналитиком и отвечаете за формулировку требований к системе NextGen. Для этого вам нужно исследовать задачи пользователей. На семинаре по формулировке требований может состояться такой диалог.

> **Системный аналитик**: "Каковы ваши задачи в контексте использования POS-системы?"
>
> **Кассир**: "Во-первых, быстро зарегистрироваться. Во-вторых, оформлять продажи."
>
> **Системный аналитик**: "Какая задача более высокого уровня приводит, на ваш взгляд, к необходимости выделения отдельной задачи регистрации?"
>
> **Кассир**: "Мне необходимо "представиться" системе, а она должна проверить, имею ли я право ею пользоваться."

**Системный аналитик:** "Какова еще более глобальная задача?"
**Кассир:** "Предотвратить утечку или повреждение данных."

Обратите внимание на стратегию аналитика выстроить иерархию целей и выявить таким образом нужный уровень для элементарного бизнес-процесса. Это позволяет лучше понять мотивацию действий исполнителей и их задачи.

Предотвращение утечки данных — это цель более высокого уровня, чем задача пользователя. Поэтому пока мы ее рассматривать не будем, хотя она является чрезвычайно важной для данной системы. (Самым кардинальным решением этой задачи является полный отказ от POS-системы и услуг кассира.)

Цель следующего уровня иерархии ("представиться" системе и выполнить аутентификацию) несколько ближе к задачам пользователя. Но относится ли она к уровню элементарных бизнес-процессов? Ее решение не добавляет ощутимого результата или измеримого бизнес-значения. Если на вопрос о том, чем кассир занимался сегодня на работе, прозвучит ответ: "Я 20 раз зарегистрировался!", то вряд ли такой ответ устроит начальника. Значит, это второстепенная задача, которая служит достижению важной цели, но не относится к уровню ЕВР. Уровню ЕВР точнее всего соответствует задача оформления продажи.

В качестве другого примера можно рассмотреть процесс регистрации выручки, когда кассир задвигает ящик с наличностью и закрывает его в системе, регистрируется и вводит в систему сумму выручки. `Регистрация выручки` — это прецедент уровня ЕВР (или уровня задач пользователя), а регистрация в системе — это один из его шагов, выполняющий вспомогательную задачу, а не отдельный прецедент.

### *Вспомогательные задачи и прецеденты*

Несмотря на то, что задача "представиться" системе и выполнить аутентификацию (или задача регистрации) не была отнесена к уровню задач пользователя, она все же является целью, но на более низком уровне. Такие задачи называют *вспомогательными* (subfunctional goal), поскольку они призваны обеспечивать выполнение задач пользователя. Для вспомогательных задач отдельные прецеденты создаются очень редко, хотя специалисты по написанию прецедентов зачастую рекомендуют улучшать (обычно упрощать) набор прецедентов.

Для вспомогательных задач писать прецеденты не запрещается, однако это не всегда нужно, поскольку при этом усложняется модель прецедентов. Количество вспомогательных задач для системы может исчисляться сотнями, но стоит ли создавать так много прецедентов?

Важно понимать, что с увеличением числа прецедентов возрастает сложность задачи формулировки и управления требованиями, а значит, увеличивается также время решения этой задачи.

Основным мотивом написания прецедента для вспомогательной задачи должна служить повторяемость этой задачи или ее важное значение как предусловия для множества других прецедентов. Этому принципу удовлетворяет задача авторизации, которая обеспечивает предусловие для большинства, если не всех остальных прецедентов уровня задач пользователя.

Таким образом, вполне логично создать прецедент `Аутентификация пользователя`.

### *Составные задачи и прецеденты*

Задачи могут принадлежать к различным уровням сложности и являться составными, начиная от уровня предприятия ("быть прибыльным"), до задач среднего уровня ("регистрация торговых операций") и вспомогательных задач в рамках приложения ("проверка правильности ввода").

Аналогично, и прецеденты могут относиться к различным уровням сложности и состоять из прецедентов более низкого уровня.

| Наличие нескольких уровней сложности задач и прецедентов может ввести в заблуждение при определении соответствующего уровня для основных прецедентов. Для отсеивания слишком детальной информации следует использовать принцип нахождения элементарных бизнес-процессов. |
|---|

## 6.9. Определение основных исполнителей, задач и прецедентов

Прецеденты предназначены для удовлетворения потребностей основных исполнителей. Поэтому для выделения прецедентов используется следующая процедура.

1. Определите рамки системы: является ли она программным приложением, аппаратно-программным комплексом, включает ли в себя своих пользователей или всю организацию?
2. Идентифицируйте основных исполнителей, потребности (цели) которых удовлетворяются с помощью системы.
3. Для каждого исполнителя определите его задачи. Составьте иерархию в соответствии с рекомендациями по выделению EBP.
4. Определите прецеденты, удовлетворяющие потребности каждого исполнителя, и присвойте им имена в соответствии с задачами. Обычно основные прецеденты соответствуют задачам пользователей, за одним исключением, о котором речь пойдет ниже.

### *Шаг 1. Определение рамок системы*

Для данного прецедента разрабатываемой системой является сама POS-система. Все, что находится за ее пределами, включая кассира, службу авторизации платежей и т.д., в эти рамки не включается.

Для определения рамок системы следует, в первую очередь, указать, что к ней не относится, т.е. определить внешних основных и вспомогательных исполнителей. После идентификации внешних исполнителей рамки системы очерчиваются более четко. Например, возлагается ли на систему полная ответственность за авторизацию платежей? Нет, эту задачу выполняет внешний исполнитель — служба авторизации платежей.

### *Шаги 2 и 3. Определение основных исполнителей и задач*

Нельзя однозначно указать последовательность определения исполнителей и задач. Обычно на семинаре по определению требований методом мозгового штурма идентифицируются и те и другие артефакты. Иногда исполнители определяются после формулировки задач, а иногда наоборот.

> *Совет*
>
> В процессе мозгового штурма основное внимание следует уделить определению основных исполнителей, поскольку это расширит возможности для дальнейшего исследования.

### Вопросы, задаваемые при определении исполнителей и задач

При определении основных исполнителей и задач пользователей следует ответить на следующие вопросы, чтобы не упустить из виду некоторые неочевидные моменты.

Кто запускает и выключает систему?

Кто является системным администратором?

Кто осуществляет управление пользователями и безопасностью?

Относится ли время к числу исполнителей, другими словами, должна ли система выполнять какие-либо действия в ответ на события времени?

Существует ли процесс мониторинга, благодаря которому система перезапускается в случае сбоя?

Кто контролирует деятельность и производительность системы?

Как выполняется обновление программного обеспечения?

Кто анализирует журналы регистрации? Можно ли обеспечить удаленный доступ к ним?

### Основные и вспомогательные исполнители

Напомним, что основные исполнители — это те, чьи потребности удовлетворяются с помощью системы. Для решения своих задач они используют систему. В отличие от них, *вспомогательные исполнители* (supporting actor) занимаются обслуживанием системы. Пока сосредоточимся на идентификации основных исполнителей.

Напомним также, что основными исполнителями, среди прочего, могут быть другие компьютерные системы.

> *Совет*
>
> Отсутствие внешних компьютерных систем среди основных исполнителей должно насторожить разработчиков.

### Перечень исполнителей и их задач

Составьте список основных исполнителей и их задач. В терминах артефактов унифицированного процесса этот список должен быть разделом артефакта "Видение", описываемого в следующей главе. Рассмотрим следующую таблицу.

| Исполнитель | Задачи | Исполнитель | Задачи |
|---|---|---|---|
| Кассир | Оформляет продажи<br>Оформляет кредиты<br>Выполняет возврат товара<br>Регистрирует выручку<br>... | Системный администратор | Добавляет пользователей<br>Изменяет параметры пользователей<br>Удаляет пользователей<br>Управляет безопасностью<br>Управляет системными таблицами |

| Исполнитель | Задачи | Исполнитель | Задачи |
|---|---|---|---|
| Менеджер | Включает систему<br>Выключает систему<br>... | Система анализа торговой деятельности | Анализирует информацию о продажах и оценивает производительность |
| ... | ... | ... | ... |

Система анализа торговой деятельности (Sales Activity System) — это удаленное приложение, которое достаточно часто будет запрашивать данные от каждого узла POS-системы по сети.

***Вопросы планирования проекта***

На практике в этот список добавляются дополнительные столбцы с информацией о приоритетах, затратах и рисках. Эти вопросы кратко обсуждаются в главе 36.

***"Суровая действительность"***

Этот перечень выглядит достаточно просто и понятно. Однако на самом деле создать его довольно сложно. На это уходит масса усилий и требуется несколько интенсивных семинаров, организованных по методу мозгового штурма. Вернемся к приведенному выше примеру, иллюстрирующему применение правила EBP к задаче регистрации в системе. На семинаре, посвященном составлению этого списка, кассир может предложить рассматривать регистрацию как одну из задач уровня пользователя. Системный аналитик, углубляясь в проблему, может расширить эту задачу до уровня "идентификации и аутентификации" пользователя. Затем он может осознать, что эта задача не относится к уровню EBP и исключить ее из списка задач пользователя. На самом деле, в реальности все будет выглядеть несколько иначе, поскольку опытный аналитик, как правило, знает набор эвристик, сформулированных на основе его прежнего опыта, одна из которых сводится к следующему: регистрация пользователя редко относится к уровню задач EBP. Поэтому он достаточно быстро исключит ее из рассмотрения.

Основной исполнитель и задачи системы зависят от ее рамок

Почему основным исполнителем для прецедента `Оформление продажи` является кассир, а не покупатель? Почему покупатель не включен в список исполнителей?

Ответ определяется рамками разрабатываемой системы, как показано на рис. 6.1. Если предприятие или торговую организацию рассматривать как агрегатную систему, то для нее основным исполнителем должен являться покупатель, задача которого — приобретение товаров или услуг. Однако с точки зрения самой POS-системы (которая определяет рамки системы для данного прецедента), основным исполнителем является кассир, задача которого — обслуживание продаж.

## Определение исполнителей и задач путем анализа событий

Для определения исполнителей, их задач и прецедентов можно также использовать внешние события. Что это означает? Зачастую к одному и тому же уровню EBP или прецеденту, например, относится целая группа событий.

| Внешнее событие | Инициатор | Задача |
|---|---|---|
| Ввод информации о наименовании товара | Кассир | Оформить продажу |
| Ввод информации о платеже | Кассир или покупатель | Оформить продажу |
| ... | ... | ... |

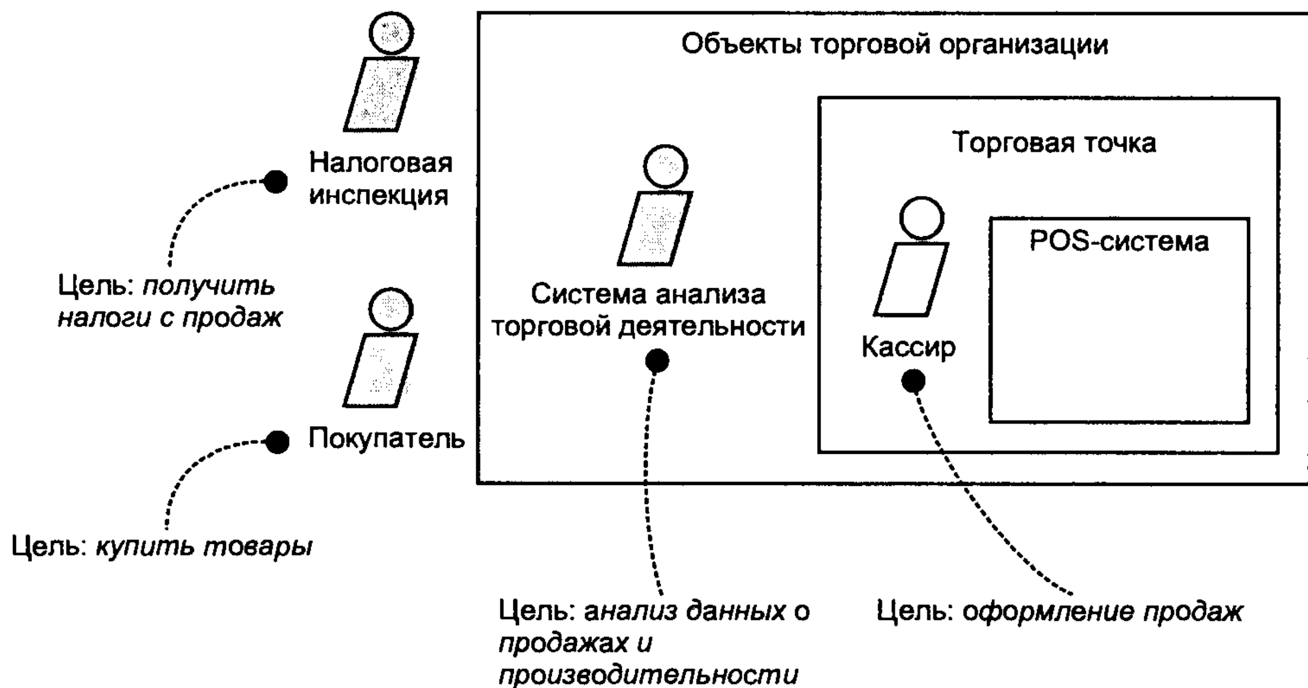



*Рис. 6.1. Основные исполнители и их задачи при изменении рамок системы*

### Шаг 4. Определение прецедентов

Как правило, каждой задаче пользователя соответствует один прецедент уровня EBP. Его имя должно соответствовать названию задачи, например, задаче оформления продажи должен соответствовать прецедент Оформление продажи.

> Как правило, имя прецедента начинается с существительного, описывающего действие.

Типичным исключением из правила соответствия задач и прецедентов является прецедент, решающий четыре задачи — создание, восстановление, обновление и удаление. Обычно такой прецедент называется Управление <*чем-либо*>. Например, задачи "изменение информации о пользователях", "удаление пользователей" и т.д. решаются в рамках прецедента Управление пользователями.

Определение прецедентов выполняется в несколько этапов, одни из которых занимают несколько минут (например, присвоение имен прецедентам), а другие — по несколько дней или недель (развернутое описание). В последующих разделах этой главы, посвященных унифицированному процессу, эти этапы будут рассмотрены в контексте итеративной разработки.

## 6.10. Поздравляем: прецеденты описаны плохо

### Необходимость обсуждений и совместных усилий

Для описания прецедентов для POS-системы NextGen в течение нескольких итераций созывались семинары по определению требований, постепенно количество прецедентов увеличивалось, требования уточнялись и дорабатывались с учетом обратной связи. В процессе описания прецедентов активно участвовали эксперты предметной области, кассиры и программисты. Между ними не было никаких посредников, все общение происходило напрямую.

Хорошо, но не очень. Спецификация требований создает лишь иллюзию корректности. Можно с уверенностью утверждать, что прецеденты и другие требования в этом документе не корректны. В описании не хватает важной информации и содержатся неверные утверждения. Для решения этой проблемы мы не призываем вас следовать рекомендациям однопроходного процесса разработки и приложить больше усилий к полному описанию требований на начальных этапах разработки, хотя, безусловно, эту работу нужно выполнить максимально добросовестно. Однако этого не достаточно.

Здесь требуется другой подход. В основном проблема решается за счет итеративности процесса разработки, но в дополнение к нему иногда требуется *постоянное личное общение* — ежедневное обсуждение между всеми разработчиками при участии специалиста по предметной области, который уполномочен принимать решения о требованиях к системе. Нужен некто, к кому программисты могут подойти и за несколько секунд снять возникшие у них вопросы. Например, в рамках подхода XP [11] существует отличный принцип: пользователь должен постоянно находиться среди участников проекта, в том же помещении.

## 6.11. Описание прецедентов, относящихся к интерфейсу пользователя, в свободном стиле

### *Новая и важная информация! Отпечатки пальцев*

Исследование целей, а не обязанностей и процедур позволяет сосредоточить внимание на основных требованиях. Например, на одном из семинаров кассир может сказать, что одной из его целей является регистрация в системе. При этом он может иметь в виду элементы интерфейса пользователя, соответствующие диалоговые окна, ввод идентификатора и пароля. Однако все это — механизмы достижения цели, а не сама цель. Изучая иерархию целей (отвечая на вопрос "какова цель этой задачи?"), системный аналитик приходит к формулировке, независимой от механизма реализации: "идентифицировать себя и выполнить аутентификацию", или на более высоком уровне — "предотвратить утечку информации".

Такой процесс исследования позволяет получить новые и более эффективные решения. Например, в настоящее время достаточно распространены и недорого стоят клавиатура и мышь с устройствами считывания биометрической информации, в частности отпечатков пальцев. Если целью является идентификация и аутентификация, то почему бы для ее достижения не использовать эффективное и быстрое средство считывания биометрических данных с клавиатуры? Однако отвечая на этот вопрос, следует принимать во внимание удобство использования. В данном случае придется установить профили типичных пользователей. А если их пальцы чем-то испачканы? А если они травмированы?

### *Базовый стиль описания*

Эта идея была сформулирована в различных рекомендациях типа "не уделяйте внимания вопросам интерфейса пользователя, сосредоточьте внимание на содержательной стороне вопроса" [33]. Подобные идеи и предложения наиболее полно были изучены Лерри Константином (Larry Constantine) в контексте создания наиболее удачного интерфейса пользователя и исследования удобства ис-

пользования программных продуктов [28, 36]. Если описание не содержит подробной информации о реализации пользовательского интерфейса, а основное внимание в нем сосредоточено на содержательных моментах, то такой стиль описания Константин называет *базовым* (essential).[5]

Базовый стиль описания предполагает изложение на уровне *намерений* пользователя и *обязанностей* системы, а не на уровне их конкретных действий. При таком стиле описания не нужно углубляться в детали технологии и механизма реализации, особенно при рассмотрении вопросов, связанных с интерфейсом пользователя.

> Описывайте прецеденты в базовом стиле. Не уделяйте внимания интерфейсу пользователя, а сосредоточьтесь на содержательной стороне вопроса.

Все приведенные выше примеры прецедентов были выдержаны в базовом стиле описания, в том числе прецедент Оформление продажи.

Заметим, что понятия *цель* и *намерение* являются синонимами [84]. Этим подтверждается взаимосвязь между предложенным Константином *базовым* стилем описания и ориентацией *на цели* (задачи), предлагаемой выше в данной главе. Действительно, многие намерения исполнителей можно трактовать как вспомогательные цели.

## Контрпримеры

### Базовый стиль

Допустим, идентификация и аутентификация выполняются в рамках прецедента Управление пользователями. Базовый стиль описания предполагает оформление описания в виде двух колонок. Однако можно, например, описывать все действия и в одной колонке.

...

| **Намерение исполнителя** | **Отклик системы** |
|---|---|
| 1. Администратор идентифицирует себя. | 2. Выполняет аутентификацию. |
| 3. ... | |

При оформлении в виде одной колонки это описание будет иметь следующий вид.

...
1. Администратор идентифицирует себя.
2. Система выполняет аутентификацию.
3. ...

Для реализации этих намерений и обязанностей можно использовать широкий спектр проектных решений: устройства считывания биометрической информации, графический интерфейс пользователя и т.д.

### Конкретный стиль — избегайте его на ранних этапах формулировки требований

Существует и другой стиль описания прецедентов — *конкретный* (concrete). При таком стиле описания проектные решения, относящиеся к пользователь-

---

[5] Этот термин происходит от термина “базовая модель” (essential model) из области системного анализа [83].

скому интерфейсу, внедряются в описание прецедента. В тексте описания могут, например, даже содержаться копии экранов, описываться элементы управления и другие элементы пользовательского интерфейса.

...
1. Администратор вводит идентификатор и пароль в диалоговом окне.
2. Система аутентифицирует администратора.
3. Система отображает окно **Изменение пользователей**.
4. ...

Такое конкретное описание потребуется на следующих этапах проектирования GUI, а не стадии анализа требований. В процессе формулировки требований "не уделяйте внимания вопросам интерфейса пользователя, сосредоточьте внимание на содержательной стороне вопроса".

## 6.12. Исполнители

Исполнитель (actor) — это сущность, обладающая поведением. К числу исполнителей может относиться и сама рассматриваемая система, если она вызывает службы других систем.[6] В прецеденте могут участвовать основные и вспомогательные (второстепенные) исполнители. Исполнителями являются не только люди, но и организации, машины и программы. Существует три типа внешних по отношению к разрабатываемой системе исполнителей.

- **Основной исполнитель** (primary actor) — его задачи выполняются с использованием системы. Примером основного исполнителя является кассир.
  - Зачем его идентифицировать? Чтобы определить цели пользователя, на основе которых формулируются прецеденты.
- **Вспомогательный исполнитель** (supporting actor) — обслуживает систему (например, предоставляет информацию). Примером вспомогательного исполнителя является служба авторизации платежей.
  - Зачем его идентифицировать? Чтобы определить внешние интерфейсы и протоколы.
- **Закулисный исполнитель** (offstage actor) — заинтересован в реализации прецедента, но не является основным или вспомогательным исполнителем. Примером закулисного исполнителя является налоговая служба.
  - Зачем его идентифицировать? Чтобы удостовериться, что *все* интересы определены и удовлетворены. Интересы закулисных исполнителей обычно не очевидны и их легко упустить из виду, если не идентифицировать их в явной форме.

## 6.13. Диаграммы прецедентов

В языке UML существует система обозначений для диаграммы прецедентов, иллюстрирующей имена прецедентов, исполнителей и взаимосвязи между ними (рис. 6.2).

---

[6] Это модифицированное и усовершенствованное определение исполнителя, основанное на определении, принятом в более ранних версиях UML и UP [32]. В прежней версии этого определения система никогда не включалась в число исполнителей. Все сущности, включая разрабатываемую систему, могут играть различные роли.

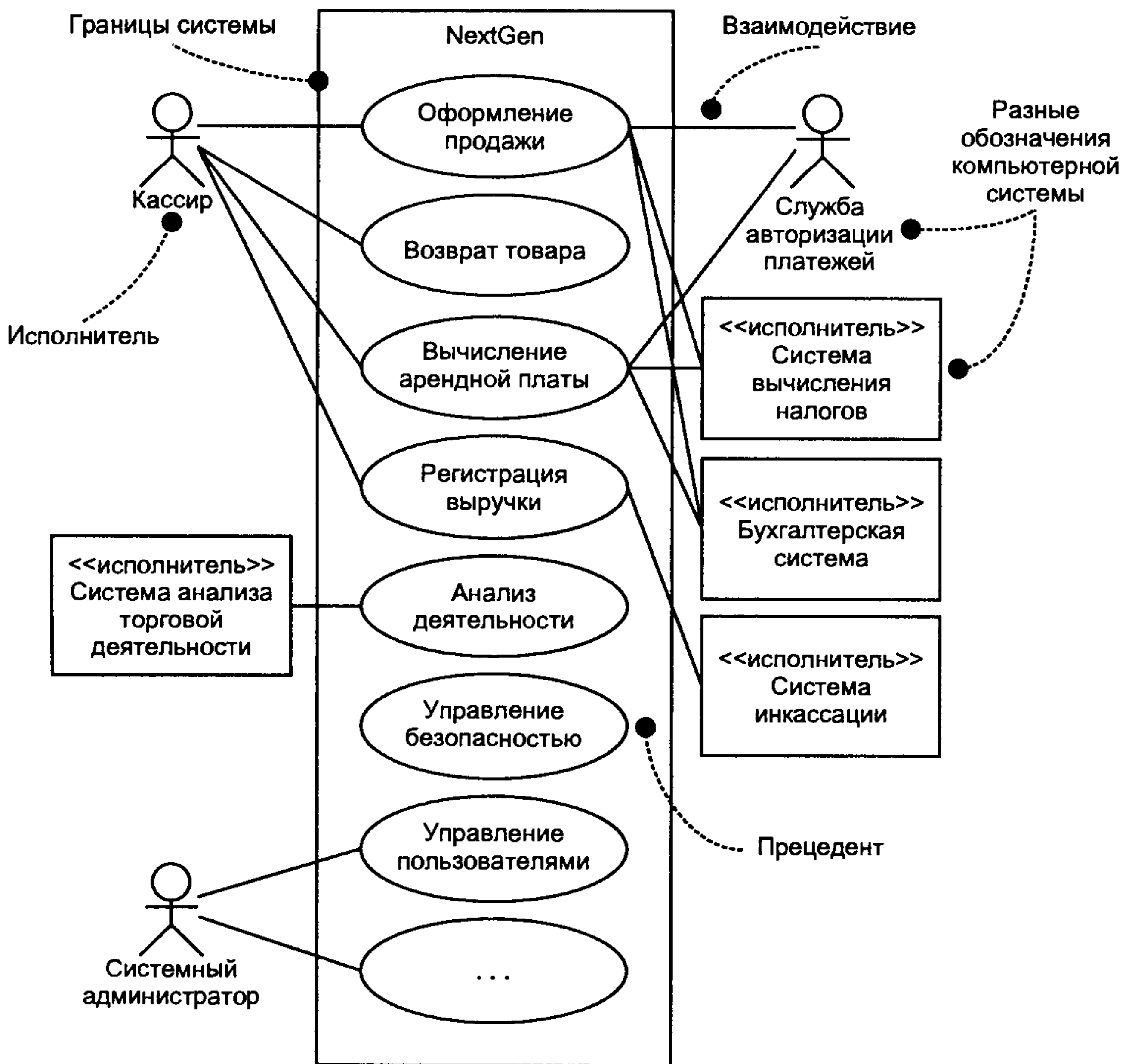



*Рис. 6.2. Фрагмент диаграммы прецедентов*

> Диаграммы прецедентов и их взаимосвязей имеют второстепенное значение при работе над прецедентами. Прецеденты — это текстовые документы. Работать над прецедентами — значит составлять текстовые описания.

Обычно новички в области моделирования прецедентов начинают с составления диаграмм прецедентов и их взаимоотношений, а не с составления текстовых описаний. Однако специалисты по написанию прецедентов мирового класса, к числу которых относятся Андерсон (Anderson), Фовлер (Fowler), Кокбурн (Cockburn) и др., основное внимание уделяют составлению текстовых описаний, а не диаграмм. В таком ракурсе диаграмма лишь иллюстрирует способы использования системы внешними исполнителями.

> *Совет*
>
> Стройте простые диаграммы прецедентов в соответствии со списком исполнителей и их задач.

Диаграмма прецедентов — это отличное изображение системного контекста, поскольку она отображает границы системы, внешние для системы понятия и способы использования системы. Она подытоживает поведение системы и ее исполнителей. Фрагмент простой диаграммы прецедентов для системы NextGen показан выше на рис. 6.2.

## Система обозначений для диаграммы прецедентов

На рис. 6.3 показаны некоторые обозначения для диаграммы прецедентов. Обратите внимание на блок с надписью <<исполнитель>>. Так в UML обозначается *стереотип* (stereotype) — механизм выделения категорий элементов. Имя стереотипа заключается в двойные угловые кавычки.

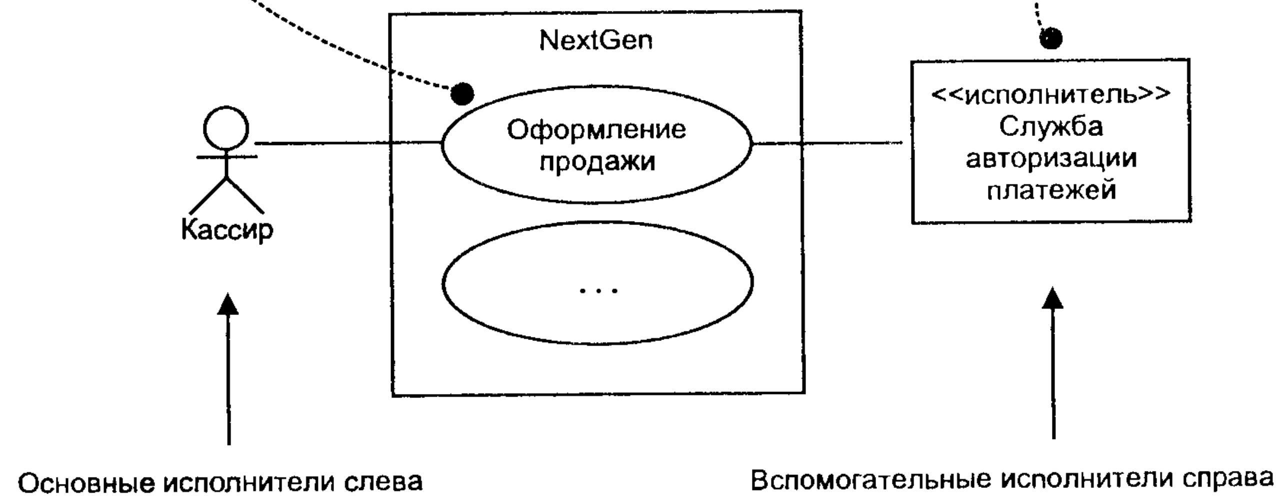



*Рис. 6.3. Предлагаемые обозначения*

Заметим, что некоторые предпочитают выделять внешних исполнителей иначе, чем показано на рис. 6.4.

## Предупреждение

Еще раз повторим, что основное внимание при работе над прецедентами следует уделять написанию текстовых документов, а не составлению диаграмм. Если в организации слишком много времени (несколько часов, а то и дней) уделяется составлению диаграммы прецедентов и обсуждению их взаимосвязей, то из виду будут упущены другие более важные моменты.

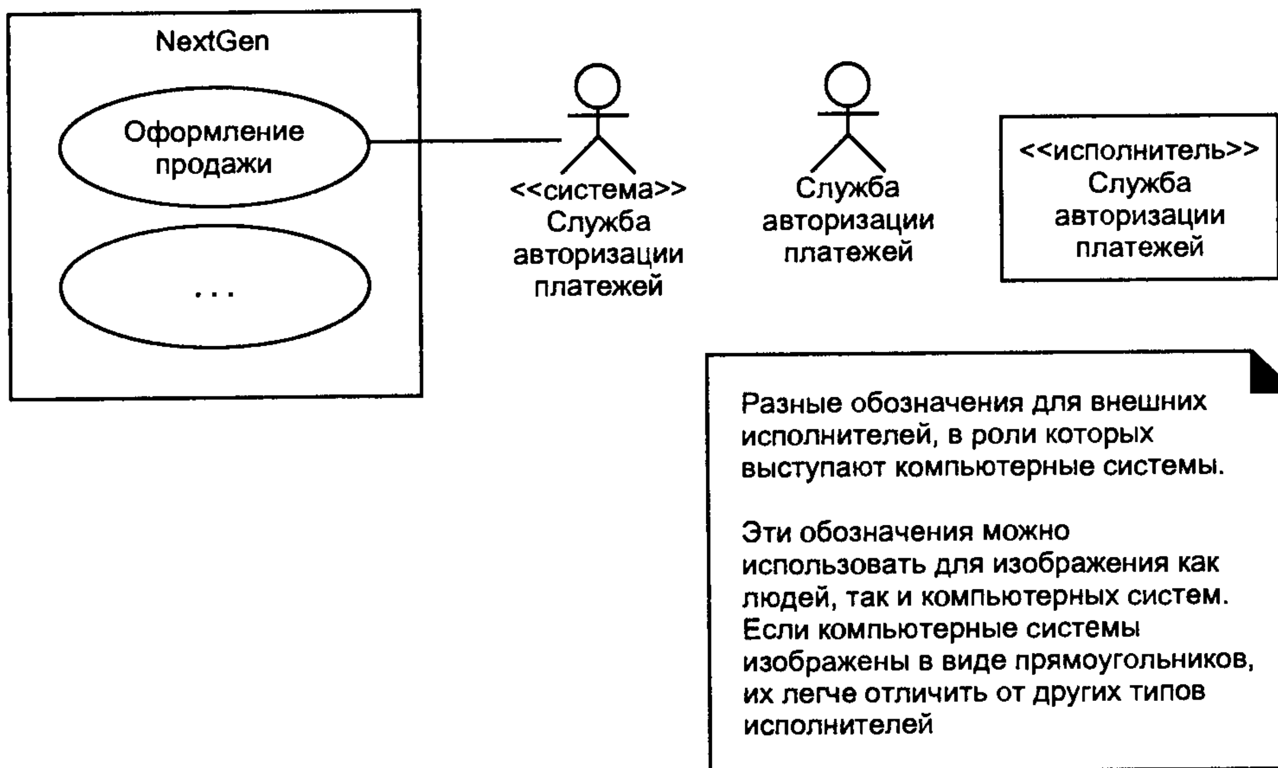



*Рис. 6.4. Другое обозначение исполнителей*

## 6.14. Требования и списки низкоуровневых свойств

Как следует из названия книги [54], основным мотивом описания прецедентов является определение требований в контексте задач (целей) и сценариев использования системы. Это очень хорошо, поскольку позволяет углубить понимание системы. Однако прецеденты — это не единственные артефакты формулировки требований. Некоторые нефункциональные требования, правила и другие элементы лучше описать в дополнительной спецификации, рассмотрению которой будет посвящена следующая глава.

Основная идея — заменить детальные списки низкоуровневых свойств (которые обычно составлялись при использовании традиционного метода формулировки требований) прецедентами (за некоторыми исключениями). Эти списки были сгруппированы по функциональному назначению и выглядели примерно следующим образом.

| Идентификатор | Свойство |
|---|---|
| Свойство 1.9 | Система будет позволять вводить новые идентификаторы |
| ... | ... |
| Свойство 2.4 | Система будет регистрировать платежи по кредитной карточке в бухгалтерской системе |

Такие детальные списки тоже могут оказаться полезными. Однако один подобный список занимает не полстраницы, а десятки или сотни страниц. Это вызывает некоторые проблемы, решить которые позволяют прецеденты. К числу проблем относятся следующие.

- Длинные, подробные списки функций не отражают требований в общем контексте. Различные функции и свойства организованы в обычный список

разрозненных элементов. А при описании прецедентов требования рассматриваются в контексте задач и целей системы.

- Если используются и описания прецедентов и списки функций, то происходит дублирование информации. На это затрачивается больше труда, требуется больше усилий для их осмысления и согласования.

> *Совет*
>
> Старайтесь заменить подробные списки низкоуровневых свойств описанием прецедентов.

### Списки высокоуровневых свойств системы вполне допустимы

Достаточно часто функциональные свойства системы описываются в кратких списках свойств высокого уровня в рамках документа "В*и*дение". Это вполне оправдано. В отличие от списка низкоуровневых свойств, перечисленных на 100 страницах, список основных свойств системы должен включать не более нескольких десятков элементов. В нем перечислены лишь функциональные свойства системы, не упомянутые при описании прецедентов.

**Список свойств системы**

- Регистрация продаж.
- Авторизация платежей (кредитных, дебитных, чековых).
- Системное администрирование для управления пользователями, безопасностью, таблицами кодов и констант и т.п.
- Автоматическая обработка информации о продажах в фоновом режиме в случае отказа внешних компонентов.
- Взаимодействие в реальном времени с внешними системами (на основе промышленных стандартов), включая систему складского учета, вычисления налоговых отчислений, бухгалтерскую систему, систему управления человеческими ресурсами и службы авторизации платежей.
- Определение и выполнение настраиваемых бизнес-правил в фиксированных точках сценария.
- ...

Такие списки будут рассматриваться в следующей главе.

### Когда нужен подробный список свойств?

Иногда прецедентов действительно недостаточно. Для некоторых систем нужно разработать детальный список свойств. Например, серверы приложений, системы управления базами данных и другие базовые системы нужно рассматривать в терминах *свойств* ("Необходима поддержка следующей версии XML"). Прецеденты не годятся для разработки таких приложений.

## 6.15. Объектно-ориентированный подход и прецеденты

При описании прецедентов объектно-ориентированный подход не используется. Это не недостаток, а просто констатация факта. Действительно, прецеденты — это широко применяемое средство анализа требований, которое можно использовать и в не объектно-ориентированных проектах. Необходимость в описании прецедентов возникает на этапе анализа требований. Однако, как будет видно из дальнейшего, прецеденты вносят основной вклад в классические виды деятельности ООА/П.

## 6.16. Прецеденты в рамках унифицированного процесса

Прецеденты играют жизненно важную роль при реализации унифицированного процесса, поскольку вся разработка в рамках этого подхода осуществляется *под управлением прецедентов* (use-case driven development). Это означает следующее.

- Требования в основном формулируются при описании прецедентов (в модели прецедентов). Остальные требования (если таковые существуют) являются либо техническими (например, список функций), либо второстепенными.
- Прецеденты — важный этап итеративного планирования. На каждой итерации реализуются некоторые сценарии или целые прецеденты. Поэтому описания прецедентов вносят существенный вклад в оценивание результата.
- Разработка приложения состоит в реализации прецедентов. То есть группа разработчиков продумывает способы взаимодействия объектов или архитектуру подсистем для реализации прецедентов.

В рамках UP выделяют два вида прецедентов: системные и бизнес-прецеденты. *Системные прецеденты* (system use-case) — это такие, которые рассматривались в этой главе, например `Оформление продажи`. Они создаются в рамках дисциплины "Требования" и являются частью модели прецедентов.

*Бизнес-прецеденты* (business use-case) используются гораздо реже. При необходимости они создаются в рамках дисциплины "Бизнес-моделирование" как часть крупномасштабного бизнес-процесса или для облегчения понимания контекста новой системы. Они описывают последовательность действий в целом, выполняемых *бизнес-исполнителем* (business actor) (исполнителем в бизнес-среде, например, потребителем). В частности, для ресторана можно выделить бизнес-прецедент `Приготовление блюда`.

### *Прецеденты и спецификация требований в рамках итеративного процесса*

В этом разделе мы вернемся к основной идее UP и итеративной разработки. Рассмотрим временные рамки, выделяемые на формулировку требований на каждой итерации. В табл. 6.1 представлен пример (не претендующий на рекомендацию) определения требований в процессе реализации стратегии UP.

Обратите внимание, что группа технической разработки приступает к созданию ядра системы в тот момент, когда детализированы лишь 10% требований, далее следует намеренная задержка, и формулировка требований откладывается до конца первой итерации стадии развития.

В этом состоит ключевое отличие от однопроходного процесса — промышленная разработка ядра системы начинается достаточно быстро, задолго до полного завершения формулировки требований.

Обратите внимание, что в конце первой итерации стадии развития проводится второй семинар по формулировке требований, по окончании которого 30% прецедентов описаны подробно. При таком планировании можно учесть обратную связь и оценки реализованной части системы. Обратная связь поступает от пользователей, специалистов по тестированию и позволяет "познать непознанное". То есть в процессе разработки системы очень быстро возникают вопросы, требующие немедленного решения.

**Таблица 6.1. Пример формулировки требований на начальных итерациях разработки проекта**

| Дисциплина | Артефакт | Комментарии по поводу формулировки требований | | | | |
|---|---|---|---|---|---|---|
| | | Начало 1 неделя | Развитие 1 4 недели | Развитие 2 4 недели | Развитие 3 3 недели | Развитие 4 3 недели |
| Требования | Модель прецедентов | Двухдневный семинар по формулировке требований. Для большинства прецедентов определяются имена и создаются краткие описания. Только 10% прецедентов описываются подробно | В конце этой итерации проводится двухдневный семинар по формулировке требований. При этом учитываются результаты выполненной работы, 30% прецедентов описываются подробно | В конце этой итерации проводится двухдневный семинар по формулировке требований. При этом учитываются результаты выполненной работы, 50% прецедентов описываются подробно | То же самое, 70% прецедентов описываются подробно | 80-90% прецедентов описываются подробно. Только небольшая часть прецедентов реализуется на стадии развития, реализация остальных переносится на стадию конструирования |
| Проектирование | Модель проектирования | | Проектирование для небольшого числа архитектурно важных требований с высокой степенью риска | То же самое | То же самое | То же самое. Все архитектурно важные элементы с высокой степенью риска должны быть спроектированы к этому моменту |
| Реализация | Модель реализации (код) | | Реализация проектного решения | То же самое. 5% окончательной системы построено | То же самое. 10% окончательной системы построено | То же самое. 15% окончательной системы построено |
| Управление проектом | План разработки | Достаточно общая оценка требуемых затрат и ресурсов | Оценки начинают конкретизироваться | Несколько больше... | Несколько больше... | Общая длительность проекта, основные этапы, затраты и ресурсы оценены достаточно точно |

## *Временные рамки создания артефактов UP*

В табл. 6.2 приводятся некоторые артефакты UP и примерный график их создания. К разработке модели прецедентов нужно приступать на начальной стадии, при этом 10% прецедентов желательно описать подробно. Основная часть прецедентов постепенно описывается на стадии развития (в этот же период готовится дополнительная спецификация), поэтому к завершению этой фазы обычно формируется реалистичное представление о сроках реализации проекта.

**Таблица 6.2. Пример планирования сроков реализации артефактов UP (н — начало, р — развитие)**

| Дисциплина | Артефакт<br>Итерация→ | Начало<br>I1 | Развитие<br>E1..En | Конструирование<br>C1..Cn | Передача<br>T1..Tn |
|---|---|---|---|---|---|
| Бизнес-моделирование | Модель предметной области | | н | | |
| Требования | Модель прецедентов | н | р | | |
| | Видение системы | н | р | | |
| | Дополнительная спецификация | н | р | | |
| | Словарь терминов | н | р | | |
| Проектирование | Модель проектирования | | н | р | |
| | Описание архитектуры | | н | | |
| | Модель данных | | н | р | |
| Реализация | Модель реализации | | н | р | р |
| Управление проектом | План разработки | н | р | р | р |
| Тестирование | Модель тестирования | | н | р | |
| Окружение | Набор документов | н | р | | |

### *Прецеденты начальной фазы*

Рассмотрим подробнее табл. 6.1.

На начальной стадии не все прецеденты описываются детально. В этот период для описания прецедентов целесообразно организовать двухдневный семинар. В первой половине первого дня можно определить задачи и круг заинтересованных лиц, а также обсудить, какие задачи относятся к проекту, а какие выходят за его рамки. С помощью компьютерного проектора можно составить и демонстрировать таблицу исполнителей и их задач. Можно приступить к построению диаграммы прецедентов. В течение нескольких часов вполне реально определить порядка 20 задач пользователей (и соответствующих им прецедентов), в том числе прецеденты `Оформление продажи`, `Возврат товара` и т.д. За это время большинство интересных, сложных прецедентов с высоким уровнем риска должны быть описаны в сжатом формате. На описание каждого такого прецедента понадобится около 2 минут. Разработчики могут приступить к составлению высокоуровневой схемы функциональности системы.

После этого 10–20% прецедентов, представляющих сложные функции или высокую степень риска, нужно описать подробно. При этом разработчики лучше оценят сложность проекта, его рамки, выявят проблемы, возникающие в процессе разработки. Самыми важными прецедентами являются, пожалуй, `Оформление продажи` и `Возврат товара`.

Для описания прецедентов нужно использовать средства управления требованиями со встроенным текстовым процессором, а результаты работы желательно демонстрировать участникам семинара с помощью проектора. Для этих прецедентов нужно составить список заинтересованных лиц и их интересов. Это поможет глубже понять неочевидные функциональные и основные нефункциональные требования, в частности к надежности и производительности системы.

Задачей такого анализа является не исчерпывающее описание прецедентов, а более глубокое понимание проблемы.

Спонсоры проекта должны решить, стоит ли вкладывать деньги в дальнейшие исследования (на стадии развития). На начальной стадии не ставится задача напрямую исследовать целесообразность капиталовложений. На этом этапе нужно определить масштаб проекта, основные риски, реалистичность этого проекта — т.е. получить исчерпывающую информацию для принятия последующих решений о продолжении или прекращении проекта.

Предположим, начальная фаза проекта NextGen заняла 5 дней. В результате двухдневного семинара и последующей работы по анализу прецедентов было принято решение о целесообразности проекта.

### Прецеденты на стадии развития

Рассмотрим следующие столбцы табл. 6.1.

Фаза развития включает несколько итераций фиксированной длительности (скажем, четыре недели), в течение которых реализуются наиболее рискованные, значимые и архитектурно важные части системы, а также идентифицируется и проясняется бо́льшая часть требований. Благодаря наличию обратной связи разработчики лучше понимают требования, постепенно уточняют и конкретизируют их. На каждой итерации желательно провести двухдневный семинар. Однако на каждом семинаре не нужно обсуждать все прецеденты. Их нужно упорядочить по приоритетности и сначала рассматривать самые важные прецеденты.

На каждом следующем семинаре нужно дорабатывать и уточнять основные требования. На начальных итерациях процесс изменения требований может происходить достаточно интенсивно. Однако в дальнейшем он стабилизируется. В итеративном процессе происходит постепенная доработка наиболее важных частей системы.

На каждом семинаре список задач пользователей и прецедентов постепенно уточняется. По окончании фазы развития большинство прецедентов должны быть описаны или переписаны в развернутом формате (примерно 80–90%). Если для POS-системы выделено 20 прецедентов, то как минимум 15 наиболее сложных и рискованных из них должны быть определены полностью.

Заметим, что на стадии развития разработка части системы доводится до программной реализации. Поэтому по окончании этой фазы команда разработчиков системы NextGen будет иметь не только достаточно полно определенные прецеденты, но и качественный исполняемый код.

### Прецеденты на стадии конструирования

Этап конструирования состоит из некоторого числа итераций фиксированной длительности (например, 20 итераций по 2 недели каждая), в течение которых основное внимание уделяется доработке системы, поскольку наиболее принципиальные вопросы уже решены на стадии развития. На этом этапе тоже некоторое внимание уделяется описанию прецедентов, однако значительно меньшее, чем на стадии развития. К этому моменту бо́льшая часть функциональных и нефункциональных требований должна быть постепенно стабилизирована. Это не означает необходимости замораживания требований и завершения исследований. Однако теперь степень модификации требований от итерации к итерации значительно снижается.

## 6.17. Пример: прецеденты начальной фазы проекта NextGen

Как видно из предыдущих разделов, на начальной стадии не все прецеденты описываются детально. Модель прецедентов на этом этапе может быть разработана на следующем уровне детализации.

| В развернутом формате | В свободном формате | Краткое описание |
|---|---|---|
| Оформление продажи<br>Возврат товара | Вычисление арендной платы<br>Анализ торговой деятельности<br>Управление безопасностью<br>... | Регистрация выручки<br>Управление пользователями<br>Запуск системы<br>Завершение работы<br>Управление системными таблицами<br>... |

## 6.18. Дополнительная литература

Наиболее популярным руководством по описанию прецедентов является книга [33], переведенная на несколько языков. Эта книга настолько популярна, что ее можно поставить первой в списке рекомендованной литературы. Данная глава основывается на основных положениях этой книги. Предупреждение: пусть вас не смущает пристрастие автора к использованию пиктограмм для прецедентов разного уровня и слишком пристальное внимание к выделению уровней и систематизации прецедентов. Пиктограммы имеют лишь второстепенное значение. И хотя для новичков в области описания прецедентов обсуждение уровней и задач может показаться отступлением от темы, более опытные специалисты знают, что эти вопросы играют очень важную роль, поскольку их недопонимание может привести к усложнению моделирования прецедентов.

Самой популярной статьей в Internet, посвященной описанию прецедентов, является работа [32], которую можно найти по адресу www.usecases.org.

Еще одной полезной книгой является [54]. Ее основным "лейтмотивом" (как следует из самого названия) является мысль о том, что прецеденты — это не дополнительный артефакт дисциплины определения требований, а основная "движущая сила" процесса анализа требований.

Стоит упомянуть еще одну ценную книгу [101], написанную опытным инструктором по описанию прецедентов и практическим специалистом, постигшем на своем опыте премудрости применения прецедентов в итеративном жизненном цикле.

## 6.19. Артефакты UP в контексте процесса

Как видно из рис. 6.5, прецеденты связаны со многими артефактами UP.

В рамках UP работа над прецедентами относится к дисциплине определения требований. На рис. 6.6 предложены некоторые рекомендации относительно места и времени для проведения семинаров по определению требований.

*Рис. 6.5. Пример взаимосвязи артефактов UP*

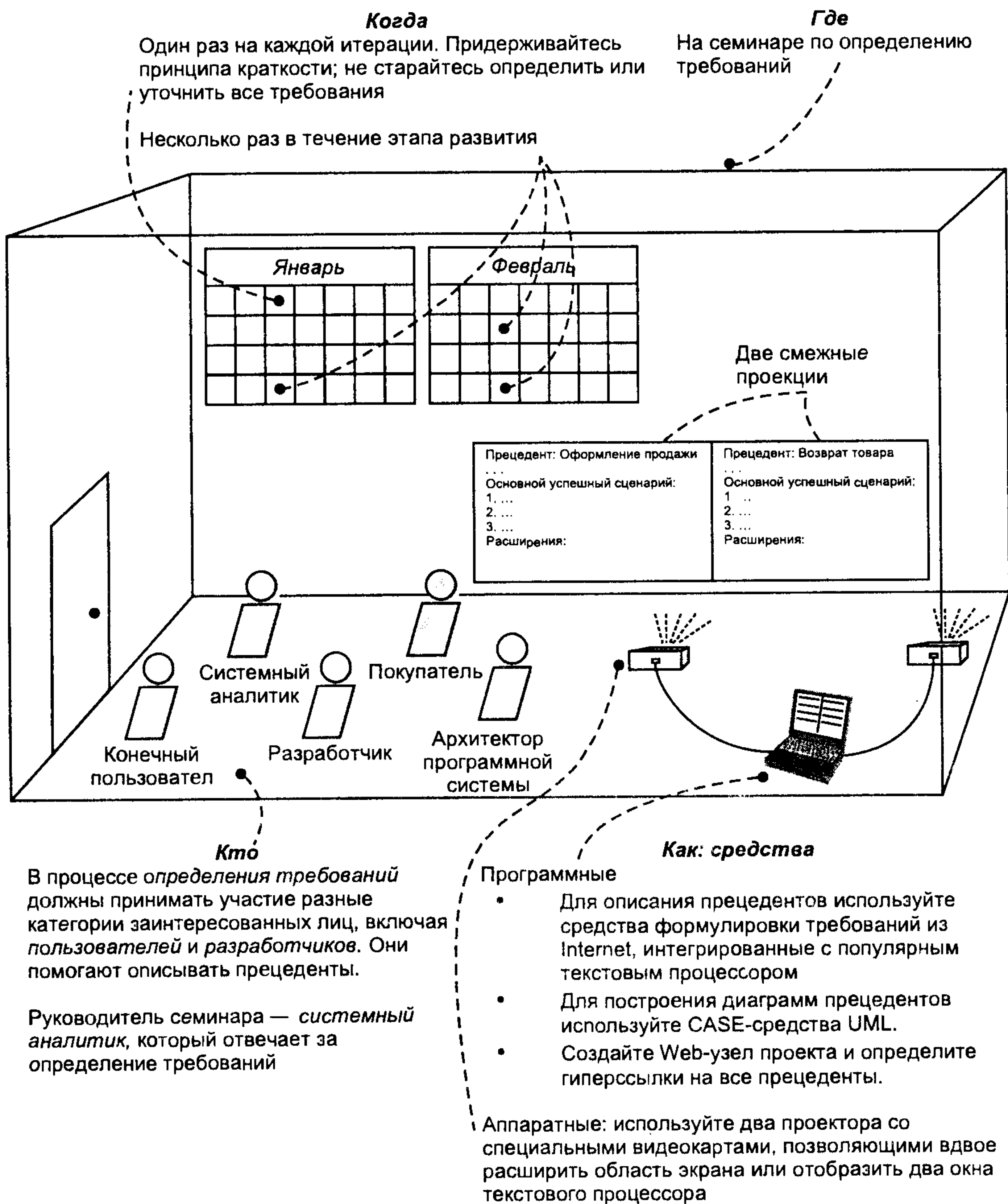



*Рис. 6.6. Рекомендации по организации семинаров*

Глава

# Определение остальных требований

*Когда не хватает идей, на помощь приходят слова.*

*Иоган Вольфганг Гете (Johann Wolfgang von Goethe)*

---

**Основные задачи**

- **Разработать дополнительную спецификацию, словарь терминов и документ "Видение".**
- **Сопоставить свойства системы с прецедентами.**
- **Связать видение системы с другими артефактами и итеративной разработкой.**
- **Определить атрибуты качества.**

---

## Введение

Для определения требований одного описания прецедентов недостаточно. Необходимо определить и другие виды требований, в частности, соглашения о лицензировании, возможностях поддержки системы и т.д. Все эти требования описываются в *дополнительной спецификации* (supplementary specification).

В *словарь терминов* (glossary) включаются термины и определения. Он также может служить словарем данных.

Документ "Видение" определяет видение проекта. В нем кратко описываются основные цели проекта, проблемы, указывается круг заинтересованных лиц, их потребности, а также основные идеи предложенного решения.

Приведем цитату.

> "Документ "Видение" определяет точку зрения заинтересованных лиц на разрабатываемый продукт в терминах их основных потребностей и свойств продукта. Этот документ содержит описание неочевидных основных требований и составляет основу для более детального описания технических требований" [97].

## 7.1. Примеры для системы NextGen

Целью последующих примеров является не подробное составление документов "Видение", "Словарь терминов" и "Дополнительная спецификация".[1] Задачей этой книги является изучение объектного проектирования, анализа требований в контексте описания прецедентов и объектно-ориентированного анализа, а не изучение проблематики POS-систем или разработка конкретной системы. Поэтому основной пример этой книги будет упомянут только в некоторых разделах, чтобы сохранить последовательность изложения материала, выделить основные моменты и обеспечить быстрое продвижение вперед.

## 7.2. Пример NextGen: фрагмент дополнительной спецификации

**Дополнительная спецификация**

**Даты внесения изменений**

| Версия | Дата | Описание | Автор |
|---|---|---|---|
| Черновой начальный вариант | 10 января, 2031 | Первый черновой вариант. Будет уточнен на стадии развития | Крэг Ларман |

**Введение**

В этом документе описаны все требования к POS-системе NextGen, не вошедшие в описание прецедентов.

**Функциональность**

(*Имеющая отношение ко многим прецедентам*)

***Регистрация событий и обработка ошибок***

Все ошибки регистрируются на постоянном носителе.

***Подключаемые бизнес-правила***

Необходимо обеспечить возможность настройки функциональности системы в различных точках сценариев нескольких прецедентов (эти точки нужно определить) на основе заданных правил.

***Безопасность***

Необходимо выполнять аутентификацию всех пользователей.

**Удобство использования**

***Человеческие факторы***

Пользователь POS-системы будет работать с большим монитором, поэтому необходимо следующее.

- Текст должен быть виден с расстояния 1м
- Нужно избегать мерцающих цветов

Быстрая, простая и корректная обработка информации — вот главные принципы системы автоматизации торговли, поскольку пользователь хочет поскорее совершить покупку. В противном случае ему не понравится этот магазин (и продавец).

Кассир зачастую смотрит не на экран компьютера, а на покупателя или товары. Поэтому предупреждающие сообщения нужно сопровождать звуковыми сигналами, а не только графически отображать на экране.

---

[1] Для нас важно не определить все требования, а *описать* процесс их определения.

### Надежность

#### *Возможность восстановления информации*

При сбоях в работе внешних систем (службы авторизации платежей, бухгалтерской системы и т.д.) нужно обеспечить возможность локальной обработки данных (их сохранение и последующую передачу внешним системам). Этот вопрос требует более детальной проработки.

### Производительность

Как указывалось выше, покупатель хочет сделать покупку *как можно скорее*. Задержка этого процесса может быть связана с внешней службой авторизации. Наша задача — выполнить авторизацию не более чем за 1 минуту в 90% случаев.

### Возможности поддержки

#### *Адаптация системы*

Различные пользователи POS-системы NextGen могут устанавливать свои бизнес-правила для обработки данных о продажах. Поэтому в нескольких заранее определенных точках сценария (например, при инициализации новой продажи или при добавлении нового наименования товара) нужно обеспечить возможность подключения бизнес-правил.

#### *Конфигурирование*

Сетевые конфигурации различных пользователей POS-системы могут отличаться. Могут использоваться архитектуры "тонкого" и "толстого" клиентов, двухуровневые и многоуровневые архитектуры и т.д. Кроме того, конфигурация ресурсов каждого клиента может изменяться со временем, отражая производственные потребности и потребности в производительности. Следовательно, система должна быть настраиваемой и отражать потребности пользователей. Этот вопрос требует тщательной дополнительной проработки, изучения степени гибкости и способов ее достижения.

### Ограничения

Руководство проекта NextGen настаивает на применении технологии Java, поскольку это улучшит возможности по поддержке системы и ее переходу на различные платформы, а также обеспечит простоту разработки.

### Приобретаемые компоненты

- Система вычисления налоговых платежей. Разрабатываемая система должна поддерживать работу с подключаемыми внешними системами различных стандартов для разных стран.

### Бесплатные компоненты на основе открытого кода

В целом, рекомендуется максимальное использование в этом проекте компонентов на основе открытого кода в рамках Java-технологии.

Несмотря на то, что пока преждевременно определять проектные решения, предлагаются следующие варианты.

- Контур регистрации Jlog
- ...

### Интерфейсы

#### *Важные интерфейсы и аппаратные средства*

- Сенсорный монитор (воспринимаемый операционной системой как обычный монитор; прикосновения обрабатываются как события мыши).
- Лазерный сканер для считывания штрих-кодов (обычно подключаемый к специальной клавиатуре; считанный код обрабатывается как последовательность нажатия клавиш).
- Устройство для печати чеков.
- Устройство считывания данных с кредитной/дебитной карточки.
- Устройство считывания подписи (не в первой версии системы).

### *Программные интерфейсы*

Для большинства внешних систем (вычисления налоговых платежей, системы складского учета, бухгалтерской системы и т.д.) необходимо обеспечить возможность подключения через различные интерфейсы.

### Бизнес-правила

| Имя | Правило | Возможность изменения | Источник |
|---|---|---|---|
| ПРАВ1 | Для платежей по кредитной карточке требуется подпись | Подпись покупателя будет нужна, но через 2 года большинство пользователей захотят применять цифровое устройство для ввода подписи, а через пять лет может понадобиться поддержка уникальной цифровой закодированной подписи, введенной в настоящее время в США | Политика практически всех служб авторизации платежей |
| ПРАВ2 | Правила вычисления налоговых платежей. С продаж отчисляются налоги. Правила налогообложения изложены в официальных документах | Высокая вероятность изменения. Законы налогообложения изменяются ежегодно на правительственном уровне | Закон |
| ПРАВ3 | При возврате товара, купленного по кредитной карточке, возврат денег осуществляется не наличными, а путем перевода на кредитную карточку | Низкая вероятность изменения | Политика компаний авторизации платежей |
| ПРАВ4 | Правила вычисления скидок (примеры).<br>Работник компании — скидка 20%.<br>Привилегированный покупатель — 10% | Высокая вероятность изменения. Каждая торговая организация устанавливает свои скидки | Политика торговых организаций |
| ПРАВ5 | Правила вычисления торговых скидок (на уровне транзакций). Применяются к общей стоимости покупки (без вычета налога). Примеры.<br>10% общей стоимости покупки, если стоимость превышает 100%.<br>5% по понедельникам.<br>10% для всех продаж с 10 до 15 часов ежедневно.<br>50% для всех продаж с 9 до 10 часов сегодня | Высокая вероятность изменения. В каждой торговой организации используются свои правила, которые могут изменяться ежедневно и ежечасно | Политика торговых организаций |
| ПРАВ6 | Правила вычисления торговых скидок на товары (на уровне наименований товаров). Примеры:<br>10% на все тракторы на этой неделе<br>При покупке двух единиц товара — третья бесплатно | Высокая вероятность изменения. В каждой торговой организации используются свои правила, которые могут изменяться ежедневно и ежечасно | Политика торговых организаций |

### Вопросы законодательства

Рекомендуется использование бесплатных компонентов на основе открытого кода, если их разрешено использовать в коммерческих программных продуктах.

Необходимо учитывать все необходимые налоги. Правила налогообложения могут изменяться достаточно часто.

### Информация из предметной области

#### *Ценовая политика*

Помимо описанных выше правил ценообразования, каждому товару могут соответствовать две цены: *исходная* и *постоянная сниженная цена*. Указанная цена товара (без вычисления положенных скидок) соответствует постоянной сниженной цене, если таковая имеется. Однако организации сохраняют исходную цену даже при наличии постоянной сниженной для бухгалтерской отчетности.

#### *Обработка платежей по кредитной и дебитной карточке*

Если электронные платежи по кредитной или дебитной карточке подтверждены службой авторизации платежей, то за них несет ответственность эта служба, а не сам покупатель. Следовательно, для каждого платежа продавец должен зафиксировать получение денежных сумм от службы авторизации. Обычно перевод денег за все выполненные в текущий день платежи на счет торговой организации выполняется в ночное время, при этом за каждую транзакцию служба взимает небольшой взнос.

#### *Вычисление налогов*

Налоги могут вычисляться по сложным схемам, и суммы отчислений могут часто изменяться на правительственном уровне. Поэтому желательно возложить задачу вычисления налоговых платежей на отдельную программу. Налоги могут взиматься городскими, региональными властями или на уровне государства. Некоторые ставки налогов могут зависеть от категории покупателя или назначения товара (например, товары для детей или фермеров).

#### *Идентификаторы товаров (UPC, EAN, SKU, штрих-коды и сканеры)*

Система NextGen должна поддерживать различные типы идентификаторов товаров. Наиболее популярными системами идентификации товаров являются UPC (Universal Product Codes), EAN (European Article Numbering) и SKU (Stock Keeping Units). Японская система JAN (Japanese Article Number) является вариантом европейской версии EAN.

В системе SKU идентификаторы товаров назначаются произвольным образом торговыми организациями.

Системы UPC и EAN имеют свои стандарты и регулируемые компоненты. Хорошее описание этих систем можно найти по адресам `www.adams1.com/pub/rus-sadam/upccode.html`, `www.uc-council.org` и `www.ean-int.org`.

## 7.3. Дополнительная спецификация (комментарии)

В дополнительной спецификации содержатся требования, ограничения и другая информация, не вошедшая в описание прецедентов или словарь терминов, включая атрибуты качества и специальные требования. Заметим, что требования, связанные с прецедентами, должны быть представлены в описаниях прецедентов в разделе “Специальные требования”, однако некоторые предпочитают также включать их в дополнительную спецификацию. В дополнительную спецификацию можно включать следующие элементы.

- Требования согласно модели FURPS+ — функциональные, требования к удобству использования, надежности, производительности и возможности поддержки.
- Отчеты.
- Ограничения на аппаратные и программные средства (операционные и сетевые системы и т.д.).
- Ограничения, накладываемые на процесс разработки (например, процесс или средства разработки).
- Другие ограничения проектирования или реализации.

- Международные соглашения (единицы измерения, языки и т.д.).
- Документация (пользовательская, руководство по установке и администрированию) и справочная информация.
- Соглашения о лицензировании или другие юридические соглашения.
- Разбиение на пакеты.
- Стандарты (технические, обеспечения качества и безопасности).
- Физические требования к окружению (например, температурный режим эксплуатации или ограничения на вибрацию).
- Операционные требования (например, способ обработки ошибок, частота архивации).
- Информация из предметной области (например, о полном цикле обработки платежа по кредитной карточке).

*Ограничения* (constraints) относятся не к поведению системы, а к проектированию. Они тоже являются требованиями, но название указывает на их ограничительный характер. Приведем следующий пример.

> Необходимо использовать продукты Oracle (поскольку с этой компанией существует лицензионное соглашение).
>
> Система должна работать в операционной системе Linux (она бесплатна).

> *Совет*
>
> Не стоит принимать проектные решения и устанавливать ограничения на ранних этапах разработки, особенно на начальной стадии, когда имеется слишком мало информации о системе. Однако некоторые ограничения неизбежны, например, интерфейсы с внешними системами или юридические ограничения.

### Атрибуты качества

Некоторые требования называются *атрибутами качества* (quality attributes) системы [9]. К ним относятся удобство использования, надежность и т.д. Заметим, что это качественные характеристики системы, однако не все они должны обеспечивать высокое качество соответствующего процесса (слово "качество" имеет два разных значения). Например, возможности поддержки (качественная характеристика) могут быть низкими, если данный программный продукт не предназначен для долгосрочной эксплуатации.

Атрибуты качества делятся на два типа.

1. Наблюдаемые в процессе работы системы (функциональность, удобство использования, надежность, производительность и т.д.).
2. Ненаблюдаемые в процессе работы системы (возможность поддержки, удобство тестирования и т.д.).

Функциональность системы определяется в описании прецедентов (например, требования к производительности указываются при описании прецедента `Оформление продажи`).

Другие атрибуты качества, согласно модели FURPS+, определяются в дополнительной спецификации.

Несмотря на то, что функциональность системы относится к качественным характеристикам, ее зачастую не включают в число атрибутов качества. Однако термин "атрибуты качества" не является синонимом термина "нефункциональные требования". Последнее понятие гораздо шире, поскольку включает в себя все характеристики, за исключением функциональности (например, лицензионные соглашения или соглашения о пакетировании).

Атрибуты качества (а значит, и дополнительная спецификация, в которой они описываются) относятся к сфере интересов системного архитектора, поскольку (как будет видно из главы 32) архитектурный анализ и проектирование подразумевают идентификацию и обоснование атрибутов качества в контексте функциональных требований. Например, предположим, что одним из атрибутов качества системы NextGen должна быть ее абсолютная надежность при сбоях внешних систем. С точки зрения архитектора, это требование накладывает отпечаток на крупномасштабные проектные решения.

Все атрибуты качества взаимосвязаны. В качестве простого примера можно рассмотреть требования "высокой надежности" и "простоты тестирования". Эти требования несколько противоречивы, поскольку для распределенной системы существует множество причин и способов выхода из строя.

## Бизнес-правила

Бизнес-правила или правила предметной области [54, 94] определяют процессы в предметной области. Это требования не к конкретному приложению, а ко всем приложениям для данной предметной области. Типичными бизнес-правилами являются политика компаний, а также физические или государственные законы.

Бизнес-правила влияют на другие требования к системе, обычно определяемые прецедентами, поскольку они проясняют многие неясные при описании прецедентов моменты. Например, если при описании прецедента `Оформление продажи` кому-то взбредет в голову осуществлять платежи по кредитной карточке без подписи покупателя, то такая возможность будет пресечена наличием бизнес-правила, учитывающего требования служб авторизации платежей (ПРАВ1).

> *Предупреждение*
>
> Правила — это не требования к приложению. Не включайте в список правил свойства системы. Правила описывают процессы из предметной области, а также их ограничения, а не свойства приложения.

## Информация из предметной области

Зачастую очень важно внести в дополнительную спецификацию некоторую информацию из предметной области, имеющую отношение к разрабатываемой системе (продажам и бухгалтерскому учету, геофизических свойствах потоков нефти, воды или газа и т.д.) и позволяющую разработчикам глубже понять происходящие процессы. В этот раздел можно включать ссылки на литературу, мнения экспертов, формулы, законы и другие ссылки. Например, в раздел информации из предметной области можно включить начальные сведения о схемах кодирования UPC и EAN а также штрих-кодах, необходимые команде разработчиков системы NextGen.

# 7.4. Пример для системы NextGen: видение (фрагмент)

**Видение**

**Даты внесения изменений**

| Версия | Дата | Описание | Автор |
|---|---|---|---|
| Черновой начальный вариант | 10 января, 2031 | Первый черновой вариант. Будет уточнен на стадии развития | Крэг Ларман |
| | | | |

**Введение**

Нам видится надежное приложение автоматизации розничной торговли следующего поколения (POS-система NextGen), обеспечивающее гибкую поддержку различных бизнес-правил, механизмы поддержки различных терминалов и интерфейсов пользователя, а также интеграцию с различными внешними вспомогательными системами.

*Анализ в этом примере носит иллюстративный характер.*

**Позиционирование**

***Экономические предпосылки***

Существующие программные продукты не обеспечивают настройку на потребности различных пользователей, в частности добавление различных бизнес-правил или поддержку разных сетевых архитектур (например, на основе "толстого" или "тонкого" клиента, двух-, трех- или четырехуровневые архитектуры). Кроме того, они плохо масштабируются. Ни одна из известных систем не обеспечивает автоматический переход из интерактивного в автономный режим при сбоях внешних систем. Отсутствует простая возможность интеграции с внешними системами. Существующие системы не поддерживают новые терминальные технологии. Негибкость существующих систем открывает новую нишу на рынке программного обеспечения POS-систем.

***Формулировка проблемы***

Традиционные POS-системы не обладают гибкостью, неустойчивы к сбоям и не обеспечивают интеграцию с внешними системами. Это приводит к проблемам с оформлением продаж, несоответствию программного обеспечения экономическим потребностям предприятий, невозможности точной и своевременной обработки данных и поддержки планирования. Эти проблемы касаются кассиров, менеджеров по продажам, системных администраторов и руководителей предприятий.

***Место системы***

Указать, для кого предназначена система, описать ее свойства и отличия от продуктов конкурирующих организаций.

**Заинтересованные лица**

*Необходимо определить, для кого предназначена система и каковы проблемы заинтересованных лиц.*

***Демографические особенности рынка...***

***Заинтересованные лица, не являющиеся пользователями системы...***

***Пользователи системы...***

***Основные задачи высокого уровня и проблемы заинтересованных лиц***

*Необходимо объединить информацию из списка исполнителей и задач, а также из раздела описания прецедентов, отражающего потребности заинтересованных лиц.*

Однодневный семинар по определению требований с приглашением специалистов по предметной области и заинтересованных лиц позволит выделить следующие основные цели и проблемы.

| Цель высокого уровня | Приоритет | Проблемы и замечания | Текущие решения |
|---|---|---|---|
| Быстрая, робастная и интегрированная обработка информации о продажах | Высокий | С увеличением нагрузки скорость падает. При выходе из строя компонентов невозможно обрабатывать информацию о продажах.<br>Не хватает свежей и точной информации от бухгалтерской и других систем из-за отсутствия интеграции с существующими бухгалтерскими системами, системами складского учета и т.д. Усложняет анализ и планирование.<br>Невозможно настраивать бизнес-правила с учетом особых требований.<br>Сложно добавлять новые типы терминалов или интерфейсы пользователей | Существующие POS-продукты обеспечивают базовую обработку информации о продажах, но не решают все возникающие проблемы |
| ... | ... | ... | ... |

### *Задачи уровня пользователя*

*Сюда можно включить список исполнителей и их задач, разработанный в процессе моделирования прецедентов, либо более сжатую информацию.*

Пользователи (и внешние системы) используют данную систему в таких целях.

- *Кассир.* Оформляет продажи, возврат товаров, регистрирует выручку.
- *Системный администратор.* Управляет пользователями, безопасностью и системными таблицами.
- *Менеджер.* Осуществляет запуск и завершает работу системы.
- *Система анализа торговой деятельности.* Анализирует данные о продажах.
- ...

### *Окружение...*

## Обзор

### *Перспективы продукта*

Система NextGen обычно будет устанавливаться в магазинах, при использовании мобильных терминалов они будут располагаться вблизи сети магазинов, либо внутри магазинов. Система будет обслуживать пользователей и взаимодействовать с другими системами, как показано на рис. Видение 1.

*Диаграмма строится на основе диаграммы прецедентов.*

*Контекстные диаграммы можно строить в различных форматах с разной степенью детализации, но все они отражают взаимодействие внешних исполнителей с системой.*

### *Преимущества системы*

*Подобно перечню исполнителей и их задач, в этой таблице указаны задачи, их решения и преимущества, однако на более высоком уровне, чем при описании прецедентов.*

*Здесь описывается основное значение и отличительные свойства продукта.*

| Свойство | Преимущества для заинтересованных лиц |
|---|---|
| Система будет обеспечивать всю основную функциональность, необходимую торговым организациям, включая обработку информации о продажах, авторизацию платежей, оформление возврата товаров и т.д. | Быстрая работа торговых точек в автоматическом режиме |
| Автоматическое выявление сбоев, переход в автономный режим работы | Возможность продолжения торговли при выходе из строя внешних компонентов |
| Подключаемые в различных точках сценария бизнес-правила | Гибкая настройка бизнес-логики |

| Свойство | Преимущества для заинтересованных лиц |
|---|---|
| Интерактивное взаимодействие с внешними системами на основе стандартных протоколов | Своевременное и точное оформление продаж, подготовка бухгалтерской документации и данных складского учета, поддержка планирования |
| ... | ... |

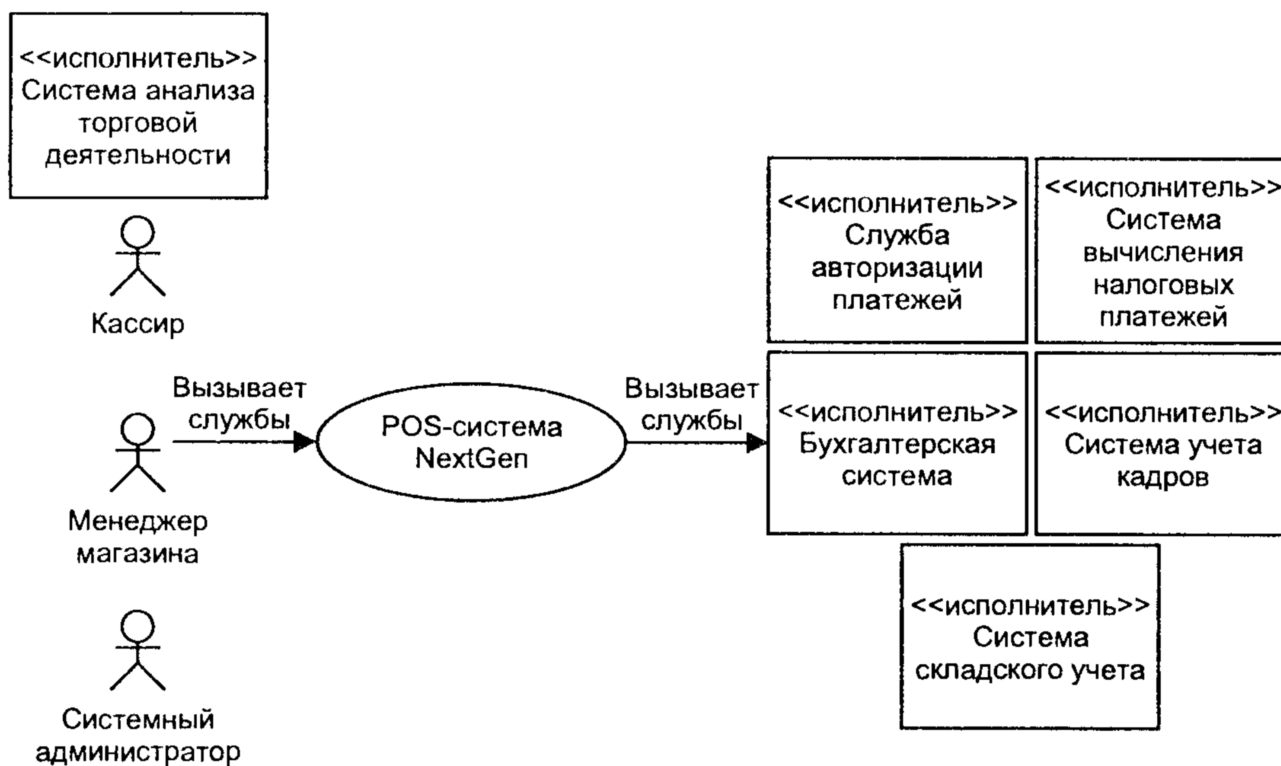



*Рис. Видение 1. Контекстная диаграмма POS-системы NextGen*

***Предположения и зависимости...***

***Стоимость и ценообразование...***

***Лицензирование и установка...***

**Основные свойства системы**

*Как было упомянуто выше, свойства системы описываются сжато путем перечисления основных функций.*

- Оформление продаж.
- Авторизация платежей (по кредитной или дебитной карточке, чеком).
- Системное администрирование и управление пользователями, безопасностью, таблицами констант и кодов и т.д.
- Автоматический переход в автономный режим работы при выходе из строя внешних систем.
- Транзакции в реальном времени на основе промышленных стандартов с внешними системами, включая бухгалтерскую систему, систему складского учета, учета человеческих ресурсов, вычисления налогов, службы авторизации платежей.
- Определение и выполнение настраиваемых бизнес-правил в фиксированных точках выполнения сценариев.
- ...

***Другие требования и ограничения***

Ограничения для процесса проектирования, удобства использования, надежности, производительности, перечень документации и т.д. описаны в дополнительной спецификации и модели прецедентов.

## 7.5. Видение (комментарий)

### *Ту ли проблему мы решаем?*

#### Формулировка проблемы

На начальной стадии разработки при определении требований необходимо кратко сформулировать суть проблемы. Это позволит избежать недопонимания между заинтересованными лицами. Зачастую разные участники проекта ставят перед собой различные задачи.

В шаблоне RUP для формулировки проблемы предлагается использовать не текстовый, а табличный формат.

| Проблема состоит в | ... |
|---|---|
| Затрагивает | ... |
| Влияет на | ... |
| Ее успешное решение основывается на | ... |

#### Основные высокоуровневые цели и потребности заинтересованных лиц

В этой таблице указываются задачи и проблемы более высокого уровня, чем при описании прецедентов, а также нефункциональные цели, затрагивающие несколько различных прецедентов, например, следующие.

- Необходимо обеспечить бесперебойное оформление продаж.
- Необходимо обеспечить возможность корректировки бизнес-правил.

#### Каковы основные проблемы и цели?

Заинтересованные лица зачастую формулируют свои потребности в форме предполагаемых решений, например: "Нам необходим штатный программист для настройки бизнес-правил при их изменении". Такие решения иногда оправданны, поскольку заинтересованные лица хорошо понимают проблемы своей предметной области и возможные пути их решения. Однако иногда возникают попытки предложить решения, которые являются неоптимальными или выходят за рамки основной проблемы.

Поэтому проблему и основные цели должен исследовать системный аналитик. Как было описано в предыдущих главах, он сможет выстроить иерархию проблем и определить приоритеты их решения.

#### Методы групповой работы

В процессе определения высокоуровневых требований и целей большое внимание уделяется групповой работе. Вот некоторые приемы групповой работы, облегчающие процесс изучения проблемы и определения основных задач: построение диаграмм Парето, многовариантное голосование, точечное голосование, номинальный групповой процесс, мозговой штурм и группировка по признаку подобия. Информацию об этих и других приемах можно почерпнуть из Internet. Автор предпочитает применять несколько перечисленных методик в процессе одного семинара для изучения общих проблем и требований с разных точек зрения.

### *Системные свойства — функциональные требования*

Прецеденты — не единственный способ выражения функциональных требований. Это объясняется несколькими причинами.

- Их описания слишком подробны. Заинтересованные лица зачастую хотят видеть краткий перечень важных функций.
- В краткий список функциональных требований можно было бы включить имена прецедентов (Оформление продажи, Возврат товара и т.д.). Однако такой список тоже может оказаться слишком длинным. Кроме того, имена прецедентов могут не отражать важные свойства, представляющие интерес для пользователей. Уровень детализации при определении прецедентов может не соответствовать уровню формулировки основных целей приложения. Например, в описание прецедента Оформление продажи может входить функция автоматической авторизации платежей. Тогда из списка имен прецедента не будет ясно, что система автоматически выполняет авторизацию платежей. Более того, некоторые стараются объединить несколько прецедентов и сформулировать таким образом одно свойство. Например, "Системное администрирование и управление пользователями, безопасностью, таблицами кодов и констант и т.д.".
- Некоторые важные функции системы легче выразить в краткой форме без привязки к именам прецедентов или задачам элементарных бизнес-процессов. Функции системы и прецеденты могут перекрываться или быть ортогональны друг другу. Например, на первом семинаре по определению требований для системы NextGen один из участников может сказать: "Система должна обеспечивать передачу данных существующим внешним системам бухгалтерского и складского учета, а также обмен данными с системой вычисления налоговых платежей". Это утверждение не относится к одному прецеденту, но достаточно четко отражает свойства системы.
    - Поясним это. Функциональность некоторых систем легче выразить через свойства системы, чем через описание прецедентов. Это относится, например, к серверам приложений, для них очень сложно сформулировать прецеденты. При обсуждении требований к следующей версии приложения один из участников семинара может внести предложение: "Следующая версия должна поддерживать компоненты EJB 2.0". Это требование сформулировано в виде свойства системы, а не в виде прецедента.

Следовательно, помимо прецедентов, функции системы можно выразить через ее свойства, а точнее *системные свойства* (system features), представляющие собой высокоуровневые, краткие утверждения, описывающие функции системы. Более строго в контексте UP системное свойство определяется как "наблюдаемая извне и обеспечиваемая системой функция, которая напрямую удовлетворяет потребности заинтересованного лица" [72].

> Свойства — это то, что может делать система. Свойства можно облечь в следующую лингвистическую форму (и использовать ее в качестве теста).
>
> *Система будет выполнять <свойство X>.*

Например, система будет выполнять авторизацию платежей.

Напомним, что документ "Видение" можно рассматривать как формальное или неформальное соглашение между разработчиками и заказчиками. Системные свойства — это механизм обобщенного выражения предназначения системы. Они дополняют описания прецедентов и выражаются в краткой форме.

Свойства отличаются от различных нефункциональных требований и ограничений, например: “Система должна работать в операционной системе Linux и поддерживать интерфейс с сенсорным экраном”. Обратите внимание, что это утверждение не удовлетворяет требованиям лингвистического теста.

Иногда бывает сложно сформулировать “наблюдаемую извне функцию”. Например, можно ли считать свойством следующее выражение.

> Система будет выполнять транзакции с внешними системами бухгалтерского и складского учета, учета человеческих ресурсов и системой вычисления налоговых платежей.

Оно описывает поведение, имеющее значение для заинтересованных лиц, но само взаимодействие может быть невидимо извне. Такое утверждение можно включить в число свойств, не особенно беспокоясь о точном соответствии определению.

И наконец, отметим, что большинство системных свойств более полно описаны в описании прецедентов.

### Обозначения и организация

В большинстве случаев краткие высокоуровневые описания играют важную роль, поскольку их можно быстро прочитать.

Каждую фразу начинать словами “Система должна выполнять...” не обязательно. Однако многие разработчики поступают именно так.

Вот пример описания свойств высокого уровня для большого проекта, одним из элементов которого является POS-система.

*К основным свойствам относятся следующие.*

- *Службы POS*
- *Управление запасами*
- *Торговля через Web*
- ...

Зачастую системные свойства организуют в двухуровневую иерархию, однако в документе “Видение” сложная иерархия приводит к необходимости излишней детализации. В этом документе описывается лишь основная функциональность, без ее представления в виде длинного списка деталей. Вот разумный пример детализации.

*К основным свойствам относятся следующие.*

- *Службы POS*
    - *Оформление продаж*
    - *Авторизация платежей*
    - ...
- *Управление запасами*
    - *Автоматическое переупорядочение*
    - ...

Иногда свойства второго уровня соответствуют прецедентам (или задачам уровня пользователя), но это не обязательно. Тем не менее, основные системные свойства подробно описываются в модели прецедентов.

Сколько системных свойств нужно включить в документ “Видение”?

> *Совет*
>
> В документ "В*и*дение" желательно включать менее 50 свойств. Если их больше, попробуйте сгруппировать некоторые из них и сформулировать на более высоком уровне абстракции.

### Другие требования в документе "Видение"

В документе "Видение" системные свойства вкратце отражают функциональные требования, которые более детально изложены в описании прецедентов. В этом документе можно также отразить другие требования (например, к надежности и удобству в использовании), которые более подробно описаны в разделе "Специальные требования" модели прецедентов и в дополнительной спецификации. Однако здесь возникает риск неоправданного дублирования. Например, в шаблонах продуктов, поддерживающих RUP, содержатся идентичные или схожие разделы требований к надежности, производительности, удобству в использовании и т.д. При таком дублировании очень сложно отслеживать изменения во всех документах. Кроме того, при этом нужен одинаковый уровень детализации дополнительной спецификации и документа "В*и*дение". То есть краткие и детальные описания требований становятся идентичными.

> *Совет*
>
> Старайтесь избегать дублирования требований в дополнительной спецификации, документе "В*и*дение" и описании прецедентов. Лучше сформулировать их в дополнительной спецификации или описании прецедентов, а в документе "В*и*дение" сделать ссылку на эти описания.

Этот совет позволит упростить разработку моделей. Однако если вы предпочитаете использовать стандартные шаблоны, то это тоже не запрещается.

### Видение, свойства или прецеденты — что раньше?

О порядке разработки артефактов говорить неуместно. Различные артефакты, относящиеся к требованиям, создаются параллельно. Тем не менее, можно порекомендовать такую последовательность разработки.

1. Создайте краткий черновой вариант документа "В*и*дение".
2. Идентифицируйте задачи пользователей и соответствующие прецеденты.
3. Опишите некоторые прецеденты и приступите к разработке дополнительной спецификации.
4. На основе полученной информации уточните документ "В*и*дение".

## 7.6. Пример системы NextGen: словарь терминов (фрагмент)

**Словарь терминов**

**Даты внесения изменений**

| Версия | Дата | Описание | Автор |
|---|---|---|---|
| Черновой начальный вариант | 10 января, 2031 | Первый черновой вариант. Будет уточнен на стадии развития | Крэг Ларман |

**Определения**

| Термин | Определение | Синоним |
| --- | --- | --- |
| Товар | Продаваемый продукт или услуга | |
| Авторизация платежа | Подтверждение гарантии оплаты от внешней службы авторизации платежей | |
| Запрос на авторизацию платежа | Набор элементов, отправляемых по электронной почте службе авторизации платежей, обычно в виде массива символов. К этим элементам относятся: идентификатор магазина, номер счета покупателя, сумма платежа и временная метка | |
| UPC | Двенадцатизначный числовой код для идентификации продукта. Обычно он представляется в виде штрих-кода. Более подробная информации содержится по адресу `http:\\www.uc-council.org` | Universal Product Code |
| ... | ... | ... |

## 7.7. Комментарии: словарь терминов

В простейшем варианте *словарь терминов* (glossary) представляет собой список важных понятий и их определение. Очень часто возникает ситуация, когда технический или другой термин используется заинтересованными лицами в несколько различных значениях. Такие противоречия необходимо разрешить для облегчения общения и однозначной формулировки требований.

> *Совет*
>
> Начинайте разработку словаря терминов на ранних этапах выполнения проекта. Мне приходилось работать в коллективах, где один и тот же термин поразному трактовался различными разработчиками.

В словарь терминов нужно вносить не все возможные понятия, а только неясные, неоднозначные или требующие дополнительного изучения, например, информацию о форматировании или правила верификации.

### Словарь терминов в роли словаря данных

В рамках UP словарь терминов также играет роль *словаря данных* (data dictionary) — документа, содержащего информацию о данных. На начальной стадии проекта словарь терминов включает лишь основные термины и их описание, а на стадии развития его можно превратить в словарь данных.

К атрибутам терминов относятся следующие.

- Синонимы
- Описание
- Формат (тип, длина, единицы измерения)
- Взаимосвязи с другими элементами
- Диапазон значений
- Правила проверки корректности

Обратите внимание, что диапазон значений и правила проверки корректности в словаре терминов соответствуют требованиям к поведению системы.

### Единицы измерения

Как отмечает Мартин Фовлер (Martin Fowler) [46], единицы измерения (тип валюты, единицы измерения физических величин) играют важную роль, особенно в современном мире интернационализации программных приложений. Например, возможно, что система NetxtGen будет продаваться в различных странах, поэтому цена товара не может быть представлена в виде числа. Она должна измеряться в определенных единицах, выбираемых в каждом случае отдельно.

### Составные термины

В словарь терминов заносятся не только простейшие понятия, такие как "цена товара". В него необходимо включать и сложные элементы, например, "продажа" (это понятие включает в себя другие элементы, такие как дата и место продажи), а также аббревиатуры, используемые для описания передачи данных между исполнителями в рамках одного прецедента. Например, рассмотрим следующий фрагмент описания прецедента `Оформление продажи`.

> Система отправляет запрос на авторизацию платежа внешней службе авторизации платежей и запрос на подтверждение платежа.

Термин "запрос на авторизацию платежа" обозначает набор данных, содержание которых необходимо пояснить в словаре терминов.

## 7.8. Надежные спецификации — нет ли здесь противоречия?

При описании требований может сложиться впечатление, что реальные требования уже хорошо определены и вполне понятны и их можно использовать для надежного планирования и объективного оценивания состояния проекта. Однако это иллюзия. Разработчики программного обеспечения из собственного опыта знают, насколько это не так. Об этом свидетельствует и высказывание Гете, выбранное в качестве эпиграфа к данной главе.

На самом деле реальное значение имеет лишь соответствие программы тестам, определенным пользователями и заинтересованными лицами, а также выполнение поставленных задач (которые зачастую нельзя четко сформулировать до начала работы с программой).

Создание дополнительной спецификации и документа "Видение" — это упражнение, позволяющее несколько прояснить поставленные задачи, обосновать назначение продукта и описать основные идеи. Однако эти документы, как и любой артефакт дисциплины определения требований, не являются надежной спецификацией. Только в процессе написания кода, его тестирования, получения обратной связи, взаимодействия с пользователями и потребителями можно составить реальное представление о сущности системы.

Это не призыв к отказу от анализа и быстрому написанию кода, а лишь рекомендации по правильному восприятию требований и их постоянной доработке.

## 7.9. Размещение артефактов на Web-узле проекта

Рассмотренные в данной книге примеры и прецеденты являются статическими, поскольку они сформулированы на бумаге. Тем не менее их желательно оформить в электронном виде и разместить на Web-узле проекта. Тогда между документами можно установить гиперссылки и добавить ссылки на другие средства. Например, словарь терминов можно разместить в таблице базы данных.

## 7.10. Не слишком ли много UML на начальной стадии проекта?

Основная задача начальной фазы — накопить достаточно информации для составления общего видения проекта, принятия решения о его достижимости и целесообразности. Поэтому на этом этапе не требуется создавать подробных диаграмм в системе обозначений языка UML за исключением простых диаграмм прецедентов. На данном этапе необходимо сосредоточить внимание на осмыслении масштаба проекта и 10% требований. Разрабатываемые документы являются текстовыми. Большая часть диаграмм приходится на следующую фазу — фазу развития.

## 7.11. Остальные артефакты дисциплины определения требований в рамках UP

Как и в предыдущей главе, в табл. 7.1 представлены примеры артефактов и время их создания. Начало разработки артефактов, связанных с определением требований, приходится на начальную фазу и продолжается на стадии развития.

**Таблица 7.1. Пример планирования сроков реализации артефактов UP (н — начало, р — развитие)**

| Дисциплина | Артефакт<br>Итерация→ | Начало<br>I1 | Развитие<br>E1..En | Конструирование<br>C1..Cn | Передача<br>T1..Tn |
|---|---|---|---|---|---|
| Бизнес-моделирование | Модель предметной области | | н | | |
| Требования | Модель прецедентов | н | р | | |
| | **Видение системы** | н | р | | |
| | **Дополнительная спецификация** | н | р | | |
| | **Словарь терминов** | н | р | | |
| Проектирование | Модель проектирования | | н | р | |
| | Описание архитектуры | | н | | |
| | Модель данных | | н | р | |
| Реализация | Модель реализации | | н | р | р |
| Управление проектом | План разработки | н | р | р | р |
| Тестирование | Модель тестирования | | н | р | |
| Окружение | Набор документов | н | р | | |

### *Начальная фаза*

Разработка артефактов определения требований не завершается на начальной стадии реализации проекта. Она только начинается.

Заинтересованные лица должны решить, стоит ли проводить серьезные исследования, т.е. приступать к этапу развития. На начальной стадии в документе "Видение" описываются идеи, позволяющие принять решение о целесообразности продолжения работы над проектом.

Поскольку основная работа по определению требований приходится на фазу развития, то на начальном этапе дополнительная спецификация только начинает разрабатываться. В ней освещаются важные качественные показатели (например, возможность восстановления данных системы при выходе из строя внешних служб), определяющие основные риски в процессе разработки системы.

Создание артефактов можно начать на семинаре по определению требований. На этом семинаре не будут полностью составлены готовые документы, их доработает впоследствии системный аналитик.

### *Фаза развития*

На стадии развития уточняется видение системы на основе обратной связи от пользователей и разработчиков различных частей системы. При этом на нескольких итерациях разработки проводятся семинары по определению требований.

В процессе итеративной разработки требования все более проясняются и заносятся в дополнительную спецификацию. Атрибуты качества (например, надежность), описанные в дополнительной спецификации, помогают определить базовую архитектуру, разрабатываемую на стадии развития. Они также выявляют основные факторы риска. Например, требование обеспечения функционирования системы при сбоях внешних компонентов детально прорабатывается на стадии развития.

На этой стадии уточняется большинство терминов в словаре.

К моменту завершения фазы развития завершается и работа над описанием прецедентов, составлением дополнительной спецификации и документа "Видение", которые теперь отражают основные свойства и другие требования. Это не означает, что по завершении фазы развития дополнительную спецификацию и документ "Видение" нужно заморозить и не изменять в дальнейшем. Напомним, что основным "лейтмотивом" итеративной разработки и UP является возможность постоянной адаптации системы.

Поясним эту мысль подробнее. По завершении фазы развития целесообразно согласовать с заинтересованными лицами задачи следующего этапа проекта и разработать соглашение (возможно, письменное) относительно требований и графика их реализации. В какой-то момент (в рамках UP — по завершении фазы разработки) необходимо точно знать "что, когда и сколько". В этом смысле формальное соглашение о требованиях к системе является нормальным и целесообразным. Необходимо также определить процесс управления изменениями (одна из лучших идей UP), предполагающий формальное рассмотрение и одобрение вносимых изменений, а не хаотичную и бесконтрольную модификацию.

Поэтому приходим к следующим выводам.

- В рамках итеративной разработки и UP в детально проработанную спецификацию требований могут вноситься незначительные изменения. Эти изменения могут затрагивать свойства системы, обеспечивающие ее конкурентоспособность или улучшающие функциональность.

- В рамках итеративной разработки основное внимание уделяется получению обратной связи от заинтересованных лиц и реализации проекта в нужном направлении. Благодаря этому заинтересованные лица не могут "умыть руки" и ожидать в стороне реализации "замороженного" набора требований в окончательном программном продукте. При "замораживании" требований система вряд ли будет отражать реальные потребности заинтересованных лиц.

### *Фаза конструирования*

На этапе конструирования основные требования остаются стабильными, но это не означает, что они не могут незначительно изменяться. Однако дополнительная спецификация и документ "В*и*дение" не претерпевают существенных изменений.

## 7.12. Дополнительная литература

Дополнительная спецификация и документ "В*и*дение" использовались во многих проектах и описаны в многочисленных книгах. Однако большинство таких книг неявно предполагают однопроходный процесс реализации проекта и предусматривают детальное и корректное создание этих документов на начальных этапах до перехода к разработке и реализации. В этом смысле традиционная литература не соответствует современному представлению об итеративной разработке. Однако в каждой книге можно найти полезные разделы, посвященные содержательному наполнению этих документов.

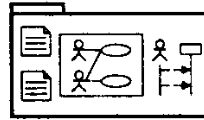
В большинстве книг по архитектуре программных систем описывается процесс анализа требований и атрибутов качества приложения, поскольку эти атрибуты оказывают влияние на архитектуру системы. Примером такой литературы является [9]. Бизнес-правила подробно описываются в [94]. В этой книге подробно изложена обширная теория формулировки бизнес-правил, однако предложенный метод не связан с другими современными приемами анализа требований, такими как прецеденты или итеративная разработка.

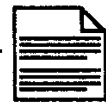
## 7.13. Артефакты UP в контексте процесса

Взаимосвязь артефактов для документов "В*и*дение", "Дополнительная спецификация" и "Словарь терминов" показана на рис. 7.1.

В рамках UP работа над документами "В*и*дение" и "Дополнительная спецификация" относится к дисциплине определения требований и начинается на семинарах по определению требований наряду с анализом прецедентов. На рис. 7.2 предложены некоторые рекомендации относительно места и времени проведения этой работы.

**Примеры артефактов UP**





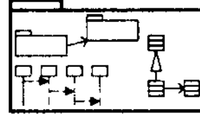


*Рис. 7.1. Пример взаимосвязи артефактов UP*

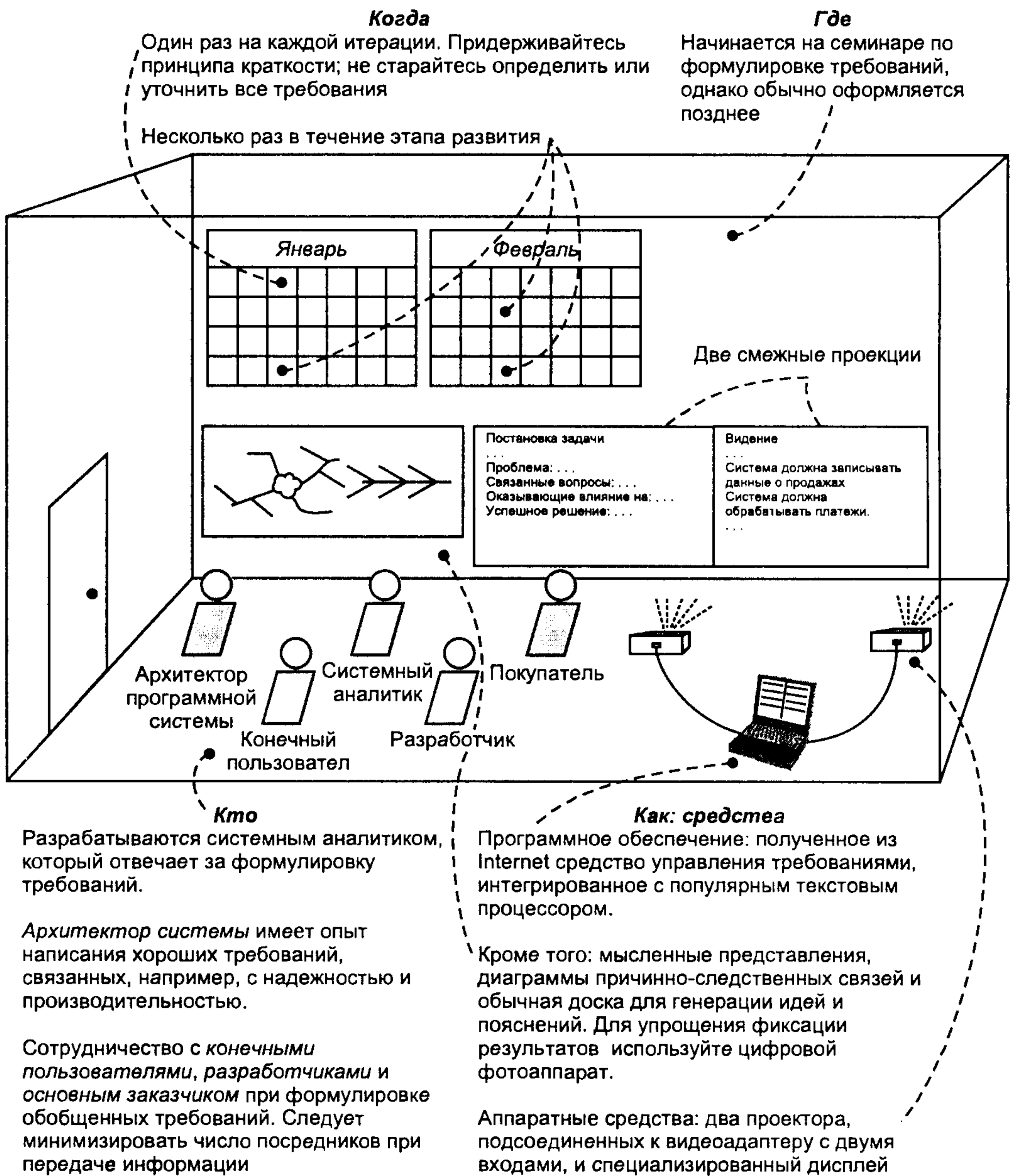



*Рис. 7.2. Рекомендации по организации семинаров*

Глава

# От начальной фазы к стадии развития

*Твердое и жесткое ломается. Гибкое торжествует.*

*Тао Те Чинг (Tao Te Ching)*

---

Основные задачи

- Определить стадию развития.
- Обосновать содержание последующих глав.

---

## Введение

Фаза развития — это первая последовательность итераций, в течение которых решаются следующие задачи.

- Изучается и стабилизируется большая часть требований
- Обосновываются и устраняются основные риски
- Реализуются и тестируются базовые архитектурные элементы

Разработка архитектуры приложения не сопряжена с риском в очень редких случаях, например, при наличии у разработчиков опыта создания аналогичных Web-узлов с помощью тех же средств и с теми же требованиями. Только в этих редких случаях вопросам архитектуры не уделяется пристальное внимание на стадии развития. Тогда на этом этапе реализуются другие принципиально важные прецеденты или свойства системы, не имеющие отношения к архитектуре.

На этом этапе мы приступим к изучению вопросов ООА/П, применения UML и шаблонов.

## 8.1. Контрольная точка: что сделано на начальной стадии?

На начальную фазу проекта POS-системы NextGen выделялась неделя. Созданные на этой стадии артефакты должны быть краткими и неполными, поскольку этот этап длится очень недолго.

На этой стадии не нужно определять требования к проекту, а лишь уяснить для себя основные риски, масштаб задачи, ее реалистичность и решить, стоит ли приступать к серьезным исследованиям, т.е. к этапу развития. В предыдущих главах были рассмотрены не все виды деятельности, выполняемые на начальной стадии. Основное внимание уделялось лишь вопросам определения требований. К числу видов деятельности и артефактов начальной фазы относятся следующие.

- Краткий семинар по определению требований.
- Выделение основных исполнителей, задач и прецедентов.
- Описание большинства прецедентов в сжатом формате. Полное описание 10–20% прецедентов для более глубокого понимания масштаба и сложности задачи.
- Идентификация наиболее важных требований, связанных с высокими рисками.
- Разработка первой версии документов "В*и*дение" и "Дополнительная спецификация".
- Составление списка рисков.
    - Например, руководство проекта очень хочет получить демонстрационную версию продукта к началу выставки POSWorld в Гамбурге, т.е. за 18 месяцев. Однако без более глубокого исследования нельзя даже примерно оценить необходимые для этого ресурсы.
- Технические прототипы, обеспечивающие обоснование концепции, а также другие исследования достижимости конкретных требований (корректно ли работает Java Swing с сенсорными экранами?).
- Разработка прототипа интерфейса пользователя для выявления функциональных требований.
- Формулировка рекомендаций по использованию компонентов на стадии развития (что нужно приобрести, разработать или повторно использовать).
    - Например, может понадобиться приобрести пакет вычисления налоговых платежей.
- Разработка высокоуровневой примерной архитектуры и предложений по использованию компонентов.
    - Это не детальное описание архитектуры, претендующее на корректность или окончательный вариант. Это лишь краткие размышления относительно начальной точки исследований на стадии развития. Например: "Будем использовать клиентское приложение Java без сервера приложений, а в качестве базы данных выберем Oracle...". На стадии развития эти идеи могут быть поддержаны или отвергнуты.
- Разработка плана первой итерации.
- Составление списка необходимых средств.

## 8.2. Фаза развития

Фаза развития — это первая последовательность итераций, в течение которых команда разработчиков выполняет серьезные исследования базовых элементов архитектуры, их реализацию (написание программного кода и его тестирование), опреде-

ляет для себя большинство требований и снимает вопросы, связанные с наиболее высокими рисками. В контексте UP термин "риск" имеет экономическое значение. То есть на ранних этапах реализации проекта должны разрабатываться сценарии, достаточно важные, но не обязательно сопряженные с техническим риском.

Фаза развития зачастую состоит из 2–4 итераций, каждая из которых длится от двух до шести недель. Время каждой итерации жестко фиксировано. Если поставленные задачи сложно выполнить к назначенному сроку, то некоторые требования переносятся на последующие итерации, чтобы данная итерация завершилась вовремя и в результате был получен устойчивый и протестированный код.

Фаза развития — это не стадия проектирования или подготовки к реализации, как было принято в рамках каскадного процесса (в стиле "водопада").

На этой стадии создаются не прототипы, а полностью разрабатывается некоторый фрагмент системы. В некоторых описаниях UP для представления части системы используется термин *архитектурный прототип* (architectural prototype), который может быть неверно истолкован. В данном случае прототип — это не экспериментальный образец, а рабочее подмножество окончательной системы. Эту часть системы также называют *исполняемой архитектурой* (executable architecture) или *архитектурной основой* (architectural baseline).

*Коротко о фазе развития*

Построение базовой архитектуры, разрешение высоких рисков, определение большинства требований и оценка общего графика реализации и необходимых ресурсов.

Приведем некоторые основные идеи и рекомендации по реализации стадии развития.

- Планирование кратких итераций, связанных с основными рисками.
- Раннее начало программирования.
- Адаптивное проектирование, реализация и тестирование основных элементов архитектуры.
- Раннее, частое и реалистичное тестирование.
- Адаптация системы на основе обратной связи от специалистов по тестированию, пользователей и разработчиков.
- Подробное описание большинства прецедентов и других требований, проведение серии семинаров, по одному на каждой итерации.

### Что является архитектурно важным на стадии развития?

На начальных итерациях разрабатывается и обосновывается базовая архитектура. Для проекта NextGen к задачам этого этапа относятся следующие.

- Разработка и реализация системы "не вглубь, а вширь" или, по определению Греди Буча (Grady Booch), "разработка по пластам".
  - Это означает идентификацию отдельных процессов, слоев, пакетов и подсистем, их высокоуровневых функций и интерфейсов. Затем выполняется реализация некоторых элементов для уточнения интерфейсов. Модули могут содержать в основном заглушки.

- Уточнение локальных и удаленных интерфейсов между модулями (включая мельчайшие детали, параметры и возвращаемые значения).
  - Например, разработка интерфейса с объектом, обеспечивающим доступ к внешней бухгалтерской системе.
  - Первая версия интерфейса вряд ли окажется идеальной. Благодаря тестированию в критических режимах на ранних этапах реализации проекта обеспечивается возможность последующего уточнения интерфейсов и параллельной работы всей команды на базе устойчивых интерфейсов.
- Интегрирование существующих компонентов.
  - Например, системы вычисления налоговых платежей.
- Реализация упрощенных сценариев, обеспечивающих параллельное проектирование, программирование и тестирование нескольких основных компонентов.
  - Например, разработка основного успешного сценария `Оформление продажи` и сценария расширения, связанного с реализацией платежа по кредитной карточке.

На этапе развития очень важную роль играет тестирование, обеспечивающее обратную связь, возможность адаптации и обоснование робастности ядра системы. Раннее тестирование для проекта NextGen включает следующие этапы.

- Тестирование удобства интерфейса пользователя для прецедента `Оформление продажи`.
- Тестирование работоспособности системы и возможности восстановления информации при выходе из строя удаленных служб, например службы авторизации платежей.
- Тестирование системы при критических нагрузках на удаленные службы, например, на службу вычисления налоговых платежей.

## 8.3. Планирование следующей итерации

Планирование и управление проектом — очень важные, но обширные задачи. Некоторые основные моменты этих видов деятельности будут рассмотрены в этой главе, а более детальное их описание приводится в главе 36.

Требования и итерации систематизируются в соответствии с рисками, границами и критичностью.

- *Риск* (risk) — это техническая сложность или другой фактор, например, отсутствие информации о необходимых затратах или ресурсах.
- *Границы* (coverage) — на начальных итерациях нужно определить все основные части системы, т.е. выполнить реализацию множества компонентов “не вглубь, а вширь”.
- *Критичность* (criticality) — требуется реализовать функции, имеющие важное значение для системы.

Эти критерии используются для распределения работы по итерациям. Прецеденты или их отдельные сценарии ранжируются с целью определения приоритетов при реализации. На начальных итерациях реализуются прецеденты с высоким рейтингом. Некоторые требования выражаются как высокоуровневые

свойства системы, не связанные с прецедентами, например, служба регистрации событий. Такие требования тоже ранжируются.

Ранжирование выполняется перед началом первой итерации, затем перед началом второй и т.д. Такой план является адаптивным, а не зафиксированным на начальной стадии проекта.

Требования обычно группируют следующим образом.

| **Приоритет** | **Требование (прецедент или свойство)** | **Комментарий** |
|---|---|---|
| Высокий | Оформление продажи<br>Регистрация<br>... | Самый высокий приоритет<br>Сложно добавить позднее<br>... |
| Средний | Поддержка пользователей<br>... | Влияет на безопасность<br>... |
| Низкий | ... | ... |

На основе такого ранжирования можно выделить архитектурно важные сценарии прецедента Оформление продажи, которые необходимо реализовать на начальных итерациях. Этот список не полон, в нем нужно отразить и остальные требования. Кроме того, на каждой итерации в явном или неявном виде реализуется прецедент Запуск системы.

Рассмотрим некоторые вопросы планирования в терминах артефактов UP.

- Требования, выбранные для реализации на следующей итерации, описываются в документе "План итерации" (Iteration plan). Это план не всех итераций проекта, а только следующей итерации.
- Если краткого плана итерации недостаточно, то задание на следующую итерацию можно подробнее описать в отдельном документе "Запрос на изменение" (Change request).
- Ранжированные требования описываются в "Плане разработки программного продукта" (Software development plan).

## 8.4. Требования и акценты первой итерации: основные вопросы ООА/П

На первой итерации стадии развития рассматриваемого примера основное внимание уделяется фундаментальным вопросам ООА/П и построения объектной системы, в частности, распределению обязанностей между объектами. Конечно же, для построения программной системы необходимо рассмотреть и множество других вопросов, таких как разработка базы данных, исследование удобства использования и проектирование интерфейса пользователя. Однако эти вопросы выходят за рамки основ унифицированного процесса и ООА/П.

### *Первая итерация: требования*

Ниже перечислены требования для первой итерации POS-системы NextGen.

- Реализация базового сценария прецедента Оформление продажи — ввод наименований товара и оплата наличными.

- Реализация прецедента Запуск системы, поскольку он необходим для инициализации основного сценария этой итерации.
- Никакие сложные сценарии не реализуются, зато простой успешный сценарий доводится до программной реализации.
- Взаимодействие с внешними службами, такими как база данных продуктов и система вычисления налогов, не реализуется.
- Сложные правила ценообразования не применяются.

Проектируются и реализуются также некоторые элементы интерфейса пользователя.

Дальнейшая функциональность будет наращиваться на последующих итерациях.

### *Инкрементальная разработка одного и того же прецедента в течение нескольких итераций*

Заметим, что на первой итерации реализуются не все требования прецедента Оформление продажи. Зачастую работа над различными сценариями одного и того же прецедента выполняется в течение нескольких итераций (рис. 8.1). В то же время короткие, простые прецеденты могут быть полностью реализованы в течение одной итерации.

*Рис. 8.1. Реализация прецедента в течение нескольких итераций*

## 8.5. Разработка каких артефактов начинается на стадии развития?

В табл. 8.1 приводится примерный список артефактов, разработка которых может начинаться на этапе развития. Некоторые из этих артефактов подробнее рассматриваются в последующих главах, в частности модель предметной области

и модель проектирования. В эту таблицу не включены артефакты, начало разработки которых приходится на начальную фазу (эти артефакты перечислены в главе 4). Заметим, что их создание не завершается на одной итерации, а продолжается в течение нескольких итераций.

**Таблица 8.1. Пример артефактов фазы развития**

| Артефакт | Комментарий |
|---|---|
| Модель предметной области | Она представляет собой визуализацию понятий предметной области, напоминающую статическую модель сущностей предметной области |
| Модель проектирования | Это набор диаграмм, описывающих логику проектного решения. Сюда относятся диаграммы программных классов, диаграммы взаимодействия объектов, диаграммы пакетов и т.д. |
| Описание программной архитектуры | Это документ, в котором рассмотрены основные архитектурные моменты и способы их реализации. В нем приводятся основные идеи проектного решения и обосновывается их целесообразность для данной системы |
| Модель данных | Включает схему базы данных и стратегию отображения объектов в необъектное представление |
| Модель тестирования | Описание задач и способов тестирования |
| Модель реализации | Это реальная реализация — с исходным кодом, исполняемыми файлами, базой данных и т.д. |
| Прототипы интерфейса пользователя | Описание интерфейса пользователя, способов навигации и т.д. |

## 8.6. Вы не поняли, что такое фаза развития, если...

- Для большинства проектов она длится дольше, чем несколько месяцев.
- Она содержит лишь одну итерацию (за редким исключением).
- Большинство требований определены до начала фазы развития.
- На этом этапе не затрагиваются базовые архитектурные элементы и элементы с высокими рисками.
- В результате реализации этой фазы не получен реальный исполняемый код.
- Эта фаза рассматривается в основном как стадия формулировки требований, предшествующая стадии реализации.
- Делаются попытки разработки полного и точного проектного решения до начала программирования.
- Обратная связь и адаптация задействованы минимально, пользователи редко привлекаются для оценки системы.
- Отсутствует раннее и реалистичное тестирование.
- Архитектура системы разработана до начала программирования.
- Эта фаза рассматривается как этап проверки концепции, а не как этап реализации ядра системы.
- Не проводятся краткие семинары по определению требований, призванные уточнять и корректировать требования на основе обратной связи.

Если налицо подобные признаки, значит, разработчики не понимают назначения стадии развития.

# Часть III

# Первая итерация фазы развития

Глава

# Модель прецедентов: диаграммы последовательностей

*Теоретически, между теорией и практикой нет различий.*
*Но на практике они есть.*

*Жан Л.А. ван де Снепшот (Jan L.A. van de Snepscheut)*

**Основные задачи**

- **Идентифицировать системные события.**
- **Создать диаграммы последовательностей для прецедентов.**

## Переход к первой итерации

Проект разработки POS-системы NextGen перешел на стадию реализации первой реальной итерации. На начальной стадии были выполнены некоторые исследования требований, на основании которых было решено продолжить реализацию проекта. Был разработан план первой итерации, согласно которому на этой итерации нужно разработать простой успешный сценарий прецедента Оформление продажи, когда оплата осуществляется наличными (без участия внешних систем). Таким образом, была поставлена задача разработки "не вглубь, а вширь", поскольку при реализации этого простого сценария задействованы многие основные архитектурные элементы новой системы. На первой итерации решение многих задач связано с установкой окружения (средств, процессов, параметров и людей). Однако эти виды деятельности мы опускаем.

Основное внимание будет уделено анализу прецедента и моделированию предметной области. До начала проектирования целесообразно выполнить дальнейшее исследование предметной области, в частности уточнить входные и выходные события, связанные с данной системой, которые можно проиллюстрировать на диаграмме последовательностей в системе обозначений UML.

## Введение

Диаграмма последовательностей — это быстро и легко создаваемый артефакт, иллюстрирующий входные и выходные события, связанные с разрабатываемой системой. Для иллюстрации событий, связывающих внешних исполнителей с системой, в языке UML существуют специальные обозначения, позволяющие создавать диаграммы последовательностей.

## 9.1. Поведение системы

Прежде чем приступать к проектированию логики работы программного приложения, необходимо исследовать и определить ее поведение как "черного ящика". *Поведение системы* (system behavior) представляет собой описание того, *какие* действия выполняет система, без определения механизма их реализации. Одной из частей такого описания является диаграмма последовательностей. К остальным частям относятся прецеденты и описания системных операций (которые будут рассмотрены в последующих главах).

## 9.2. Диаграммы последовательностей системы

Прецеденты определяют, как исполнители взаимодействуют с программной системой. В процессе этого взаимодействия исполнителем генерируются события, передаваемые системе, которые представляют собой запросы на выполнение некоторой операции. Например, кассир, введя идентификатор товара, тем самым предписывает, чтобы система POS записала данные о приобретении товара. Это событие инициирует в системе выполнение некоторой операции.

Было бы неплохо отделить и проиллюстрировать операции системы, выполнение которых запрашивает внешний исполнитель, поскольку они важны для понимания поведения системы. В качестве системы обозначений в состав языка UML входят *диаграммы последовательностей* (sequence diagram). С их помощью можно проиллюстрировать взаимодействие исполнителя с системой и операции, выполнение которых при этом инициируется.

*Диаграмма последовательностей системы* (system sequence diagram) — это схема, которая для определенного сценария прецедента[1] показывает генерируемые внешними исполнителями события, их порядок, а также события, генерируемые внутри самой системы. При этом все системы рассматриваются как "черный ящик". Назначение данной диаграммы — отображение событий, передаваемых исполнителями системе через ее границы.

> Диаграмму последовательностей нужно создать для основного успешного сценария прецедента, а при необходимости и для наиболее существенных и сложных альтернативных сценариев.

В контексте языка UML нет понятия "диаграмма последовательностей системы", есть просто "диаграмма последовательностей". Это уточнение автор использовал для того, чтобы сделать акцент на рассмотрении системы в виде "черного ящика". В дальнейшем диаграммы последовательностей будут рассмотрены в другом контексте — для иллюстрации взаимодействия разрабатываемых программных объектов.

---

[1] Сценарий прецедента — это его частный случай или реальный путь его реализации.

## 9.3. Пример диаграммы последовательностей

На диаграмме последовательностей для определенного хода событий, описанного в прецеденте, отображаются внешние исполнители, которые взаимодействуют непосредственно с системой, сама система (как "черный ящик"), а также системные события, инициируемые исполнителями (рис. 9.1). При этом порядок событий должен соответствовать их последовательности в описании прецедента.

Системные события могут включать различные параметры.

Пример, приведенный на рис. 9.1, соответствует основному успешному сценарию прецедента `Оформление продажи`. Он иллюстрирует, что для POS-системы кассир генерирует системные события `makeNewSale`, `enterItem`, `endSale` и `makePayment`.

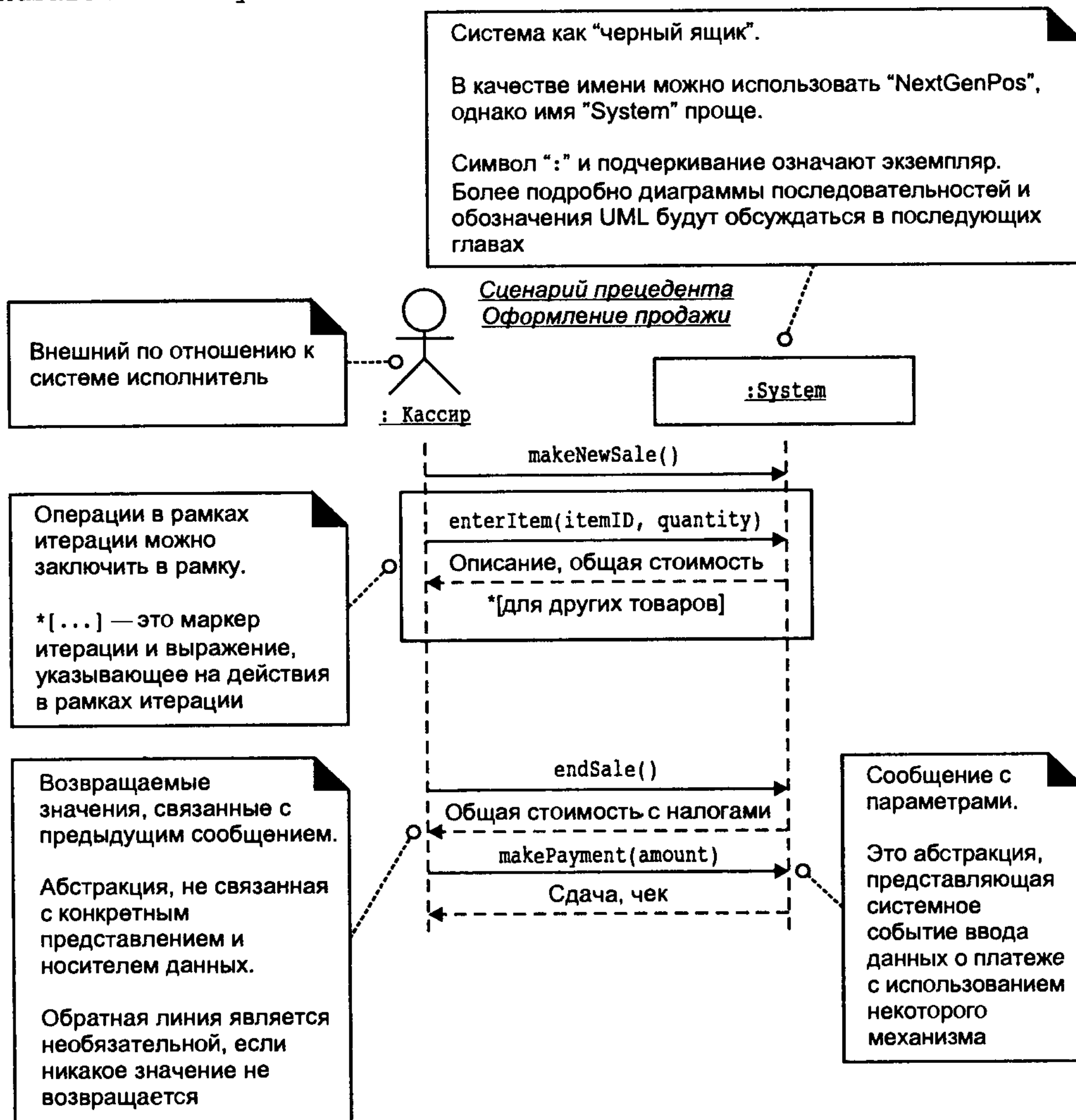



*Рис. 9.1. Диаграмма последовательностей для сценария прецедента* `Оформление продажи`

## 9.4. Межсистемные диаграммы последовательностей

Диаграммы последовательностей можно также использовать для иллюстрации взаимодействия между системами, например, между POS-системой NextGen и внешней службой авторизации платежей по кредитной карточке. Однако этот вопрос мы отложим до следующих итераций, поскольку на данном этапе нас не интересует взаимодействие с удаленными системами.

## 9.5. Системные события и прецеденты

На диаграмме последовательностей отображаются системные события сценария некоторого прецедента. Поэтому сама диаграмма строится на основе описания прецедента (рис. 9.2).

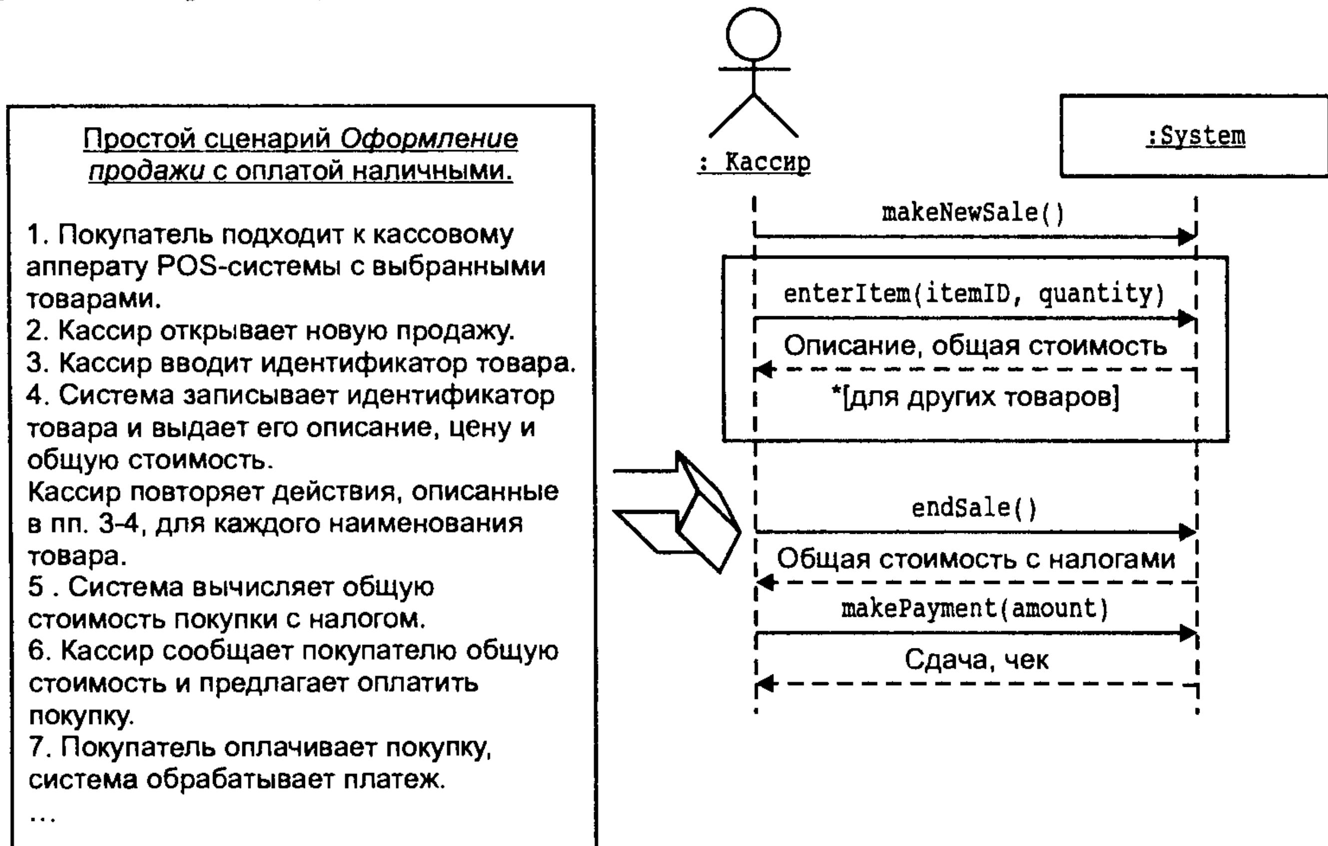



*Рис. 9.2. Диаграммы последовательностей строятся на основе описания прецедентов*

## 9.6. Системные события и границы системы

Как уже упоминалось в предыдущей главе, при идентификации системных событий необходимо четко определить, где проходят границы системы. При разработке программной системы ее границы обычно выбираются в соответствии с ее программной частью (а возможно, и аппаратной). В этом контексте системным является внешнее событие, оказывающее влияние непосредственно на программное обеспечение (рис. 9.3).

Выполним идентификацию системных событий для прецедента Оформление продажи. Сначала необходимо определить исполнителей, которые взаимодействуют с программной системой непосредственно. Покупатель взаимодействует с кассиром, однако при этом его связь с программным обеспечением POS-системы отсутствует. Этот процесс осуществляется только кассиром. Следовательно, генератором системных событий является лишь кассир.

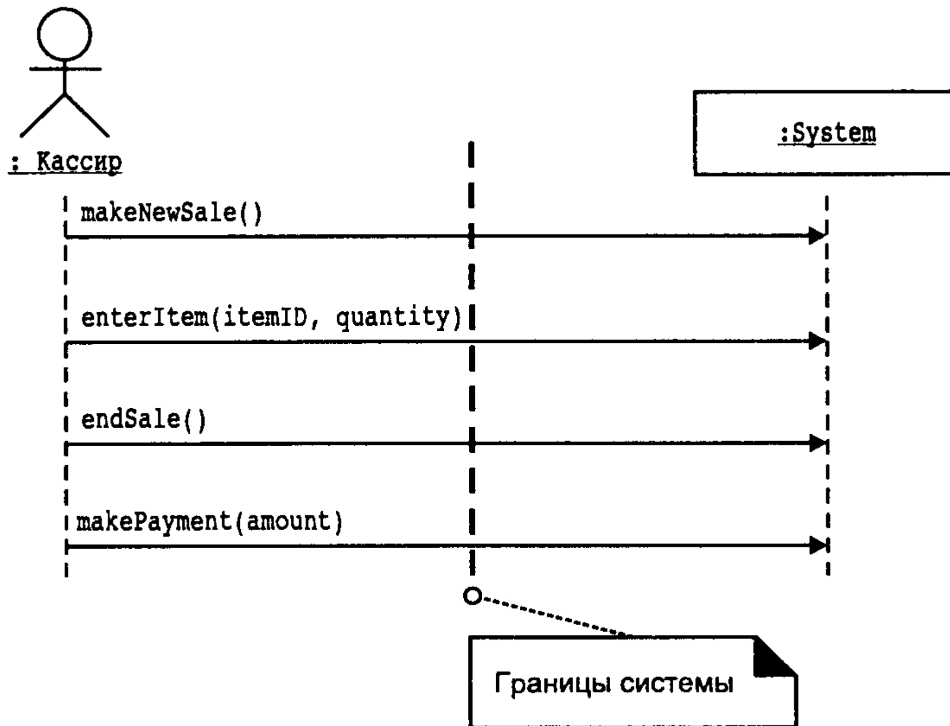



*Рис. 9.3. Определение границ системы*

## 9.7. Имена системных событий и операций

Системным событиям (и связанным с ними системным операциям) имена нужно присваивать осмысленно, а не в терминах входных физических носителей данных или управляющих элементов пользовательского интерфейса.

Имя системного события лучше всего начинать с глагола: `add...` (добавить), `enter...` (ввести), `end...` (завершить), `make...` (выполнить) (рис. 9.4). Это повышает читабельность имен и, кроме того, дополнительно подчеркивает направление передачи событий.

Таким образом, имя `enterItem` гораздо лучше, чем имя `scan` (в смысле считывания лазерным сканером), поскольку в нем отражается смысл операции. К тому же такое имя является абстрактным, не зависящим от выбора управляющих элементов интерфейса, которые будут участвовать в непосредственной обработке системного события.

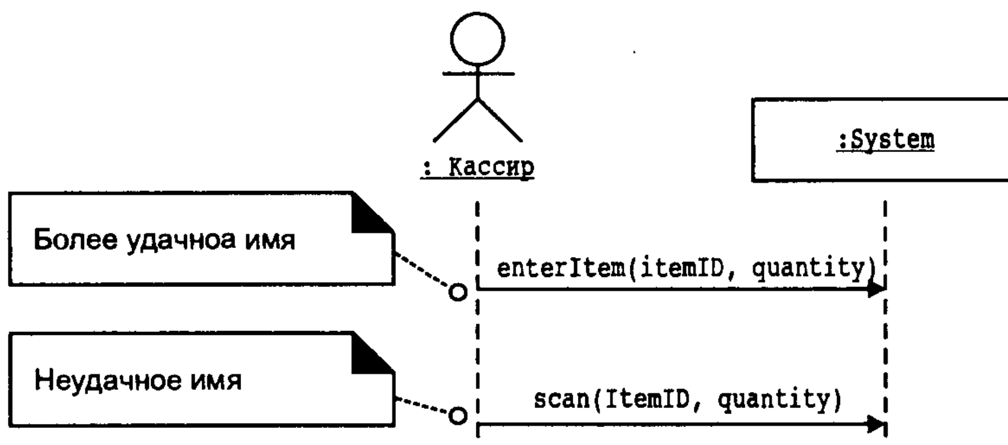



*Рис. 9.4. Выбирайте имена событий и операций на абстрактном уровне*

## 9.8. Отображение текста из описания прецедента

В некоторых случаях на диаграмму последовательностей совсем нелишне помещать хотя бы фрагменты из описания прецедента, что позволит проиллюстрировать строгую связь между этими двумя артефактами (рис. 9.5).

Простой сценарий *Оформление продажи* с оплатой наличными.

1. Покупатель подходит к кассовому аппарату POS-системы с выбранными товарами.
2. Кассир открывает новую продажу.
3. Кассир вводит идентификатор товара.
4. Система записывает идентификатор товара и выдает его описание, цену и общую стоимость.

Кассир повторяет действия, описанные в пп. 3-4, для каждого наименования товара.

5. Система вычисляет общую стоимость покупки с налогом.
6. Кассир сообщает покупателю общую стоимость и предлагает оплатить покупку.
7. Покупатель оплачивает покупку, система обрабатывает платеж.

...

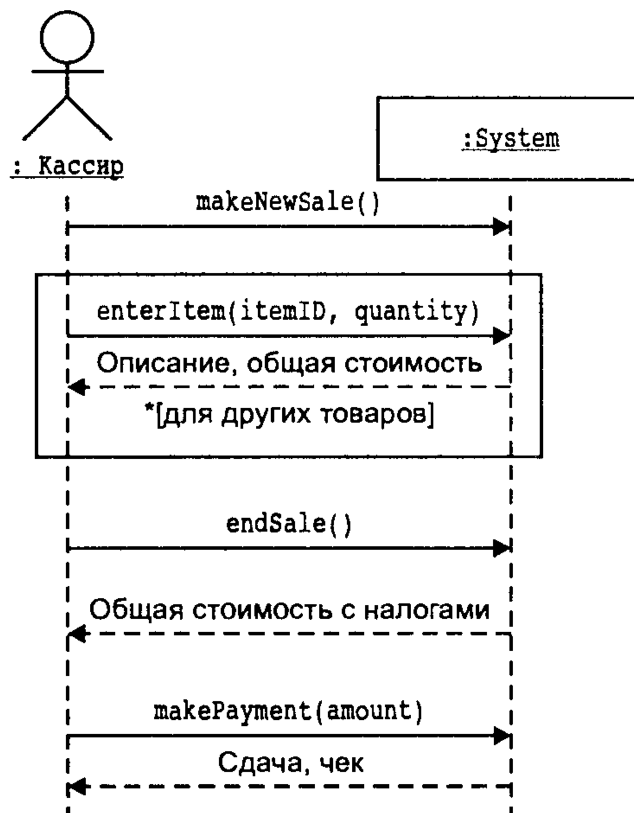



*Рис. 9.5. Диаграмма последовательностей системы с текстом из описания прецедента*

## 9.9. Диаграммы последовательностей и словарь терминов

На диаграмме последовательностей термины (операции, параметры, возвращаемые значения) отображаются в краткой форме. Поэтому на этапе проектирования могут понадобиться некоторые пояснения этих терминов. Если значение терминов не разъясняется в описании прецедентов, то их нужно внести в словарь терминов.

Однако, как обычно при обсуждении процесса создания артефактов, отличных от программного кода (составляющего "сердцевину" проекта), нужно соблюдать осторожность. Необходимо тщательно обдумывать вносимую в словарь информацию, поскольку иначе его заполнение превратится в ненужную и утомительную работу.

## 9.10. Диаграммы последовательностей в контексте UP

Диаграммы последовательностей — это часть модели прецедентов. Они обеспечивают визуализацию взаимодействия объектов при реализации прецедента. В исходном описании унифицированного процесса диаграммы последовательностей не упоминаются явно, хотя разработчики UP хорошо понимают значимость и полезность подобных диаграмм. Диаграммы последовательностей — это пример артефактов анализа и проектирования или видов деятельности, не упомянутых в документах по описанию UP или RUP.

### *Фазы*

**Начало** — на этой стадии диаграммы последовательностей не строятся.

**Развитие** — большая часть диаграмм последовательностей разрабатывается на стадии развития, когда необходимо определить детали системных событий и прояснить основные системные операции, составить описания системных операций (этот процесс будет описан в главе 13) и выполнить некоторые оценки (например, макрооценивание на основе ненастраиваемых функциональных точек и принципов COCOMO II).

Заметим, что совсем не обязательно создавать диаграммы последовательностей для всех сценариев прецедента, во всяком случае одновременно. На данной итерации нужно рассматривать только некоторые выбранные сценарии (табл. 9.1).

И наконец, напомним, что на создание диаграммы последовательностей должно отводиться от нескольких минут до получаса.

**Таблица 9.1. Пример планирования сроков реализации артефактов UP (н — начало, р — развитие)**

| Дисциплина | Артефакт<br>Итерация→ | Начало<br>I1 | Развитие<br>E1..En | Конструирование<br>C1..Cn | Передача<br>T1..Tn |
|---|---|---|---|---|---|
| Бизнес-моделирование | Модель предметной области | | н | | |
| Требования | **Модель прецедентов** | н | р | | |
| | Видение системы | н | р | | |
| | Дополнительная спецификация | н | р | | |
| | Словарь терминов | н | р | | |
| Проектирование | Модель проектирования | | н | р | |
| | Описание архитектуры | | н | | |
| | Модель данных | | н | р | |
| Реализация | Модель реализации | | н | р | р |
| Управление проектом | План разработки | н | р | р | р |
| Тестирование | Модель тестирования | | н | р | |
| Окружение | Набор документов | н | р | | |

## 9.11. Дополнительная литература

Различные типы диаграмм, иллюстрирующих входные и выходные события системы, рассматриваемой в качестве "черного ящика", известны и широко использовались в течение десятилетий. Например, в области телекоммуникаций широко известны диаграммы потоков звонков. Они особенно популярны в различных методах объектно-ориентированного подхода, в частности используются в методе Fusion [34]. В этой книге детально рассматриваются вопросы взаимосвязи диаграмм последовательностей и системных операций с другими артефактами анализа и проектирования.

## 9.12. Артефакты UP

Пример взаимосвязи диаграмм последовательностей с другими артефактами показан на рис. 9.6.

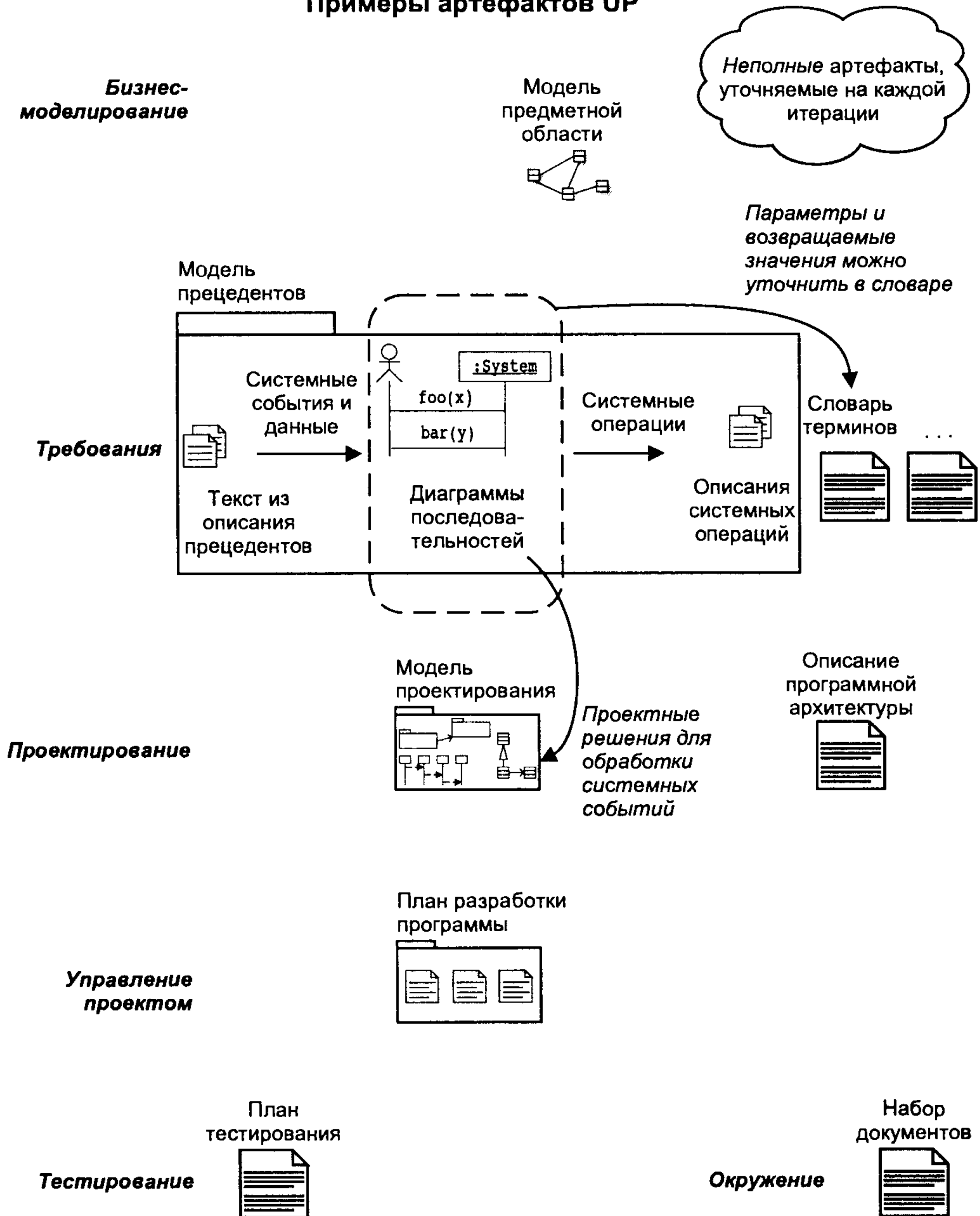



*Рис. 9.6. Пример взаимосвязи артефактов UP*

Глава

# Модель предметной области: визуализация понятий

*Это очень хорошо на практике, но теоретически оно никогда не будет работать.*

*Неизвестный управленец*

---

**Основные задачи**

- Идентифицировать классы понятий (концептуальные классы), соответствующие требованиям текущей итерации.
- Создать исходную модель предметной области.
- Определить корректные и некорректные атрибуты.
- При необходимости добавить спецификацию концептуальных классов.
- Провести сравнительный анализ концептуального представления и реализации.

---

## Введение

Модель предметной области широко используется в качестве основы для разработки программных объектов и обеспечивает важную входную информацию для создания нескольких последующих артефактов, описываемых в этой книге. Следовательно, если вы не знакомы с вопросами моделирования предметной области, вам необходимо ознакомиться с этой главой.

Модель предметной области отображает основные (с точки зрения моделирующего) классы понятий (концептуальные классы) предметной области. Она является наиболее важным артефактом, создаваемым на этапе объектно-ориентированного анализа[1]. В этой главе приводятся начальные сведения о создании моделей предмет-

[1] Прецеденты — это важные артефакты этапа анализа требований, но на самом деле они не являются *объектно*-ориентированными. Они концентрируют внимание разработчиков на процессах, происходящих в предметной области.

ной области. В следующих двух главах содержится более подробная информация о концептуальных моделях с точки зрения атрибутов и ассоциаций.

Основной задачей объектно-ориентированного анализа является идентификация большого количества разнообразных объектов или понятий, а также точная оценка усилий в терминах отдачи на стадиях проектирования и реализации.

Идентификация классов понятий или концептуальных классов — составная часть исследования предметной области. Модели предметной области на языке UML строятся в форме диаграмм классов.

> *Основная идея*
>
> Модель предметной области представляет классы понятий реального мира, а не программные компоненты. Это *не* набор диаграмм, описывающих программные классы или программные объекты с их обязанностями.

## 10.1. Модели предметной области

Основной составляющей объектно-ориентированного анализа или исследования является декомпозиция проблемы на отдельные классы понятий (концептуальные классы) или объекты. *Модель предметной области* — это *визуальное* представление концептуальных классов или объектов реального мира в терминах предметной области [46, 81]. Такие модели называют также *концептуальными моделями* (именно этот термин был принят в первом издании этой книги), *моделями объектов предметной области* или *объектными моделями анализа*.[2]

В контексте, UP Модель предметной области[3] — это один из артефактов, создаваемых в рамках дисциплины бизнес-моделирования.

На языке UML модель предметной области представляется в виде набора *диаграмм классов*, на которых не определены никакие операции. Модель предметной области может отображать следующее.

- Объекты предметной области или концептуальные классы.
- Ассоциации между концептуальными классами.
- Атрибуты концептуальных классов.

Например, на рис. 10.1 представлен фрагмент модели предметной области, из которого ясно, что с точки зрения предметной области концептуальными классами являются `Payment` (Платеж) и `Sale` (Продажа). Как видно из диаграммы, эти понятия связаны между собой и понятию `Sale` соответствуют определенная дата и время. Пока мы не станем концентрировать внимание читателя на системе обозначений.

---

[2] Они также связаны с моделями взаимоотношений концептуальных сущностей, отображающими только концептуальное представление предметной области, однако интерпретируемыми в более широком смысле как модели данных для разработки баз данных. Модели предметной области — это не модели данных.

[3] В данном случае термин *Модель предметной области* начинается с прописной буквы, чтобы акцентировать внимание на том, что это официальная модель, определенная в рамках UP, а не просто обычное понятие.

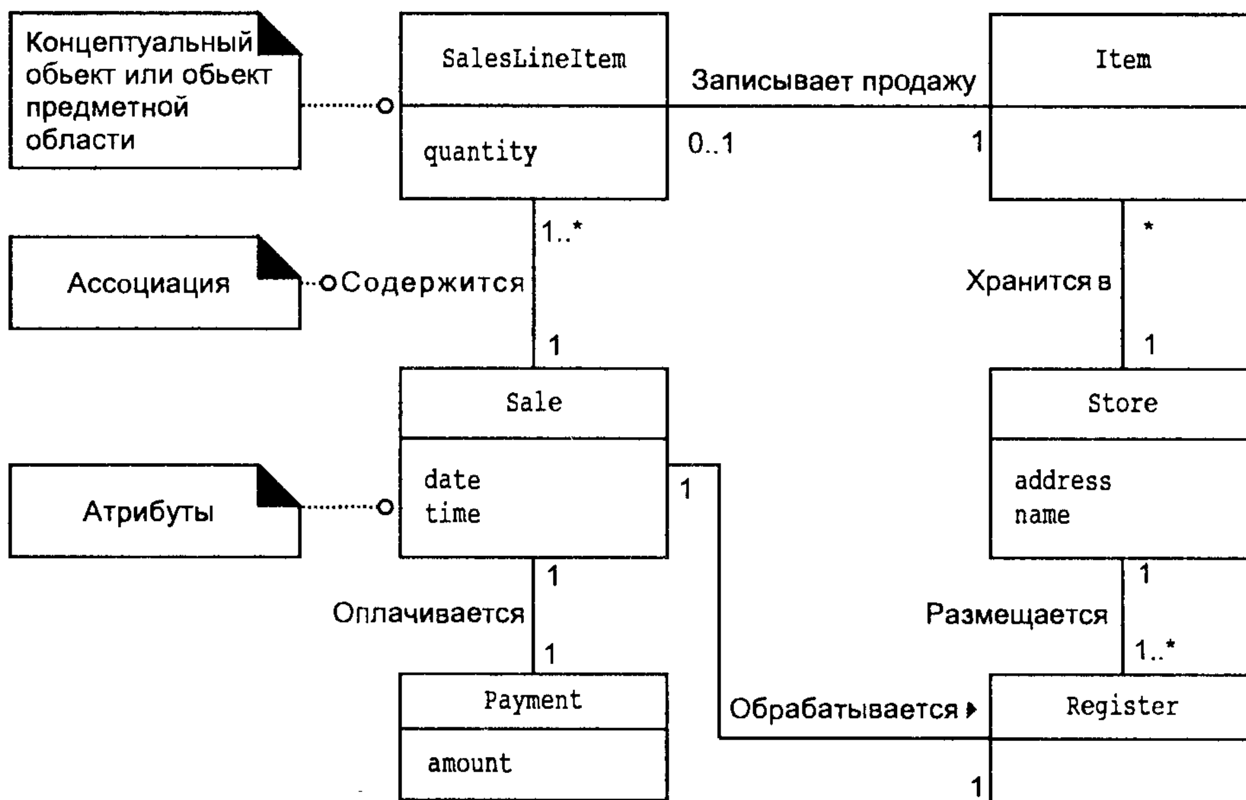



*Рис. 10.1. Фрагмент модели предметной области — визуальный словарь. Числа на концах соединительных линий обозначают кратность связи, которая более подробно рассматривается в последующих главах*

## Основная идея: модель предметной области — это визуальный словарь абстракций

Обратимся еще раз к рис. 10.1. На нем представлены некоторые понятия или концептуальные классы предметной области и связи между ними. На самом деле здесь показаны абстракции концептуальных классов, поскольку с каждой продажей, реестром и другим понятием связано множество свойств и характеристик. Данная модель отображает обобщенное представление или абстракцию и не учитывает неинтересные (с точки зрения моделирующего) детали.

Представленную на диаграмме (в системе обозначений UML) информацию можно также выразить в виде словесного описания, терминов словаря и т.д. Однако элементы и их взаимосвязи легче представить на этом визуальном языке, поскольку одним из преимуществ человеческого интеллекта является хорошая способность обработки визуальной информации.

Таким образом, модель предметной области можно рассматривать как *визуальный словарь* важных абстракций или словарь предметной области.

## Концептуальная модель — это не модель программных компонентов

Как видно из рис. 10.2, модель предметной области — это результат визуализации понятий реального мира в терминах предметной области, а не программных элементов, таких как классы Java или C++ (рис. 10.3). Следовательно, в модели предметной области не используются следующие элементы.

- Артефакты программирования наподобие окон или базы данных, если только разрабатываемая система не является моделью программного средства, например моделью графического интерфейса пользователя.

- Обязанности или методы[4].

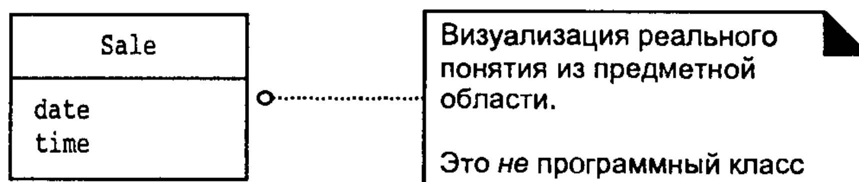



*Рис. 10.2. Модель предметной области отображает понятия реального мира, а не программные классы*

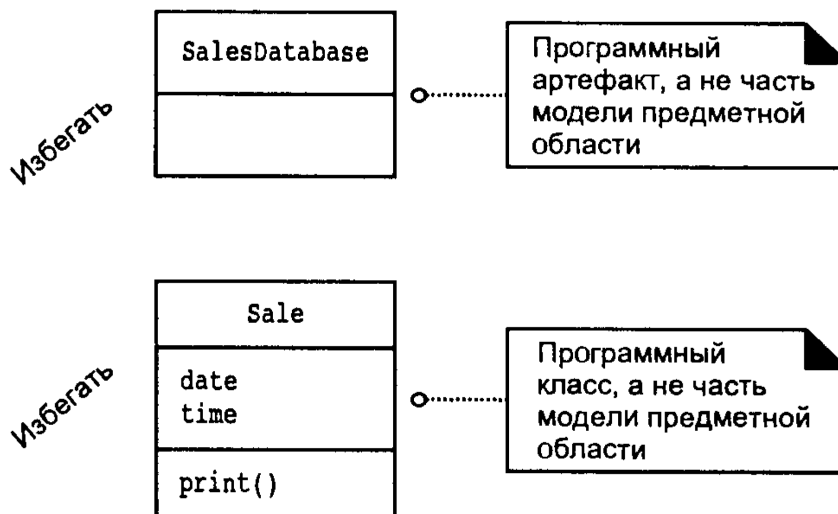



*Рис. 10.3. Модель предметной области не отображает программные артефакты или классы*

### Концептуальные классы

Модель предметной области иллюстрирует концептуальные классы или словарь предметной области. Неформально, *концептуальный класс* — это представление идеи или объекта. Если говорить более строго, то концептуальный класс можно рассматривать в терминах символов, содержания и расширения [81] (рис. 10.4).

- *Символы* (symbol) — слова или образы, представляющие концептуальный класс.
- *Содержание* (intension) — определение концептуального класса.
- *Расширение* (extension) — набор примеров, к которым применим концептуальный класс.

Например, рассмотрим концептуальный класс для события осуществления покупки. Его можно обозначить символом `Sale`. Содержанием этого понятия является "представление события осуществления покупки в определенный день и определенное время". В качестве расширения можно рассматривать все примеры покупок (другими словами, все множество покупок).

---

[4] В объектном моделировании обязанности обычно связаны с программными компонентами. Методы — это тоже чисто программные понятия. Однако модель предметной области описывает понятия реального мира, а не программные компоненты. Обязанности рассматриваются на стадии проектирования и не являются частью данной модели. Обязанности могут отображаться в модели предметной области только в том случае, если к понятиям предметной области относятся роли, исполняемые людьми (например, кассир), и их обязанности нужно зафиксировать в модели.

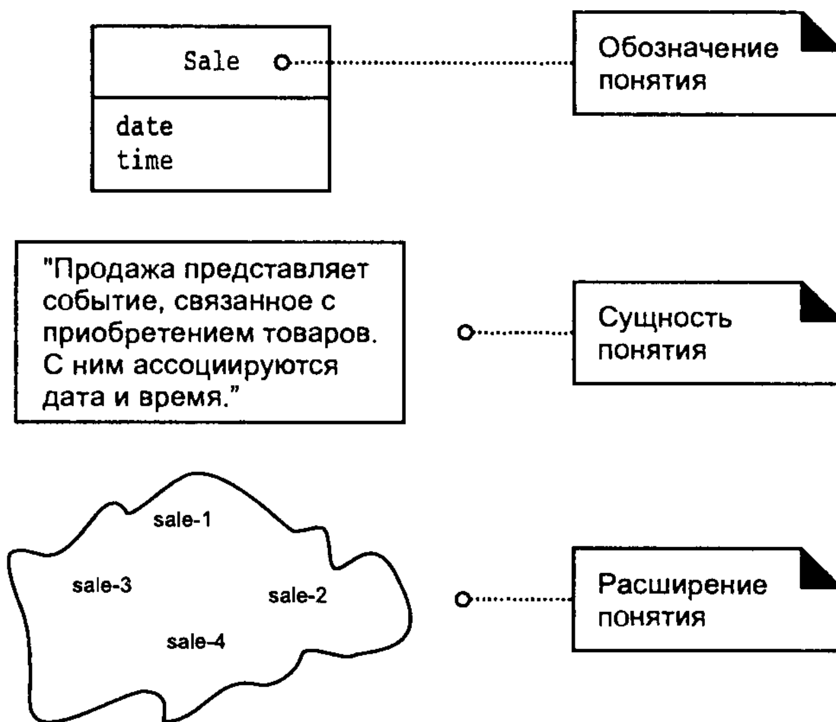



*Рис. 10.4. Концептуальному классу соответствуют символ, содержание и расширение*

При создании модели предметной области обычно рассматриваются символьное описание и содержание концептуального класса, поскольку именно они представляют наибольший практический интерес.

## Модели предметной области и декомпозиция

Программные системы могут быть очень сложными. Поэтому декомпозиция (по принципу "разделяй и властвуй") — это общая стратегия борьбы со сложностью проблемы за счет ее разделения на мелкие составные части. При *структурном подходе* к проектированию систем задача разбивается на процессы или функции, а при объектно-ориентированном — на основные понятия или сущности предметной области.

> Основное отличие объектно-ориентированного анализа от структурного состоит в декомпозиции проблемы на понятия (объекты), а не на функции.

Следовательно, основной задачей на стадии анализа является идентификация различных понятий из предметной области и представление результатов в виде модели предметной области.

## Концептуальные классы предметной области торговли

Например, в предметной области розничной торговли можно выделить концептуальные классы `Store` (Магазин), `Register` (Реестр) и `Sale` (Продажа). Следовательно, модель предметной области системы автоматизации розничной торговли должна включать элементы, представленные на рис. 10.5.

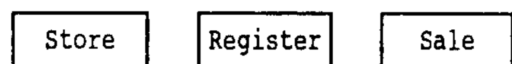



*Рис. 10.5. Фрагмент модели предметной области для системы розничной торговли*

## 10.2. Идентификация концептуальных классов

Нашей задачей является создание модели предметной области, отражающей интересные и важные понятия рассматриваемой предметной области розничной торговли. В данном случае речь идет о понятиях, связанных с прецедентом `Оформление продажи`.

При итеративной разработке модель предметной области строится в течение нескольких итераций фазы развития. На каждой итерации к модели добавляются концептуальные классы рассматриваемых сценариев, а не делается попытка "объять необъятное" и сразу же построить исчерпывающую модель всех возможных концептуальных классов и их взаимоотношений. Например, если на данной итерации рассматривается упрощенный сценарий прецедента `Оформление продажи`, описывающий продажу товаров за наличный расчет, то в разрабатываемом фрагменте модели предметной области нужно отобразить понятия, относящиеся только к этому сценарию.

Основная задача — идентифицировать концептуальные классы, связанные с разрабатываемым сценарием.

При идентификации концептуальных классов целесообразно руководствоваться следующим принципом.

> Лучше излишне детализировать модель предметной области, чем недоопределить ее.

Не следует думать, что модель предметной области тем лучше, чем меньше в ней концептуальных классов. Истина состоит в обратном.

Зачастую на начальной стадии идентификации некоторые концептуальные классы упускаются из виду, а появляются позднее, при рассмотрении атрибутов и ассоциаций, или даже на стадии проектирования. Обнаруженные новые понятия добавляются в модель предметной области.

Не следует исключать из рассмотрения концептуальные классы только на том основании, что из анализа требований не следует очевидная необходимость их запоминания (этот критерий зачастую применяется при разработке реляционных баз данных, однако не годится для создания моделей предметной области) или концептуальный класс не имеет атрибутов.

Концептуальные классы без атрибутов вполне допустимы, как допустимы и концептуальные классы, играющие "поведенческую", а не информационную роль в предметной области.

### Стратегии идентификации концептуальных классов

В следующих разделах представлены два способа выявления концептуальных классов.

1. С использованием списка категорий концептуальных классов.
2. На основе выделения существительных.

Для моделирования предметной области применяется и другой замечательный прием — *шаблоны анализа*. Шаблоны — это готовые модели предметной области, созданные специалистами. Они широко описаны в литературе, например в [46] и [61].

### *Использование списка категорий концептуальных классов*

Приступая к созданию модели предметной области, целесообразно составить список кандидатов на роль концептуальных классов. В табл. 10.1 содержится множество стандартных категорий, которые обычно имеют важное значение. В этом перечне они приводятся в произвольном порядке и не упорядочены по степени важности. Примеры взяты из предметной области системы торговли и резервирования авиабилетов.

**Таблица 10.1. Список категорий концептуальных классов**

| **Категория концептуальных классов** | **Примеры** |
|---|---|
| Физические или материальные объекты | `Register` (Реестр),<br>`Airplane` (Самолет) |
| Спецификации, элементы проектных решений или описания объектов | `ProductSpecification` (Спецификация товара),<br>`FlightDescription` (Описание полета) |
| Места | `Store` (Магазин),<br>`Airport` (Аэропорт) |
| Транзакции | `Sale` (Продажа), `Payment` (Платеж),<br>`Reservation` (Резервирование) |
| Элементы транзакций | `SalesLineItem` (Элемент продажи) |
| Роли людей | `Cashier` (Кассир),<br>`Pilot` (Пилот) |
| Контейнеры других объектов | `Store` (Магазин), `Bin` (Бункер),<br>`Airplane` (Самолет) |
| Содержимое контейнеров | `Item` (Элемент),<br>`Passenger` (Пассажир) |
| Другие компьютеры или электромеханические системы, внешние по отношению к данной системе | `CreditPaymentAuthorizationSystem` (Система авторизации кредитных платежей),<br>`AirTrafficControl` (Система управления движением) |
| Абстрактные понятия | `Hunger` (Голод),<br>`Acrophobia` (Акрофобия) |
| Организации | `SalesDepartment` (Отдел продаж),<br>`ObjectAirline` (Авиалинии) |
| События | `Sale` (Продажа), `Payment` (Платеж), `Meeting` (Встреча),<br>`Flight` (Полет), `Crash` (Крушение), `Landing` (Приземление) |
| Процессы (зачастую *не* представляются в виде понятий) | `SellingAProduct` (Продажа продукта),<br>`BookingASeat` (Бронирование места) |
| Правила и политика | `RefundPolicy` (Правила возврата товара)<br>`CancellationPolicy` (Политика аннулирования заказа) |
| Каталоги | `ProductCatalog` (Каталог товаров),<br>`PartsCatalog` (Каталог частей) |
| Записи финансовой, трудовой, юридической и другой деятельности | `Receipt` (Чек), `Ledger` (Гроссбух), `EmploymentContract` (Трудовой контракт),<br>`MaintenanceLog` (Журнал обслуживания) |
| Финансовые инструменты и службы | `LineOfCredit` (Кредитная линия),<br>`Stock` (Акция) |
| Руководства, документы, статьи, книги | `DailyPriceChangeList` (Бюллетень ежедневного изменения цен),<br>`RepairManual` (Руководство по восстановлению) |

### *Определение концептуальных классов с помощью выявления существительных*

Еще один полезный (и очень простой) прием идентификации концептуальных классов на основе лингвистического анализа предлагается в [1]. Он состоит в выделении существительных из текстовых описаний предметной области и их выборе в качестве кандидатов в концептуальные классы или атрибуты.

> Этот метод следует применять с осторожностью. Между существительными и концептуальными классами нет взаимно однозначного соответствия, а слова естественного языка могут иметь по несколько значений.

Тем не менее это информация к размышлению. Для реализации подобного подхода удобно использовать развернутые описания прецедентов, например, основной сценарий прецедента `Оформление продажи`.

**Основной успешный сценарий (или основной процесс)**

1. Покупатель подходит к **кассовому аппарату POS-системы** с выбранными **товарами**.
2. **Кассир** открывает новую **продажу**.
3. **Кассир** вводит **идентификатор товара**.
4. Система записывает **наименование товара** и выдает его **описание**, **цену** и **общую стоимость**. Цена вычисляется на основе набора правил.

Кассир повторяет действия, описанные в пп. 3–4 для каждого наименования товара.

5. Система вычисляет общую стоимость покупки с **налогом**.
6. Кассир сообщает покупателю общую стоимость и предлагает оплатить покупку.
7. Покупатель оплачивает покупку, система обрабатывает **платеж**.
8. Система регистрирует **продажу** и отправляет информацию о ней внешней **бухгалтерской системе** (для обновления бухгалтерских документов и начисления **комиссионных**) и **системе складского учета** (для обновления данных).
9. Система выдает **товарный чек**.
10. Покупатель покидает магазин с чеком и товарами (если он что-то купил).

**Расширения (или альтернативные потоки)**

...

7а. Оплата наличными

1. Кассир вводит предложенную покупателем **сумму**.
2. Система вычисляет положенную **сдачу** и открывает **кассу с наличностью**.
3. Кассир складывает поученные деньги и выдает сдачу покупателю.
4. Система регистрирует платеж наличными.

Модель предметной области — это визуализация важных понятий из словаря предметной области. Откуда брать необходимые термины? Из описания прецедентов. Эти описания — богатый источник для идентификации существительных.

Одни из этих существительных могут быть представлены в виде концептуальных классов, другие могут представлять концептуальные классы, не имеющие отношения к данной итерации (например, "бухгалтерская система" или "комиссионные"), а третьи — в виде атрибутов этих концептуальных классов. Более подробная информация об отличиях между концептуальными классами и атрибутами содержится в следующем разделе и следующей главе.

Недостатком данного подхода является выразительность естественного языка. Для описания одного и того же концептуального класса или атрибута могут использоваться различные существительные, и в то же время некото-

рые существительные могут иметь по несколько значений. Тем не менее этот подход рекомендуется использовать в сочетании с методом выбора понятий по списку категорий.

## 10.3. Кандидатуры на роль концептуальных классов для предметной области торговли

Пользуясь списком категорий и методом анализа словесного описания, мы составили список кандидатур на роль концептуальных классов для предметной области розничной торговли. Он соответствует требованиям и принятым упрощениям для основного сценария прецедента `Оформление продажи`.

| | |
|---|---|
| Register | ProductSpecification |
| Item | SalesLineItem |
| Store | Cashier |
| Sale | Customer |
| Payment | Manager |
| ProductCatalog | |

Не существует понятия "правильный" список. Это просто произвольный набор абстракций и понятий из словаря предметной области, которые, по мнению разработчика модели, являются важными. Тем не менее, если для выделения концептуальных классов использовать описанные выше стратегии, то различные специалисты по моделированию составят примерно одинаковые списки.

### *Объекты отчета: включать ли понятие "товарный чек" в модель*

Товарный чек — это документ, принимающий участие в продаже товара и его оплате. Он может рассматриваться как концептуальный класс из предметной области. Возникает вопрос: нужно ли его включать в модель предметной области?

Для ответа на этот вопрос необходимо учитывать следующие факторы.

- Товарный чек — это своеобразный отчет о сделанной покупке. В принципе, в модель предметной области не следует включать объекты отчета, поскольку вся содержащаяся в них информация получена из других источников. Это является одной из причин исключения понятия "чек" из модели предметной области.
- Товарный чек выполняет конкретную роль при реализации бизнес-правил: обычно он обеспечивает право на возврат товара. Этот фактор говорит в пользу включения товарного чека в модель предметной области.

Поскольку возврат товара не рассматривается на данной итерации разработки, понятие `Receipt` (Товарный чек) не включается в модель предметной области. Оно будет включено в модель системы на той итерации разработки, на которой будет реализовываться прецедент `Возврат товара`.

## 10.4. Принципы создания модели предметной области

### *Как создать модель предметной области*

Для создания модели предметной области выполните следующие действия.

1. Составьте список кандидатов на роль концептуальных классов на основе списка категорий и метода анализа текстового описания для текущей итерации разработки.
2. Отобразите их в модели предметной области.
3. Добавьте необходимые ассоциации, отражающие связи, для которых требуется выделение памяти (обсуждается в следующей главе).
4. Добавьте атрибуты, необходимые для выполнения информационных требований (обсуждается в следующей главе).

Полезно также изучить и скопировать шаблоны анализа, описываемые в следующей главе.

### Имена и модели: стратегия построения карт

При построении моделей предметной области применяется та же стратегия, что и при создании карт.

Модель предметной области следует создавать согласно принципам картографии.

- Использовать применяемые на данной территории названия
- Исключать несущественные детали
- Не добавлять объекты, которые отсутствуют на данной территории

Модель предметной области — это своеобразная разновидность карты понятий некоторой предметной области. Отсюда следуют аналитическая роль модели предметной области, а также справедливость следующих замечаний.

- Картографы используют названия, применяемые на данной территории. Они не изменяют названия городов на карте. С точки зрения модели предметной области это означает необходимость *использования словаря предметной области при именовании концептуальных классов и атрибутов*. Например, при разработке модели библиотеки в качестве понятия, означающего потребителя, следует выбрать `Borrower` (Читатель), т.е. использовать терминологию библиотекарей.
- Картограф не наносит на карту объекты, не имеющие отношения к основному ее назначению, например не отображает топографию или состав населения. Аналогично, в модели предметной области не должны содержаться понятия из предметной области, не имеющие отношения к требованиям. Например, из нашей модели предметной области можно исключить понятия `Pen` (Ручка) и `PaperBag` (Папка), поскольку они не имеют отношения к требованиям.
- Картограф не отображает на карте отсутствующие объекты, например, горы на карте равнинной местности. Точно так же, в модели предметной области не должны содержаться понятия, не имеющие отношения к рассматриваемой проблеме.

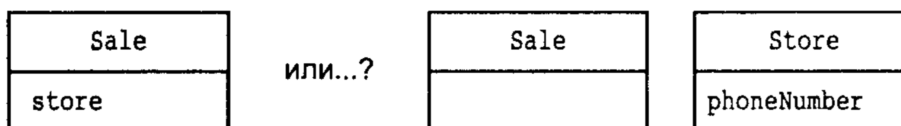
Этот принцип называют также стратегией использования словаря предметной области [30].

### *Типичная ошибка при выделении концептуальных классов*

Возможно, наиболее типичной ошибкой при создании модели предметной области является отнесение некоторого объекта к атрибутам, в то время как он должен относиться к концептуальным классам. Чтобы избежать этой ошибки, следует придерживаться простого правила.

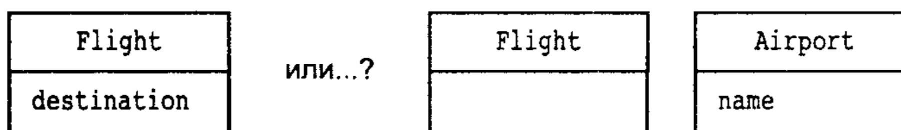
> Если некоторый объект $X$ в реальном мире не является числом или текстом, значит, это скорее концептуальный класс, чем атрибут.

Например, является ли `store` (магазин) атрибутом объекта `Sale` (Продажа) или отдельным концептуальным классом `Store`?




В реальном мире магазин не является числом или текстом, он представляет реальную сущность, организацию, занимающую некоторое место. Следовательно, `Store` нужно рассматривать в качестве концептуального класса.

В качестве другого примера рассмотрим предметную область системы резервирования авиабилетов. Нужно ли рассматривать место назначения как атрибут `destination` понятия `Flight` (Полет) или как отдельное понятие `Airport` (Аэропорт)?




В реальном мире аэропорт места назначения не является ни текстом, ни числом: это массивный объект, занимающий определенное пространство. Следовательно, в модели предметной области он должен быть представлен понятием `Airport`.

> Если у вас возникают сомнения при разграничении понятий и атрибутов, создавайте отдельный концептуальный класс. Атрибуты редко отображаются в модели предметной области.

## 10.5. Разрешение конфликта сходных концептуальных классов: `Register` или POST

POST означает терминал торговой точки (point-of-sale terminal). В компьютерном мире терминал — это любое устройство или система для ввода и вывода информации, например клиентский компьютер, беспроводное сетевое устройство PDA и т.д. В старые времена, задолго до появления терминалов POST, в каждом магазине вели реестр — книгу, в которой регистрировались все продажи и платежи. Со временем этот процесс был автоматизирован с помощью механических кассовых аппаратов. На сегодняшний день роль реестра выполняет терминальная система розничной торговли (рис. 10.6).

Сходные понятия с различными именами

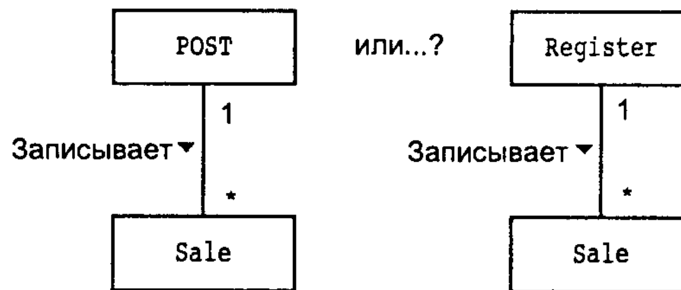



*Рис. 10.6. Концептуальные классы* `POST` *и* `Register` *аналогичны*

Таким образом, реестр — это объект, в который записываются сведения о продажах и платежах. Такую же функцию выполняет и терминальная система розничной торговли. Однако термин "реестр" является несколько более абстрактным, чем POST, и не ориентированным на реализацию. Так может быть в модели предметной области вместо концептуального класса `POST` следует использовать понятие `Register` (Реестр)?

> Необходимо помнить, что модель предметной области не может быть абсолютно корректной или абсолютно ложной, но может быть полезной в большей или меньшей степени. Она представляет собой средство общения.

Согласно одному из принципов картографии, термин "POST" более распространен на данной территории, поэтому с точки зрения общения символьное обозначение `POST` является более полезным. Для создания абстрактной модели, не зависящей от конкретной реализации, более предпочтительным и полезным является концептуальный класс `Register`[5].

Достойны внимания оба понятия. Мы остановились на понятии `Register`, но можно было бы воспользоваться и концептуальным классом `POST`.

## 10.6. Моделирование "нереального" мира

Предметная область некоторых приложений имеет очень слабое отношение к реальному миру. Примерами таких приложений могут быть системы телекоммуникаций. Для подобных систем тоже можно создавать модели предметной области, но для этого требуется высокая степень абстракции и отход от стандартных принципов разработки.

Например, в качестве концептуальных классов, связанных с предметной областью системы телекоммуникаций, могут выступать `Message` (Сообщение), `Connection` (Соединение), `Dialog` (Диалог), `Road` (Маршрут) и `Protocol` (Протокол).

---

[5] Заметим, что в прежние времена реестр представлял собой лишь одну из возможных реализаций системы записей о проданных товарах. Со временем этот термин приобрел обобщенное значение.

## 10.7. Спецификация или описание концептуальных классов

Рассматриваемые в данном разделе вопросы, на первый взгляд, могут показаться достаточно частными. Однако необходимость спецификации концептуальных классов возникает для множества различных предметных областей. На этом и будет сделан основной акцент.

Примем следующие допущения.

- Термин Item (Товар) представляет физический товар в магазине, а значит, он может иметь серийный номер.
- Понятию Item соответствуют описание, цена и идентификатор, которые более нигде не записываются.
- Каждый сотрудник магазина страдает амнезией.
- При каждой продаже реального физического товара удаляется соответствующий программный экземпляр Item.

Что произойдет при выполнении следующего сценария при сделанных предположениях?

В магазине повысился спрос на новую разновидность вегетарианских горячих бутербродов ObjectBurger. Идет бойкая торговля, и при продаже каждого объекта ObjectBurger экземпляр Item удаляется из памяти компьютера.

Однако возникает проблема, связанная с тем, что никто не может ответить на вопрос "Сколько стоит ObjectBurger?", поскольку его цена связана с каждым экземпляром, который удаляется из памяти в случае его продажи.

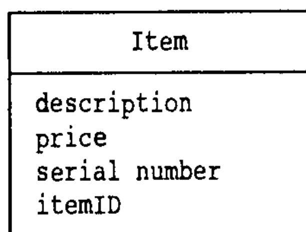
Заметим также, что при программной реализации описанной модели происходит дублирование данных, и память используется неэффективно, поскольку описание, цена и идентификатор товара связаны с каждым экземпляром одного и того же товара.

### *Необходимость спецификаций или описание концептуальных классов*

Описанная проблема иллюстрирует необходимость использования понятий, представляющих собой спецификации или описание других объектов. Для решения этой проблемы необходимо ввести концептуальный класс ProductSpecification (Спецификация товара) (или ItemSpecification, ProductDescription и т.д.), содержащий информацию о товаре. Понятие ProductSpecification не совпадает с понятием Item, а представляет собой лишь описание этого товара. Заметим, что даже после продажи всех единиц товара и удаления из памяти всех программных элементов Item понятие ProductSpecification остается в памяти компьютера.

Объекты описаний или спецификации тесно связаны с описываемыми понятиями. В модели предметной области связь между ними принято обозначать следующим образом: XSpecification описывает X (рис. 10.7).

Понятия-спецификации обычно используются при описании предметной области, связанной с продажами или товарами. Они также часто используются при описании производства, для обозначения отличий одних производимых товаров от других. Эти понятия идентифицируются одними из первых, поскольку являются очень типичными.



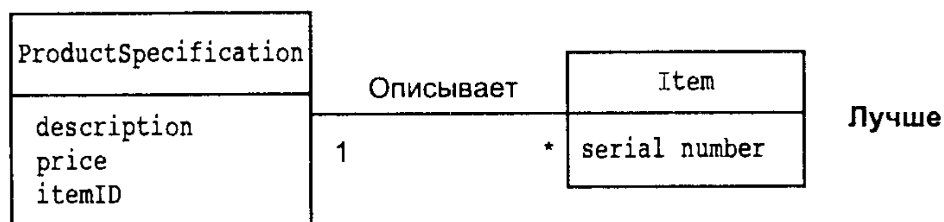




*Рис. 10.7. Спецификации или описания других понятий. Символ * означает "множественную" связь и указывает на то, что одно понятие ProductSpecification может описывать много (*) понятий Item*

## Когда требуются понятия-спецификации

Рассматривая вопрос о необходимости введения понятий-спецификаций, нужно принимать во внимание следующие соображения.

> Концептуальный класс спецификации или описания (например, ProductSpecification) вводится в таких случаях.
>
> - Существует необходимость описания элементов или служб независимо от существования конкретных экземпляров этих объектов.
> - Если удаление экземпляров описываемых им понятий (например, Item) приводит к потере важной информации в связи с некорректной ассоциацией этой информации с удаляемым экземпляром.
> - Если при наличии понятия устраняется дублирование информации.

## Еще один пример спецификации

В качестве следующего примера рассмотрим авиакомпанию, которая пострадала в связи с крушением одного из ее самолетов. Предположим, что все рейсы этой авиакомпании отменены на шесть месяцев до завершения расследования причин авиакатастрофы. Предположим также, что после отмены рейсов все программные объекты Flight (Полет) были удалены из памяти компьютера.

Если сведения о полетах связаны только с экземплярами объектов Flight, которые представляют конкретные рейсы, выполняемые в конкретное время, то после крушения самолета в авиакомпании не останется никакой информации об осуществляемых ею рейсах.

Для решения этой проблемы необходимо ввести понятие FlightDescription (Описание полета) или FlightSpecification (Спецификация полета), описывающее абстрактный рейс без привязки к дате и времени (рис. 10.8).

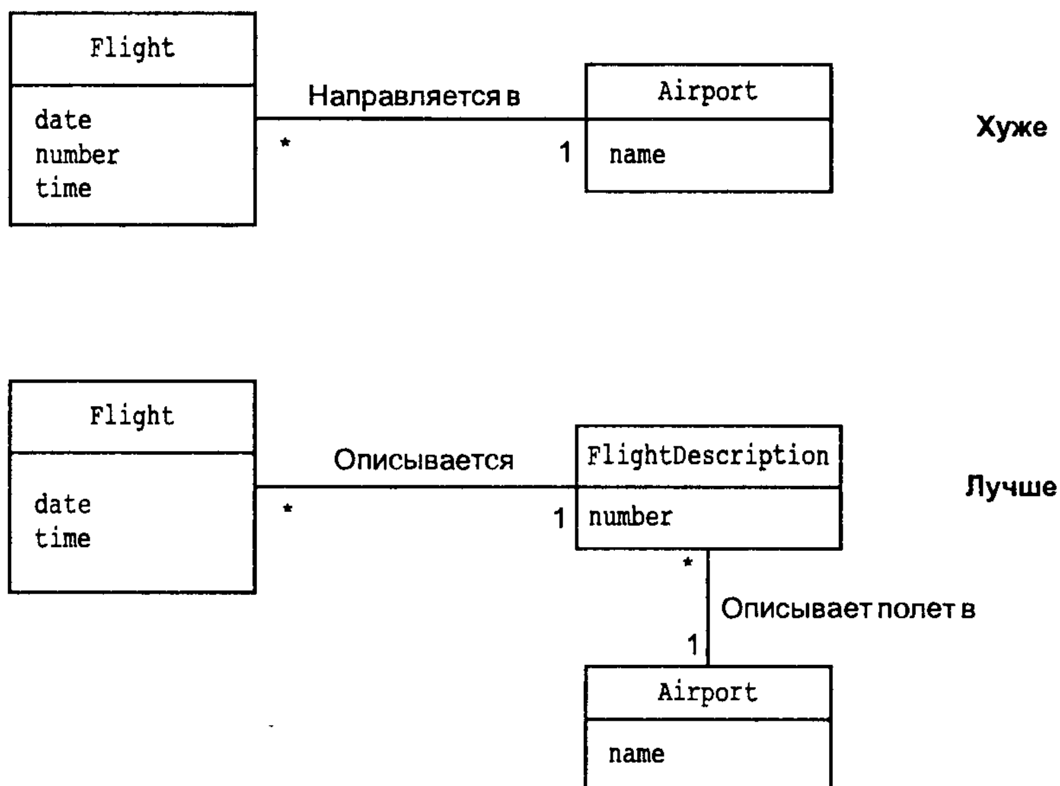



*Рис. 10.8. Спецификации различных объектов*

### *Описания услуг*

Обратите внимание, что последний пример относится к услугам (полетам), а не к товарам (вегетарианским горячим бутербродам). Описания услуг играют не менее важную роль.

В качестве следующего примера рассмотрим телекоммуникационную компанию по продаже пакетов услуг мобильной телефонной связи наподобие "бронзовый", "золотой" и т.д. Такой компании необходимо описание пакета (некоторый текст, описывающий стоимость минуты разговора, поддержку выхода в Internet, стоимость этой услуги и т.д.). Его нужно хранить отдельно от реально проданных пакетов (например, пакет "золотой" продан Крэгу Ларману 1 января 2002 года за $55 в месяц). С точки зрения маркетологов, для описания этой услуги понятие `MobileCommunicationsPackageDescription` нужно определить еще до начала продаж реальных пакетов.

## 10.8. Термины языка UML, модели и методы: различные ракурсы

В контексте UP определено понятие модели предметной области, иллюстрируемой с помощью систем обозначений языка UML. Однако в официальной документации по UML термин "модель предметной области" отсутствует. Это указывает на следующий важный момент.

> Язык UML просто описывает типы диаграмм, например диаграммы классов или последовательностей. Он не привязан ни к какому методу моделирования или ракурсу моделирования. В отличие от этого, в рамках процесса (например, UP) обозначения языка UML применяются в контексте методологически определенной модели.

Например, обозначения для диаграммы классов языка UML можно использовать для обозначения концептуальных классов или понятий (в модели предметной области), программных классов, таблиц реляционных баз данных и т.д.

Поэтому не нужно путать базовую систему обозначений языка UML с его применениями к визуализации различных видов моделей, определенных специалистами по методологии (рис. 10.9). Это относится не только к элементам диаграммы классов, но и к большинству других обозначений языка UML.

Рассмотрим еще один пример диаграмм, которые по разному интерпретируются в различных моделях. Диаграммы последовательностей языка UML можно использовать для иллюстрации передачи сообщений между программными объектами (например, в модели проектирования UP) или взаимодействия между людьми и частями реального мира (как в объектной бизнес-модели UP).

Такой подход декларируется в рамках одного из объектно-ориентированных методов [26] и Мартином Фовлером (Martin Fowler) в [49]. То есть одни и те же обозначения элементов диаграмм можно применять для построения трех ракурсов и типов моделей.

1. **Концептуальный аспект** (conceptual perspective) — диаграммы описывают сущности реального мира или предметной области.
2. **Аспект спецификации** (specification perspective) — диаграммы (с использованием обозначений, принятых в рамках концептуального аспекта) описывают программные абстракции или компоненты со своими спецификациями и интерфейсами, однако без привязки к конкретной реализации (например, без конкретного соответствия языку C# или Java).
3. **Аспект реализации** (implementation perspective) — диаграммы (с использованием обозначений, принятых в рамках концептуального аспекта) интерпретируются как описания программной реализации на базе конкретной технологии или языка (например, Java).

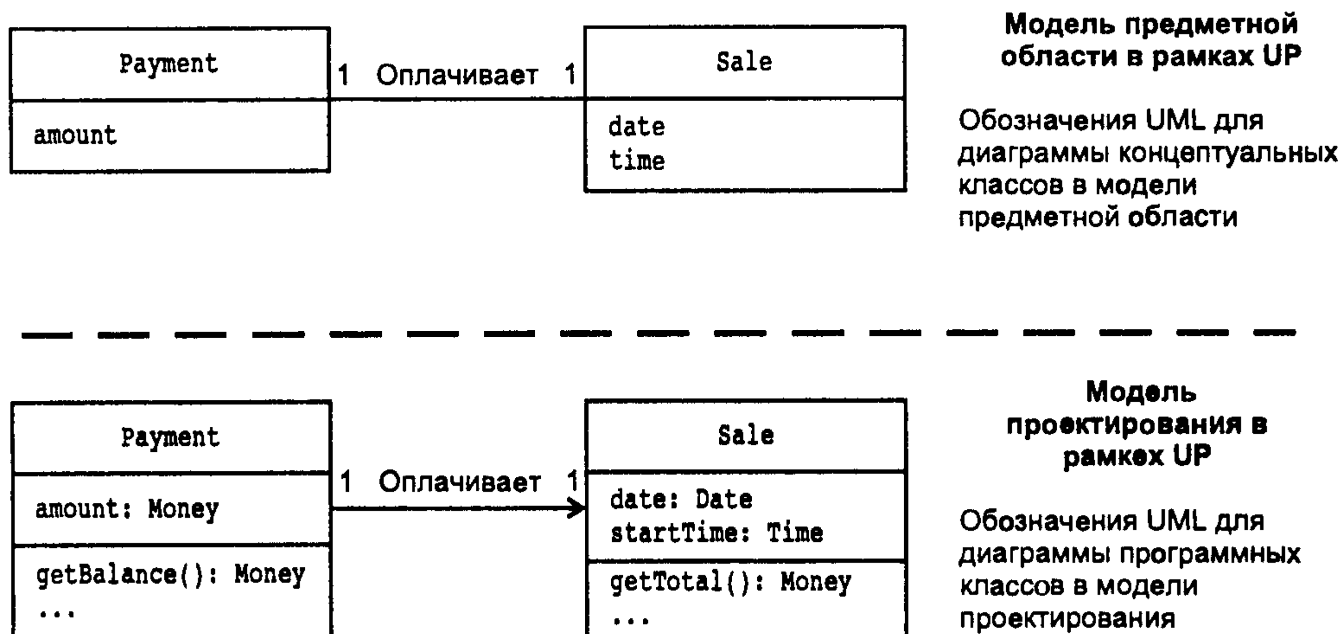



*Рис. 10.9. Система обозначений языка UML применима к различным аспектам и моделям, определенным в рамках некоторого процесса или метода*

### *Согласование терминологии, принятой в различных методах и UML*

На языке UML прямоугольники, показанные на рис. 10.9, обозначают *классы*. Однако этот термин может означать различные феномены: физические понятия, программные компоненты, события и т.д.[6] Наряду с обозначениями языка UML в рамках конкретного процесса или метода используется альтернативная терминология. Например, в контексте UP прямоугольные обозначения UML, представленные в модели предметной области, могут называться *понятиями предметной области* или *концептуальными классами*, поскольку модель предметной области обеспечивает концептуальный ракурс рассмотрения. Если эти же обозначения UML использованы в модели проектирования, то в контексте UP они официально называются *классами проектирования*, поскольку модели проектирования соответствует ракурс спецификации или реализации.

Однако независимо от формального определения следует отличать терминологию анализа предметной области, оперирующую понятиями реального мира (в концептуальном ракурсе), такими как продажа, от программных сущностей наподобие программного класса `Sale` (в ракурсе спецификации или реализации).

Язык UML можно использовать для иллюстрации обоих ракурсов. При этом используются аналогичная система обозначений и терминология. Поэтому очень важно постоянно помнить о том, в каком ракурсе рассматривается задача в данный момент.

Рассмотрим некоторые термины, связанные с унифицированным процессом разработки и используемые в UML.

- *Концептуальный класс* — понятие из реального мира. Рассматривается в концептуальном ракурсе. В рамках UP модель предметной области содержит концептуальные классы.
- *Программный класс* — класс, представляющий спецификацию или реализацию программного компонента, независимо от процесса или метода.
- *Класс проектирования* — элемент модели проектирования UP. Это синоним программного класса, однако по определенным причинам автору хотелось акцентировать внимание на том, что данный класс имеет отношение к модели проектирования. В контексте UP классы проектирования относят либо к ракурсу спецификации, либо к реализации.
- *Класс реализации* — класс, реализованный на объектно-ориентированном языке, например на Java.
- *Класс* — в языке UML — это общий класс, представляющий либо понятие реального мира (концептуальный класс), либо программную сущность (программный класс).

---

[6] В контексте UML, класс — это специальный вид достаточно общего элемента модели, получившего название *классификатора* (classifier). Классификатор — это сущность со структурными свойствами и/или поведением, в частности, класс, исполнитель, интерфейс и прецедент.

## 10.9. Сокращение разрыва в представлениях

Обратимся к рис. 10.10. Почему во многих книгах при обсуждении вопросов проектирования объектов рассматриваются только программные классы, имена которых отражают понятия предметной области? Почему для программного класса выбирается имя `Sale` и что оно означает?

В целом выбор имен, отражающих понятия предметной области, улучшает восприятие программы и понимание предназначения программного класса `Sale`. У каждого человека уже сформировались определенные понятия из конкретной предметной области (например о том, что в магазине продаются товары). Из жизни мы знаем, что каждой продаже соответствует дата. Следовательно, создавая на языке Java программный класс `Sale`, которому соответствует реальная продажа и время, мы привносим в программную реализацию свои интуитивные представления о предметной области.

> Модель предметной области — это визуальный словарь понятий, на основании которого можно сформировать имена некоторых программных элементов.

Данный вопрос связан с семантическим разрывом между воображаемой моделью предметной области и ее представлением в программном обеспечении.

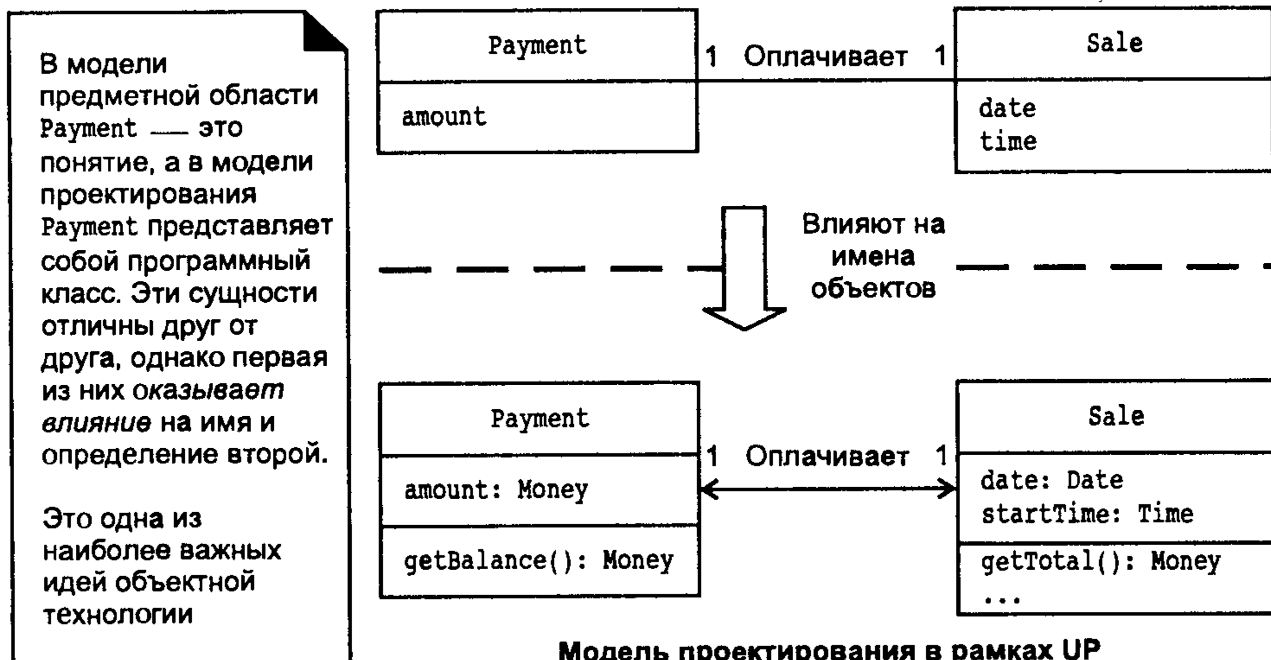



*Рис. 10.10. В объектном проектировании и программировании зачастую создаются программные классы, имена и содержание которых соответствуют сущностям реального мира*

Можно представить себе утрированную ситуацию, когда POS-система NextGen разрабатывается напрямую в двоичном коде с использованием набора инструкций процессора. При этом семантический разрыв чрезвычайно велик, хотя его масштабы очень сложно оценить, поскольку такую программу трудно соотнести с предметной областью проблемы. Альтернативой является использование объектных технологий, при которых имена классов в точности отражают понятия предметной области. Если в реальном мире существует сущность (или событие) "продажа", то в программной продаже ему соответствует класс Sale. При таком однозначном соответствии словаря предметной области словарю программных элементов семантический разрыв сокращается. При этом облегчается понимание существующего кода (поскольку он работает именно так, как ожидается исходя из знания предметной области) и обеспечивается возможность естественного расширения кода. То есть программная модель напоминает концептуальную модель и работает соответствующим образом.

Такое сокращение семантического разрыва между программной и концептуальной моделями имеет свое практическое преимущество. Большинство разработчиков программного обеспечения знают об этом, хотя и не всегда по достоинству его оценивают. Действительно, специалисты знают, что из байт-кода Java очень сложно путем обратного проектирования получить исходный код, поскольку в байт-коде имена классов и методов неразборчивы и не соответствуют представлениям о предметной области.

Конечно же, объектная технология обладает и другими преимуществами. Она позволяет разрабатывать "элегантные", легко масштабируемые системы, о чем речь пойдет в последующих главах. Сокращения семантического разрыва — это полезное, но второстепенное преимущество объектного подхода по сравнению с легкостью модификации и расширения, а также поддержкой борьбы со сложностью.

## 10.10. Пример: модель предметной области POS-системы NextGen

Список концептуальных классов предметной области POS-системы NextGen можно представить графически (рис. 10.11) в рамках модели предметной области.

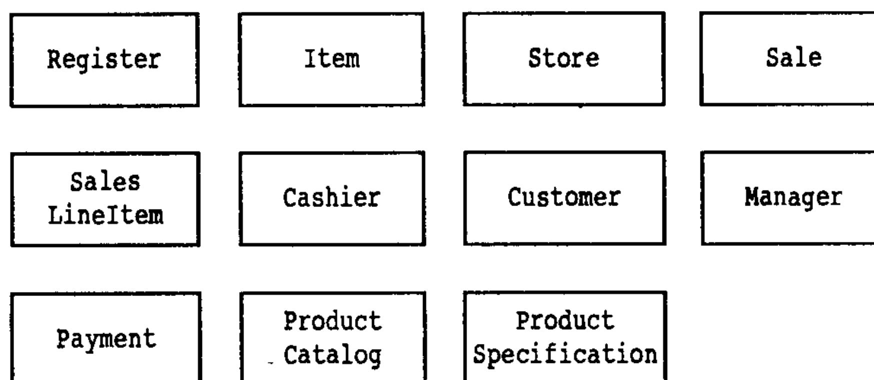



*Рис. 10.11. Исходная модель предметной области*

Атрибуты и ассоциации, связанные с этой моделью предметной области, будут рассмотрены в последующих главах.

## 10.11. Модели предметной области в рамках UP

Как видно из табл. 10.2, начало и завершение модели предметной области приходится на фазу развития.

### Начальная фаза

Модели предметной области не рассматриваются на начальной стадии, поскольку на этом этапе не выполняются серьезные исследования, а принимается решение о целесообразности выполнения более глубоких исследований на стадии развития.

**Таблица 10.2. Пример планирования сроков реализации артефактов UP (н — начало, р — развитие)**

| Дисциплина | Артефакт<br>Итерация→ | Начало<br>I1 | Развитие<br>E1..En | Конструирование<br>C1..Cn | Передача<br>T1..Tn |
|---|---|---|---|---|---|
| Бизнес-моделирование | Модель предметной области | | н | | |
| Требования | Модель прецедентов | н | р | | |
| | Видение системы | н | р | | |
| | Дополнительная спецификация | н | р | | |
| | Словарь терминов | н | р | | |
| Проектирование | Модель проектирования | | н | р | |
| | Описание архитектуры | | н | | |
| | Модель данных | | н | р | |
| Реализация | Модель реализации | | н | р | р |
| Управление проектом | План разработки | н | р | р | р |
| Тестирование | Модель тестирования | | н | р | |
| Окружение | Набор документов | н | р | | |

### Фаза развития

Модель предметной области в основном создается в процессе нескольких итераций фазы развития, когда появляется необходимость понять сущности реального мира и отразить их в процессе проектирования в программные классы.

И хотя вопросу моделирования объектов предметной области был посвящен значительный объем материала, в опытных руках разработчиков построение модели предметной области (частичной и постоянно расширяемой) на каждой итерации занимает не более нескольких часов. При использовании стандартных шаблонов анализа этот процесс еще более ускоряется.

### Объектная бизнес-модель и модель предметной области в рамках UP

Модель предметной области — это официальный вариант менее распространенной объектной бизнес-модели UP (UP Business Object Model — UP BOM). Модель UP BOM не нужно путать с моделью BOM, определенной в рамках других методов другими авторами. Модель UP BOM — это вид модели предприятия, используемый для описания всей его экономической деятельности. Ее можно ис-

пользовать для проектирования бизнес-процессов или их обратного проектирования независимо от каких-либо программных приложений (таких как POS-система NextGen). Приведем цитату.

> "[Модель UP BOM] служит абстракцией связи и взаимодействия бизнес-сущностей и сотрудников в процессе экономической деятельности." [97]

Модель BOM состоит из нескольких различных диаграмм (классов, видов деятельности и последовательностей), иллюстрирующих экономические процессы в рамках всего предприятия. Она наиболее полезна для проектирования бизнес-процессов, однако такая деятельность менее типична, чем разработка отдельных программных приложений.

Поэтому в UP модель предметной области определена как более типичный артефакт или специализация модели BOM. Приведем цитату.

> "Вы можете разработать "неполную" объектную бизнес-модель, сконцентрировав внимание на пояснении сущностей и понятий, важных для данной предметной области. ...Такую модель часто называют моделью предметной области." [97]

## 10.12. Дополнительная литература

В [81] рассмотрены вводные вопросы концептуального моделирования предметной области. Для изучения этих же вопросов также полезна книга [26].

В книге Фовлера [46] рассматриваются шаблоны для построения моделей предметной области. Еще одной хорошей книгой с точки зрения описания шаблонов для моделей предметной области является [61]. В ней приводятся советы экспертов в области моделирования данных, которые хорошо понимают различие между чисто концептуальными моделями и схемами баз данных, что очень полезно при моделировании объектов предметной области.

Хорошей книгой по моделированию предметной области является [27]. Авторы этой книги выделяют стандартные шаблоны типов и ассоциаций. При этом цвета, как следует из названия этой книги, действительно используются для визуализации стандартных категорий этих типов. Так, описания выделяются голубым цветом, роли — желтым, а временн*ы*е сущности — розовым. Выделение цветом используется для большей наглядности шаблонов.

В рамках объектно-ориентированного анализа используются достаточно сложные лингвистические приемы, получившие название моделирования естественного языка. Об этом рассказывается, например в [82].

## 10.13. Артефакты UP

Взаимосвязи артефактов UP в контексте модели предметной области показаны на рис. 10.12.

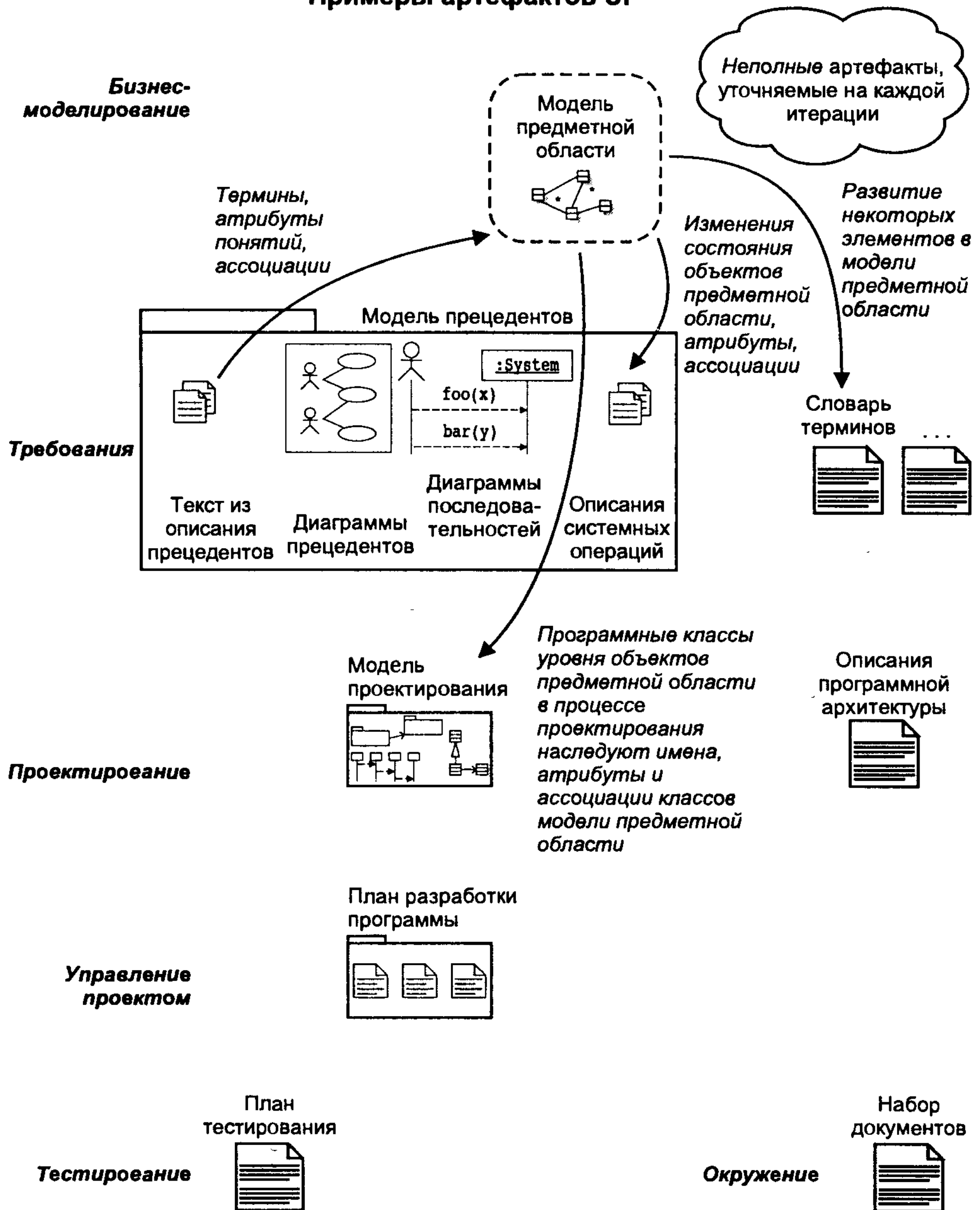



*Рис. 10.12. Пример взаимосвязи артефактов UP*

Глава

# Модель предметной области: добавление ассоциаций

**Основные задачи**

- Определить ассоциации в рамках модели предметной области.
- Выделить важные и второстепенные ассоциации.

## Введение

В процессе разработки модели предметной области необходимо идентифицировать связи (ассоциации) между концептуальными классами, удовлетворяющие информационным требованиям разрабатываемых на текущей итерации сценариев, а также выделить те из них, которые способствуют лучшему пониманию модели предметной области. В данной главе сначала рассматривается процесс определения соответствующих ассоциаций, а затем полученные навыки применяются к добавлению ассоциаций в модель предметной области POS-системы NextGen.

## 11.1. Ассоциации

*Ассоциация* (association) — это связь между типами (или точнее, экземплярами типов), отражающая некоторое значимое и полезное отношение между ними (рис. 11.1).

В языке UML ассоциации описываются как "семантические взаимосвязи между двумя или несколькими классификаторами и их экземплярами".

*Рис. 11.1. Ассоциации*

### *Поиск ассоциаций*

Заслуживающие внимания ассоциации обычно содержат знания о взаимосвязи между объектами, которые должны сохраняться в течение некоторого периода. Этот период может измеряться в миллисекундах или годах в зависимости от конкретного контекста. Другими словами, о связи между какими объектами нужно помнить? Например, нужно ли помнить о том, что экземпляры объекта `SalesLineItem` (Элемент продажи) ассоциированы с экземпляром объекта `Sale` (Продажа)? Очевидно да, поскольку в противном случае будет невозможно восстановить данные о продаже, распечатать товарный чек или вычислить итоговую сумму.

> В модель предметной области целесообразно включать следующие ассоциации.
>
> - Ассоциации, знания о которых нужно сохранять в течение некоторого периода (важные ассоциации).
> - Ассоциации, производные от содержащихся в списке стандартных ассоциаций.

Возникает еще один вопрос: нужно ли сохранять информацию о связи между текущим экземпляром объекта `Sale` и `Manager`? Нет, не каждая связь является необходимой. Связь между объектами `Sale` и `Manager` нельзя считать неверной, однако в контексте наших требований она не является ни обязательной, ни полезной.

Это достаточно важный момент. Если в модели предметной области содержится N различных концептуальных классов, то между ними можно установить N*(N-1) ассоциацию, а это может быть достаточно большое число. Многие линии связей такой диаграммы будут просто вносить визуальный шум и ухудшать ее наглядность. Поэтому при добавлении ассоциаций нужно придерживаться принципа минимализма. Критерии необходимости ассоциаций будут предложены ниже в этой главе.

## 11.2. Система обозначений для ассоциаций языка UML

Ассоциация обозначается проведенной между классами линией, с которой связано определенное имя. Обычно ассоциация является двунаправленной. Это означает, что от одного объекта любого типа возможен логический переход к другому объекту. Такой переход является абсолютно абстрактным. Он не определяет тип взаимосвязей между программными сущностями.

На концах линии, которая обозначает ассоциацию, могут содержаться выражения, определяющие количественную связь между экземплярами классов.

Дополнительная стрелка рядом с именем ассоциации указывает, в каком направлении нужно читать ее имя. Она не определяет направление видимости или перемещения.

Если такая стрелка отсутствует, то имена ассоциаций следует читать с использованием общепринятых соглашений, а именно — слева направо и сверху вниз. Однако в языке UML в явной форме это правило отсутствует (рис. 11.2).

> Стрелка направления чтения не имеет семантического значения. Она лишь указывает направление чтения диаграмм.

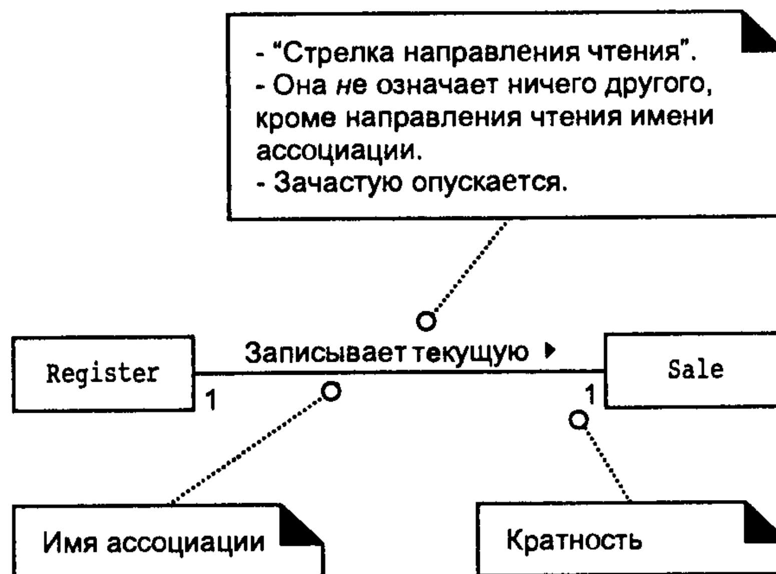



*Рис. 11.2. Система обозначения ассоциаций в языке UML*

## 11.3. Поиск ассоциаций: список стандартных ассоциаций

Приступим к добавлению ассоциаций с использованием списка, представленного в табл. 11.1.

В нем указаны стандартные категории, которыми обычно не следует пренебрегать. Примеры ассоциаций взяты из предметной области резервирования авиабилетов и розничной торговли.

**Таблица 11.1. Список стандартных ассоциаций**

| Категория | Примеры |
|---|---|
| А является физической частью В | `Drawer-Register` (Устройство печати торговых чеков-Реестр)<br>`Wing-Airplane` (Крыло-Самолет) |
| А является логической частью В | `SalesLineItem-Sale` (Элемент продажи-Продажа)<br>`FlightLeg-FlightRoute` (Отрезок пути-Маршрут полета) |
| А физически содержится в/на В | `Register-Store` (Реестр-Магазин), `Item-Shelf` (Товар-Полка)<br>`Passenger-Airplane` (Пассажир-Самолет) |
| А логически содержится в В | `ItemDescription-Catalog` (Описание товара-Каталог)<br>`Flight-FlightSchedule` (Полет-График полетов) |
| А является описанием В | `ItemDescription-Item` (Описание товара-Товар)<br>`FlightDescription-Flight` (Описание полета-Полет) |
| А является элементом транзакции или отчета В | `SalesLineItem-Sale` (Элемент продажи-Продажа)<br>`MaintenanceJob-MaintenanceLog` (Услуга по техническому обслуживанию-Журнал обслуживания) |
| А известен/ зарегистрирован/ записан/включен в В | `Sale-Register`(Продажа-Реестр)<br>`Reservation-FlightManifest` (Заказ билета-Декларация) |
| А является членом В | `Cashier-Store` (Кассир-Магазин)<br>`Pilot-Airline` (Пилот-Самолет) |
| А является организационной единицей В | `Department-Store` (Отдел-Магазин)<br>`Maintenance-Airline` (Служба поддержки-Самолет) |

| Категория | Примеры |
|---|---|
| А использует или управляет В | `Cashier-Register` (Кассир-Реестр)<br>`Pilot-Airplane` (Пилот-Самолет) |
| А взаимодействует с В | `Customer-Cashier` (Покупатель-Кассир)<br>`ReservationAgent-Passenger` (Агент по резервированию билетов-Пассажир) |
| А связан с транзакцией В | `Customer-Payment` (Покупатель-Платеж)<br>`Passenger-Ticket` (Пассажир-Билет) |
| А является транзакцией, которая связана с другой транзакцией В | `Payment-Sale` (Платеж-Продажа)<br>`Reservation-Cancellation` (Заказ билета-Отмена заказа) |
| А следует за В | `SalesLineItem-SalesLineItem` (Наименование товара-Следующее наименование товара)<br>`City-City` (Город-Город) |
| А является "собственностью" В | `Register-Store` (Реестр-Магазин)<br>`Plane-Airplane` (Крыло-Самолет) |
| А является событием, связанным с В | `Sale-Customer` (Продажа-Покупатель), `Sale-Store` (Продажа-Магазин), `Departure-Flight` (Отправление-Рейс) |

### *Ассоциации с высоким приоритетом*

В приведенной таблице имеются категории ассоциаций с высоким приоритетом, которые чрезвычайно полезно включать в модель предметной области.

- А является *физической* (physical) или *логической* (logical) частью В.
- А *физически* или *логически* содержится в/на В.
- А *записан* (recorded) в В.

## 11.4. Рекомендации по назначению ассоциаций

- Основное внимание нужно уделить тем ассоциациям, знания о которых нужно сохранять в течение некоторого периода (важным ассоциациям).
- Гораздо важнее определить концептуальные классы, чем выделить ассоциации.
- Слишком большое количество ассоциаций приводит к ошибкам в модели предметной области, а не к ее упрощению. Изучение ассоциаций не должно отнимать слишком много времени и вместе с тем должно приносить максимальную пользу.
- Избегайте отображения избыточных или не имеющих самостоятельного значения ассоциаций.

## 11.5. Роли

Каждый конец ассоциации называется *ролью* (role). Роль дополнительно может иметь следующие характеристики.

- Имя
- Кратность

- Направление связи

В данной главе будет рассмотрена кратность, а две другие характеристики ассоциации будут обсуждаться в последующих главах.

## Кратность

*Кратность* (multiplicity) определяет, сколько экземпляров класса А может быть ассоциировано с одним экземпляром класса В (рис. 11.3).

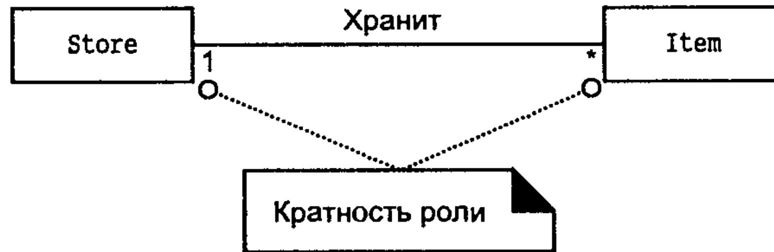



*Рис. 11.3. Кратность ассоциации*

Например, один экземпляр класса Store (Магазин) может быть ассоциирован с несколькими (ни с одним или с несколькими, что отображается символом *) экземплярами класса Item.

Некоторые примеры кратных ассоциаций представлены на рис. 11.4.

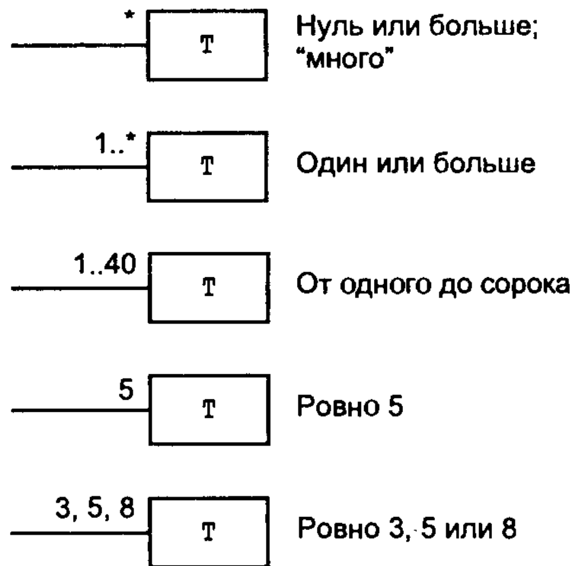
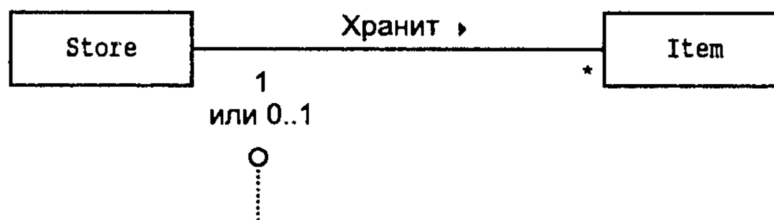



*Рис. 11.4. Значения кратности*

Значение кратности определяет, сколько экземпляров одного класса может быть корректно связано с экземпляром другого класса в некоторый конкретный момент, а не на всем промежутке времени. Например, некий подержанный автомобиль через определенные промежутки времени может быть многократно продан одному и тому же продавцу подержанных автомобилей. Однако в каждый конкретный момент этот автомобиль продан только *одному* владельцу. Автомобиль не может быть продан *нескольким* владельцам одновременно. Аналогично, в странах, подчиняющихся законам моногамии, каждый человек в конкретный момент может состоять в браке только с одним супругом, хотя в течение длительного промежутка времени он может вступать в брак неоднократно.

Значение кратности зависит от интересов разработчика модели и программы, поскольку ограничения предметной области должны быть отражены в программном обеспечении. На рис. 11.5 приводится пример подобной ситуации с разъяснениями.




Какую кратность использовать: "1" или "0..1"?

Ответ зависит от целей, преследуемых в процессе моделирования. Обычно на практике кратность обусловлена ограничениями из предметной области, которые нужно учитывать в программе, если эта связь отражена в программных объектах или базе данных. Например, определенный товар может быть куплен или списан, и, следовательно, он больше не будет храниться в магазине. С этой точки зрения логично использовать кратность "0..1", однако...

Внимательно ли вы продумали этот аспект? Если эта связь реализована в программе, то необходимо обеспечить связь экземпляра каждого программного объекта `Item` с определенным экземпляром `Store`. В противном случае в пограммных компонентах возникает сбой или потеря данных.

В этом фрагменте модели предметной области не представлены программные объекты, однако в ней указана кратность связей (ограничения), отражающая особенности реальной предметной области. С этой точки зрения более предпочтительным значением может оказаться "1"

*Рис. 11.5. Значение кратности является контекстно-зависимым*

Румбах [96] приводит еще один прекрасный пример объектов `Person` (Человек) и `Company` (Компания), связанных ассоциацией `Works-for` (Работает на). Отображение того факта, что один экземпляр объекта `Person` "работает" на один или несколько экземпляров объекта `Company`, зависит от контекста модели. Отдел налогообложения будет больше интересовать значение *много*, тогда как профсоюз, возможно, — значение *один*. Поэтому выбор на практике обычно зависит от заказчика программного обеспечения.

## 11.6. Насколько важны ассоциации

Ассоциации чрезвычайно важны, однако стандартной ошибкой при создании моделей предметной области являются слишком большие затраты времени на их изучение.

Очень важно принять во внимание следующее замечание.

> Гораздо важнее определить *концептуальные классы*, чем найти ассоциации. Большая часть времени, затраченного на создание модели предметной области, должна быть посвящена идентификации концептуальных классов, а не ассоциаций.

## 11.7. Имена ассоциаций

Имена ассоциаций базируются на формате *ИмяТипа-ГлагольнаяФраза-ИмяТипа*, где глагольная фраза представляет собой последовательность, которая читается и является значимой в контексте модели.

Имена ассоциаций должны начинаться с прописной буквы, поскольку ассоциация обычно представляет классификатор связей между экземплярами. В языке UML имя классификатора начинается с прописной буквы. Для имен ассоциаций принято использовать два формата.

- `Paid-by` (с дефисом)
- `PaidBy` (без дефиса)

На рис. 11.6 при чтении имен ассоциаций используется направление, принятое по умолчанию, а именно — слева направо и сверху вниз. Это правило относится не только к языку UML, а является относительно стандартным.

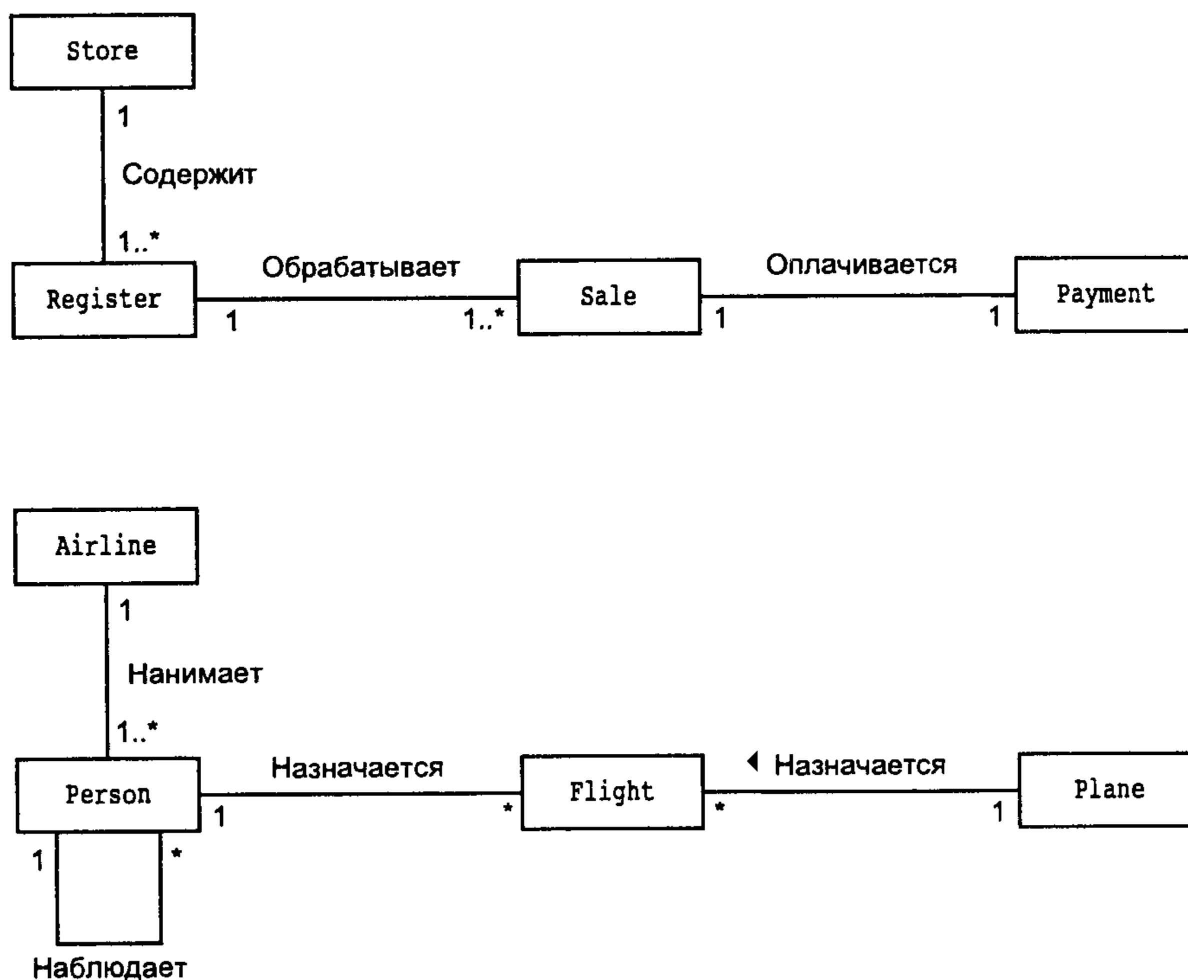



*Рис. 11.6. Имена ассоциаций*

## 11.8. Несколько ассоциаций между двумя типами

Между двумя типами может быть установлено несколько ассоциаций. И в этом нет ничего необычного. В рассматриваемой нами POS-системе нет соответствующего примера, однако из предметной области системы управления полетами в

качестве подобного примера можно привести отношение между объектами Flight (Полет) и Airport (Аэропорт) (рис. 11.7). Ассоциации Flies-to (Летит в) и Flies-from (Летит из) являются принципиально различными и должны отображаться отдельно.

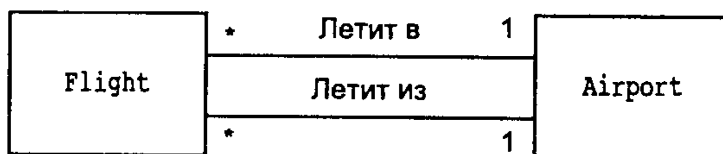



*Рис. 11.7. Несколько ассоциаций*

## 11.9. Ассоциации и их реализация

На стадии создания модели предметной области ассоциация *не* является описанием потоков данных, экземпляров переменных или взаимодействия объектов программной системы. Она представляет собой лишь описание значимого отношения, имеющего чисто концептуальный смысл, почерпнутый из реального мира. На практике многие из таких ассоциаций обычно реализуются в программном обеспечении как методы перемещения и средства обеспечения видимости, однако их присутствие в концептуальном представлении модели предметной области не требует какой бы то ни было реализации.

При создании модели предметной области можно определить ассоциации, которые не понадобятся во время проектирования. И наоборот, можно обнаружить ассоциации, которые должны быть реализованы, однако на стадии моделирования предметной области еще отсутствовали. В таком случае для отражения изменений модель предметной области нужно обновить.

> *Предупреждение*
>
> Нужно ли обновлять разработанные ранее модели, в том числе модель предметной области, по результатам программной реализации этой модели? Этого не стоит делать, если в дальнейшем эта модель больше не понадобится. Если модель просто является временным артефактом, используемым для более глубокого изучения проблемы на следующем шаге, и не будет применяться в дальнейшем, ее не стоит обновлять по результатам выполнения каждой итерации проекта. Старайтесь обновлять и поддерживать в актуальном состоянии только ту документацию и модели, которые будут реально использоваться при дальнейшей разработке.

Позднее мы рассмотрим реализацию ассоциаций на объектно-ориентированном языке программирования (как правило, для этого используется атрибут, который указывает на экземпляр ассоциированного класса). Однако в данный момент ассоциации лучше рассматривать как чисто концептуальные выражения, которые *не* содержат описания базы данных или каких-либо других частей программной системы. Как всегда, принимаемые при проектировании допущения позволяют освободиться от излишней информации и необходимости принятия решений при создании модели анализа и, кроме того, увеличивают свободу действий на более поздних стадиях разработки.

## 11.10. Ассоциации для предметной области POS-системы NextGen

Теперь приступим к добавлению ассоциаций в модель предметной области рассматриваемой POS-системы. Требуется добавить те ассоциации, которые должны сохраняться согласно определенным требованиям (например, прецедентам), или те, которые вытекают из нашего представления о предметной области. При решении новой проблемы должны быть пересмотрены и проанализированы ранее определенные стандартные категории ассоциаций, поскольку они представляют многие из актуальных ассоциаций, которые обычно следует учитывать.

### *Отношения в магазине, которые должны быть учтены*

Важными являются следующие ассоциации. Они основываются на описанных выше прецедентах.

| | |
|---|---|
| Register Records Sale (Register Фиксирует Sale) | Для того чтобы получить информацию о текущей продаже, необходимо вычислить общую сумму покупки и напечатать чек |
| Sale Paid-by Payment (Sale Оплачивается посредством Payment) | Чтобы узнать, была ли оплачена покупка, необходимо сравнить внесенную сумму с общей стоимостью покупки и напечатать чек |
| ProductCatalog Records ProductSpecification (ProductCatalog Записывает ProductSpecification) | Чтобы получить спецификацию товара ProductSpecification, необходимо знать его идентификатор itemID |

### *Использование списка категорий ассоциаций*

Проанализируем список категорий ассоциаций, основываясь на ранее идентифицированных типах и учитывая требования разрабатываемых на данной итерации прецедентов.

| Категория | Примеры |
|---|---|
| A является физической частью B | Register-CashDrawer |
| A является логической частью B | SalesLineItem-Sale |
| A физически содержится в/на B | Register-Store<br>Item-Store |
| A логически содержится в B | ProductSpecification-ProductCatalog<br>ProductCatalog-Store |
| A является описанием B | ProductSpecification-Item |
| A является элементом транзакции или отчета B | SalesLineItem-Sale |
| A известен/зарегистрирован/записан/включен в B | (Совершенные покупки) Sales-Store<br>(Текущая продажа) Sale-Register |
| A является членом B | Cashier-Store |
| A является организационной единицей B | Не применяется |
| A использует или управляет B | Cashier-Register<br>Manager-Register<br>Manager-Cashier; однако, возможно, не применяется |
| A взаимодействует с B | Customer-Cashier (Покупатель-Кассир) |

| Категория | Примеры |
|---|---|
| А связан с транзакцией В | Customer-Payment<br>Cashier-Payment |
| А является транзакцией, которая связана с другой транзакцией В | Payment-Sale (Платеж-Продажа) |
| А следует за В | SalesLineItem-SalesLineItem |
| А является "собственностью" В | Register-Store |

## 11.11. Модель предметной области POS-системы NextGen

В показанной на рис. 11.8 модели предметной области представлен набор концептуальных классов и ассоциаций, являющихся кандидатами на включение в POS-приложение. Ассоциации предварительно были выбраны из списка ассоциаций-кандидатов.

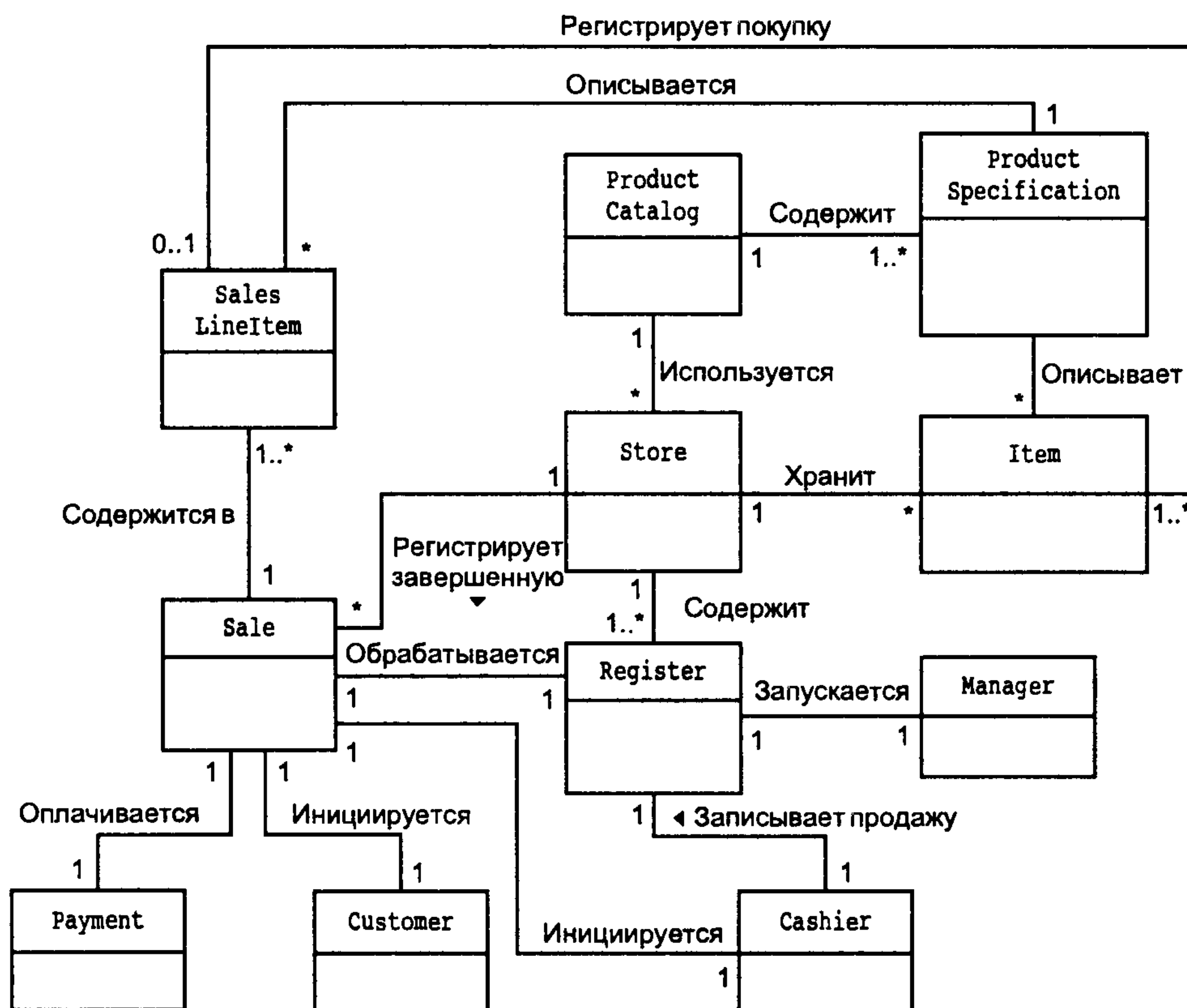



*Рис. 11.8. Фрагмент модели предметной области*

### *Сохранение только важных ассоциаций*

Ассоциации, представленные в модели предметной области на рис. 11.8, были, по существу, механически выбраны из списка ассоциаций. Однако при включении ассоциаций в модель предметной области желательно быть более из-

бирательным. Используя их как средство взаимодействия, не стоит перегружать модель предметной области ассоциациями, которые не требуются наверняка или не отражают результатов исследования предметной области. Слишком много ненужных ассоциаций скорее запутывают, чем вносят ясность.

Как упоминалось выше, при выборе ассоциаций руководствуйтесь следующими рекомендациями.

- Сосредоточьтесь на тех ассоциациях, данные о которых должны сохраняться в течение некоторого времени (важные ассоциации).
- Избегайте использования избыточных или повторяющихся ассоциаций.

Из приведенных рекомендаций следует, что не каждая из представленных ассоциаций является необходимой. Примите во внимание следующее.

| Ассоциация | Описание |
|---|---|
| `Sale Entered-by Cashier` (`Sale Вводится Cashier`) | Требования не отражают необходимости хранить или записывать информацию о кассире. Кроме того, эта ассоциация напрямую следует из ассоциации `Register Used-by Cashier` (`Register Используется Cashier`) |
| `Register Used-by Cashier` (`Register Используется Cashier`) | Требования не отражают необходимости хранить или записывать информацию о кассире |
| `Register Started-by Manager` (`Register Запускается Manager`) | Требования не отражают необходимости хранить или записывать информацию о менеджере, запустившем систему |
| `Sale Initiated-by Customer` (`Sale Инициируется Customer`) | Требования не отражают необходимости хранить или записывать информацию о текущем покупателе, инициировавшем продажу |
| `Store Stocks Item` (`Store Хранит Item`) | Требования не отражают необходимости хранить или поддерживать информацию об инвентаризации |
| `SalesLineItem Records-sale-of Item` (`SalesLineItem Записывает-информацию-о-продаже Item`) | Требования не отражают необходимости хранить или поддерживать информацию об инвентаризации |

Обратите внимание, что размещение ассоциаций по степени важности зависит от требований. Внесение в них явных изменений, таких как требование наличия на товарном чеке идентификатора кассира, приведет к необходимости хранения данных о новой связи.

Основываясь на проведенном анализе, можно сказать, что эти ассоциации *можно* удалить.

### Важные и второстепенные ассоциации

При использовании только важных ассоциаций будет сгенерирована минимальная "информационная модель", содержащая лишь те знания о предметной области, которые определяются принятыми текущими требованиями. Однако подобный подход может привести к созданию модели, которая не по-

зволяет получить полное представление о предметной области (вам или другому человеку).

Иногда модель предметной области рассматривается не как строгая информационная модель, а как средство, с помощью которого предпринимается попытка разобраться в важных понятиях и их отношениях. С этой точки зрения удаление некоторых не очень важных ассоциаций может привести к созданию модели, которая не отражает ключевых идей и связей.

Для примера еще раз обратимся к POS-системе розничной торговли. Хотя с точки зрения важности ассоциация `Sale Initiated-by Customer` (Продажа, инициированная покупателем) не является необходимой, ее отсутствие оставит за пределами рассмотрения важный аспект в понимании предметной области, заключающийся в том, что продажу инициирует покупатель.

В терминах ассоциаций хорошая модель находится где-то между моделью с важными ассоциациями и моделью, в которой представлены все возможные ассоциации. Таким образом, можно сформулировать следующий основной критерий оценки модели: "Позволяют ли все важные и дополнительные второстепенные ассоциации разобраться в важных понятиях предметной области и их отношениях?".

> Уделяйте особое внимание важным ассоциациям, однако не забывайте учитывать и второстепенные. Таким образом можно достичь лучшего понимания процессов, происходящих в предметной области.

Глава

# Модель предметной области: добавление атрибутов

*Любое достаточно серьезное упущение неотличимо от свойства.*

*Рич Кулавьек (Rich Kulawiec)*

**Основные задачи**

- **Идентифицировать атрибуты в модели предметной области.**
- **Выделить корректные и некорректные атрибуты.**

## Введение

Необходимо идентифицировать атрибуты концептуальных классов, которые удовлетворяют информационным требованиям разрабатываемых в текущий момент сценариев. В данной главе сначала рассматривается процесс выделения соответствующих атрибутов, а затем полученные навыки применяются при добавлении атрибутов в модель предметной области системы NextGen.

## 12.1. Атрибуты

*Атрибут* (attribute) — это абстрактное свойство объекта.

> В модель предметной области включаются те атрибуты, для которых определены соответствующие требования (например, прецеденты) или для которых необходимо хранить определенную информацию.

Например, в товарном чеке (представляющем собой отчет о некоторой продаже) обычно указываются дата и время. Эта информация необходима менеджерам компании. Следовательно, для концептуального класса `Sale` (Продажа) требуются атрибуты `date` (дата) и `time` (время).

## 12.2. Система обозначений атрибутов в языке UML

Атрибуты помещаются во второй раздел условного обозначения класса (рис. 12.1). Дополнительно может быть указан также тип атрибута.

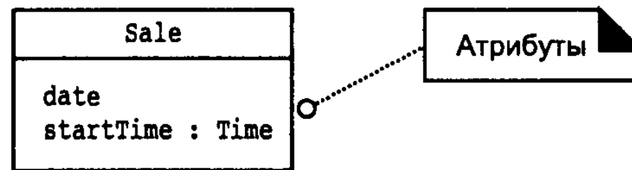



*Рис. 12.1. Класс и его атрибуты*

## 12.3. Корректные типы атрибутов

Некоторые сущности гораздо удобнее представлять не в виде атрибутов, а в форме ассоциаций. Изучению корректных атрибутов и посвящен данный раздел.

### *Атрибуты должны быть простыми*

Типы большинства простых атрибутов зачастую рассматриваются как примитивные типы данных. Обычно тип атрибута не должен быть сложным понятием предметной области, таким как `Sale` (Продажа) или `Airport` (Аэропорт). Например, атрибут `currentRegister` класса `Cashier` (Кассир) лучше не использовать (рис. 12.2), поскольку он имеет тип `Register`, не являющийся простым типом атрибута (таким, как `Number` (Число) или `String` (Строка)). Если объект `Cashier` использует объект `Register`, то лучше всего выразить этот факт с помощью ассоциации, а не атрибута.

> В модели предметной области атрибуты должны быть *простыми атрибутами* (simple attributes) или *простыми типами данных* (data types).
>
> К стандартным типам атрибутов относятся `Boolean`, `Date`, `Number`, `String(Text)`, `Time`.
>
> Другими стандартными типами являются следующие: `Address` (адрес), `Color` (цвет), `Geometrics` (`Point`, `Rectangle`) (геометрические фигуры: точка, прямоугольник), `Phone Number` (номер телефона), `Social Security Number` (номер страхового полиса), `Universal Product Code` (UPC) (универсальный код товара), `SKU`, `ZIP` или `postal code` (почтовый индекс), перечисляемые типы.

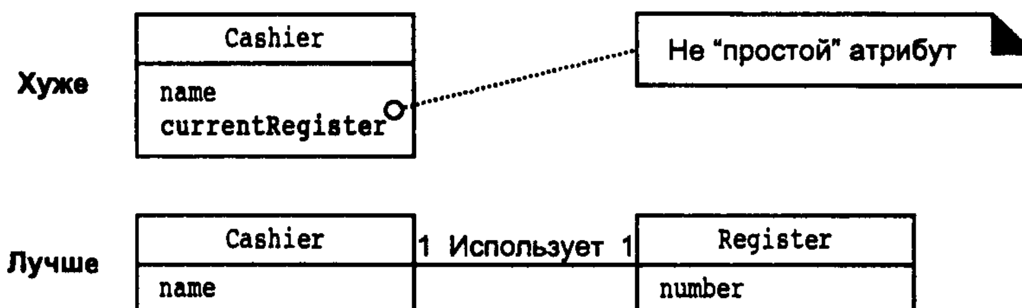



*Рис. 12.2. Связь с помощью ассоциаций, а не атрибутов*

Как видно из приведенного примера, стандартной ошибкой является моделирование сложного понятия предметной области в форме атрибута. Другими словами, аэропорт назначения на самом деле является не строкой, а сложной сущностью с территорией, протянувшейся на многие километры. Таким образом, объект `Flight` (Полет) должен быть связан с объектом `Airport` (Аэропорт) с помощью ассоциации, а не атрибута (рис. 12.3).

| Связывайте концептуальные классы с использованием ассоциаций, а не атрибутов. |
|---|

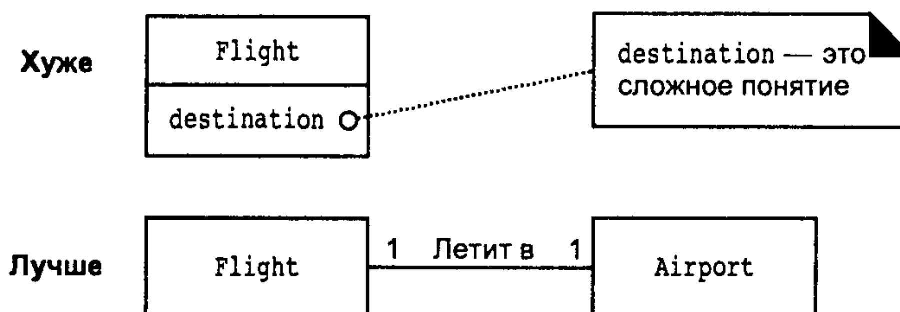



*Рис. 12.3. Избегайте представления сложных понятий предметной области в виде атрибутов; используйте ассоциации*

### Концептуальный аспект и аспект реализации: реализация атрибутов в программном коде

Требование того, что атрибуты модели предметной области должны описываться только простыми типами данных, не предполагает, что на языках программирования C++, Java и Smalltalk атрибутами (данными-членами, полями) должны быть простые, примитивные типы данных. Модель предметной области концентрирует внимание лишь на аналитическом исследовании проблемы предметной области, а не на программных сущностях.

Позднее, на стадиях проектирования и реализации, вы увидите, что ассоциации между объектами модели предметной области зачастую реализуются как атрибуты, указывающие на другие сложные программные объекты. Однако это далеко не единственная возможность реализации ассоциации, и соответствующее решение нужно отложить на более поздний срок.

### Типы данных

Атрибутами должны быть данные простых типов (или в терминах UML *типов данных* — data types), для которых совершенно незначимой является уникальная тождественность (в контексте модели или системы) [93]. Например, обычно не существует различия между:

- отдельными экземплярами числа 5 ( тип `Number`);
- отдельными экземплярами строк 'cat' (тип `String`);
- отдельными объектами `PhoneNumber`, содержащими один и тот же номер телефона;
- отдельными объектами `Address`, содержащими один и тот же адрес.

Однако существенными являются различия между двумя отдельными объектами `Person` (Человек), даже если в обоих объектах содержится имя "Jill Smith", поскольку два экземпляра могут представлять отдельных людей с одним и тем же именем.

В терминах программных систем существует несколько случаев, когда нет необходимости сравнивать адреса памяти экземпляров объектов `Number`, `String`, `PhoneNumber` или `Address`, достаточно лишь выполнить сравнение их значений. Однако для того чтобы различить объекты `Person`, даже если они имеют одинаковые значения атрибутов, все же придется сравнить адреса памяти, поскольку в данном случае важна их уникальность.

Таким образом, все простые типы (числа, строки) считаются типами данных UML, однако не все типы данных являются простыми. Например, `PhoneNumber` (Номер телефона) — это не простой (или не примитивный) тип данных.

Данные простых типов называются также *объектами значений* (value objects).

Сущность данных простых типов трудноуловима. Для стандартной проверки "простоты" руководствуйтесь следующим правилом: если атрибут можно рассматривать как число, строку, логическое значение, дату или время (и т.д.), то следует оставить его в качестве атрибута; в противном случае его нужно представить отдельным концептуальным классом.

> Если у вас имеются какие-либо сомнения, то лучше создайте отдельный концептуальный класс, а не атрибут.

## 12.4. Непримитивные типы классов

Тип атрибута может рассматриваться как непримитивный класс в его собственном значении в модели предметной области. Например, в POS-системе имеется универсальный идентификатор товара. Обычно он рассматривается как простое число. Что же должно быть представлено как непримитивный класс? Руководствуйтесь следующими рекомендациями.

> Тип данных, изначально считающийся примитивным (такой, как число или строка), может быть представлен в виде непримитивного класса в следующих случаях.
>
> - Если он составлен из отдельных частей.
>   - Номер телефона, имя человека.
> - Если с этим типом обычно ассоциируются операции, такие как синтаксический анализ и проверка.
>   - Номер страхового полиса.
> - Он содержит другие атрибуты.
>   - Для льготной цены могут устанавливаться сроки действия (начало и конец).
> - Если этот тип используется для задания количества с единицами измерения.
>   - Стоимость товара измеряется в некоторых единицах.
> - Это абстракция одного или нескольких типов.
>   - Идентификатор товара — это некое обобщение типов UPC (Universal Product Code) и EAN (European Article Number).

Применение этих рекомендаций к атрибутам модели предметной области POS-системы приведет к формулировке следующих идей.

- *Идентификатор товара* — это абстракция, построенная на основе различных стандартных схем кодирования, включая UPC-A, UPC-E и семейство схем EAN. Эти схемы кодирования могут использоваться при подсчете контрольной суммы или иметь другие атрибуты (например, код изготовителя, товара, страны). Следовательно, это должен быть непримитивный класс `ItemID`.
- Атрибуты `price` (цена) и `amount` (сумма) должны иметь тип `Quantity` (Количество) или `Money` (Валюта), поскольку они представляют собой количество, выраженное в денежных единицах.
- Атрибут `address` (адрес) должен относиться к непримитивному типу `Address`, поскольку в нем имеются отдельные разделы.

Классы `ItemID`, `Address` и `Quantity` относятся к данным простых типов (уникальная тождественность является несущественной), так что их без проблем можно помещать в раздел атрибутов, а не связывать с использованием ассоциаций.

## Как иллюстрировать классы типов данных

Должен ли тип `ItemID` представляться в модели предметной области как отдельный концептуальный класс? Это зависит от того, хотите ли вы выделить его на диаграмме. Если тип `ItemID` представляет данные простых типов (уникальная тождественность не существенна), то его можно поместить в раздел атрибутов соответствующего класса, как показано на рис. 12.4. Если же `ItemID` является непримитивным типом и у него имеются собственные атрибуты и ассоциации, то больший интерес он будет представлять в качестве отдельного концептуального класса. Однако это не совсем корректный ответ. На самом деле все зависит от того, как модель предметной области используется в качестве средства взаимодействия и от значимости понятия в предметной области.

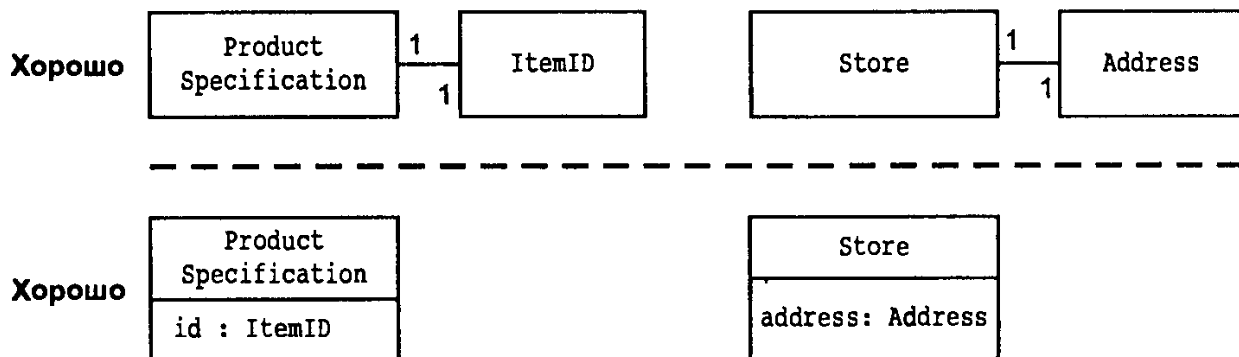



*Рис. 12.4. Если сущность относится к данным простых типов, то ее можно поместить в раздел атрибутов*

| Модель предметной области является средством общения. Выбор элементов, которые нужно отображать в этой модели, зависит от принятых предположений. |
|---|

## 12.5. Совет разработчикам: не используйте атрибуты в качестве внешних ключей

Атрибуты не должны использоваться для связи концептуальных классов в модели предметной области. Наиболее распространенным нарушением этого принципа является добавление некоторой разновидности *атрибута внешнего ключа* (foreign key attribute), что обычно происходит при разработке реляционных баз данных. Например, на рис. 12.5 атрибут `currentRegisterNumber` использовать нежелательно, поскольку его назначением является связывание объекта `Cashier` с объектом `Register`. Объекты `Cashier` и `Register` лучше связать с помощью ассоциации, а не атрибута внешнего ключа. Не лишним будет повторить еще раз: связывайте типы с помощью ассоциаций, а не атрибутов.

Связать объекты можно множеством различных способов, и внешние ключи являются далеко не единственной возможностью. Для успешного создания системы на стадии проектирования необходимо решить, как реализовать требуемые связи.

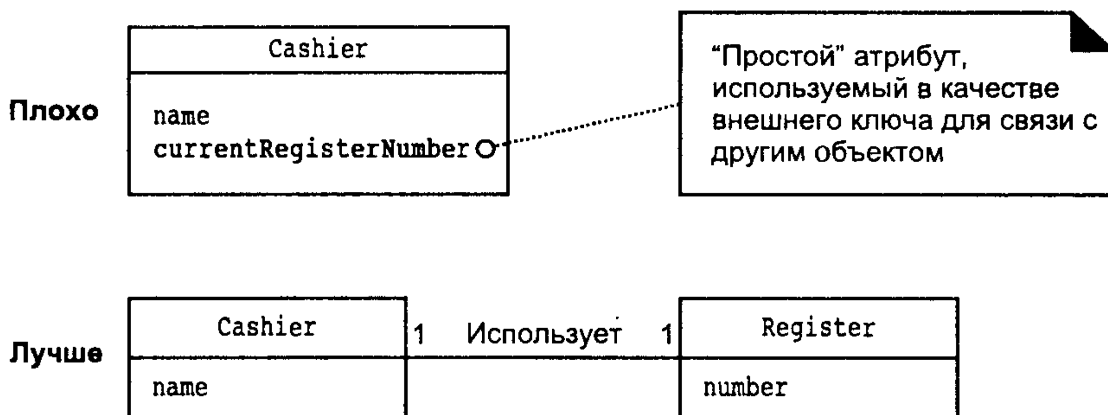



*Рис. 12.5. Не используйте атрибуты в качестве внешних ключей*

## 12.6. Моделирование атрибутов `Quantity` и `Unit`

Большинство количественных сущностей нельзя представить в виде числовых типов данных. Возьмем, к примеру, цену или скорость. С этими количественными сущностями связаны определенные единицы измерения, необходимые для поддержки функций конвертирования. Программная система NextGen предназначена для использования в различных странах, поэтому она должна поддерживать различные типы валют. Решение заключается в создании отдельного концептуального класса `Quantity` (Количество), с которым будет ассоциироваться класс `Unit` (Единица измерения) [46]. Поскольку количество рассматривается в качестве типа данных, соответствующий атрибут можно поместить в раздел атрибутов, как показано на рис. 12.6. Обычно количество измеряют в некоторых единицах. Деньги — это количество, единицей измерения которого служит тип валюты. Вес — это количество, измеряемое в килограммах или фунтах.

## 12.7. Атрибуты модели предметной области системы NextGen

Теперь необходимо сформировать список атрибутов для отображения требований данной итерации или список атрибутов для сценариев прецедента `Оформление продажи`.

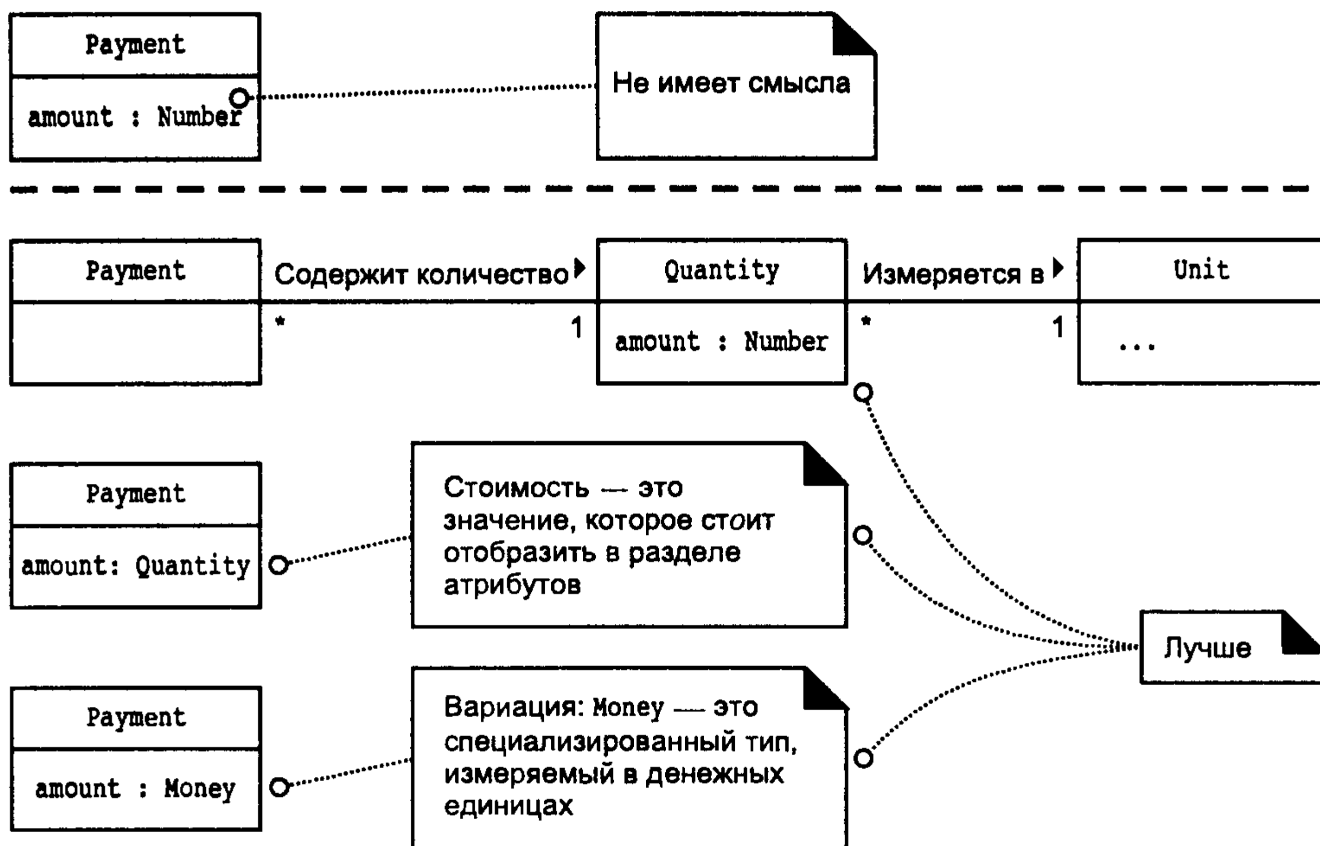



*Рис. 12.6. Моделирование количества*

| | |
|---|---|
| Payment (Платеж) | amount (сумма). Если нужно определить, достаточную ли сумму заплатил покупатель, и вычислить разность между этой суммой и стоимостью покупок, необходимо иметь атрибут amount (известный также как “предложенная сумма”) |
| ProductSpecification (Спецификация товара) | description (описание). Оказывается полезным, если описание товара нужно отобразить на экране или распечатать на товарном чеке<br>id (идентификатор товара). Если необходимо просмотреть данные объекта ProductSpecification, соответствующего введенному значению кода itemID, то эту связь можно обеспечить с помощью данного атрибута<br>price (цена). Необходим для вычисления итоговой суммы и отображения цены единицы товара |
| Sale (Продажа) | date (дата), time (время). Товарный чек представляет собой распечатанные на бумаге данные о продаже. Как правило, на нем указываются также дата и время продажи |
| SalesLineItem (Элемент продажи) | quantity (количество). Требуется для записи введенного количества товаров, если покупатель приобретает несколько единиц одного и того же товара (например, пять коробок конфет) |
| Store (Магазин) | address (адрес), name (название). На товарном чеке требуется указывать название и адрес магазина |

На рис. 12.7 представлена модель предметной области с атрибутами.

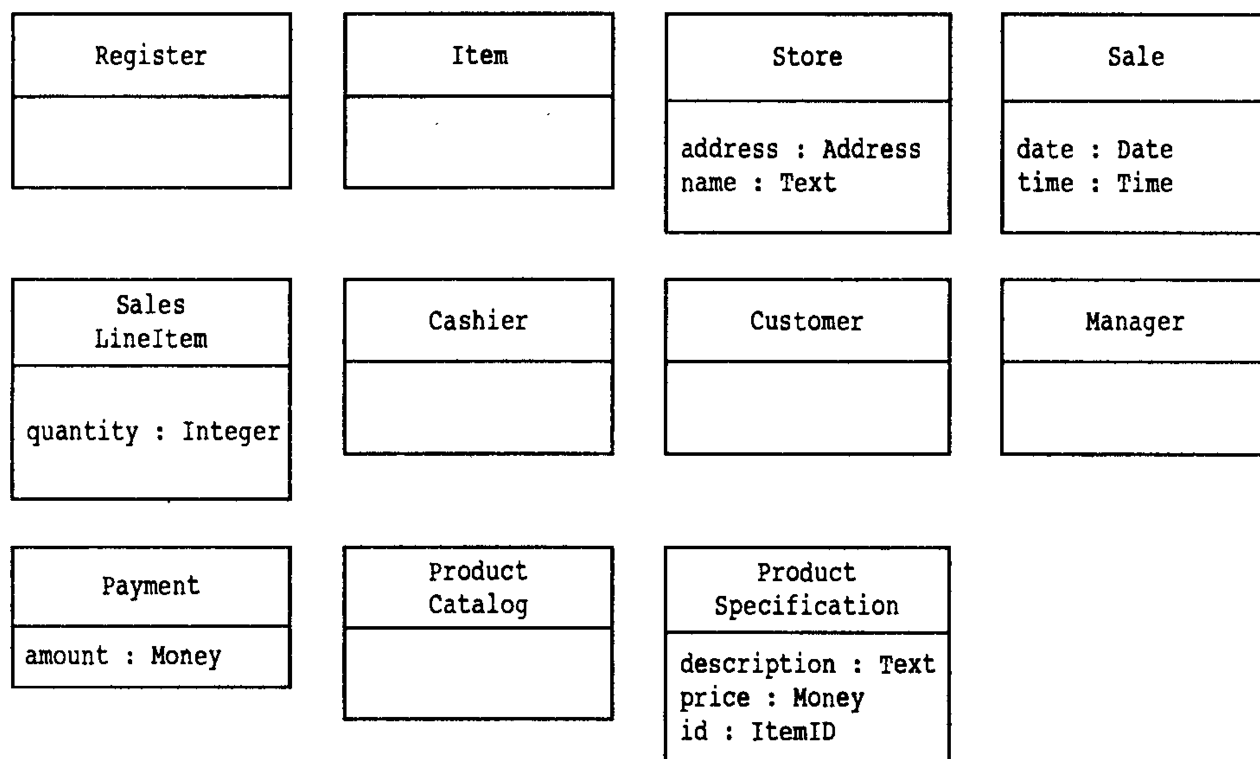



*Рис. 12.7. Модель предметной области с атрибутами*

## 12.8. Кратная ассоциация между понятиями SalesLineItem и Item

Вполне возможна такая ситуация, когда покупатель приобретает несколько единиц одного товара (например, шесть пакетиков с конфетами). Тогда кассиру достаточно сначала один раз ввести идентификатор товара (itemID), а затем — его количество (например, шесть). В результате отдельный объект SalesLineItem будет связан с несколькими экземплярами объекта Item.

Введенное кассиром количество может быть записано как атрибут объекта SalesLineItem (Элемент продажи) (рис. 12.8). Однако количество может быть определено из реального значения кратности, заданной для связи. Поэтому его можно охарактеризовать как *производный атрибут* (derived attribute), который можно получить из других данных. В языке UML производный атрибут отображается с использованием символа "/".

## 12.9. Заключительные замечания о модели предметной области

После объединения концептуальных классов, ассоциаций и атрибутов, рассмотренных в предыдущих разделах, модель предметной области системы розничной торговли примет вид, показанный на рис. 12.9.

Нами была создана достаточно полезная модель предметной области POS-системы. Не существует такого понятия, как "единственно верная модель". Все модели являются аппроксимациями изучаемой предметной области. В хорошей модели представлены наиболее существенные абстракции и данные, которые

требуются для понимания предметной области в контексте текущих требований. С помощью такой модели можно досконально разобраться в предметной области, ее понятиях, терминологии и взаимосвязях между различными элементами.

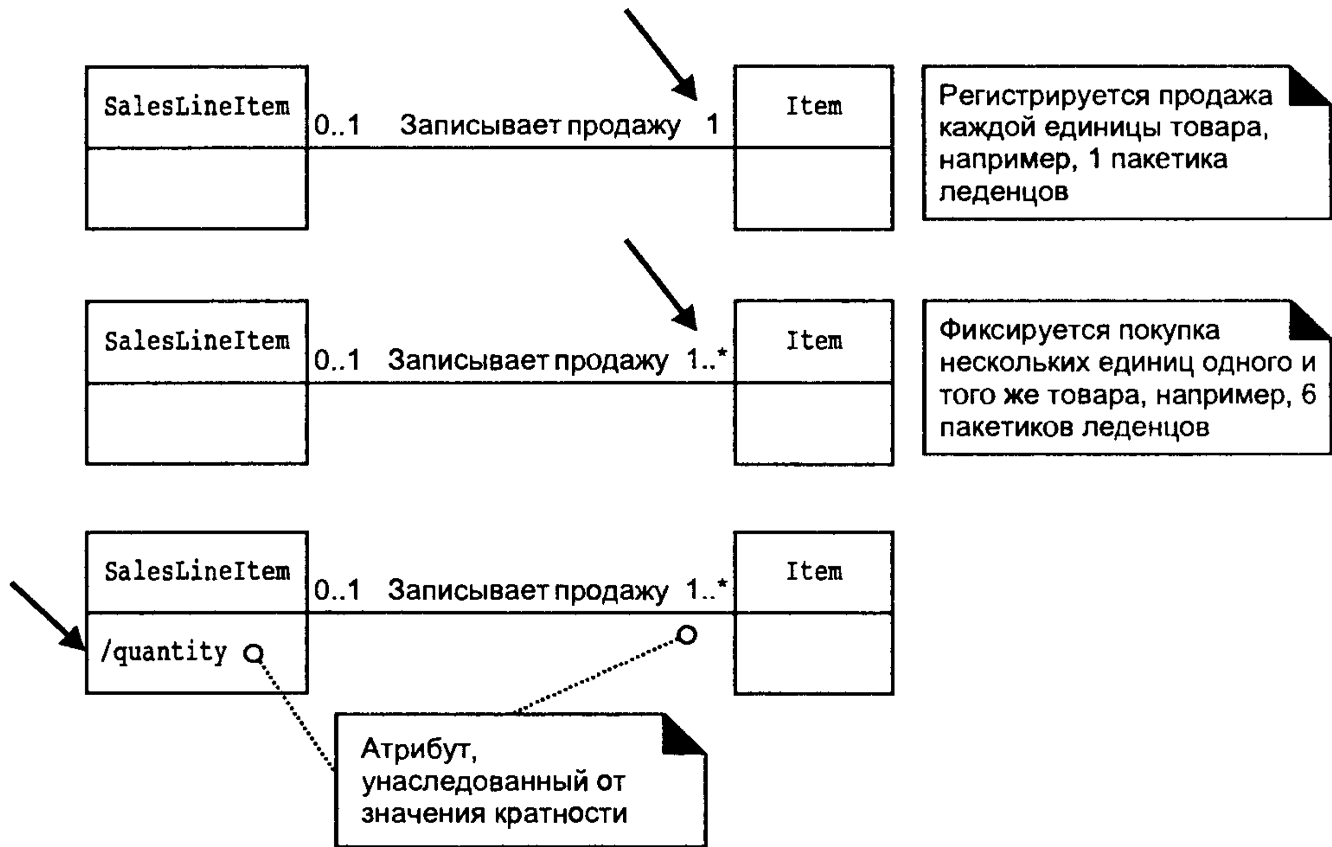



*Рис. 12.8. Запись количества приобретенных элементов товара в объект* `SalesLineItem`

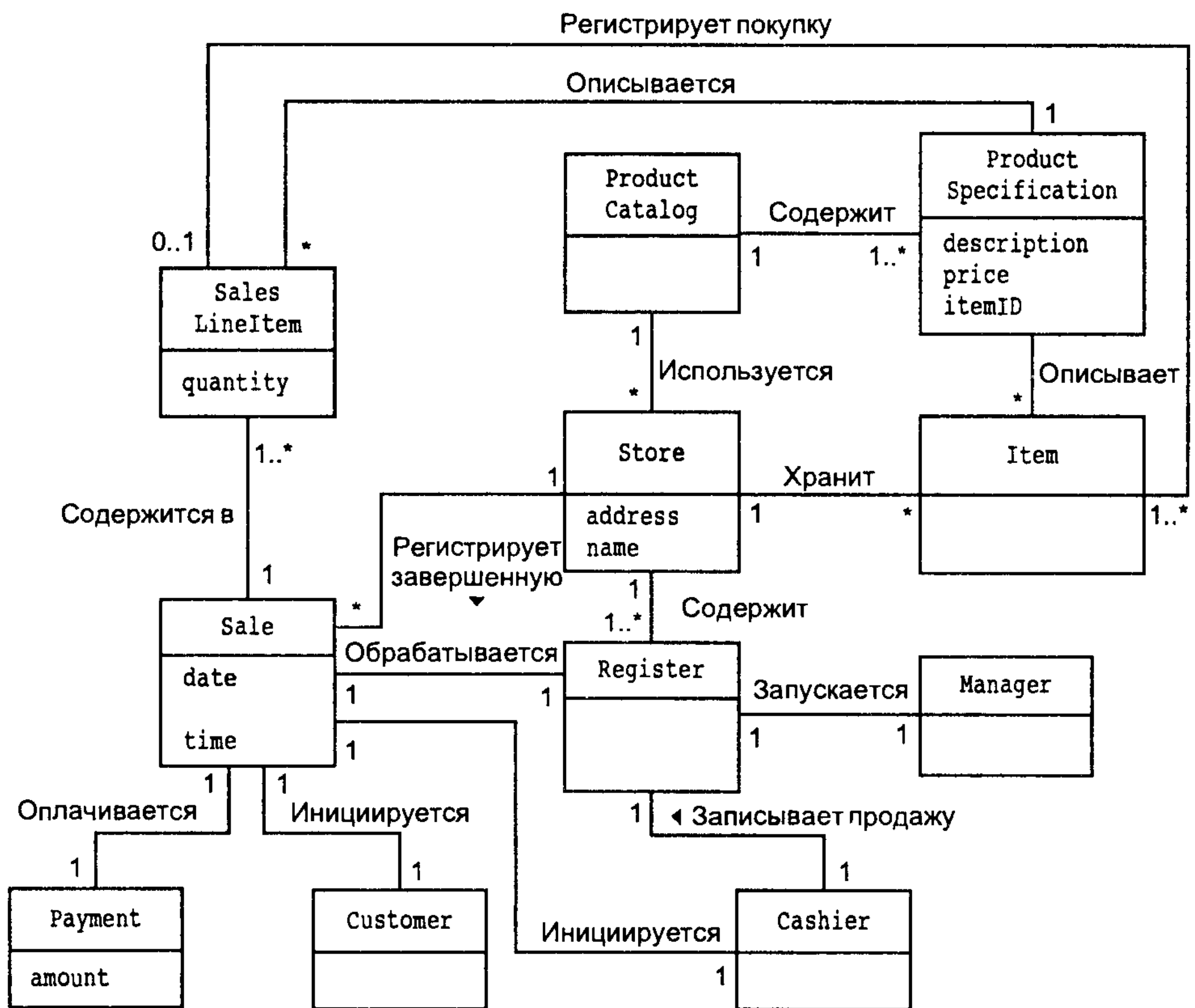



*Рис. 12.9. Фрагмент модели предметной области*

Глава 13

# Модель прецедентов: детализация с помощью описаний операций

*Многословный контракт не стоит даже той бумаги, на которой он написан.*

*Сэмюэль Голдвин (Samuel Goldwyn)*

**Основная задача**

- **Создать описания системных операций.**

## Введение

Описания системных операций помогают определить поведение системы. Они описывают влияние операций на систему. Эта глава посвящена использованию описаний.

## 13.1. Описания

Основным механизмом описания поведения в UP являются прецеденты. Зачастую приводимой в них информации достаточно для описания поведения системы. Однако иногда требуется более детальное описание поведения. *Описания системных операций* (system operation contract) описывают детальное поведение системы в терминах изменения состояния объектов модели предметной области после выполнения системных операций.

### *Системные операции и системный интерфейс*

Описания определяются для *системных операций* (system operations). Системными называются операции, входящие в открытый интерфейс системы для обработки входных системных событий, которые система выполняет как черный ящик. Системные операции можно идентифицировать на основе системных событий (рис. 13.1).

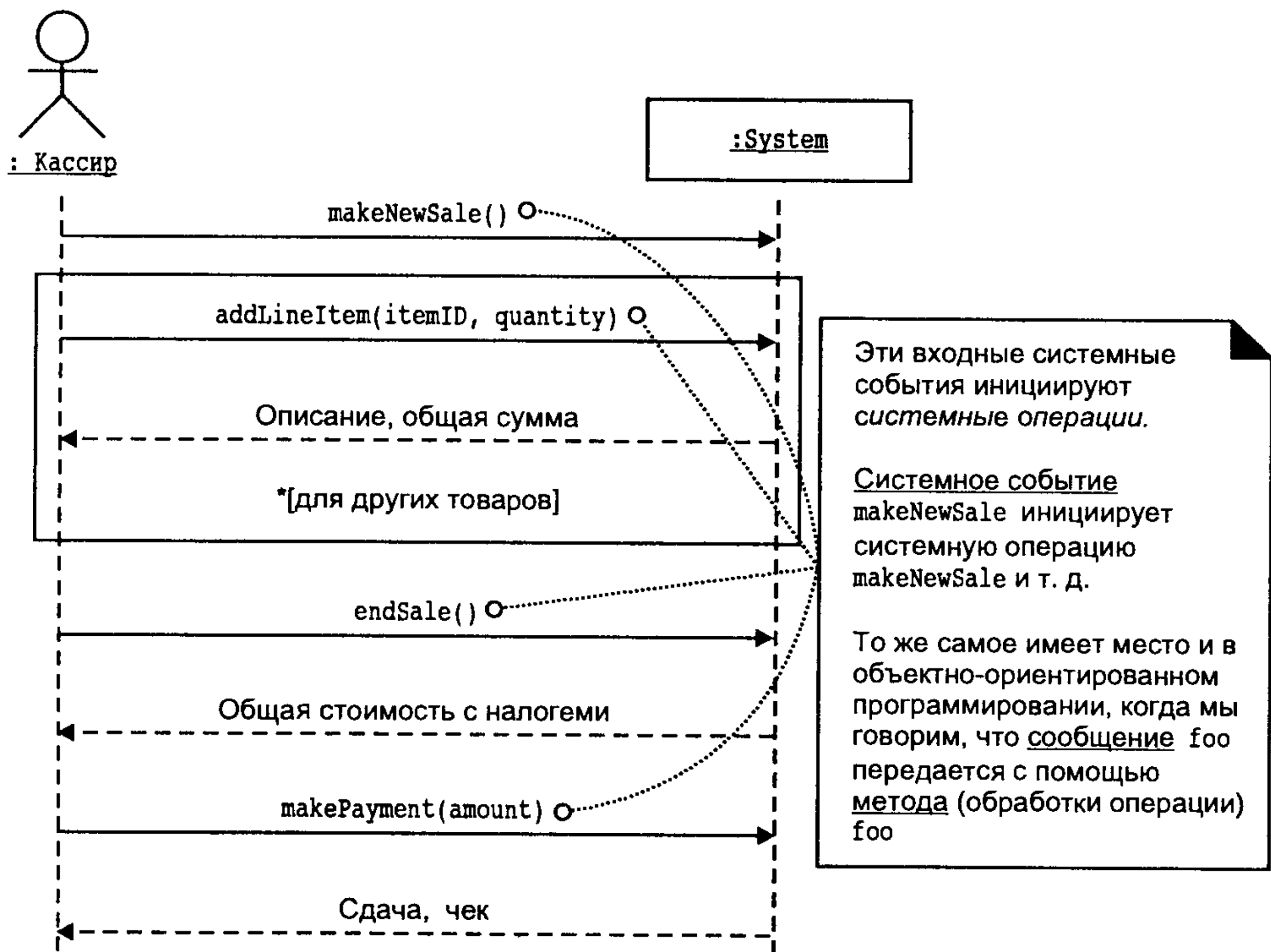



*Рис. 13.1. Системные операции обрабатывают входные системные события*

Весь набор системных операций, выполняемых в процессе всех прецедентов, определяет открытый системный интерфейс, в ракурсе которого система рассматривается как единый компонент или класс. В UML систему в целом можно представить в виде одного класса.

## 13.2. Пример описания системной операции: enterItem

Прежде чем исследовать необходимость описания системных операций, рассмотрим пример. В этом примере приводится описание системной операции enterItem.

### Описание операции ОП2: enterItem

| | |
|---|---|
| **Операция** | enterItem(itemID: ItemID, quantity: integer) |
| **Ссылки** | Прецеденты: Оформление продажи |
| **Предусловия** | Инициирована продажа |
| **Постусловия** | - Создан экземпляр sli класса SalesLineItem (*создание экземпляра*)<br>- Экземпляр sli связан с текущим экземпляром класса Sale (*формирование ассоциации*)<br>- Атрибуту sli.quantity присвоено значение quantity (*модификация атрибута*)<br>- Экземпляр sli связан с классом ProductSpecification на основе соответствия идентификатора товара itemID (*формирование ассоциации*) |

## 13.3. Разделы описания

Ниже рассмотрен каждый из разделов описания.

| | |
|---|---|
| **Операция** | Имя операции и ее параметры |
| **Ссылки** | (не обязательный) Прецеденты, в рамках которых может выполняться эта операция |
| **Предусловия** | Предположения о состоянии системы или объектов модели предметной области до выполнения операции. Выполнение этих условий не проверяется в рамках логики выполнения данной операции, а предполагается, что они истинны. Это нетривиальные условия, на которые читатель должен обратить внимание |
| **Постусловия** | Состояние объектов модели предметной области после завершения операции (подробнее обсуждается в следующем разделе) |

## 13.4. Постусловия

Обратите внимание, что для каждого из постусловий в примере описания системной операции `enterItem` указана категория, например "Создание экземпляра" или "Формирование ассоциации".

> В разделе "Постусловия" декларируются изменения состояния объектов модели предметной области. К таким изменениям относятся создание экземпляра, формирование или разрыв ассоциации, или изменение атрибута.

Постусловия — это не действия, выполняемые в процессе операции, а лишь декларация об изменении состоянии объектов модели предметной области после выполнения операции (когда рассеется туман).

> Существуют следующие категории постусловий описаний.
>
> - Создание и удаление экземпляра.
> - Модификация атрибута.
> - Формирование и разрыв ассоциаций.

В качестве примера разрыва ассоциации рассмотрим операцию удаления наименования товара. Постусловие этой операции может иметь вид: "Ассоциация выбранного класса `SalesLineItem` с экземпляром класса `Sale` разорвана". Если рассматривать другие предметные области, то разрыв ассоциации происходит при выплате кредита или выходе человека из некоторого сообщества.

Реже всего встречаются постусловия удаления экземпляра, поскольку явное разрушение объекта реального мира обычно не имеет значения для разработчиков программной системы. Однако рассмотрим следующий пример. Во многих странах, согласно законодательству, через семь–десять лет после объявления о банкротстве все записи об этом банкротстве должны быть удалены. Заметим, что это относится к концептуальному аспекту, а не к аспекту реализации. Это не означает необходимость очистки памяти компьютера, занимаемой программными объектами.

Важной характеристикой постусловий является констатация изменения состояния, а не действия. Постусловия — это декларации о результатах или состояниях, а не описание выполняемых действий или проектного решения.

### Связь постусловий с моделью предметной области

Постусловия формулируются в контексте модели предметной области. Экземпляры каких объектов могут создаваться? Из модели предметной области. Какие ассоциации могут формироваться? Опять же, ассоциации, предусмотренные в модели предметной области, и т.д.

### Преимущества описания постусловий

Поскольку описания формируются в терминах изменения состояний, их очень удобно использовать для анализа требований, описывающих изменение состояния (в терминах объектов модели предметной области), не вдаваясь в детали реализации операций. Другими словами, на этом этапе можно абстрагироваться от проектных решений и сосредоточить внимание на анализе происходящих в системе событий. Более того, постусловия обеспечивают достаточно высокий уровень детализации результатов операций.

Этот уровень детализации можно поддерживать и при описании прецедентов, однако это нежелательно, поскольку тогда эти описания будут слишком многословными.

Рассмотрим постусловия описанной ранее операции.

| | |
|---|---|
| **Постусловия** | - Создан экземпляр `sli` класса `SalesLineItem` (*создание экземпляра*)<br>- Экземпляр `sli` связан с текущим экземпляром класса `Sale` (*формирование ассоциации*)<br>- Атрибуту `sli.quantity` присвоено значение `quantity` (*модификация атрибута*)<br>- Экземпляр `sli` связан с классом `ProductSpecification` на основе соответствия идентификатора товара `itemID` (*формирование ассоциации*) |

Здесь ничего не говорится о способе создания экземпляра `SalesLineItem` или установке ассоциации с экземпляром класса `Sale`. Речь может идти об описании классов на бумаге и скреплении двух таких описаний, использовании Java-технологии для создания программных объектов и их связывании или о вставке записей в реляционную базу данных.

### Дух постусловий: сцена и занавес

Постусловия должны описывать состояние системы, а не выполняемые действия. Их желательно формулировать в прошедшем времени, чтобы подчеркнуть уже произошедшие изменения, например:

- создан экземпляр `SalesLineItem`,

а не

- создание экземпляра `SalesLineItem`.

Рассуждая о постусловиях, уместно использовать следующую метафору. *Система и ее объекты находятся на сцене театра.*

1. Сфотографируйте сцену перед выполнением операций.
2. Опустите занавес и выполните системную операцию (за кулисами раздается шум, возникающий из-за передвижения декораций, голоса и прочие звуки...).

3. Откройте занавес и сделайте еще один снимок.
4. Сравните оба снимка и выразите произошедшие изменения в форме постусловий (создан экземпляр объекта `SalesLineItem`...).

### Насколько детальными должны быть постусловия описаний

Во-первых, описания операций могут вообще не понадобиться. Об этом говорилось в предыдущих разделах. Однако если нужно составить описания системных операций, то генерировать полный и детальный список постусловий на этапе анализа требований нежелательно. Их надо рассматривать как некие начальные сведения и понимать, что описания операций являются далеко не полными. Однако желательно создавать описания операций (даже неполные) на ранних этапах разработки, а не откладывать это до стадии проектирования. На стадии проектирования разработчики уже должны принимать проектные решения, а не выяснять, *что* должно быть сделано.

Детали реализации операций будут исследованы на этапе проектирования. И это хорошо. Не стоит слишком затягивать этап анализа требований. Необходимость проведения исследований неизбежно возникнет и в процессе проектирования, что приведет к формулировке новых требований для следующей итерации. В этом состоит одно из преимуществ итеративной разработки — исследования, проведенные на одной итерации, инициируют новые исследования и анализ требований на следующей.

## 13.5. Обсуждение: постусловия описания операции `enterItem`

В этом разделе подробно рассматриваются постусловия системной операции `enterItem`.

### Создание и удаление экземпляра

Какие новые объекты создаются в системе, когда кассир вводит код товара `itemID` и количество его единиц `quantity`? Должен быть создан экземпляр объекта `SalesLineItem`. Таким образом:

- создан экземпляр `sli` класса `SalesLineItem` (*создание экземпляра*).

Обратите внимание на имя экземпляра. Это имя упростит ссылки на вновь созданный экземпляр в последующих постусловиях.

### Модификация атрибута

Какие атрибуты новых или уже существующих объектов должны быть модифицированы, когда кассир вводит код товара `itemID` и количество его единиц `quantity`? Должно быть установлено количество покупаемых единиц для товара `SalesLineItem`. Таким образом:

- атрибуту `sli.quantity` присвоено значение `quantity` (*модификация атрибута*).

### *Формирование и разрыв ассоциации*

Какие ассоциации между новыми или существующими объектами должны быть сформированы или разорваны, когда кассир вводит код товара itemID и количество его единиц quantity? Новый покупаемый товар SalesLineItem должен быть связан с текущей продажей Sale, а также со своей спецификацией ProductSpecification. Таким образом:

- экземпляр sli связан с текущим экземпляром класса Sale (*формирование ассоциации*);
- экземпляр sli связан с классом ProductSpecification на основе соответствия идентификатора товара itemID (*формирование ассоциации*).

Обратите внимание на неформальную констатацию формирования взаимосвязи с конкретным экземпляром класса ProductSpecification на основе соответствия идентификатора товара itemID. Для выражения этого факта можно использовать более формальный подход, в частности, основанный на применении объектного языка ограничений OCL (Object Constraint Language). Однако автор советует придерживаться принципа простоты и не усложнять себе жизнь.

## 13.6. Описание операций приводит к изменению предметной области

В процессе составления описаний операций в модель предметной области зачастую приходится вводить новые концептуальные классы, атрибуты или ассоциации. Не привязывайтесь к существующей модели предметной области, усовершенствуйте ее в процессе исследования и описания системных операций.

## 13.7. Когда нужны описания операций? Что лучше: описания операций или прецедентов?

Прецеденты — основной источник информации о требованиях к проекту. В них могут отражаться основные или все детали, необходимые для проектирования. В этом случае описания системных операций не нужны. Однако бывают случаи, когда изменение состояния не слишком детально описывается в прецедентах из-за своей сложности.

Например, рассмотрим систему резервирования авиабилетов и системную операцию addNewReservation. Она обладает высокой сложностью, поскольку затрагивает множество объектов предметной области, состояние которых изменяется. Детали таких изменений *можно* описать в прецеденте, связанном с этой операцией, однако при этом описание прецедента станет излишне многословным (поскольку придется упомянуть каждый атрибут всех изменяемых объектов).

Обратите внимание, что формат описания постусловий предполагает использование очень точного языка с необходимой степенью детализации.

Если на основании описаний прецедентов и обсуждения проблемы со специалистами по предметной области разработчикам понятно, что нужно делать, то составлять описание системных операций не обязательно.

Однако если операции достаточно сложны и их детальное описание вносит ясность в проблему, то такое описание системной операции можно рассматривать как еще один механизм формулировки требований.

На практике описания системных операций требуются не слишком часто. Если команде разработчиков приходится составлять описания всех системных операций для каждого прецедента, значит, либо описания прецедентов плохо продуманы, либо отсутствует должный контакт со специалистами по предметной области, либо группа разработчиков готовит слишком много лишней документации.

В примере для POS-системы приводится гораздо больше описаний, чем необходимо на самом деле. Обычно большая часть деталей очевидна из описаний прецедентов. Однако "очевидность" — очень субъективное понятие.

## 13.8. Составление описания

Приведем некоторые советы по составлению описаний.

> Чтобы составить описания для каждого прецедента, выполните следующие действия.
>
> 1. Определите системные операции из диаграмм последовательностей.
> 2. Составьте описание для сложных системных операций, результаты которых с очевидностью не следуют из описания прецедента.
> 3. При описании постусловий используйте следующие категории.
>    - Создание и удаление экземпляра
>    - Модификация атрибута
>    - Формирование и разрыв ассоциаций

### *Советы по составлению описаний системных операций*

- Постусловия целесообразно описывать в декларативной форме, желательно с использованием глаголов пассивного залога в прошедшем времени, чтобы подчеркнуть факт изменения состояний, а не способ реализации, как, например, показано ниже.
  - Создан экземпляр `SalesLineItem` (лучше).
  - Создание экземпляра `SalesLineItem` (хуже).
- Не забудьте установить отношения между существующими и вновь создаваемыми объектами путем формирования ассоциаций. Например, при выполнении операции `enterItem` недостаточно просто создать новый экземпляр записи о покупке товара `SalesLineItem`. После выполнения операции этот экземпляр должен быть связан с текущей продажей `Sale`. Поэтому одним из постусловий этой операции является следующее.
  - Вновь созданный экземпляр `SalesLineItem` связан с объектом `Sale` (*формирование ассоциации*).

### *Наиболее типичная ошибка при создании описаний системных операций*

Наиболее типичной ошибкой при составлении описаний является невключение *формирования ассоциаций* в число постусловий операции. Установка ассоциаций играет особо важную роль при создании новых экземпляров. Не забывайте об этом!

## 13.9. Пример POS-системы NextGen: описания

### *Системные операции для прецедента* `Оформление продажи`

#### Описание операции ОП1: `makeNewSale`

| | |
|---|---|
| **Операция** | `makeNewSale()` |
| **Ссылки** | Прецеденты: `Оформление продажи` |
| **Предусловия** | Отсутствуют |
| **Постусловия** | - Создан экземпляр `s` объекта `Sale` (*создание экземпляра*)<br>- Экземпляр `Sale` связан с объектом `Register` (*формирование ассоциации*)<br>- Инициализированы атрибуты экземпляра `s` |

Обратите внимание на нечеткое описание последнего постусловия.

В реальном проекте все эти постусловия настолько очевидны и следуют из описания прецедентов, что создавать описание системной операции `makeNewSale` не имеет смысла.

Напомним один из основных принципов UP: стремитесь к максимальной простоте и избегайте использования артефактов, которые не добавляют новых знаний.

#### Описание операции ОП2: `enterItem`

| | |
|---|---|
| **Операция** | `enterItem(itemID: ItemID, quantity: integer)` |
| **Ссылки** | Прецеденты: `Оформление продажи` |
| **Предусловия** | Инициирована продажа |
| **Постусловия** | - Создан экземпляр `sli` класса `SalesLineItem` (*создание экземпляра*)<br>- Экземпляр `sli` связан с текущим экземпляром класса `Sale` (*формирование ассоциации*)<br>- Атрибуту `sli.quantity` присвоено значение `quantity` (*модификация атрибута*)<br>- Экземпляр `sli` связан с классом `ProductSpecification` на основе соответствия идентификатора товара `itemID` (*формирование ассоциации*) |

#### Описание операции ОП3: `endSale`

| | |
|---|---|
| **Операция** | `endSale()` |
| **Ссылки** | Прецеденты: `Оформление продажи` |
| **Предусловия** | Инициирована продажа |
| **Постусловия** | - Атрибут `Sale.iscomplete` принял значение `true` (*модификация атрибута*) |

#### Описание операции ОП4: `makePayment`

| | |
|---|---|
| **Операция** | `makePayment(amount: Money)` |
| **Ссылки** | Прецеденты: `Оформление продажи` |
| **Предусловия** | Инициирована продажа |
| **Постусловия** | - Создан экземпляр `p` класса `Payment` (*создание экземпляра*)<br>- Атрибут `p.amountTendered` принял значение `amount` (*модификация атрибута*)<br>- Экземпляр `p` связан с текущим экземпляром класса `Sale` (*формирование ассоциации*)<br>- Текущий экземпляр `Sale` связан с экземпляром класса `Store` для его добавления в журнал регистрации продаж (*формирование ассоциации*) |

## 13.10. Изменение модели предметной области

При составлении этих описаний предполагалось наличие одного объекта данных, который еще не был представлен в модели предметной области, а именно — завершение ввода информации о покупаемых товарах. Значение этого объекта изменяется при выполнении операции `endSale`, а в операции `makePayment` оно проверяется в предусловии.

Одним из способов представления этой информации является введение атрибута `isComplete` (или `entryIsComplete`) логического типа для объекта `Sale`.

| Sale |
|---|
| isComplete: Boolean<br>date<br>time |

Существуют и альтернативные решения, позволяющие представить изменение состояния системы. Одним из них является *шаблон State*, который будет описываться в главе 34. Еще одно решение состоит в использовании объектов `session`, отвечающих за состояние сеанса. Это решение тоже будет рассмотрено в последующих главах.

## 13.11. Описания, операции и UML

В UML существует формальное определение *операции* (operation). Приведем цитату.

> "Операция — это спецификация преобразования или запроса к объекту" [93].

Например, в терминах UML элементы интерфейса являются операциями. Операция — это абстракция, а не реализация. В отличие от нее, в UML *метод* (method) — это реализация операции.

В UML каждой операции соответствует *сигнатура* (signature), т.е. имя и параметры, а также *спецификация* (operation specification), описывающая результаты выполнения операции или постусловия. Для спецификаций в UML определен свободный формат. Не существует также формата описания операций. Однако в документации по UML приводятся примеры описания операций со стилями оформления пред- и постусловий. Такой подход к описанию операций наиболее распространен.

Итак, в UML определены спецификации операций, имеющие вид описаний с пред- и постусловиями. Как уже отмечалось в этой главе, в спецификации операции не отражаются алгоритм или проектное решение, а только констатируются изменения состояния или результаты выполнения операции.

Помимо описаний открытых операций для всей системы (системных операций) можно создавать описания операций любого уровня детализации, в частности описания открытых операций (или интерфейсов) подсистем, абстрактных классов и т.д. Рассмотренные в этой главе операции относятся к классу системных. В UML операции делятся на классы. Более того, в UML подсистемы моделируются в виде классов (а также в виде пакетов). В контексте UML вся система — это подсистема самого высокого уровня. Она моделируется с помощью класса `System` (на самом деле можно использовать любое имя) с открытыми операциями и спецификациями.

### *Описание операций на языке OCL*

С языком UML связан формальный объектный язык ограничений OCL (Object Constraint Language) [106], который можно использовать для описания ограничений в модели. Язык OCL можно использовать вместо неформального естественного языка, поскольку UML допускает произвольный формат спецификации операций.

> *Совет*
>
> Если нет насущной необходимости в изучении и использовании языка OCL, придерживайтесь принципа простоты и составляйте описания на естественном языке.

В языке OCL определен официальный формат задания пред- и постусловий в описаниях операций. Это демонстрирует следующий пример.

```
System::makeNewSale()
    pre : <операторы языка OCL>
    post : ...
```

Более подробное описание языка OCL не входит в задачи этой книги.

### Проектирование на основе описаний

Форму описания операций с указанием пред- и постусловий, используемую в языке UML для спецификации операций, много лет назад предложил Бертранд Мейер (Bertrand Meyer). Его идеи нашли отражение в подходе к проектированию, получившем название *проектирование на основе описаний* (Design by contract). Этот подход основан на более ранних исследованиях, проводимых в 60-х годах по использованию формальной спецификации языков. При таком подходе для операций тоже составляются описания, причем не только для открытых операций системы или подсистем.

Кроме того, в рамках этого подхода в спецификацию включается раздел инвариантов. В нем определяются элементы, состояние которых не изменяется в процессе выполнения операции. В этой главе инварианты не рассматривались из соображений простоты.

### Языки программирования и описания операций

Некоторые языки программирования, такие как Eiffel, обеспечивают поддержку инвариантов, а также пред- и постусловий. Аналогичные препроцессоры существуют и для Java.

## 13.12. Описания операций в рамках UP

Описания с указанием пред- и постусловий — это стандартный стиль спецификации операций в языке UML. В UML операции существуют на различных уровнях, начиная от системных операций и заканчивая операциями отдельных классов, таких как `Sale`. Спецификации операций системного уровня составляют часть модели прецедентов, хотя они формально не выделяются в документации по UP или RUP. Однако их соответствие модели подтверждено авторами RUP.[1]

### Фазы

**Начало** — на этом этапе описания не составляются, поскольку они должны нести слишком детальную информацию.

**Развитие** — если описания операций вообще создаются, то это происходит на этапе развития в процессе реализации большинства прецедентов. Описание следует составлять только для наиболее сложных и неоднозначных операций.

---

[1] В частной беседе.

### Взаимосвязь артефактов

Взаимосвязи между описаниями операций и другими артефактами на различных уровнях детализации показаны на рис. 13.2 и 13.3.

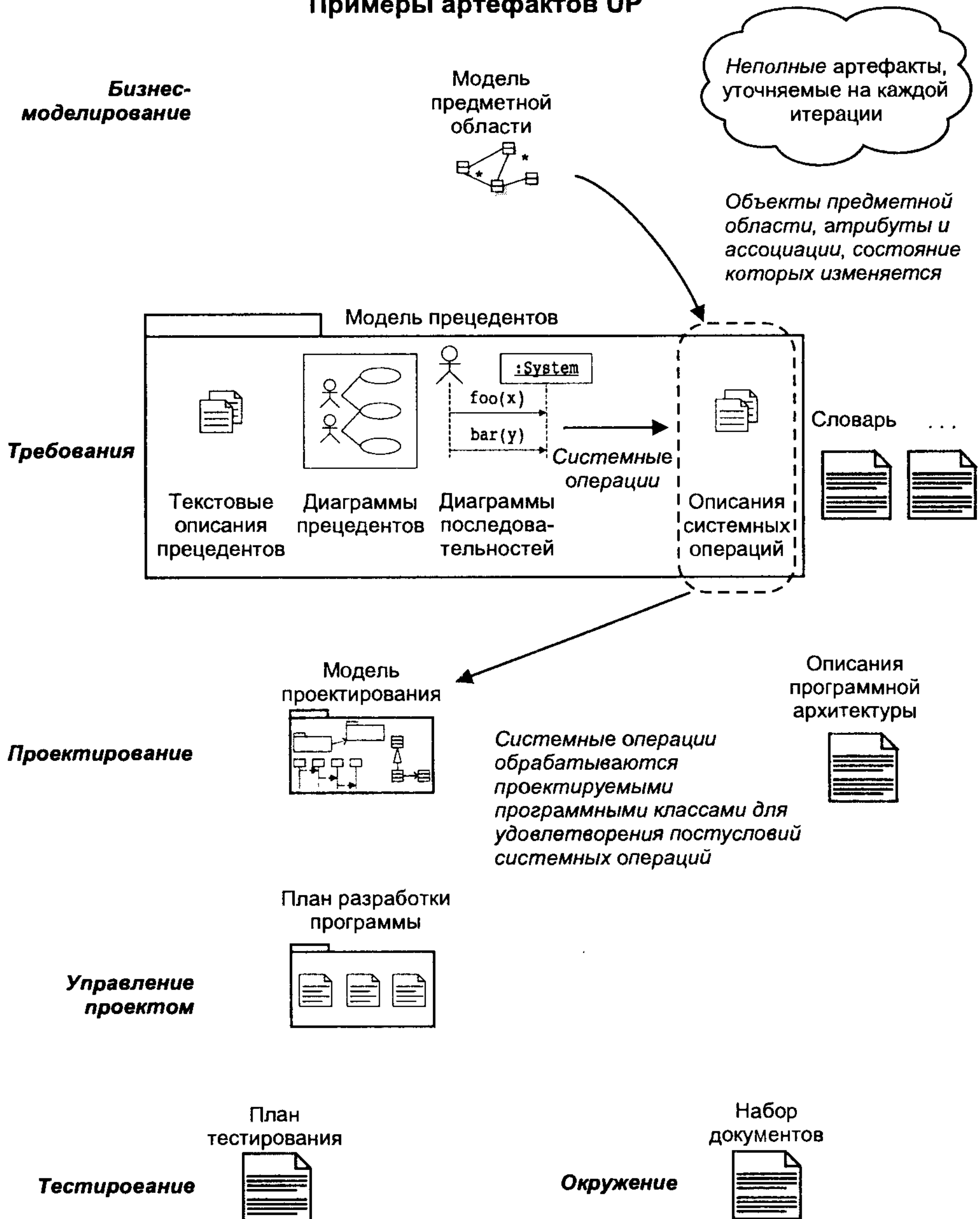



*Рис. 13.2. Пример взаимосвязи артефактов UP*

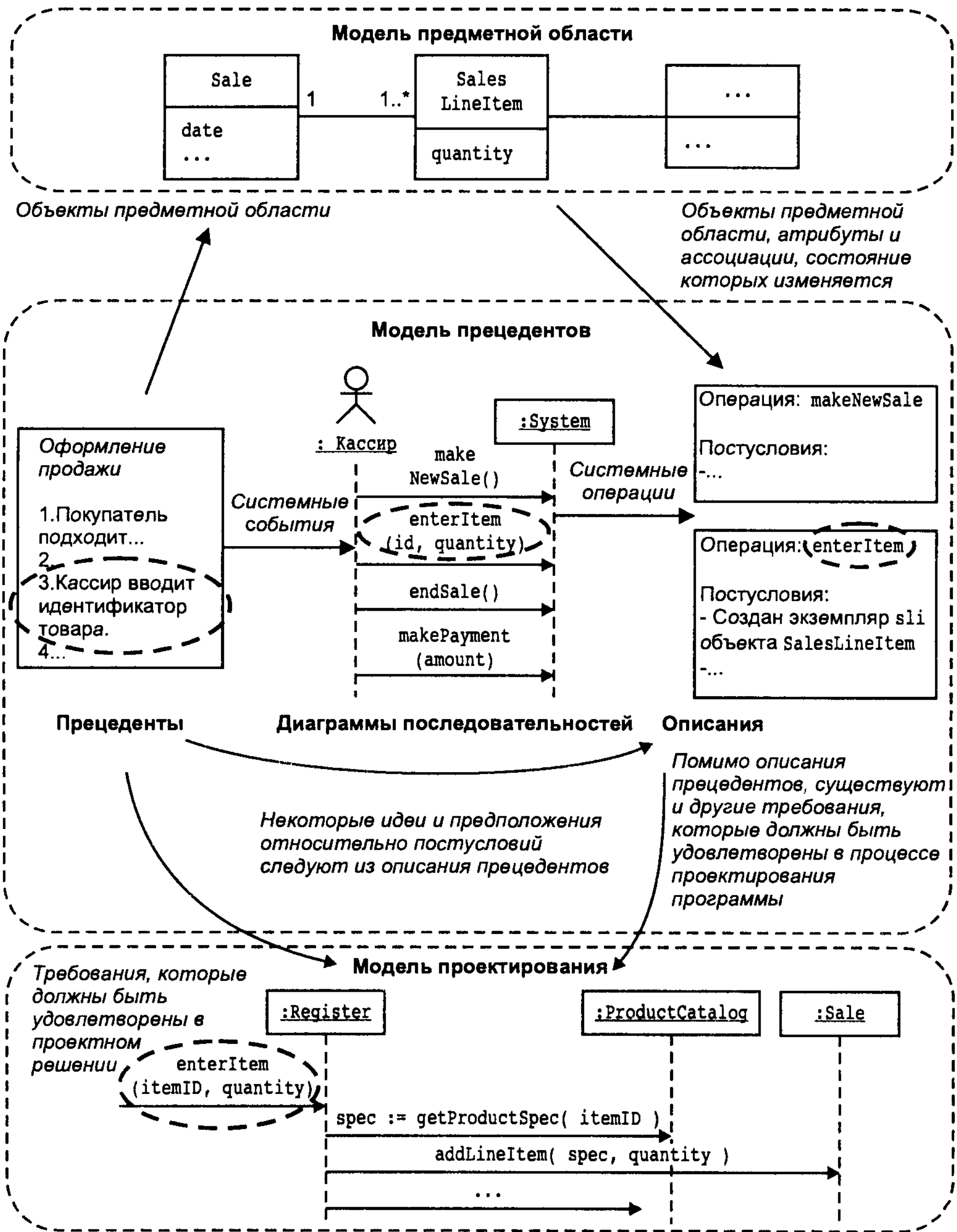



*Рис. 13.3. Взаимосвязь описаний операций с другими артефактами*

## 13.13. Дополнительная литература

Описание операций "уходит корнями" в область формальных спецификаций, которая развивалась с 60-х годов, в частности, в рамках метода разработки VDM (Vienna Development Method) [13]. По методу VDM и другим языкам формаль-

ных спецификаций существует обширная библиография. Значительный вклад в развитие формальных спецификаций и описаний с указанием пред- и постусловий в рамках языка Eiffel внес Бертранд Мейер (Bertrand Meyer). Он же ввел термин "проектирование на основе описаний".

Более строгое описание операций в рамках UML обеспечивает язык OCL, которому посвящена книга Вормера и Клеппа [106].

Глава

# От анализа требований к проектированию на данной итерации

Основные задачи

- Обосновать переход к проектированию.
- Продемонстрировать более важную роль навыков проектирования по сравнению со знанием системы обозначений UML.

## Введение

Итак, к настоящему моменту в рамках данной итерации, в основном, исследованы требования, понятия и операции, связанные с системой. Согласно принципам UP, на протяжении начальной фазы нужно проанализировать примерно 10% требований, а на первой итерации стадии развития начинаются более глубокие исследования. В следующих главах основное внимание будет смещено в область разработки проектного решения для данной итерации в терминах взаимодействия программных объектов.

## 14.1. Занимаясь итеративной разработкой, делайте это правильно

В процессе анализа требований и объектно-ориентированного анализа основное внимание уделялось правильной организации деятельности, т.е. изучению основных целей POS-системы NextGen, а также связанных с ними правил и ограничений. На следующем этапе проектирования акцент смещается в сторону правильной реализации поставленных целей, т.е. разработки грамотного проектного решения, удовлетворяющего требованиям данной итерации.

При итеративной разработке переход от анализа требований к проектированию и реализации выполняется на каждой итерации. Более того, в процессе проектирования и реализации на начальных итерациях некоторые требования естественным

образом изменяются. Благодаря этому проясняются задачи проектирования на данной итерации и уточняются требования, реализуемые на последующих итерациях. В результате, к моменту завершения фазы развития, требования к системе в основном стабилизируются, и примерно 80% из них определены детально.

## 14.2. Возможно, на это потребуются недели? Вовсе нет

После подробного обсуждения этапа анализа на страницах нескольких глав может создаться впечатление, что для моделирования требуется несколько недель. Вовсе нет. Если разработчик умеет описывать прецеденты, строить модель предметной области и т.д., то на выполнение всех видов деятельности по моделированию потребуется от силы несколько дней.

Это не означает, что проектирование системы начинается через несколько дней после начала проекта. Несколько недель занимает подготовка: создание программных прототипов для проверки правильности концепции (proof-of-concept programming), поиск ресурсов (человеческих, программных и т.д.), планирование, создание окружения и т.п.

## 14.3. Переход к проектированию

На этапе объектного проектирования разрабатывается логическое решение на основе объектно-ориентированной парадигмы. Основной задачей этого этапа является создание *диаграмм взаимодействия* (interaction diagram), иллюстрирующих взаимодействие объектов в процессе выполнения системных требований.

После создания диаграмм взаимодействия (или параллельно с ними) можно построить *диаграммы классов*, отражающие определения программных классов и интерфейсов, реализованных в программе.

В терминах UP эти артефакты являются частью *модели проектирования* (design model).

> На практике создание диаграмм взаимодействия и диаграмм классов происходит параллельно, поскольку эти процессы тесно взаимосвязаны. Однако для простоты и ясности при рассмотрении данного примера эти виды деятельности описываются последовательно.

### *Важнее иметь навыки объектного проектирования, чем знать систему обозначений UML*

В последующих главах рассматривается процесс создания этих артефактов, или точнее, навыки объектного проектирования, лежащие в основе их создания. Очень важно уметь мыслить объектами и проектировать в рамках объектного подхода. Это гораздо важнее, чем просто знать систему обозначений языка UML. В то же время стандартный язык визуального моделирования обеспечивает дополнительные возможности для выполнения этих видов деятельности, поэтому будет представлена система обозначений UML для поддержки проектирования.

Из двух упомянутых выше артефактов наиболее важными (с точки зрения разработки хорошего проекта) являются диаграммы взаимодействия, для создания которых требуются значительные творческие усилия. При построении диаграмм взаимодействия происходит распределение *обязанностей* (responsibilities) на основе ис-

пользования принципов проектирования и *шаблонов* (patterns). Поэтому в следующих главах основное внимание уделяется принципам распределения обязанностей и применению шаблонов в процессе объектного проектирования.

> *Навыки объектного проектирования и система обозначений UML*
>
> Диаграммы взаимодействия на языке UML отражают принимаемые проектные решения, поэтому навыки объектного проектирования лежат в основе построения диаграмм UML.
>
> При создании диаграмм взаимодействий необходимо знать следующее.
>
> - Принципы распределения обязанностей.
> - Шаблоны проектирования.

Глава

# Система обозначений для диаграмм взаимодействия

*Кошки умнее собак. Вы не сможете заставить восемь кошек тянуть сани по снегу.*

*Джеф Валдез (Jeff Valdez)*

Основная задача

- Ознакомиться с системой обозначений языка UML для построения диаграмм взаимодействия (последовательностей и кооперации).

## Введение

В следующих главах рассматриваются вопросы объектного проектирования. Для иллюстрации проектных решений используются, в первую очередь, диаграммы взаимодействия. Поэтому желательно познакомиться с примерами, приведенными в этой главе, и системой обозначений языка UML, используемой для построения таких диаграмм.

В состав языка UML входят обозначения для *диаграмм взаимодействий* (interaction diagrams). Они иллюстрируют способ взаимодействия объектов с помощью сообщений. В этой главе вводится система обозначений, а в последующих главах внимание читателя будет сфокусировано на их использовании в процессе объектного проектирования на примере системы NextGen.

### *Рекомендации по проектированию приводятся в следующих главах*

В этой главе лишь вводится система обозначений. Для грамотного объектного проектирования необходимо понимать его основные принципы. После ознакомления с системой обозначений для диаграмм взаимодействия в следующих главах вы ознакомитесь с этими принципами и их применением для построения диаграмм взаимодействия.

## 15.1. Диаграммы последовательностей и кооперации

Термин "диаграмма взаимодействия" используется в качестве общего названия для двух следующих конкретных типов диаграмм, которые могут использоваться для иллюстрации обмена сообщениями.

- Диаграммы кооперации (collaboration diagram)
- Диаграммы последовательностей (sequence diagram)

Чтобы подчеркнуть свободу разработчиков при выборе артефактов проектирования, в книге будут использованы оба типа диаграмм.

*Диаграммы кооперации* (collaboration diagram) иллюстрируют взаимодействие объектов в формате графа или сети, как показано на рис. 15.1. При этом объекты могут размещаться в любом месте диаграммы.

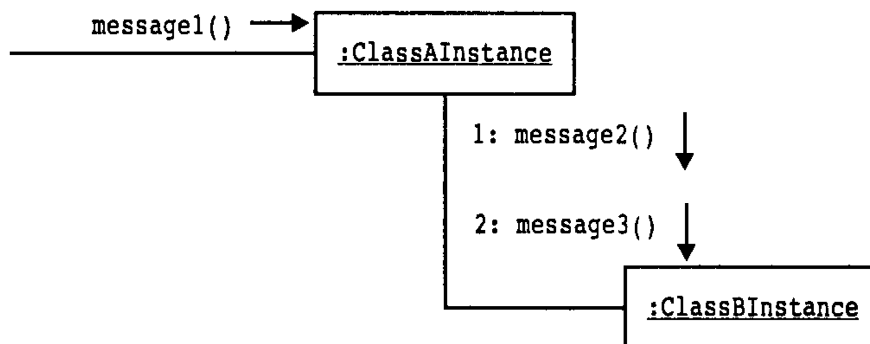



*Рис. 15.1. Диаграмма кооперации*

*Диаграммы последовательностей* (sequence diagram) иллюстрируют взаимодействие в форме, показанной на рис. 15.2. Здесь объекты располагаются слева направо.

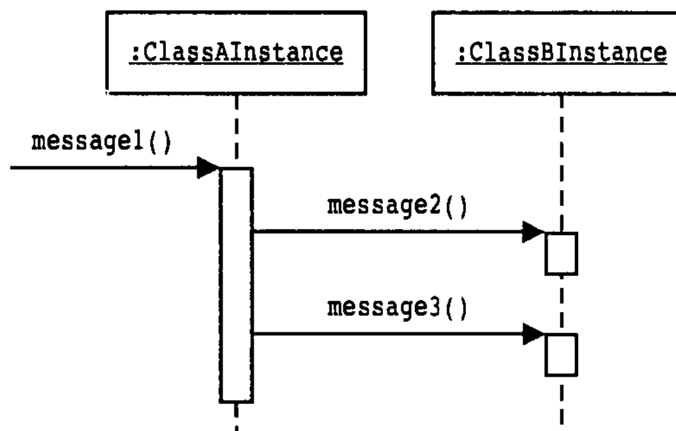



*Рис. 15.2. Диаграмма последовательностей*

Оба типа диаграмм имеют свои преимущества и недостатки. При изображении диаграмм на листах ограниченной ширины удобнее использовать диаграммы кооперации, поскольку новые объекты к этим диаграммам можно добавлять в нижней части страницы. При использовании диаграмм последовательностей новые объекты добавляются справа, что не всегда можно сделать в условиях ограниченной ширины страницы. Однако на диаграммах кооперации сложнее отслеживать последовательность передачи сообщений.

При использовании CASE-средств разработки многие предпочитают строить диаграммы последовательностей для удобства обратного проектирования — преобразования исходного кода в диаграмму взаимодействия.

| Тип диаграммы | Преимущества | Недостатки |
|---|---|---|
| Последовательностей | Ясно отображает последовательность и временной порядок сообщений. Простые обозначения | Расширяется вправо при добавлении новых объектов; занимает много места по горизонтали |
| Кооперации | Экономия пространства — возможность добавления объектов в двух направлениях. Лучше иллюстрирует сложные зависимости, итерационность и параллельные события | Сложнее отследить последовательность сообщений. Более сложная система обозначений |

## 15.2. Пример диаграммы кооперации: makePayment

Показанную на рис. 15.3 диаграмму кооперации нужно интерпретировать следующим образом.

1. Сообщение makePayment передается экземпляру объекта Register. Отправитель сообщения не определен.
2. Объект Register передает сообщение makePayment экземпляру объекта Sale.
3. Объект Sale создает экземпляр объекта Payment.

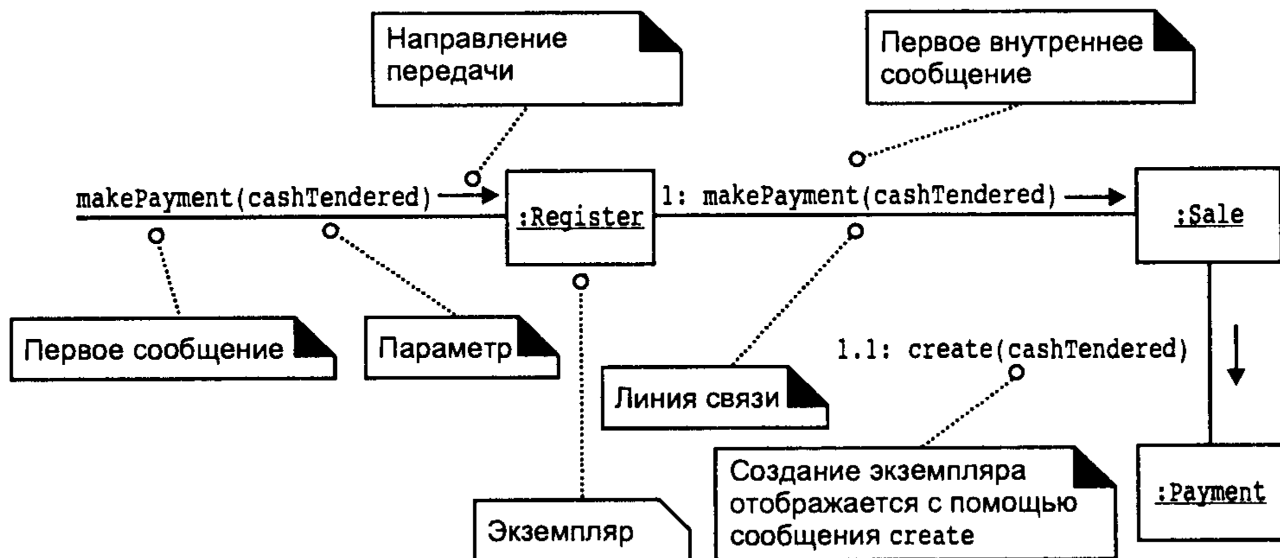



*Рис. 15.3. Диаграмма кооперации*

## 15.3. Пример диаграммы последовательностей

На рис. 15.4 в виде диаграммы последовательностей изображена информация, представленная на приведенной выше диаграмме кооперации.

## 15.4. Диаграммы взаимодействия — это важный артефакт

Стандартной проблемой проектов, в которых применяется объектная технология, является недопонимание важности создания диаграмм взаимодействий и тщательного распределения обязанностей. С ней связана и другая проблема — эти диаграммы зачастую носят неопределенный характер. А именно: на них отображается процесс передачи сообщений между объектами, требующими более

детальной проработки. Например, на диаграмме отображается передача сообщения runSimulation некоторому объекту Simulation без дальнейшего более глубокого проектирования. Разработчики иногда пребывают в заблуждении и считают, что после правильного отображения процесса передачи сообщений с продуманными именами проектирование системы магическим образом завершается.

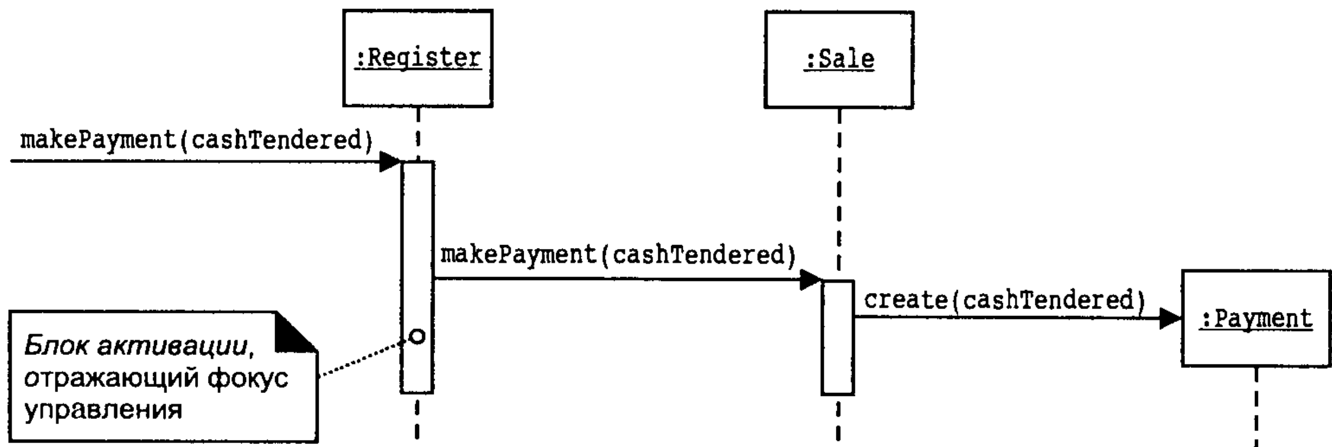



*Рис. 15.4. Диаграмма последовательностей*

Значительную часть усилий, затрачиваемых на выполнение проекта, необходимо затратить на построение диаграмм взаимодействий, отражающих продуманные детали проектного решения. Например, если длительность итерации составляет две недели, то в начале этой итерации полдня или даже целый день нужно затратить на построение диаграмм взаимодействия (а параллельно с ними и диаграмм классов). Только после этого можно приступать к программной реализации. Да, отображаемая на диаграммах информация является неполной, она будет дополнена и уточнена на стадии программирования. Диаграммы взаимодействия обеспечивают хороший "фундамент" для более детального продумывания проектного решения в процессе программной реализации.

> *Совет*
>
> Создавая диаграммы взаимодействия, разбейте разработчиков на пары. Диаграммы лучше разрабатывать парами, а не в одиночку. Это будет способствовать более эффективному проектированию и взаимному обучению членов коллектива.

На этом шаге потребуется применение соответствующих навыков разработки в терминах шаблонов, идиом и принципов. Вообще говоря, создавать прецеденты, модели предметной области и другие артефакты проще, чем распределять обязанности и создавать правильно спроектированные диаграммы взаимодействий. Это объясняется тем, что в этом процессе применяется гораздо больше "трудноуловимых" принципов разработки и имеется больше степеней свободы, чем при создании любого другого артефакта объектно-ориентированного анализа и проектирования.

> Диаграммы взаимодействий являются одним из наиболее важных артефактов, создаваемых при объектно-ориентированном анализе и проектировании.
>
> Для улучшения качества разрабатываемых диаграмм взаимодействий можно применять систематизированные шаблоны, принципы и идиомы.

Для успешного конструирования диаграмм взаимодействий принципы разработки предварительно *могут* быть систематизированы и проанализированы. Такой подход к пониманию и использованию этих принципов основывается на *шаблонах* (patterns), представляющих собой структурированные рекомендации и принципы. Таким образом, после знакомства с синтаксисом диаграмм взаимодействия основное внимание (в последующих главах) будет уделено шаблонам разработки и их применению к построению таких диаграмм.

## 15.5. Основные обозначения для диаграмм взаимодействия

### Отображение классов и экземпляров объектов

В языке UML для иллюстрации *экземпляров* объектов (а не классификаторов) используется простой и непротиворечивый подход (рис. 15.5).

- Для экземпляра любого элемента языка UML (класса, исполнителя и т.д.) используется то же самое графическое обозначение, что и для типа, однако при этом соответствующая определяющая строка *подчеркивается*.

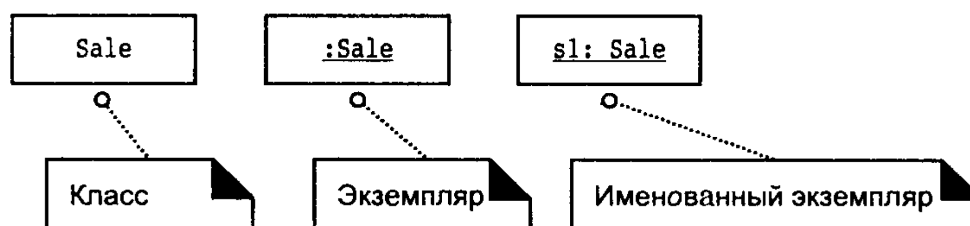



*Рис. 15.5. Классы и экземпляры объектов*

Таким образом, для отображения экземпляра класса на диаграмме взаимодействий используется обычное графическое условное обозначение класса, однако при этом его имя подчеркивается.

Для уникальной идентификации экземпляра класса может использоваться его имя. Если имя экземпляра не указано, на диаграмме кооперации перед именем класса ставится двоеточие (:).

### Синтаксис для отображения сообщений

В языке UML существует стандартный синтаксис для обозначения передачи сообщений.

```
получатель := сообщение (параметр : типПараметра) : типПолучателя
```

Информация о типах может исключаться в силу своей очевидности или незначительности. Например:

```
spec := getProductSpect(id)
spec := getProductSpect(id:ItemID)
spec := getProductSpect(id:ItemID) : ProductSpecification
```

## 15.6. Основные обозначения диаграммы кооперации

### Отображение связей

*Связь* (link) является соединением между двумя экземплярами классов, определяющим некоторую форму перемещения и видимости между ними (рис. 15.6). Более строго, можно сказать, что связь является экземпляром ассо-

циации. Например, имеется связь, или маршрут перемещения, от объекта Register к объекту Sale, в соответствии с которым могут передаваться сообщения, такие как makePayment.

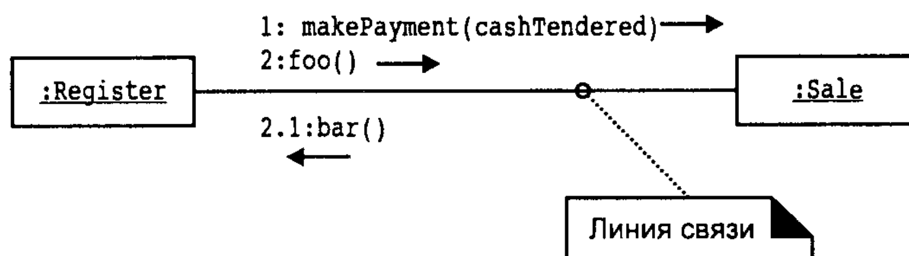



*Рис. 15.6. Линии связей*

> Обратите внимание, что по одной и той же линии связи могут передаваться несколько сообщений в обоих направлениях.

## Отображение сообщений

Передаваемые между объектами сообщения представляются в виде имен этих сообщений над линиями связей, помеченных стрелками. Над одной линией связи может быть указано любое количество сообщений (рис. 15.7). Для отображения порядка следования сообщений в текущем потоке управления рядом с сообщением приводится порядковый номер.

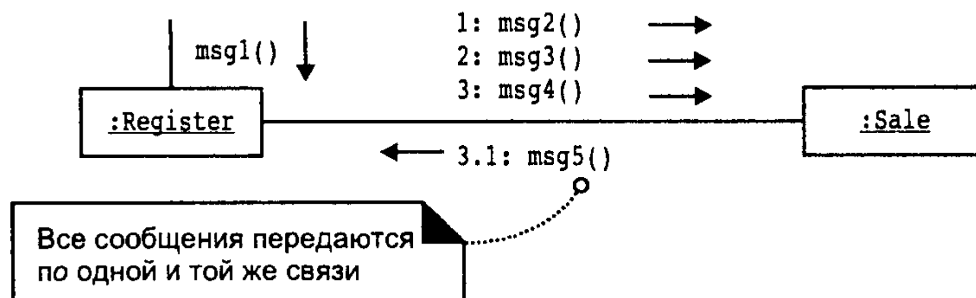



*Рис. 15.7. Сообщения*

## Сообщения, передаваемые самому объекту

Сообщение может передаваться объектом самому себе (рис. 15.8). Такой случай отображается с помощью связи объекта с самим собой, когда сообщения передаются в направлении этой связи.

## Создание экземпляров объектов

Для создания экземпляра объекта можно использовать любые сообщения. Однако в языке UML принято использовать для этой цели сообщение create (создать). При использовании другого имени (возможно, менее очевидного) сообщение следует снабдить специальным свойством, получившим название стереотипа UML, в частности <<create>>. Это сообщение передается создаваемому экземпляру объекта (рис. 15.9).

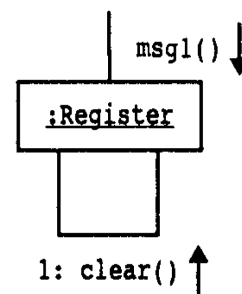



*Рис. 15.8. Сообщения, передаваемые самому объекту*

Сообщение `create` может иметь параметры, задающие начальные значения свойств создаваемого экземпляра. Такая форма записи может означать, например, вызов конструктора с параметрами на языке Java.

Более того, чтобы подчеркнуть факт создания экземпляра, новому экземпляру объекта можно дополнительно присвоить свойство `{новый}`.

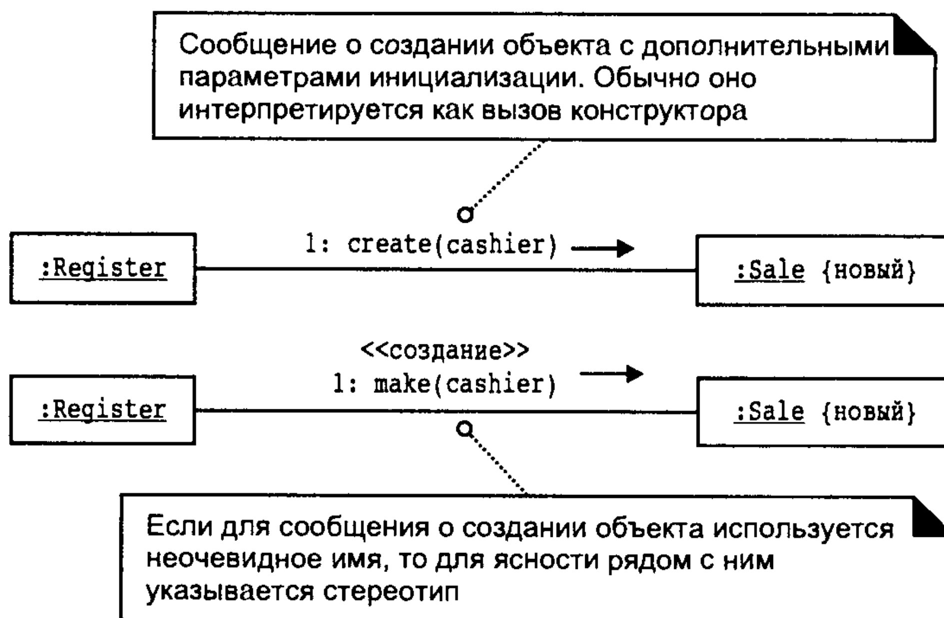



*Рис. 15.9. Создание экземпляра объекта*

## Представление порядка передачи сообщений

Порядок передачи сообщений иллюстрируется с помощью *порядковых номеров* (sequence number) (рис. 15.10). При этом используется следующая схема нумерации.

1. Первое сообщение не нумеруется. Таким образом, сообщение `msg1()` является ненумерованным.
2. Порядок и вложенность последующих сообщений отображается в соответствии с принятой схемой нумерации. При ее использовании к вложенным сообщениям добавляется номер. Вложенность означает, что к номеру исходящего сообщения добавляется номер входящего сообщения.

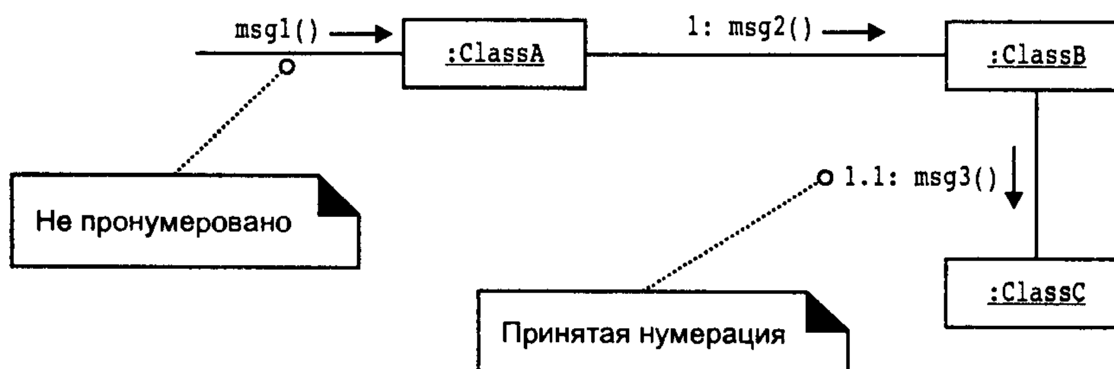



*Рис. 15.10. Нумерация последовательности сообщений*

На рис. 15.11 показан более сложный случай.

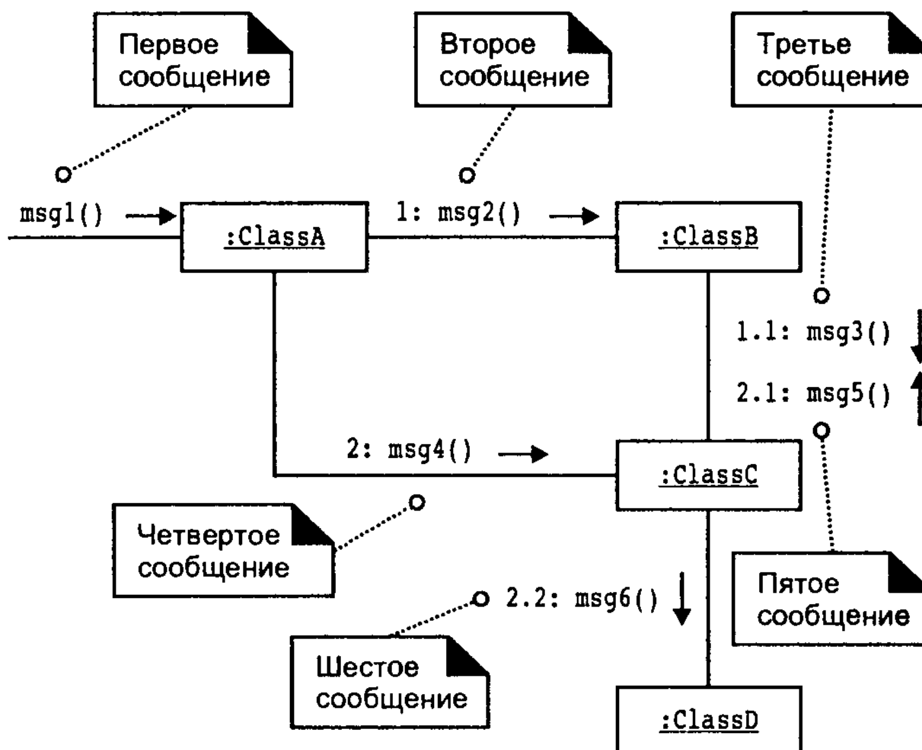



*Рис. 15.11. Сложная нумерация последовательности сообщений*

## Представление условных сообщений

Условное сообщение (рис. 15.12) изображается с помощью его номера, за которым в квадратных скобках указывается условное выражение, аналогичное условию цикла. Сообщение передается только в том случае, когда оператором возвращается значение `true`.

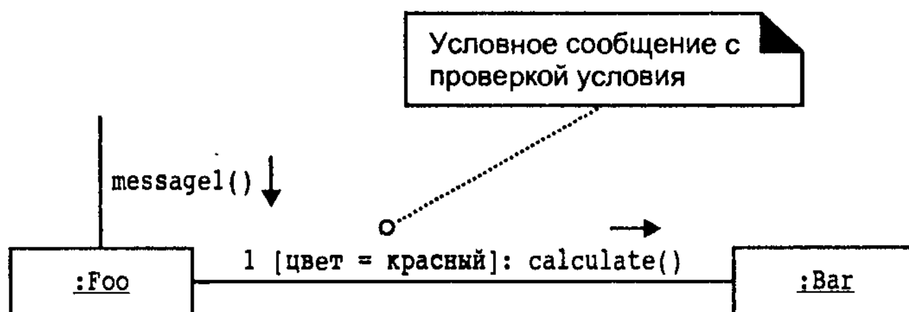



*Рис. 15.12. Условное сообщение*

## Представление взаимоисключающих условных маршрутов

Приведенный на рис. 15.13 пример иллюстрирует порядковые номера сообщений при использовании взаимоисключающих условных маршрутов.

В этом случае требуется модифицировать схему нумерации, воспользовавшись символом условного маршрута. В соответствии с принятым соглашением, первым таким символом является буква `a`. Фрагмент диаграммы, приведенной на рис. 15.13, означает, что после передачи сообщения `msg1()` будет передано либо сообщение `1a`, либо сообщение `1b`. Наличие номера `1` означает, что оба сообщения относятся к первому внутреннему сообщению.

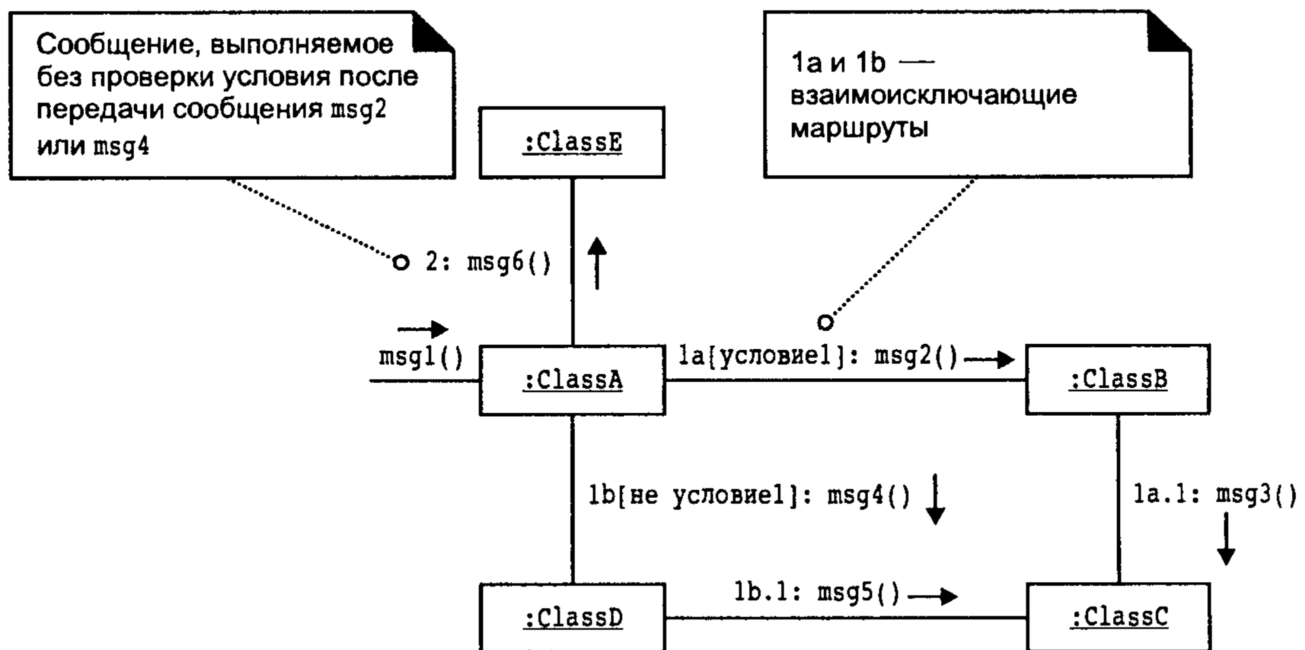



*Рис. 15.13. Взаимоисключающие сообщения*

Обратите внимание, что последующие вложенные сообщения по-прежнему нумеруются согласно соответствующим внешним сообщениям. Таким образом, `1b.1` является вложенным сообщением сообщения `1b`.

### Представление итерационного процесса или циклов

Обозначения для итерационного процесса показаны на рис. 15.14. Итерационный процесс можно отобразить, указав за порядковым номером сообщения символ *.

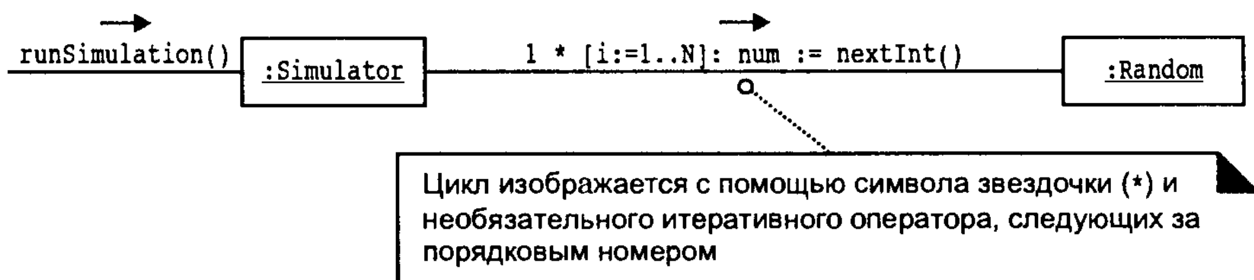



*Рис. 15.14. Итерационный процесс*

### Итерационный процесс для коллекций объектов

Зачастую сообщение передается каждому элементу коллекции (например, списку или карте). Для этого можно использовать некий вид объекта-итератора, например, в Java реализацию `java.util.Iterator` или итератор из стандартной библиотеки C++. В UML для обозначения набора экземпляров или коллекции применяется термин *сложный объект* или *мультиобъект* (multiobject). На диаграмме кооперации сложный объект отображается с использованием условного обозначения, показанного на рис. 15.15.

### Сообщения, передаваемые классу

Сообщения могут передаваться самому классу, а не его экземплярам. Это может понадобиться, например, для вызова статических методов класса. При этом сообщение изображается как обычно, однако в условном обозначении клас-

са его имя не подчеркнуто. Тем самым указывается, что это сообщение передается самому классу, а не его экземпляру (рис. 15.16).

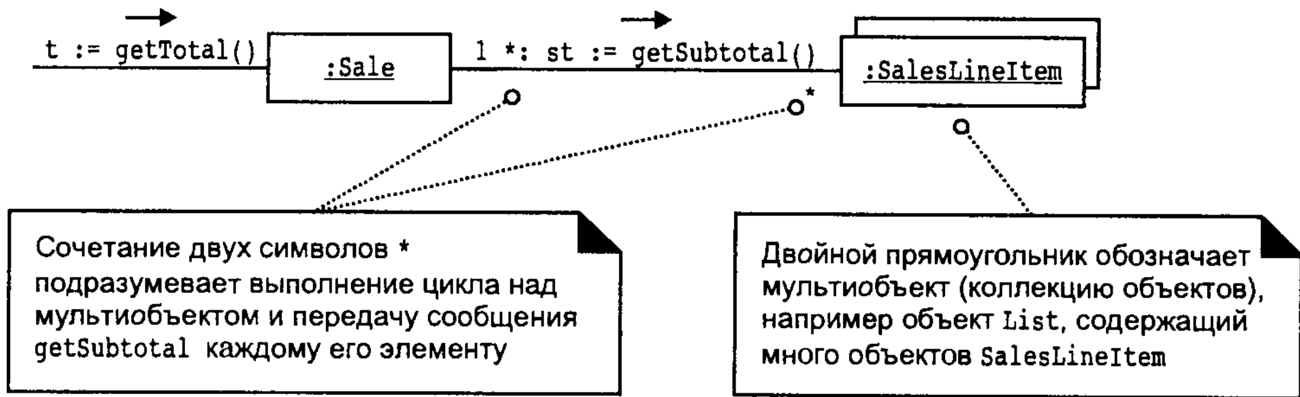



*Рис. 15.15. Итерационный процесс для сложного объекта*

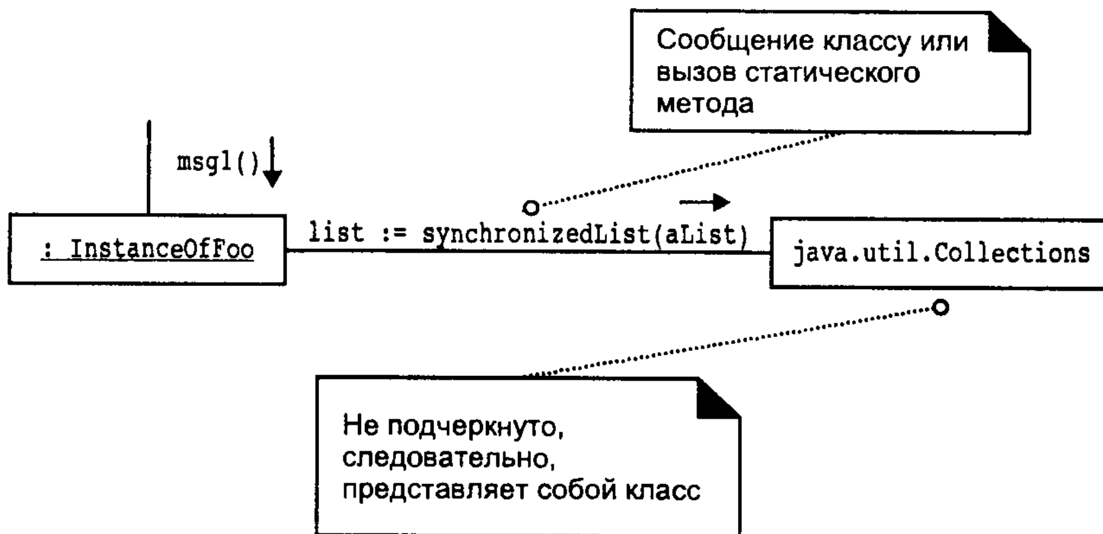



*Рис. 15.16. Сообщения, передаваемые объекту класса (вызов статических методов)*

Очень важно, чтобы там, где это нужно, имена экземпляров были подчеркнуты. В противном случае передаваемые сообщения могут быть неверно интерпретированы.

## 15.7. Основные обозначения диаграммы последовательностей

### *Связи*

В отличие от диаграмм кооперации, на диаграмме последовательностей связи не отображаются.

### *Сообщения*

Сообщения между объектами изображаются в виде соединяющих объекты линий со стрелками на конце, над которыми указывается имя сообщения. Порядок передачи сообщений определяется их расположением сверху вниз.

### Фокус управления и блоки активации

Как видно из рис. 15.17, на диаграммах последовательностей можно отображать фокус управления с использованием *блока активации* (activation box). Блоки активации указывать необязательно, но специалисты по UML их обычно используют.

### Возвращаемые значения

На диаграммах последовательностей при желании можно отражать возврат значения при передаче сообщения. Для этого используются штриховые линии со стрелками на конце, исходящие из блока активации (рис. 15.18). Многие опытные специалисты их не применяют. Однако иногда с помощью этих линий изображают возвращаемые значения (если таковые имеются).

### Создание экземпляров объектов

Обозначения, иллюстрирующие создание экземпляра, показаны на рис. 15.20.

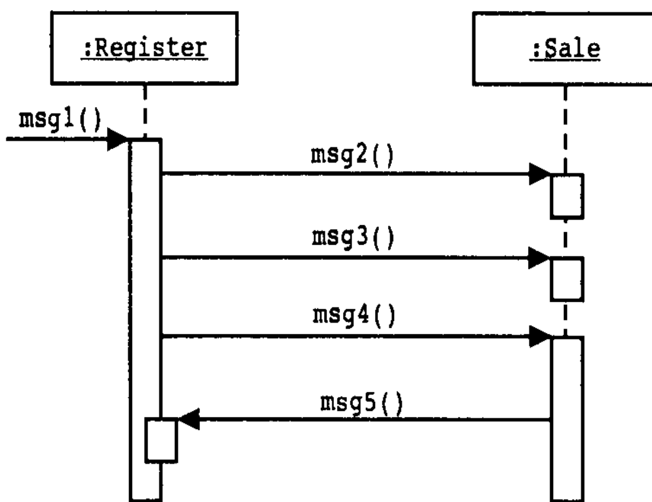



*Рис. 15.17. Сообщения и фокус управления*

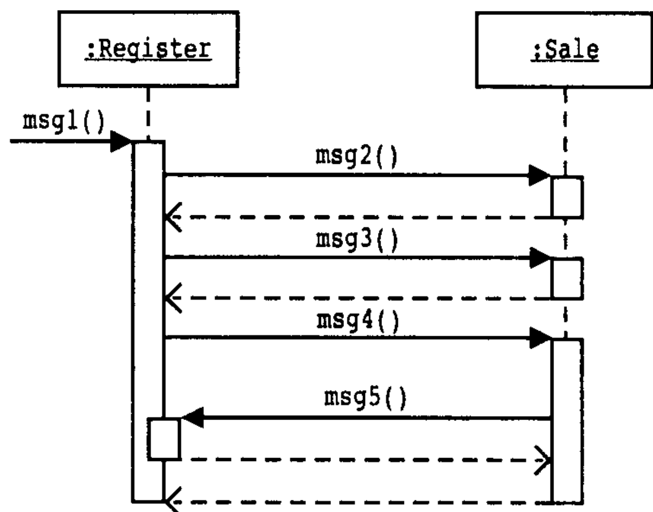



*Рис. 15.18. Отображение возврата значения*

### Сообщения, передаваемые самому объекту

Передача сообщения объектом самому себе отображается с использованием вложенных активационных блоков (рис. 15.19).

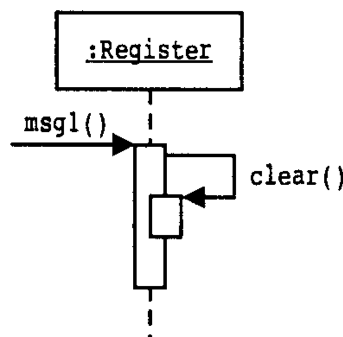



*Рис. 15.19. Сообщения, передаваемые самому объекту*

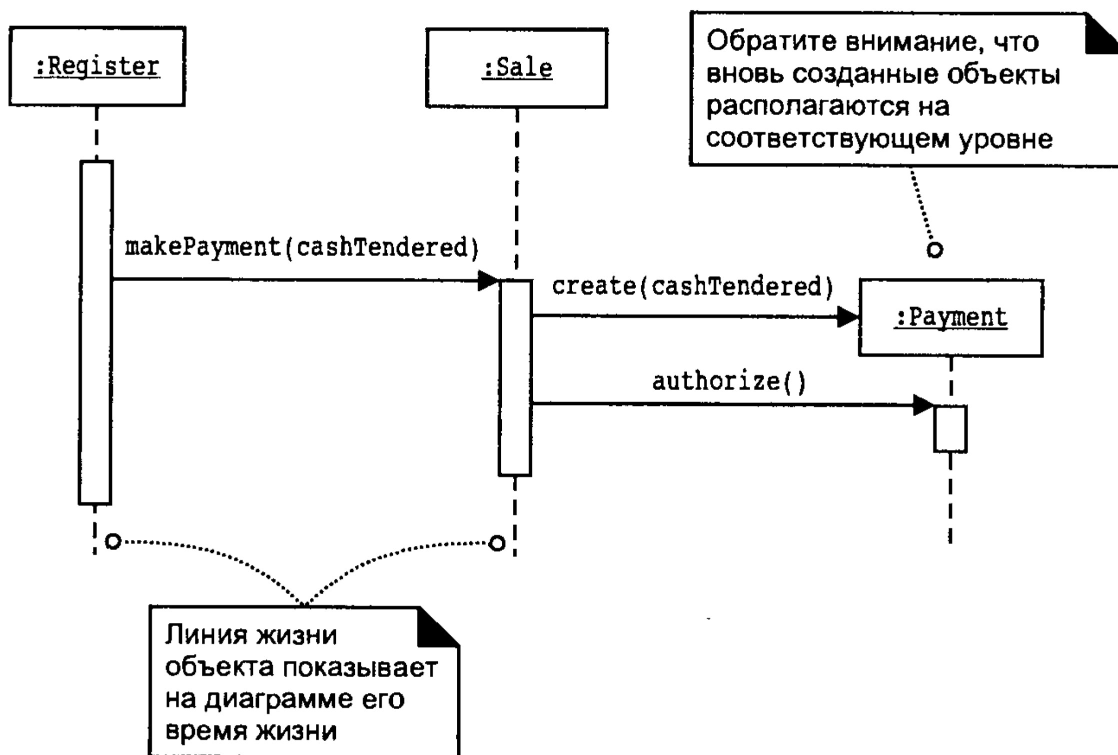



*Рис. 15.20. Создание экземпляра объекта и линии жизни*

### Линии жизни объектов и уничтожение объектов

На рис. 15.20 показаны также *линии жизни объектов* (object lifelines) — вертикальные штриховые линии, расположенные под соответствующими объектами. Иногда желательно отобразить на диаграмме факт уничтожения объекта (например, в языке C++ отсутствует механизм сборки мусора). Этот факт можно отобразить в терминах UML с помощью специального символа на линии жизни объекта (рис. 15.21).

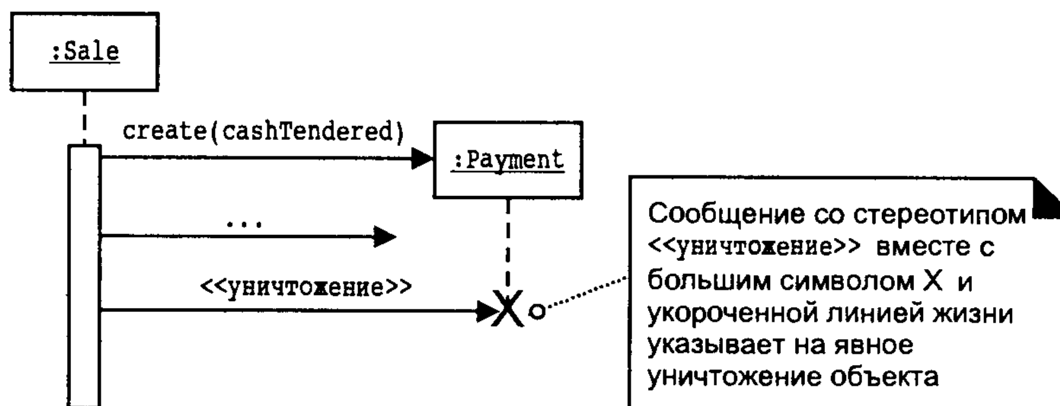



*Рис. 15.21. Уничтожение объекта*

### Представление условных сообщений

Условное сообщение показано на рис. 15.22.

### Представление взаимоисключающих условных маршрутов

Для этого случая используются линии сообщений, исходящие из одной точки под разными углами, как показано на рис. 15.23.

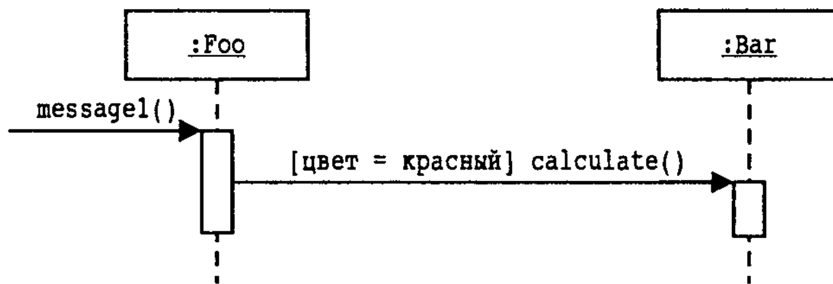



*Рис. 15.22. Условное сообщение*

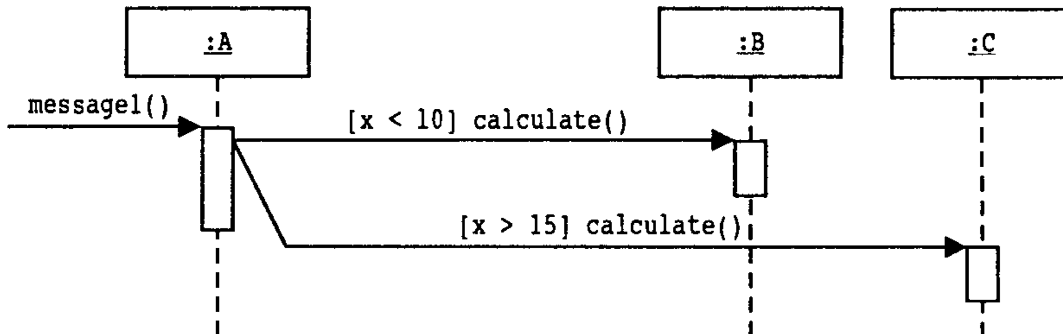



*Рис. 15.23. Взаимоисключающие сообщения*

## Представление итерационного процесса для одного сообщения

Обозначения итерационного процесса для одного сообщения показаны на рис. 15.24.

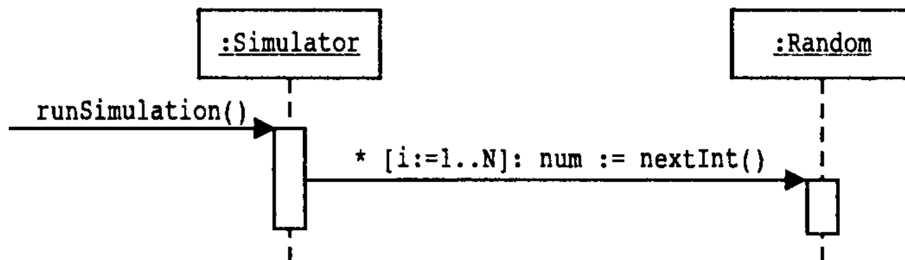



*Рис. 15.24. Итерационный процесс для одного сообщения*

## Итерационный процесс для последовательности сообщений

Обозначения итерационного процесса для последовательности сообщений показаны на рис. 15.25.

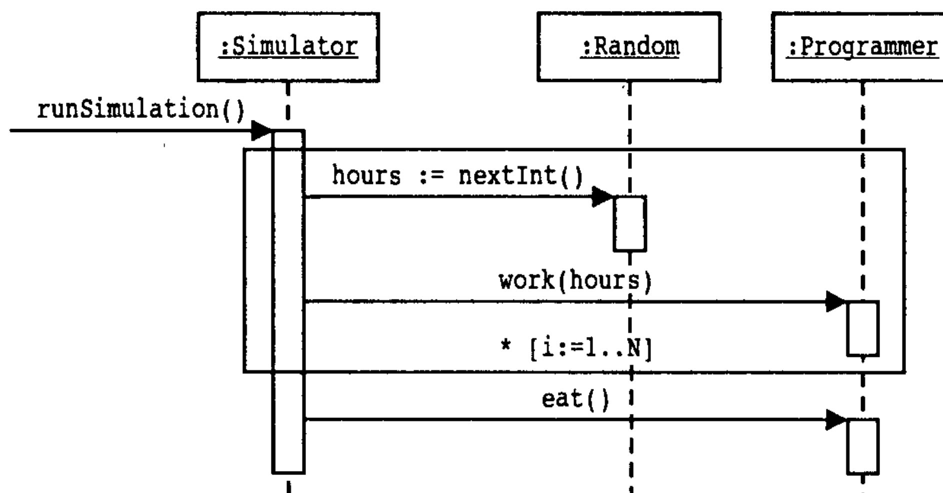



*Рис. 15.25. Итерационный процесс для последовательности сообщений*

## Итерационный процесс для коллекции (сложного объекта)

Обозначения итерационного процесса для коллекции объектов показаны на рис. 15.26.

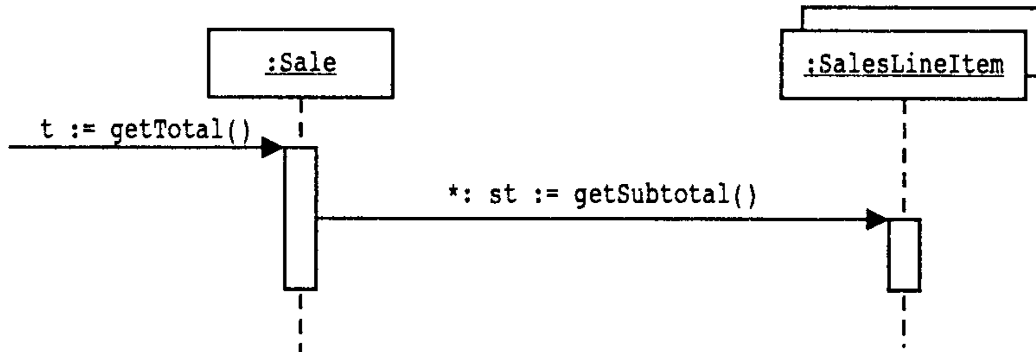



*Рис. 15.26. Итерационный процесс для сложного объекта*

На диаграмме кооперации для иллюстрации передачи сообщения каждому элементу, а не самой коллекции, в конце каждой роли указывается символ кратности "*". Для диаграммы последовательностей аналогичное обозначение в языке UML отсутствует.

## Сообщения, передаваемые классу

Как и на диаграмме кооперации, при вызове статических методов или методов класса имя классификатора не подчеркивается. Тем самым указывается, что это сообщение передается самому классу, а не его экземпляру (рис. 15.27).

*Рис. 15.27. Сообщения, передаваемые объекту класса (вызов статических методов)*

# GRASP: ШАБЛОНЫ ДЛЯ РАСПРЕДЕЛЕНИЯ ОБЯЗАННОСТЕЙ

*По мнению большинства экспертов, наиболее вероятный способ разрушения мира — это случайность. К такому же выводу приходим мы — профессионалы в компьютерной области. Нами движет случай.*

*Натаниэль Боренштейн (Nathaniel Borenstein)*

**Основные задачи**

- **Определить шаблоны.**
- **Научиться применять шаблоны GRASP.**

## Введение

Процесс объектного проектирования иногда описывают следующим образом.

> Сначала определяются требования и создается модель предметной области, затем добавляются методы программных классов, описывающие передачу сообщений между объектами для удовлетворения требованиям.

Такое описание мало полезно на практике, поскольку в нем не учтены глубинные принципы и вопросы, лежащие в основе этого процесса. Вопрос определения способов взаимодействия объектов и принадлежности методов чрезвычайно важен и отнюдь не тривиален. Его решение требует внимательного изучения в процессе построения диаграмм и программирования.

Это очень важный этап, представляющий собой "сердцевину" разработки объектно-ориентированной системы. Он не сводится к построению диаграмм предметной области, диаграмм пакетов и т.д.

### *GRASP — это методический подход к объектному проектированию*

Принципы объектного проектирования отражены в шаблонах проектирования GRASP, изучение и применение которых позволит освоить методический подход и снять завесу таинственности с процесса разработки.

Шаблоны GRASP позволяют понять основные принципы объектного проектирования и методично применять их. Эти шаблоны называют также *шаблонами распределения обязанностей* (pattern of assigning responsibilities).

## 16.1. Обязанности и методы

В UML *обязанность* (responsibility) определяется как "контракт или обязательство классификатора" [86]. Обязанности описывают поведение объекта. В общем случае можно выделить два типа обязанностей.

1. *Знание* (knowing).
2. *Действие* (doing).

Ниже перечислены обязанности, относящиеся к *действиям* объекта.

- Выполнение некоторых действий самим объектом, например, создание экземпляра или выполнение вычислений.
- Инициирование действий других объектов.
- Управление действиями других объектов и их координирование.

Ниже перечислены обязанности, относящиеся к *знаниям* объекта.

- Наличие информации о закрытых инкапсулированных данных.
- Наличие информации о связанных объектах.
- Наличие информации о следствиях или вычисляемых величинах.

Обязанности присваиваются объектам в процессе объектно-ориентированного проектирования. Например, можно сказать, что объект `Sale` отвечает за создание экземпляра `SalesLineItems` (действие) или что объект `Sale` отвечает за наличие информации о стоимости покупки (знание). Обязанности, относящиеся к разряду "знаний", зачастую вытекают из модели предметной области, поскольку она иллюстрирует атрибуты и ассоциации.

Возможность реализации обязанностей в виде классов и методов зависит от точности их описаний. Например, реализация обязанности "обеспечения доступа к реляционным базам данных" может потребовать создания десятков классов и сотен методов, а для реализации обязанности "создания экземпляра объекта `Sale`" достаточно одного или нескольких методов.

Между методами и обязанностями нельзя ставить знак равенства, однако можно утверждать, что реализация метода обеспечивает выполнение обязанностей. Обязанности реализуются посредством методов, действующих либо отдельно, либо во взаимодействии с другими методами и объектами. Например, для класса `Sale` можно определить один или несколько методов вычисления стоимости (скажем, метод `getTotal`). Для выполнения этой обязанности объект `Sale` должен взаимодействовать с другими объектами, в том числе передавать сообщения `getSubtotal` каждому объекту `SalesLineItem` о необходимости предоставления соответствующей информации этими объектами.

## 16.2. Обязанности и диаграммы взаимодействий

Назначение этой главы — помочь разработчику научиться применять фундаментальные принципы распределения обязанностей между объектами. Это обычно

выполняется на этапе программирования. В контексте артефактов UML обязанности (реализованные в виде методов) рассматриваются в процессе создания диаграмм взаимодействия (относящихся к модели проектирования), система обозначений которых рассматривалась в предыдущей главе.

Из рис. 16.1 видно, что обязанностью объектов `Sale` является создание экземпляров `Payment`. Для выполнения этой обязанности передается сообщение, реализуемое посредством метода `makePayment`. Более того, для ее выполнения требуется взаимодействие с объектами `SalesLineItem` и вызов их конструктора.

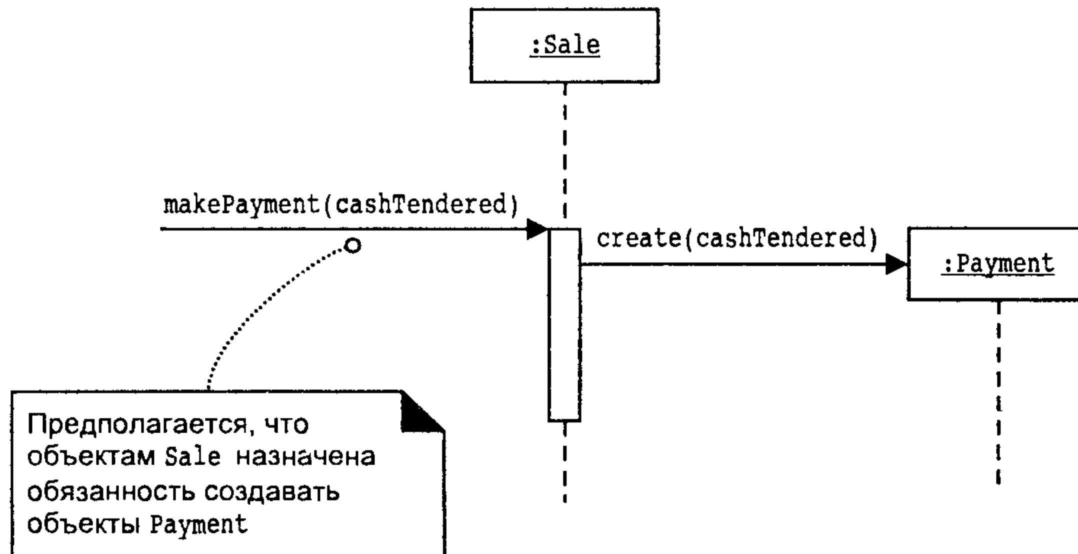



*Рис. 16.1. Связь методов и обязанностей*

Итак, диаграммы взаимодействий отражают распределение обязанностей между объектами. Распределенные обязанности отображаются в виде сообщений, отправляемых различным классам объектов. В этой главе основное внимание уделяется фундаментальным принципам распределения обязанностей, выраженным в терминах шаблонов GRASP. Процесс распределения обязанностей иллюстрируется с помощью диаграмм взаимодействия.

## 16.3. Шаблоны

Опытные разработчики объектно-ориентированных систем сформулировали общие принципы и стандартные решения, помогающие в разработке программного обеспечения. Если эти принципы и идиомы систематизировать и структурировать, а также присвоить им имена, то их можно применять в качестве *шаблонов* (patterns). Приведем пример одного из таких шаблонов.

| | |
|---|---|
| Имя шаблона | Information Expert |
| Решение | Обязанности назначаются классу, который имеет информацию, необходимую для их выполнения |
| Решаемая проблема | Каков основной принцип распределения обязанностей между объектами? |

В объектно-ориентированной технологии проектирования шаблоном называют именованное описание проблемы и ее решения, которые можно применить при разработке других систем. В идеале, шаблон должен содержать советы по поводу его применения в различных ситуациях, а также описание его преиму-

ществ и недостатков[1]. Многие шаблоны содержат рекомендации по распределению обязанностей между объектами с учетом специфики задачи.

> Проще говоря, шаблон — это именованная пара "проблема/решение", содержащая рекомендации для применения в различных конкретных ситуациях, которую можно использовать в различных контекстах.

Роль шаблонов в процессе разработки объектно-ориентированных систем очень удачно сформулирована в [52]: "Шаблон одного разработчика — это простейший строительный блок другого". В этом утверждении ничего не говорится об именах шаблонов, а лишь делается акцент на их практическом использовании. С этой точки зрения шаблоны являются основным механизмом для накопления и повторного использования полезных принципов разработки программного обеспечения.

### *Шаблоны не содержат новых идей*

Шаблон можно назвать новым, если он описывает новую идею. Однако сам термин "шаблон" означает стандартную, повторяющуюся сущность. Шаблоны не предназначены для изучения и выражения новых принципов разработки программного обеспечения. Скорее, наоборот. Они призваны систематизировать существующие знания, идиомы и принципы. Чем шире они используются, тем лучше.

Следовательно, шаблоны GRASP (с которыми мы вскоре познакомимся) не содержат новых идей, а лишь постулируют широко используемые базовые принципы.

### *Шаблоны имеют имена*

В идеале, всем шаблонам должны быть присвоены осмысленные имена. Именование шаблонов, методов и принципов имеет следующие преимущества.

- Позволяет зафиксировать понятие в памяти
- Облегчает общение

Именование сложной идеи, например шаблона, — это пример силы абстракции. Таким образом, сложная сущность сводится к простому названию за счет отбрасывания деталей. В частности, шаблоны GRASP тоже имеют осмысленные имена, например, Information Expert (Эксперт), Creator (Создатель), Protected Variations (Защищенные вариации).

#### Именованные шаблоны облегчают общение

Если шаблон имеет имя, то его легко обсуждать с другими разработчиками. Рассмотрим пример дискуссии между двумя разработчиками программного обеспечения, владеющими типичным набором шаблонов (Creator, Factory и т.д.).

**Фрэд:** "Как ты считаешь, как следует реализовать необходимость создания экземпляров `SalesLineItem`? Мне кажется, здесь подойдет Factory."

---

[1] Родоначальником использования шаблонов в процессе проектирования можно считать Кристофера Александера [2]. При разработке программного обеспечения их впервые начали применять в 80-х годах Кент Бек и Вард Каннингхем [7, 10].

**Вилма:** "Я думаю, в данном случае лучше подойдет Creator, поскольку объект `Sale` обладает всей необходимой информацией."

**Фрэд:** "Согласен."

Использование общеизвестных имен облегчает общение и позволяет обсуждать вопросы разработки на более высоком уровне абстракции.

## 16.4. Шаблоны GRASP: общие принципы распределения обязанностей

Подводя итоги, можно сделать следующие выводы.

- Высокопрофессиональное распределение обязанностей играет исключительно важную роль в объектно-ориентированном проектировании.
- Распределение обязанностей зачастую выполняется в процессе создания диаграмм взаимодействия, а также на этапе программирования.
- Шаблоны — это именованные пары утверждений "проблема/решение", описывающие важные принципы, как правило, связанные с распределением обязанностей.

| | |
|---|---|
| **Вопрос** | Что такое шаблоны GRASP? |
| **Ответ** | Это шаблоны, описывающие фундаментальные принципы распределения обязанностей между объектами. |

Очень важно понимать и уметь применять эти шаблоны при создании диаграмм взаимодействия, поскольку неопытный разработчик объектно-ориентированного программного обеспечения должен как можно скорее освоить базовые принципы. Эти принципы составляют основу проектного решения.

Аббревиатура GRASP означает General Responsibility Assignment Software Patterns (Общие шаблоны распределения обязанностей в программных системах)[2]. Такая аббревиатура очень символична, поскольку подчеркивает важность изучения этих принципов для успешного проектирования объектно-ориентированного приложения[3].

### *Как применять шаблоны GRASP*

В следующих разделах описаны первые пять шаблонов GRASP.

- Information Expert
- Creator
- High Cohesion
- Low Coupling
- Controller

---

[2] На самом деле следовало бы писать "шаблоны GRAS", но "шаблоны GRASP" звучит лучше.

[3] В переводе с английского "grasp" означает "схватывать", "постигать".— *Прим. перев.*

В последующих главах рассматриваются и другие шаблоны, а здесь основное внимание уделяется лишь первым пяти, поскольку они касаются основных и фундаментальных вопросов проектирования.

Нам предстоит изучить эти шаблоны, разобраться в правилах их использования при создании диаграмм взаимодействий и применить их для создания новых диаграмм. Сначала мы познакомимся с первыми пятью шаблонами, а затем и с остальными.

## 16.5. Система обозначений диаграммы классов в языке UML

Диаграммы классов иллюстрируют взаимоотношения программных элементов. Обозначение класса состоит из трех частей, в которых указываются имя класса, его атрибуты и методы (рис. 16.2).

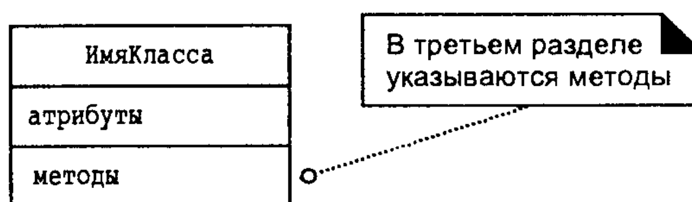



*Рис. 16.2. Имена методов программных классов*

Система обозначений диаграммы классов более подробно рассматривается в следующих главах, однако такие обозначения будут использоваться при описании шаблонов.

## 16.6. Шаблон Information Expert

**Решение.** Назначить обязанность информационному эксперту — классу, у которого имеется информация, требуемая для выполнения обязанности.

**Проблема.** Каков наиболее общий принцип распределения обязанностей между объектами при объектно-ориентированном проектировании?

В модели системы могут быть определены десятки или сотни программных классов, а в приложении может потребоваться выполнение сотен или тысяч обязанностей. Во время объектно-ориентированного проектирования при формулировке принципов взаимодействия объектов необходимо распределить обязанности между классами. При правильном выполнении этой задачи система становится гораздо проще для понимания, поддержки и расширения. Кроме того, появляется возможность повторного использования уже разработанных компонентов в последующих приложениях.

**Пример.** В приложении POS-системы NextGen некоторому классу необходимо знать общую сумму продажи.

| Начинайте распределение обязанностей с их четкой формулировки. |
|---|

С этой точки зрения можно сформулировать следующее утверждение.

*Какой класс должен отвечать за знание общей суммы продажи?*

Согласно шаблону Information Expert, нужно определить, объекты каких классов содержат информацию, необходимую для вычисления общей суммы.

Теперь возникает ключевой вопрос: на основе какой модели нужно анализировать информацию — модели предметной области или проектирования? Модель предметной области иллюстрирует концептуальные классы из предметной области системы, а в модели проектирования показаны программные классы.

Ответ на этот вопрос сводится к следующему.

1. Если в модели проектирования имеются соответствующие классы, в первую очередь, следует использовать ее.
2. В противном случае нужно обратиться к модели предметной области и постараться уточнить ее для облегчения создания соответствующих программных классов.

Например, предположим, мы находимся в самом начале этапа проектирования, когда модель проектирования представлена в минимальном объеме. Следовательно, кандидатуру на роль информационного эксперта следует искать в модели предметной области. Вероятно, на эту роль подойдет концептуальный класс `Sale`. Тогда в модель проектирования нужно добавить соответствующий программный класс под именем `Sale` и присвоить ему обязанность вычисления общей стоимости, реализуемую с помощью вызова метода `getTotal`. При таком подходе сокращается разрыв между организацией программных объектов и соответствующих им понятий из предметной области.

Чтобы рассмотреть этот пример подробнее, обратимся к фрагменту модели предметной области, представленному на рис. 16.3.

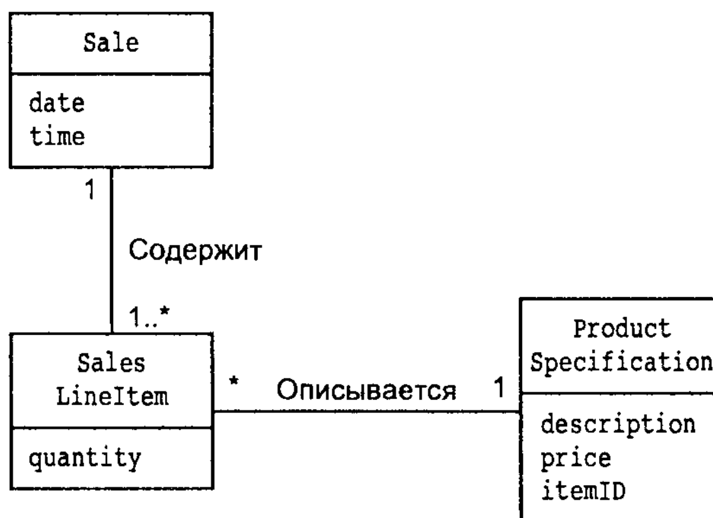



*Рис. 16.3. Ассоциации объекта* `Sale`

Какая информация требуется для вычисления общей суммы? Необходимо узнать стоимость всех проданных товаров `SalesLineItem` и просуммировать эти промежуточные суммы. Такой информацией обладает лишь экземпляр объекта `Sale`. Следовательно, с точки зрения шаблона Information Expert объект `Sale` подходит для выполнения этой обязанности, т.е. является *информационным экспертом* (information expert).

Как уже упоминалось, подобные вопросы распределения обязанностей зачастую возникают при создании диаграмм взаимодействий. Представьте, что вы приступили к работе, начав создание диаграмм для распределения обязанностей между объектами. Принятые решения иллюстрируются на фрагменте диаграммы взаимодействий, представленном на рис. 16.4.

*Рис. 16.4. Фрагмент диаграммы взаимодействий*

Однако на данном этапе выполнена не вся работа. Какая информация требуется для вычисления промежуточной суммы элементов продажи? Необходимы значения атрибутов SalesLineItem.quantity и SalesLineItem.price. Объекту SalesLineItem известно количество товара и известен связанный с ним объект ProductSpecification. Следовательно, в соответствии с шаблоном Expert, промежуточную сумму должен вычислять объект SalesLineItem. Другими словами, этот объект является *информационным экспертом.*

В терминах диаграмм взаимодействий это означает, что объект Sale должен передать сообщения getSubtotal каждому объекту SalesLineItem, а затем просуммировать полученные результаты. Этот процесс проиллюстрирован на рис. 16.5.

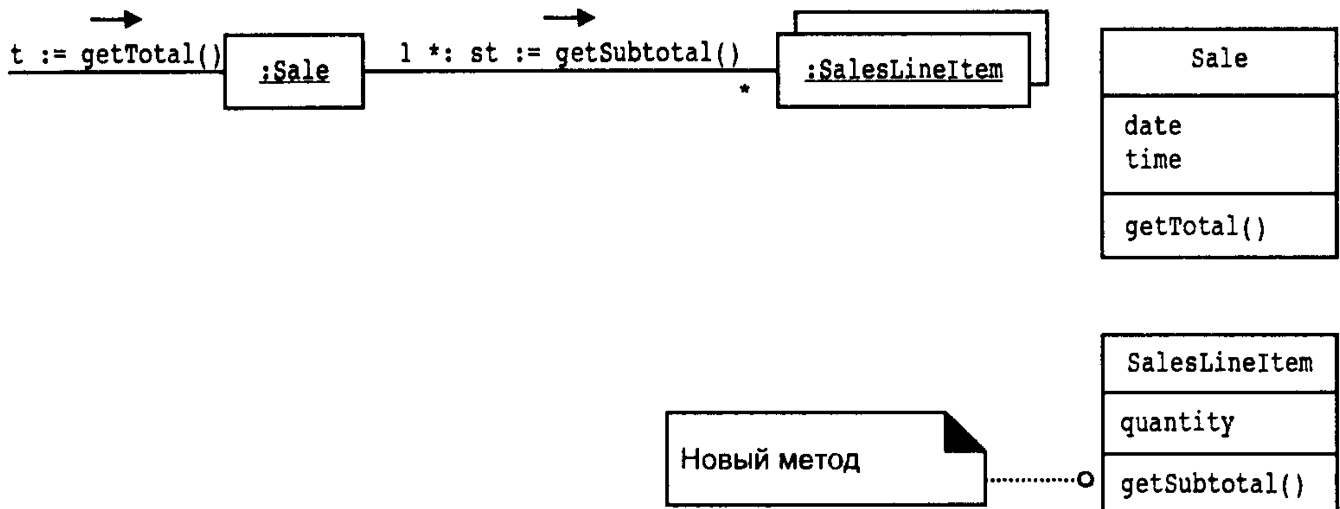



*Рис. 16.5. Вычисление общей суммы продажи*

Для выполнения обязанности, связанной со знанием и предоставлением промежуточной суммы, объекту SalesLineItem должна быть известна стоимость товара.

В данном случае в качестве информационного эксперта будет выступать объект ProductSpecification.

Результаты проектирования представлены на рис. 16.6.

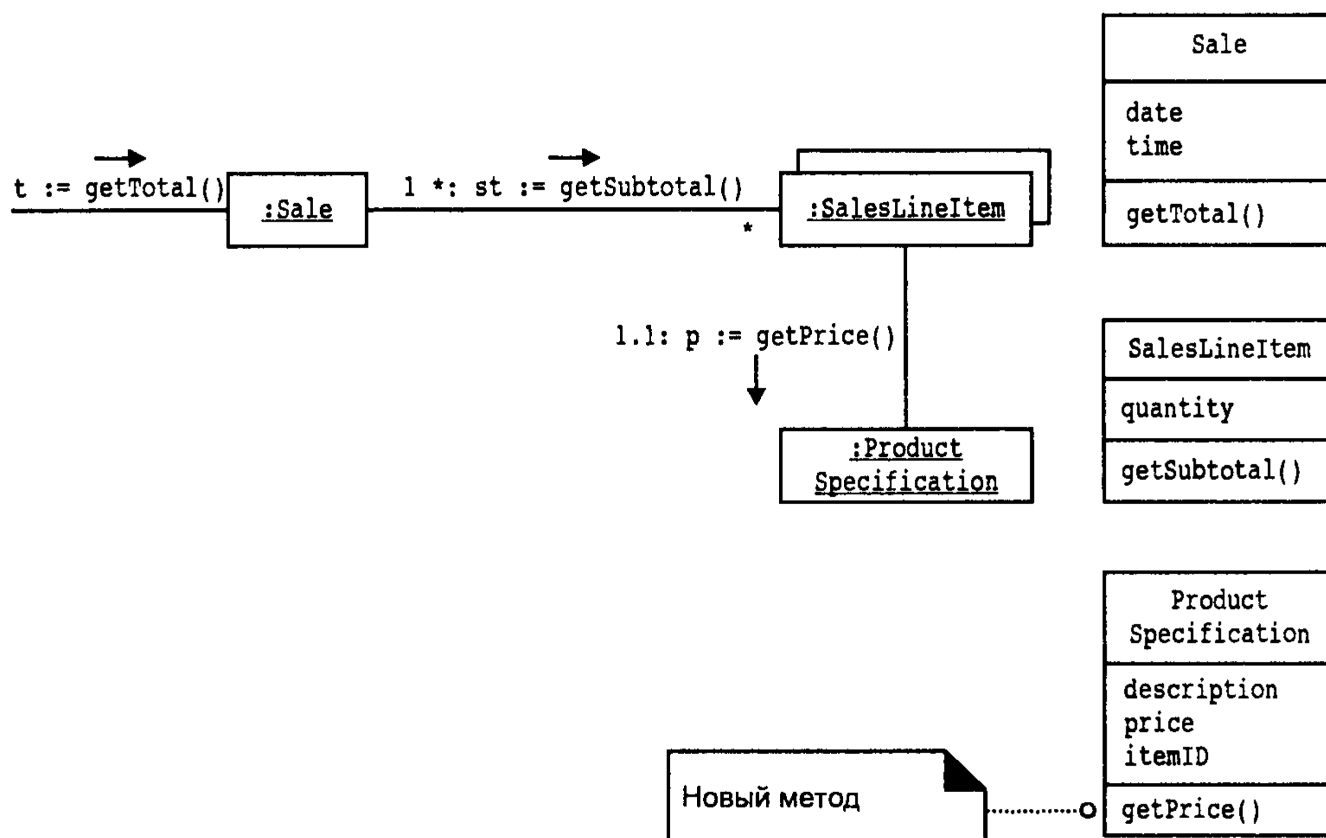



*Рис. 16.6. Вычисление общей суммы продажи*

В завершение можно сказать следующее. Для выполнения обязанности "знать и предоставлять общую сумму продажи трем объектам классов" были следующим образом присвоены три обязанности.

| **Класс** | **Обязанность** |
|---|---|
| Sale | Знание общей суммы продажи |
| SalesLineItem | Знание промежуточной суммы для данного товара |
| ProductSpecification | Знание цены товара |

Рассмотрение и распределение обязанностей выполнялись в процессе создания диаграммы взаимодействий. Затем полученные результаты могут быть реализованы в разделе методов диаграммы классов.

При назначении обязанностей, согласно шаблону Expert, был применен следующий принцип: обязанности связываются с тем объектом, который имеет информацию, необходимую для их выполнения.

**Обсуждение.** При распределении обязанностей шаблон Information Expert используется гораздо чаще любого другого шаблона. В нем определены основные принципы, которые уже давно используются в объектно-ориентированном проектировании. Шаблон Expert не содержит неясных или запутанных идей и отражает обычный интуитивно понятный подход. Он заключается в том, что объекты осуществляют действия, связанные с имеющейся у них информацией.

Обратите внимание, что для выполнения обязанности зачастую требуется информация, распределенная между различными классами или объектами. Это предполагает, что должно существовать много "частичных" экспертов, взаимодействующих при выполнении общей задачи. Например, для вычисления общей суммы продажи в конечном счете необходимо взаимодействие трех классов. Если

информация распределена между различными объектами, то при выполнении общей задачи они должны взаимодействовать с помощью сообщений.

Применение шаблона Expert приводит к тому, что объекты выполняют операции точно так, как они выполняются сущностями реального мира. Именно эти сущности и моделируют объекты. Питер Код называет это стратегией "Сделай сам" ("Do It Myself") [30]. Например, в реальном мире без применения электромеханических устройств покупка не сможет сообщить о своей стоимости. Она представляет собой неодушевленное понятие. Каждый из покупателей вычисляет сумму покупки самостоятельно. Однако в мире объектно-ориентированного программного обеспечения все программные объекты являются живыми и одушевленными, поэтому могут выполнять обязанности и осуществлять действия. В основном они выполняют действия, связанные с известной им информацией. В объектно-ориентированном проектировании этот принцип называется "оживлением" (animation) (программные объекты подобны "живым" персонажам мультипликационных фильмов).

Шаблон Information Expert, как и многие другие понятия объектной технологии, имеет соответствующую аналогию в реальном мире. Обычно мы распределяем обязанности между теми служащими, у которых имеется необходимая для выполнения поставленной задачи информация. Например, кто в коммерческом предприятии должен нести ответственность за создание отчета о прибыли и убытках? Тот из служащих, кто имеет доступ ко всей информации, необходимой для создания такого отчета. Возможно, лучше всего для выполнения этой обязанности подойдет руководитель финансового отдела предприятия. Программные объекты взаимодействуют между собой и обмениваются информацией так же, как люди. Начальник финансового отдела компании может запросить требуемые данные у бухгалтеров, работающих со счетами по дебиторской и кредиторской задолженности, чтобы составить отдельные отчеты по кредиту и дебиту.

**Когда не следует применять шаблон.** В некоторых ситуациях применение шаблона Expert нежелательно, например, в связи с проблемами со связыванием и зацеплением (эти принципы обсуждаются ниже в этой главе).

Например, какой объект должен отвечать за сохранение информации о продажах в базе данных? Безусловно, большая часть подлежащей сохранению информации "известна" объекту `Sale`, а значит, согласно шаблону Expert, на этот класс следует возложить обязанность по сохранению. Логическим следствием такого рассуждения является вывод о том, что каждый объект должен отвечать за сохранение себя в базе данных. Однако при этом возникают проблемы связывания, зацепления и дублирования. В частности, класс `Sale` должен содержать методы обращения к базе данных, т.е. быть связан с языком SQL или службами JDBC (Java Database Connectivity). Тогда этот класс не будет относиться к логике приложения и моделировать "продажу". При этом расширяется круг его обязанностей и снижается зацепление. Этот класс должен быть связан с техническими службами баз данных других подсистем, в частности со службами JDBC, а не только с программными объектами уровня предметной области. Кроме того, вероятно, подобная логика будет дублироваться во многих других классах, информация о которых подлежит постоянному хранению.

Все эти проблемы приводят к нарушению основного архитектурного принципа — проектирования с разделением основных функций системы. Логика приложения должна храниться в одном месте (на уровне программных объектов предметной области), а логика связи с базой данных — в другом (в отдельной

подсистеме служб базы данных). Различные функции не должны реализовываться в одном и том же компоненте.[4]

Принцип разделения основных функций улучшает показатели связывания и зацепления. С точки зрения этих категорий, класс `Sale` не должен отвечать за сохранение информации в базе данных.

**Преимущества**

- Шаблон Expert поддерживает инкапсуляцию. Для выполнения требуемых задач объекты используют собственные данные. Подобную возможность обеспечивает также шаблон Low Coupling, применение которого приводит к созданию более надежных и легко поддерживаемых систем. (Low Coupling является шаблоном GRASP, который будет рассмотрен чуть ниже.)
- Соответствующее поведение системы обеспечивается несколькими классами, содержащими требуемую информацию. Это приводит к определениям классов, которые гораздо проще понимать и поддерживать. Кроме того, поддерживается шаблон High Cohesion (рассматриваемый ниже в этой главе).

**Связанные шаблоны**

- Low Coupling
- High Cohesion

**Другие названия и аналогичные принципы.** "Хранение обязанностей вместе с данными", "Кто знает, тот и выполняет", "Сделай сам", "Размещайте службы вместе с их атрибутами".

## 16.7. Шаблон Creator

**Решение.** Назначить классу B обязанность создавать экземпляры класса A, если выполняется одно из следующих условий.

- Класс B *агрегирует* (aggregate) объекты A.
- Класс B *содержит* (contains) объекты A.
- Класс B *записывает* (records) экземпляры объектов A.
- Класс B *активно использует* (closely uses) объекты A.
- Класс B *обладает данными инициализации* (has the initializing data), которые будут передаваться объектам A при их создании (т.е. при создании объектов A класс B является экспертом).

Класс B — *создатель* (creator) объектов A.

Если выполняется несколько из этих условий, то лучше использовать класс B, *агрегирующий* или *содержащий* класс A.

**Проблема.** Кто должен отвечать за создание нового экземпляра некоторого класса?

Создание объектов в объектно-ориентированной системе является одним из наиболее стандартных видов деятельности. Следовательно, при назначении обязанностей, связанных с созданием объектов, полезно руководствоваться некоторым основным принципом. Правильно распределив обязанности при проектиро-

[4] Разделение функций более подробно рассматривается в главе 32.

вании, можно создать слабо связанные независимые компоненты с возможностью их дальнейшего использования, упростить их, а также обеспечить инкапсуляцию данных и их повторное использование.

**Пример.** Кто в POS-системе должен отвечать за создание нового экземпляра объекта `SalesLineItem`? В соответствии с шаблоном Creator, необходимо найти класс, агрегирующий, содержащий и т.д. экземпляры объектов `SalesLineItem`. Рассмотрим фрагмент модели предметной области, представленной на рис. 16.7.

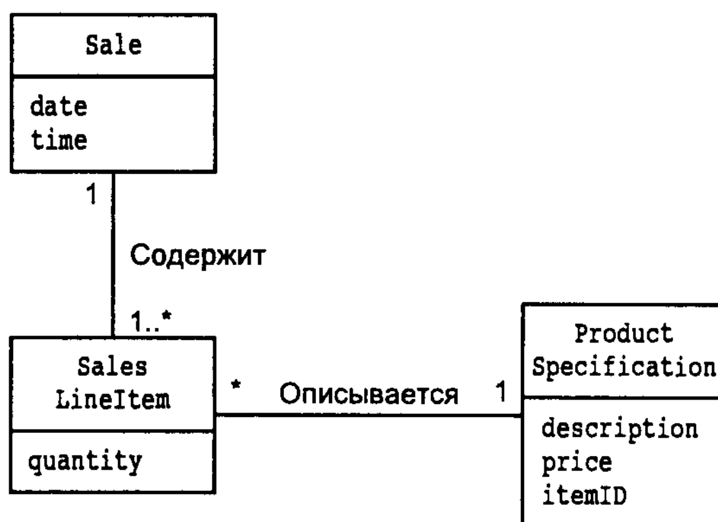



*Рис. 16.7. Фрагмент модели предметной области*

Поскольку объект `Sale` содержит (фактически — агрегирует) несколько объектов `SalesLineItem`, согласно шаблону Creator, он является хорошим кандидатом для выполнения обязанности, связанной с созданием экземпляров объектов `SalesLineItem`.

Такой подход приводит к необходимости разработки взаимодействия объектов, показанного на следующей диаграмме (рис. 16.8).

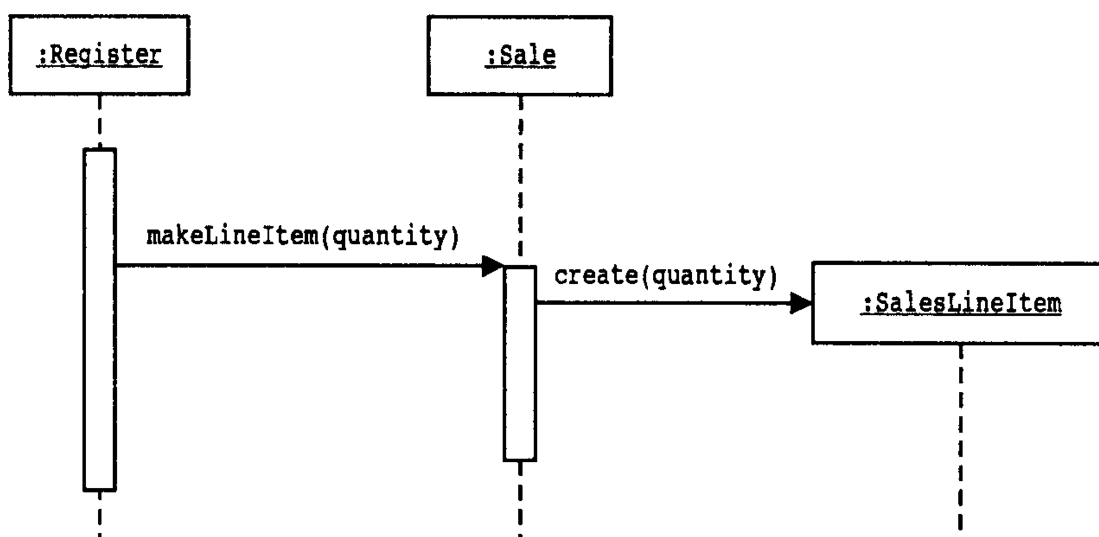



*Рис. 16.8. Создание экземпляра объекта `SalesLineItem`*

При таком распределении обязанностей требуется, чтобы в объекте `Sale` был определен метод `makeLineItem`.

Рассмотрение и распределение обязанностей выполнялись в процессе создания диаграммы взаимодействий. Затем полученные результаты могут быть реализованы в конкретных методах, помещенных в разделе методов диаграммы классов.

**Обсуждение.** Шаблон Creator определяет способ распределения обязанностей, связанный с процессом создания объектов. В объектно-ориентированных системах эта задача является одной из наиболее распространенных. Основным назначением шаблона Creator является выявление объекта-создателя, который при возникновении любого события должен быть связан со всеми созданными им объектами. При таком подходе обеспечивается низкая степень связанности.

Целое *агрегирует* свои части, контейнер *хранит* свое содержимое, регистратор ведет учет. Все эти взаимосвязи являются очень распространенными способами взаимодействия классов в диаграмме классов. В шаблоне Creator определяется, что внешний контейнер или класс-регистратор — это хорошие кандидаты на выполнение обязанностей, связанных с созданием сущностей, которые они будут содержать или регистрировать. Конечно, это утверждение является лишь рекомендацией.

Обратите внимание, что понятие "агрегация" (aggregation) используется в соответствии с его определением в шаблоне Creator. Этот термин будет подробнее рассмотрен в главе 27, однако сейчас нужно ознакомиться с кратким определением этого понятия. Агрегация подразумевает, что сущности находятся во взаимосвязи "целое–часть" или "агрегат–часть", подобно тому, как туловище агрегирует ноги или абзац содержит отдельные предложения.

В некоторых случаях в качестве создателя выбирается класс, который содержит данные инициализации, передаваемые объекту во время его создания. На самом деле это пример использования шаблона Expert. В процессе создания инициализирующие данные передаются с помощью метода инициализации некоторого вида, такого как конструктор языка Java с параметрами. Например, предположим, что при создании экземпляр объекта `Payment` нужно инициализировать с использованием общей суммы, содержащейся в объекте `Sale`. Поскольку объекту `Sale` эта сумма известна, он является кандидатом на выполнение обязанности, связанной с созданием экземпляра объекта `Payment`.

**Когда не следует применять шаблон.** Зачастую создание экземпляра — это достаточно сложная операция, выполняемая при реализации некоторого условия на основе каких-либо внешних свойств. В этом случае предпочтительнее использовать шаблон Factory (Фабрика) [52] и делегировать обязанность создания экземпляров вспомогательному классу. Шаблон Factory описывается в главе 23.

**Преимущества**

- Поддерживается шаблон Low Coupling (рассматриваемый ниже), способствующий снижению затрат на сопровождение и обеспечивающий возможность повторного использования созданных компонентов в дальнейшем. Применение шаблона Creator не повышает степени связанности, поскольку *созданный* (created) класс, как правило, оказывается видимым для класса-создателя посредством имеющихся ассоциаций.

**Связанные шаблоны и принципы**

- Low Coupling.
- Factory.
- В [15] описывается шаблон, позволяющий определить объекты-агрегаты, которые поддерживают инкапсуляцию составляющих их компонентов.

## 16.8. Шаблон Low Coupling

**Решение.** Распределить обязанности таким образом, чтобы степень связанности оставалась низкой.

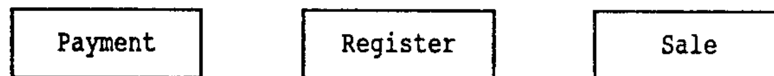
**Проблема.** Как обеспечить зависимость, незначительное влияние изменений и повысить возможность повторного использования?

*Степень связанности* (coupling) — это мера, определяющая насколько жестко один элемент связан с другими элементами, либо каким количеством данных о других элементах он обладает. Элемент с низкой степенью связанности (или слабым связыванием) зависит от не очень большого числа других элементов. Выражение "очень много" зависит от контекста, однако необходимо провести его оценку.

Класс с высокой степенью связанности (или жестко связанный) зависит от множества других классов. Однако наличие таких классов нежелательно, поскольку оно приводит к возникновению следующих проблем.

- Изменения в связанных классах приводят к локальным изменениям в данном классе.
- Затрудняется понимание каждого класса в отдельности.
- Усложняется повторное использование, поскольку для этого требуется дополнительный анализ классов, с которыми связан данный класс.

**Пример.** Рассмотрим следующий фрагмент диаграммы классов для приложения NextGen.




Предположим, что необходимо создать экземпляр класса Payment и связать его с объектом Sale. Какой класс должен отвечать за выполнение этой операции? Поскольку в реальной предметной области регистрация объекта Payment выполняется объектом Register, в соответствии с шаблоном Creator, объект Register является хорошим кандидатом для создания объекта Payment. Затем экземпляр объекта Register должен передать сообщение addPayment объекту Sale, указав в качестве параметра новый объект Payment. Приведенные рассуждения отражены на фрагменте диаграммы взаимодействий, представленной на рис. 16.9.

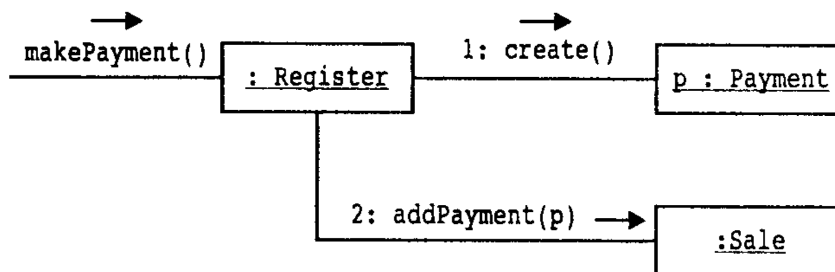



*Рис. 16.9. Новый экземпляр объекта Payment создается с помощью объекта Register*

Такое распределение обязанностей предполагает, что класс Register обладает знаниями о данных класса Payment (т.е. связывается с ним).

Обратите внимание на обозначения, принятые в языке UML. Экземпляру объекта Payment присвоено явное имя p, чтобы его можно было использовать в качестве параметра сообщения 2.

Альтернативный способ создания объекта Payment и его связывания с объектом Sale показан на рис. 16.10.

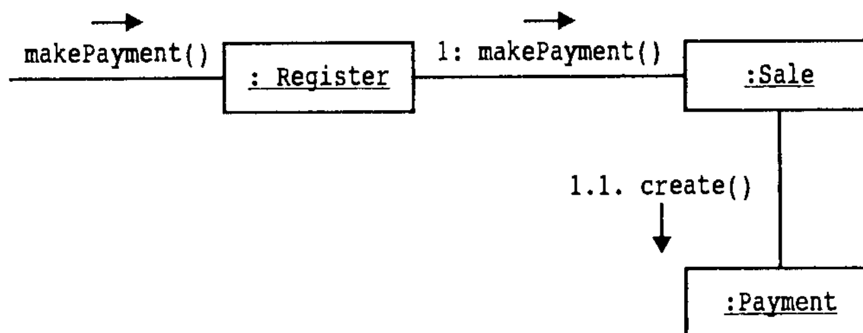



*Рис. 16.10. Новый экземпляр объекта* Payment *создается с помощью объекта* Sale

Какой из методов проектирования, основанный на распределении обязанностей, обеспечивает более низкую степень связывания? В обоих случаях предполагается, что в конечном итоге объекту Sale должно быть известно о существовании объекта Payment. При использовании первого способа, когда объект Payment создается с помощью объекта Register, между этими двумя объектами добавляется новая связь, тогда как второй способ степень связывания объектов не усиливает. С точки зрения числа связей между объектами, более предпочтительным является второй способ, поскольку в этом случае обеспечивается низкая степень связывания. Приведенная иллюстрация является примером того, как при использовании двух различных шаблонов — Low Coupling и Creator — можно прийти к двум различным решениям.

> На практике уровень связывания не рассматривается отдельно от других принципов, сформулированных в шаблонах Expert и High Cohesion. Тем не менее в шаблоне Low Coupling описан один из факторов, с использованием которого можно значительно улучшить весь проект в целом.

**Обсуждение.** В шаблоне Low Coupling описывается принцип, о котором нельзя забывать на протяжении всех стадий работы над проектом. Он является объектом постоянного внимания. Шаблон Low Coupling представляет собой средство, которое разработчик применяет при оценке всех принимаемых в процессе проектирования решений.

В объектно-ориентированных языках программирования, таких как C++, Java и C#, имеются следующие стандартные способы связывания объектов *TypeX* и *TypeY*.

- Объект *TypeX* содержит атрибут (переменную-член), который ссылается на экземпляр объекта *TypeY* или сам объект *TypeY*.
- Объект *TypeX* вызывает службы объекта *TypeY*.
- Объект *TypeX* содержит метод, который каким-либо образом ссылается на экземпляр объекта *TypeY* или сам объект *TypeX* (обычно это подразумевает

использование *TypeY* в качестве типа параметра, локальной переменной или возвращаемого значения).

- Объект *TypeX* является прямым или непрямым подклассом объекта *TypeY*.
- Объект *TypeY* является интерфейсом, а *TypeX* реализует этот интерфейс.

Шаблон Low Coupling подразумевает такое распределение обязанностей, которое не влечет за собой чрезмерное повышение степени связывания, приводящее к отрицательным результатам.

Шаблон Low Coupling поддерживает независимость классов, что, в свою очередь, повышает возможности повторного использования и обеспечивает более высокую эффективность приложения. Его нельзя рассматривать изолированно от других шаблонов, таких как Expert и High Cohesion. Скорее, он обеспечивает один из основных принципов проектирования, применяемых при распределении обязанностей.

Подкласс жестко связан со своим суперклассом. Поэтому, принимая решение о наследовании свойств объектов, следует учитывать, что отношение наследования повышает степень связанности классов. Например, предположим, объекты необходимо постоянно хранить в реляционной или объектной базе данных. В этом случае зачастую создают абстрактный суперкласс `PersistentObject`, от которого наследуют свои свойства другие классы. Недостатком такого подхода является жесткое связывание объектов с конкретной службой, а преимуществом — автоматическое наследование поведения.

Не существует абсолютной меры для определения слишком высокой степени связывания. Важно лишь понимать степень связанности объектов на текущий момент и не упустить тот момент, когда дальнейшее повышение степени связанности может привести к возникновению проблем. В целом, следует руководствоваться таким принципом: классы, которые являются достаточно общими по своей природе и с высокой вероятностью будут повторно использоваться в дальнейшем, должны иметь минимальную степень связанности с другими классами.

Крайним случаем при реализации шаблона Low Coupling является полное отсутствие связывания между классами. Такая ситуация тоже нежелательна, поскольку базовой идеей объектного подхода является система связанных объектов, которые "общаются" между собой посредством передачи сообщений. При слишком частом использовании принципа слабого связывания система будет состоять из нескольких изолированных сложных активных объектов, самостоятельно выполняющих все операции, и множества пассивных объектов, основная функция которых сводится к хранению данных. Поэтому при создании объектно-ориентированной системы должна присутствовать некоторая оптимальная степень связывания между объектами, позволяющая выполнять основные функции посредством взаимодействия этих объектов.

**Когда не следует применять шаблон.** Высокая степень связывания с устойчивыми элементами не представляет проблемы. Например, приложение J2EE можно жестко связать с библиотеками Java (`java.util` и т.п.), поскольку эти библиотеки широко распространены и стабильны.

#### Выбери свою игру

Высокая степень связывания сама по себе не является проблемой. Проблемой является жесткое связывание с неустойчивыми в некотором отношении элементами.

Важно понимать следующее. Разработчик может обеспечивать гибкость программы, реализовывать принцип инкапсуляции и придерживаться принципа слабого

связывания во многих аспектах системы. Однако без убедительной мотивации не следует во что бы то ни стало бороться за уменьшение степени связывания объектов.

Разработчики должны выбрать "свою игру", чтобы снизить степень связывания и обеспечить инкапсуляцию. При этом особое внимание нужно уделить неустойчивым или быстро изменяющимся элементам. Например, в проекте NextGen к системе планируется подключать различные программы вычисления налоговых платежей (с общим интерфейсом). Следовательно, в этой части системы нужно обеспечить низкую степень связывания.

**Преимущества**

- Изменения компонентов мало сказываются на других объектах.
- Принципы работы и функции компонентов можно понять, не изучая другие объекты.
- Удобство повторного использования.

**Основы.** Связывание и зацепление (описывается ниже) — действительно фундаментальные принципы проектирования, которые следует применять при разработке всех программных систем. Ларри Константин, разработавший в 70-х годах структурный подход к проектированию и изучающий в настоящее время принципы удобства использования программ [28], в 60-е годы занимался исследованием связывания и зацепления, как важнейших принципов проектирования [29, 35].

**Связанные шаблоны**

- Protected Variations (Защищенные вариации)

## 16.9. Шаблон High Cohesion

**Решение.** Распределение обязанностей, поддерживающее высокую степень зацепления.

**Проблема.** Как обеспечить возможность управления сложностью?

В терминах объектно-ориентированного проектирования *зацепление* (cohesion) (или, более точно, функциональное зацепление) — это мера связанности и сфокусированности обязанностей класса. Считается, что элемент обладает высокой степенью зацепления, если его обязанности тесно связаны между собой и он не выполняет непомерных объемов работы. В роли таких элементов могут выступать классы, подсистемы и т.д.

Класс с низкой степенью зацепления выполняет много разнородных функций или несвязанных между собой обязанностей. Такие классы создавать нежелательно, поскольку они приводят к возникновению следующих проблем.

- Трудность понимания.
- Сложности при повторном использовании.
- Сложности поддержки.
- Ненадежность, постоянная подверженность изменениям.

Классы со слабым зацеплением, как правило, являются слишком "абстрактными" или выполняют обязанности, которые можно легко распределить между другими объектами.

**Пример.** Для анализа шаблона High Cohesion можно использовать тот же пример, что и для Low Coupling.

Предположим, необходимо создать экземпляр объекта Payment и связать его с текущей продажей. Какой класс должен выполнять эту обязанность? Поскольку в реальной предметной области сведения о платежах записываются в реестре, согласно шаблону Creator, для создания экземпляра объекта Payment можно использовать объект Register. Тогда экземпляр объекта Register сможет отправить сообщение addPayment объекту Sale, передавая в качестве параметра новый экземпляр объекта Payment, как показано на рис. 16.11.

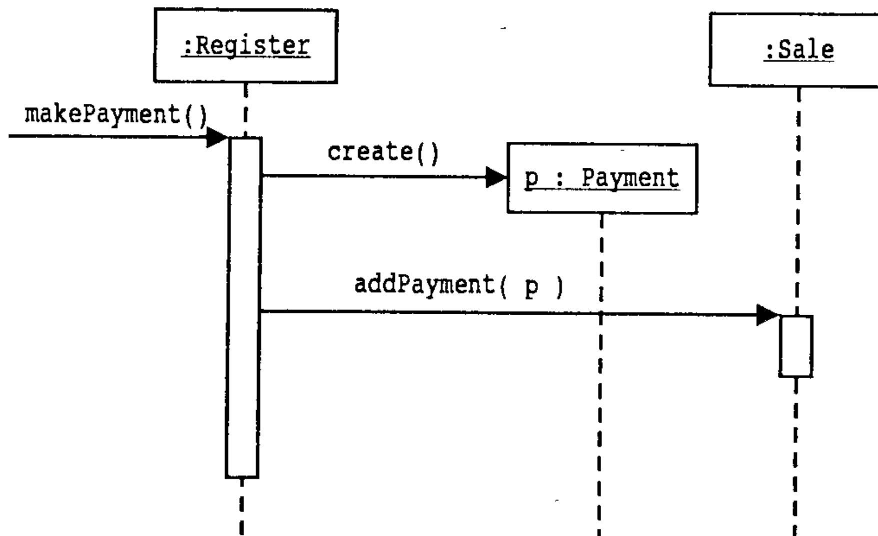



*Рис. 16.11. Экземпляр объекта Register создает объект Payment*

При таком распределении обязанностей платежи выполняет объект Register, т.е. объект Register частично несет ответственность за выполнение системной операции makePayment.

В данном обособленном примере это приемлемо. Однако если и далее возлагать на класс Register обязанности по выполнению все новых и новых функций, связанных с другими системными операциями, то этот класс будет слишком перегружен и будет обладать низкой степенью зацепления.

Предположим, приложение должно выполнять пятьдесят системных операций и все они возложены на класс Register. Если этот объект будет выполнять все операции, то он станет чрезмерно "раздутым" и не будет обладать свойством зацепления. И дело не в том, что одна задача создания экземпляра объекта Payment сама по себе снизила степень зацепления объекта Register; она является частью общей картины распределения обязанностей.

На рис. 16.12 представлен другой вариант распределения обязанностей. Здесь функция создания экземпляра платежа делегирована объекту Sale. Благодаря этому поддерживается более высокая степень зацепления объекта Register. Поскольку такой вариант распределения обязанностей обеспечивает низкий уровень связывания и более высокую степень зацепления, он является более предпочтительным.

| На практике уровень зацепления не рассматривают изолированно от других обязанностей и принципов, обеспечиваемых шаблонами Expert и Low Coupling. |
|---|

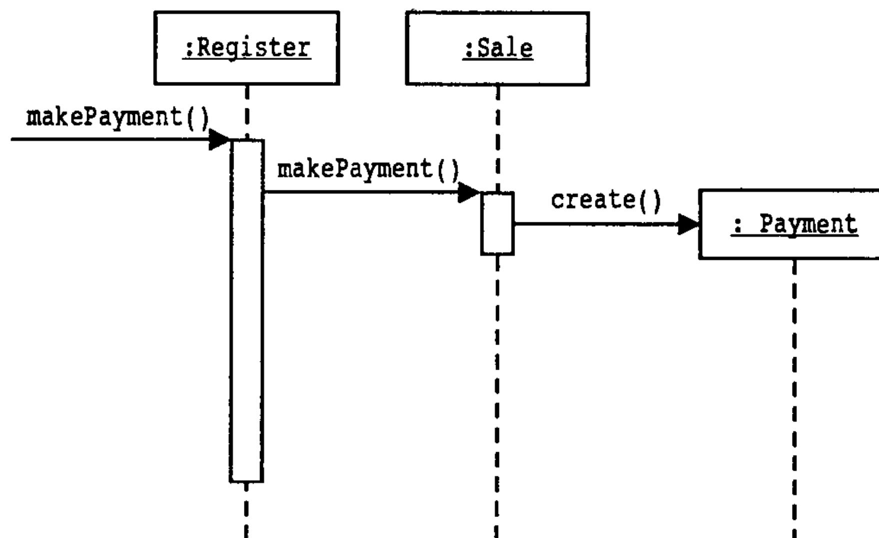



*Рис. 16.12. Объект* `Sale` *создает экземпляр объекта* `Payment`

**Обсуждение.** Как и о принципе слабого связывания, о высокой степени зацепления следует помнить в течение всего процесса проектирования. Этот шаблон необходимо применять при оценке эффективности каждого проектного решения.

Гради Буч считает, что можно говорить о высоком функциональном зацеплении между элементами одного компонента (например, класса), если "все они взаимодействуют между собой с целью обеспечения некоторого определенного поведения" [18].

Вот несколько сценариев, иллюстрирующих различную степень функционального зацепления.

1. *Очень слабое зацепление.* Только класс отвечает за выполнение множества операций в самых различных функциональных областях.

   - Допустим, существует класс `RDB-RPC-Interface`, полностью отвечающий за взаимодействие между реляционными базами данных и вызов удаленных процедур. Это две абсолютно разные функциональные области, требующие больших объемов кода. Такие обязанности можно распределить между семейством классов, связанных с доступом к реляционной базе данных, и семейством классов, отвечающих за поддержку удаленных процедур.

2. *Слабое зацепление.* Класс несет единоличную ответственность за выполнение сложной задачи из одной функциональной области.

   - Допустим, некий класс `RDBInterface` полностью отвечает за взаимодействие с реляционными базами данных. Методы этого класса связаны между собой, однако их слишком много, и их реализация требует огромных объемов кода. Таких методов может быть несколько сотен или даже тысяч. Данный класс следует разделить на несколько более мелких классов, совместно обеспечивающих доступ к реляционным базам данных.

3. *Сильное зацепление.* Класс имеет среднее количество обязанностей из одной функциональной области и для выполнения своих задач взаимодействует с другими классами.

   - Допустим, некоторый класс `RDBInterface` лишь частично отвечает за взаимодействие с реляционными базами данных. Для извлечения и со-

хранения объектов в базе данных он взаимодействует с десятком других классов.

4. *Среднее зацепление.* Класс имеет несложные обязанности в нескольких различных областях, логически связанных с концепцией этого класса, но не связанных между собой.

   - Допустим, существует класс `Company`, который несет полную ответственность за (а) знание всех сотрудников компании и (б) всю финансовую информацию. Эти две области не слишком связаны между собой, однако обе логически связаны с понятием "компания". К тому же предполагается, что такой класс содержит небольшое число открытых методов и требует незначительных объемов кода для их реализации.

Как правило, класс с высокой степенью зацепления содержит сравнительно небольшое число методов, которые функционально тесно связаны между собой, и не выполняет слишком много функций. Он взаимодействует с другими объектами для выполнения более сложных задач.

Классы с высокой степенью зацепления являются очень предпочтительными, поскольку они весьма просты в понимании, поддержке и повторном использовании. Высокая степень однотипной функциональности в сочетании с небольшим числом операций упрощают поддержку и модификацию класса, а также возможность его повторного использования.

Шаблон High Cohesion, как и другие понятия объектно-ориентированной технологии проектирования, имеет аналогию в реальном мире. Всем известно, что человек, выполняющий большое число разнородных обязанностей (особенно тех, которые можно легко распределить между другими людьми), работает не очень эффективно. Это касается менеджеров, которые не умеют распределять обязанности между своими подчиненными. Такие люди страдают от "низкой степени зацепления" и могут легко "расклеиться".

### Еще один классический принцип: модульное проектирование

Связывание и зацепление — это старые принципы проектирования программных продуктов. Объектное проектирование не идет вразрез с устоявшимися подходами. Еще одной особенностью, тесно связанной со связыванием и зацеплением, является модульность. Приведем цитату.

> "Модульность — это свойство системы, разбитой на множество модулей с высокой степенью зацепления и слабым связыванием" [18].

Модульное проектирование обеспечивается за счет создания методов и классов с высоким зацеплением. На уровне базовых объектов модульность достигается за счет проектирования каждого метода с явно выделенной конкретной целью и группирования набора взаимосвязанных методов в рамках одного класса.

### Связывание и зацепление: инь и янь проектирования программного обеспечения

Некорректное связывание порождает неправильное зацепление, и наоборот. Автор рассматривает связывание и зацепление как инь и янь проектирования программного обеспечения, поскольку эти понятия взаимозависимы. Предположим, например, что класс уровня пользовательского интерфейса, представляющий аплет, сохраняет данные в базе данных и вызывает уда-

ленные службы. Такой класс не только обладает слабым зацеплением, но и оказывается сильно связанным со многими другими элементами.

**Когда не следует применять шаблон.** Существует несколько случаев, когда низкое зацепление оказывается оправданным.

Одна из таких ситуаций возникает в том случае, когда обязанности или код группируются в одном классе или компоненте для упрощения его поддержки одним человеком. Однако в данном случае необходимо помнить о том, что такая группировка может привести и к усложнению поддержки. Например, предположим, что в приложении содержатся внедренные операторы SQL, которые в соответствии с другими шаблонами проектирования нужно распределить по десяти классам работы с базой данных. В этом случае лишь один или два эксперта в области SQL знают, как лучше всего определять и поддерживать эти операторы SQL, даже несмотря на то, что в проекте участвуют десятки программистов с опытом работы в области объектно-ориентированного программирования. Некоторые из них могут иметь достаточно высокий уровень знания объектно-ориентированных технологий. Предположим также, что эксперт в области SQL не обладает навыками программиста по созданию объектно-ориентированных программ. Архитектор программной системы может решить сгруппировать операторы SQL в одном классе `RDBOperations`, чтобы эксперту было легче работать с этими операторами в одном месте.

Другой пример слабого зацепления имеет отношение к распределенным серверным объектам. Поскольку быстродействие системы определяется производительностью удаленных объектов и их взаимодействием, иногда желательно создать несколько более крупных серверных объектов со слабым зацеплением, предоставляющих интерфейс многим операциям. Эта ситуация связана также с шаблоном Coarse-Grained Remote Interface (Укрупненный удаленный интерфейс), в рамках которого создаются укрупненные удаленные операции, выполняющие больше функций. Такое проектное решение объясняется повышенным влиянием удаленных вызовов на производительность сети. В качестве альтернативы вместо удаленного объекта с тремя операциями `setName`, `setSalary` и `setHireDate` лучше реализовать одну укрупненную удаленную операцию `setDate`, работающую с целым множеством данных. Это приведет к уменьшению числа удаленных вызовов и, как следствие, к повышению производительности.

**Преимущества**

- Повышаются ясность и простота проектных решений.
- Упрощаются поддержка и доработка.
- Зачастую обеспечивается слабое связывание.
- Улучшаются возможности повторного использования, поскольку класс с высокой степенью зацепления выполняет конкретную задачу.

## 16.10. Шаблон Controller

**Решение.** Делегирование обязанностей по обработке системных сообщений классу, удовлетворяющему одному из следующих условий.

- Класс представляет всю систему в целом, устройство или подсистему (*внешний контроллер*).

- Класс представляет сценарий некоторого прецедента, в рамках которого выполняется обработка всех системных событий, и обычно называется <*Прецедент*>`Handler`, <*Прецедент*>`Coordinator` или <*Прецедент*>`Session` (*контроллер прецедента* или *контроллер сеанса*).
    - Для всех системных событий в рамках одного сценария прецедента используется один и тот же класс-контроллер.
    - Неформально, сеанс — это экземпляр взаимодействия с исполнителем. Сеансы могут иметь произвольную длину, но зачастую организованы в рамках прецедента (сеансы прецедента).

*Следствие.* Заметим, что в этот перечень не включаются классы, реализующие окно, аплет, приложение, вид и документ. Такие классы не выполняют задачи, связанные с системными событиями. Они обычно получают сообщения и делегируют их контроллерам.

**Проблема.** Кто должен отвечать за обработку входных системных событий?

*Системное событие* (system event) — это событие высокого уровня, генерируемое внешним исполнителем (событие с внешним входом). Системные события связаны с *системными операциями* (system operation), т.е. операциями, выполняемыми системой в ответ на события.

Например, когда кассир в POS-системе щелкните на кнопке Оплатить, он генерирует системное событие, свидетельствующее о завершении торговой операции. Аналогично, когда пользователь текстового процессора выбирает команду Орфография, он генерирует системное событие "выполнить проверку орфографии".

*Контроллер* (controller) — это объект, не относящийся к интерфейсу пользователя и отвечающий за обработку системных событий. Контроллер определяет методы для выполнения системных операций.

**Пример.** В приложении NextGen существует несколько системных операций (рис. 16.13).

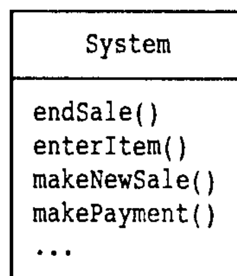



*Рис. 16.13. Системные операции связаны с системными событиями*

> В процессе анализа поведения системы системные операции относятся к классу `System`. Однако это не означает, что их выполняет программный класс `System`. Более того, обязанности по выполнению системных операций обычно возлагаются на класс Controller (рис. 16.14).

Какой класс должен выступать в роли контроллера для системных событий типа `enterItem` или `endSale`?

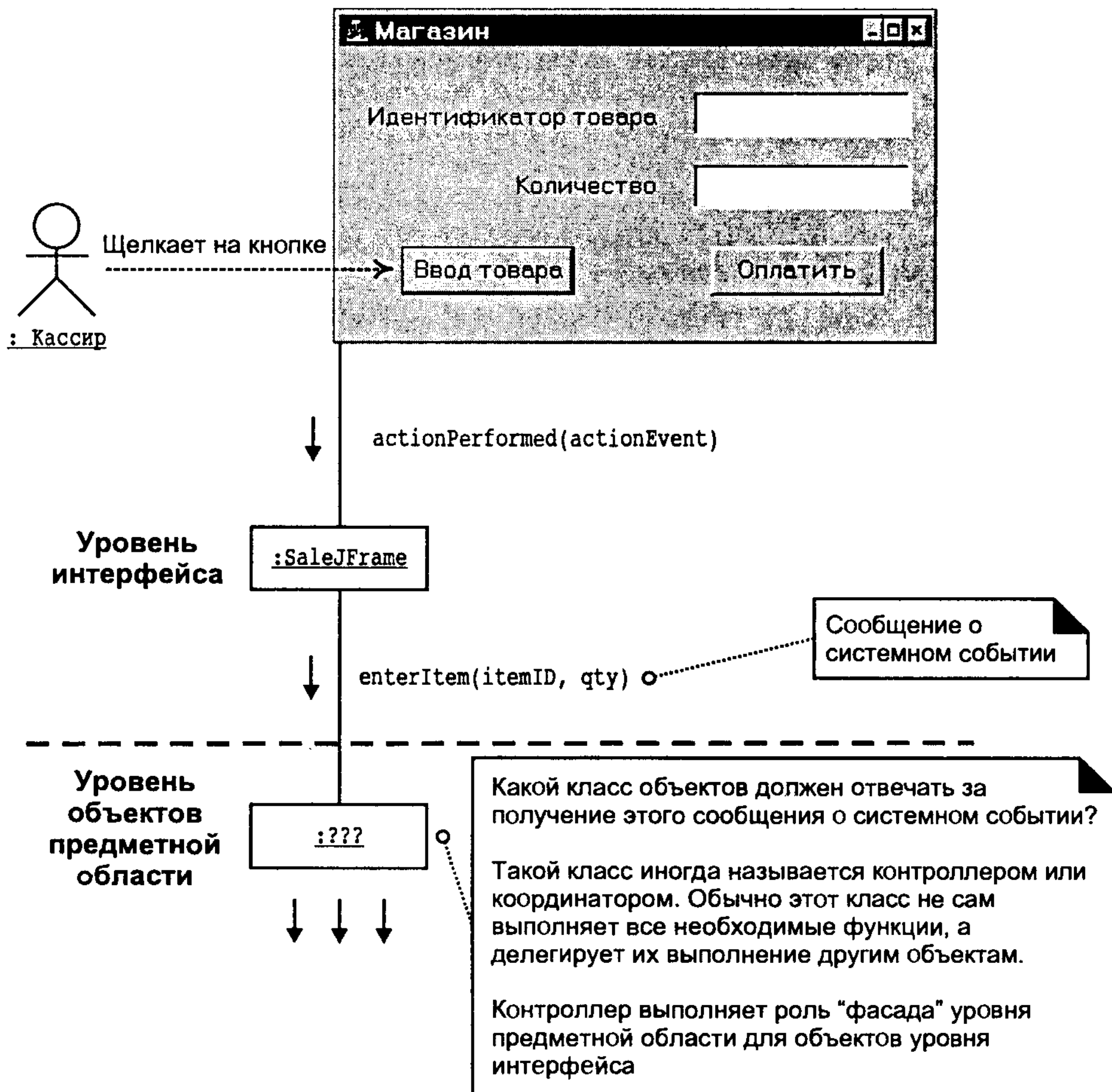



*Рис. 16.14. Кто является контроллером для события* `enterItem`?

Согласно шаблону Controller, возможны следующие варианты.

| | |
|---|---|
| Класс, представляющий всю систему в целом, устройство или подсистему. | `Register`, `POSSystem` |
| Класс, представляющий получателя или искусственный обработчик всех системных событий некоторого сценария прецедента. | `ProcessSaleHandler`, `ProcessSaleSession` |

В терминах диаграммы взаимодействий должен использоваться один из вариантов, представленных на рис. 16.15.

Выбор наиболее подходящего контроллера определяется другими факторами, в частности зацеплением и связыванием. Об этом более подробно рассказывается ниже.

Системные операции, идентифицированные в процессе анализа поведения системы, на этапе проектирования присваиваются одному или нескольким классам контроллеров, например `Register` (рис. 16.16).

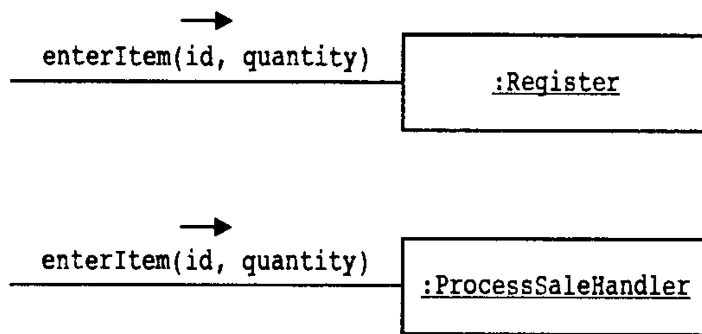



*Рис. 16.15. Варианты контроллеров*

System

endSale()
enterItem()
makeNewSale()
makePayment()

makeNewReturn()
enterReturnItem()
...

Register

...

endSale()
enterItem()
makeNewSale()
makePayment()

makeNewReturn()
enterReturnItem()
...

Системные операции, исследованные в процессе анализа поведения системы

Размещение системных операций в процессе проектирования при использовании одного внешнего контроллера

System

endSale()
enterItem()
makeNewSale()
makePayment()

enterReturnItem()
makeNewReturn()
...

ProcessSale Handler

...

endSale()
enterItem()
makeNewSale()
makePayment()

HandleReturns Handler

...

enterReturnItem()
makeNewReturn()
...

Размещение системных операций в процессе проектирования при использовании нескольких контроллеров прецедентов

*Рис. 16.16. Распределение системных операций*

**Обсуждение.** Большинство систем получает внешние события. Обычно они связаны с графическим интерфейсом пользователя. Кроме того, системе могут передаваться внешние сообщения, например, при обработке телекоммуникационных сигналов или сигналов от датчиков в системах управления.

Во всех случаях при использовании объектно-ориентированного подхода для обработки внешних событий применяются контроллеры. Шаблон Controller обеспечивает наиболее типичные проектные решения для этого случая. Как видно из рис. 16.14, контроллер — это своеобразный вид интерфейса между уровнями предметной области и графического представления.

Чтобы обеспечить возможность поддержки информации о состоянии прецедента, для обработки всех системных событий в рамках одного прецедента должен использоваться один и тот же класс контроллера. Такая информация может понадобиться, например, для идентификации момента нарушения последовательности системных событий (например, выполнение операции `makePayment` перед выполнением операции `endSale`). Для различных прецедентов можно использовать разные контроллеры.

Типичной ошибкой при создании контроллеров является возложение на них слишком большого числа обязанностей.

> Обычно контроллер должен лишь *делегировать* функции другим объектам и координировать их деятельность, а не выполнять эти действия самостоятельно.

Об этом подробнее рассказывается ниже в данном разделе.

Первым типом контроллеров является *внешний контроллер* (facade controller), представляющий всю "систему", устройство или подсистему. Основная идея сводится к выбору некоторого класса, имя которого охватывает все слои приложения. Этот класс обеспечивает главную точку вызова всех служб из интерфейса пользователя и обращения к остальным слоям. Этот класс может представлять физический объект, например систему `Register`,[5] телекоммуникационный переключатель `TelecommSwitch`, `Phone` или устройство `Robot`; всю программную часть системы, например `POSSystem`, или любые другие понятия, выбранные разработчиком для представления системы в целом.

Внешние контроллеры удобно использовать в том случае, когда существует лишь несколько системных событий или системные сообщения невозможно перенаправить другим контроллерам, наподобие системы обработки сообщений.

Если применяется *контроллер прецедента* (use case controller), то для каждого прецедента должен существовать отдельный контроллер. Заметим, что это не объект из предметной области, а искусственная конструкция, поддерживающая жизнедеятельность системы. Например, если в системе NextGen используются прецеденты `Оформление продажи` и `Возврат товара`, то в ней может быть реализован класс `ProcessSaleHandler` (Обработчик продажи).

Когда следует использовать контроллеры прецедентов? В том случае, когда применение внешнего контроллера приводит к слабой степени зацепления или высокой степени связывания. Контроллер прецедента вводится в том случае, если существующий контроллер слишком "раздувается" при возложении на него дополнительных обязанностей. Контроллеры прецедентов следует применять при наличии большого числа системных событий, распределенных между различными процессами.

---

[5] Для обозначения физического объекта POS-системы можно использовать разные термины, в том числе реестр, терминал системы розничной торговли и т.д. Термин "реестр" при этом может означать и физический объект, и логическую абстракцию, обеспечивающую регистрацию информации о продажах и платежах.

Такие контроллеры позволяют разделять обязанности их обработки между несколькими классами и отслеживать состояние текущего сценария.

В UP, а также более раннем методе Якобсона [65], существуют (необязательные) понятия пограничных, управляющих классов и классов сущностей. *Пограничные объекты* (boundary objects) — это абстракции интерфейсов, *классы сущностей* (entity objects) — это независимые от приложения (и, как правило, сохраняемые в базе данных) программные объекты, соответствующие понятиям из предметной области, а управляющие объекты (control object) — это обработчики прецедентов, подобные описываемым в шаблоне Controller.

Важным следствием применения шаблона Controller является вынесение обязанностей по выполнению системных событий за пределы уровня представления и внешнего интерфейса объектов (например, объектов окон или аплетов). Другими словами, системные операции, отражающие процессы в предметной области, должны обрабатываться на уровне логики приложения или реализации объектов, а не на уровне интерфейса. Этот вопрос более подробно рассматривается ниже.

Объект-контроллер, как правило, относится к клиентской части приложения и функционирует в рамках того же процесса, что и интерфейс пользователя (например, приложение с графическим интерфейсом Java Swing). Поэтому шаблон Controller напрямую неприменим, если в качестве клиента выступает Web-броузер. Для такой архитектуры существуют различные шаблоны обработки системных событий, основанные на использовании серверных технических каркасов, в частности на базе сервлетов Java. Чаще всего основная идея сводится к использованию контроллеров прецедентов в серверной части приложения, когда обработку каждого прецедента осуществляет либо отдельный сервлет, либо компонент-обработчик сеанса EJB (Enterprise JavaBeans). Серверный объект-обработчик сеанса обслуживает один сеанс взаимодействия с внешним исполнителем.

Если интерфейс с пользователем обеспечивается не через Web-броузер (например, через графический интерфейс Windows или Swing), но приложение вызывает удаленные службы, то такое приложение чаще всего основывается на шаблоне Controller. Из интерфейса пользователя направляется запрос локальному классу-контроллеру, относящемуся к клиентской части приложения, а этот класс может перенаправить весь запрос или его часть удаленным службам. При таком решении снижается степень связывания между интерфейсом пользователя и удаленными службами.

Таким образом, класс-контроллер получает запросы от объектов уровня интерфейса пользователя и координирует их выполнение, обычно делегируя обязанности другим объектам.

#### Преимущества

- *Улучшение условий для повторного использования компонентов.* Применение этого шаблона обеспечивает обработку процессов предметной области на уровне реализации объектов, а не на уровне интерфейса. Обязанности контроллера могут быть технически реализованы в объектах интерфейса, однако в этом случае программный код и логические решения, относящиеся к процессам предметной области, будут жестко связаны с элементами интерфейса, например с окнами. При этом снижается эффективность повторного использования компонентов в других приложениях, поскольку процессы предметной области ограничены рамками интерфейса (например, связаны с оконным объектом), что может оказаться неприемлемым в других прило-

жениях. Делегирование выполнения системных операций специальному контроллеру облегчает повторное использование логики обработки подобных процессов в последующих приложениях.

- *Контроль состояния прецедента.* Иногда необходимо удостовериться, что системные операции выполняются в некоторой определенной последовательности. Например, необходимо гарантировать, чтобы операция `makePayment` выполнялась только после операции `EndSale`, для чего необходимо накапливать информацию о последовательности событий. Для этой цели удобно использовать контроллер, особенно контроллер прецедента.

#### Проблемы и решения

### Раздутый контроллер

Плохо спроектированный класс контроллера имеет низкую степень зацепления: он выполняет слишком много обязанностей и является несфокусированным. Такой контроллер называется *раздутым* (bloated controller). Признаки раздутого контроллера таковы.

- В системе имеется *единственный* класс контроллера, получающий *все* системные сообщения, которых поступает слишком много. Такая ситуация зачастую возникает при использовании внешнего контроллера.
- Контроллер сам выполняет все задачи, не делегируя обязанности другим классам. Обычно это приводит к нарушению основных принципов шаблонов Information Expert и High Cohesion.
- Контроллер имеет много атрибутов и содержит значительный объем информации о системе или предметной области, которую необходимо распределить между другими объектами, либо дублирует информацию, хранящуюся в других объектах.

Существует несколько рецептов для устранения проблемы раздутого контроллера.

1. Добавьте несколько контроллеров; не нужно ограничиваться лишь одним. Помимо внешних контроллеров, используйте контроллеры прецедентов. Например, в системе, имеющей много системных событий (скажем, в системе резервирования авиабилетов), можно использовать следующие контроллеры.

| Контроллеры прецедентов |
|---|
| `MakeReservationHandler` (обработчик события резервирования) |
| `ManageSchedulesHandler` (менеджер расписания) |
| `ManageFaresHandler` (менеджер тарифов) |

2. Спроектируйте контроллер таким образом, чтобы он делегировал обязанности по выполнению системных операций другим объектам.

### Системные события не обрабатываются на уровне представления

Напомним, что важным следствием применения шаблона Controller является такой факт: объекты интерфейса (например, окна) не обрабатывают системные события. Например, рассмотрим систему, написанную на языке Java и использующую объект `JFrame` для отображения информации.

Допустим, в системе розничной торговли имеется окно, отображающее информацию о продаже и позволяющее кассиру выполнять необходимые операции. На рис. 16.17 показаны приемлемые отношения между объектом JFrame, контроллером и другими объектами в упрощенной версии POS-системы.

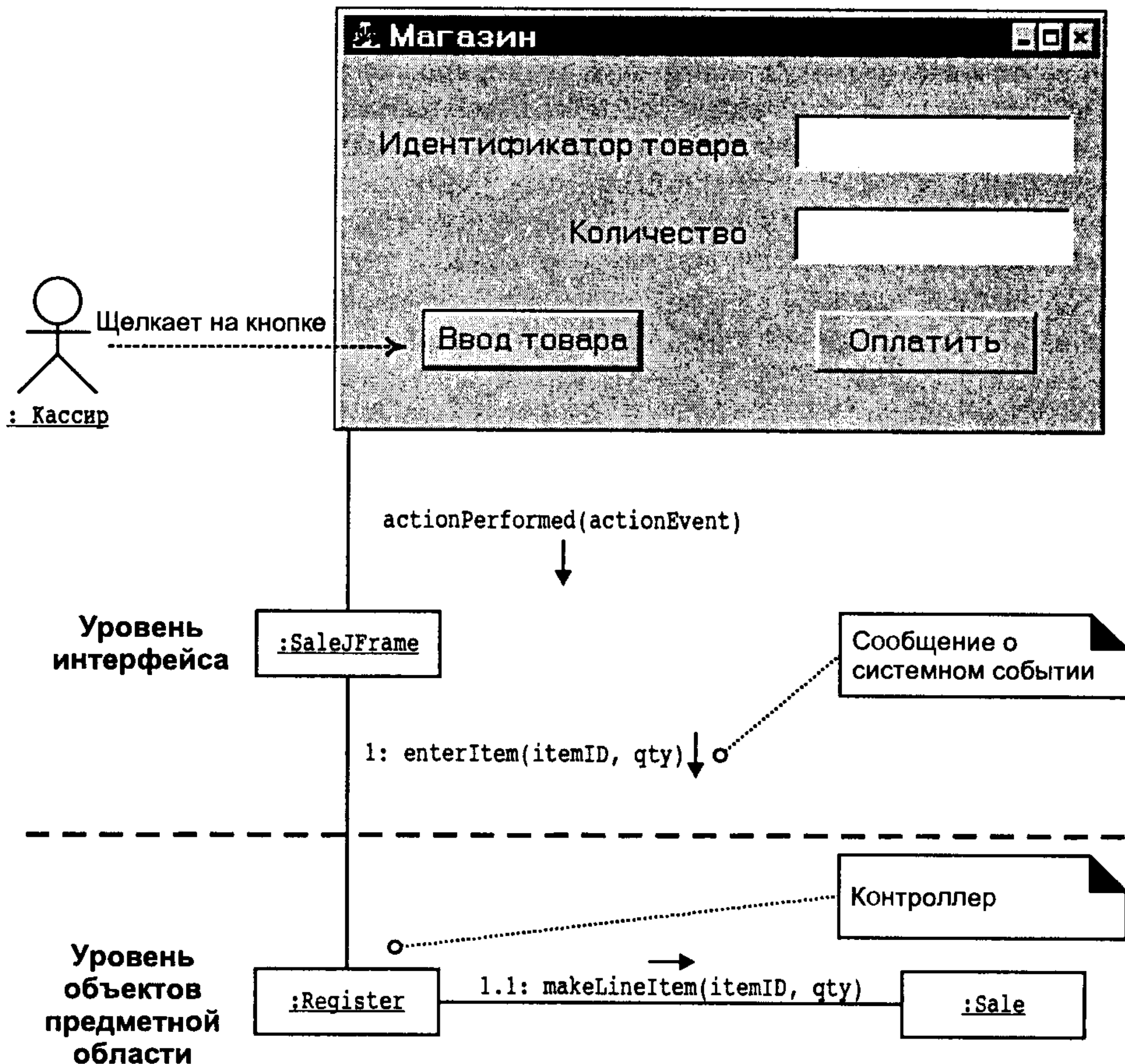



*Рис. 16.17. Связывание уровней интерфейса и предметной области*

Заметим, что класс SaleJFrame, относящийся к уровню интерфейса, передает сообщение enterItem объекту Register. Он не принимает участия в обработке этой операции, а лишь делегирует функцию уровеню реализации.

Распределение обязанностей по выполнению системных операций между объектами предметной области (согласно шаблону Controller), а не объектами уровня представления повышает эффективность повторного использования. Если объект интерфейса (скажем, SaleJFrame) обрабатывает системную операцию (составляющую часть процесса предметной области), то логика процесса предметной области должна быть заложена в интерфейсе (например, в оконном объекте). Такое проектное решение снижает эффективность повторного использования, поскольку данное приложение существенно связано с конкретным интерфейсом.

Следовательно, на рис. 16.18 представлено неприемлемое проектное решение.

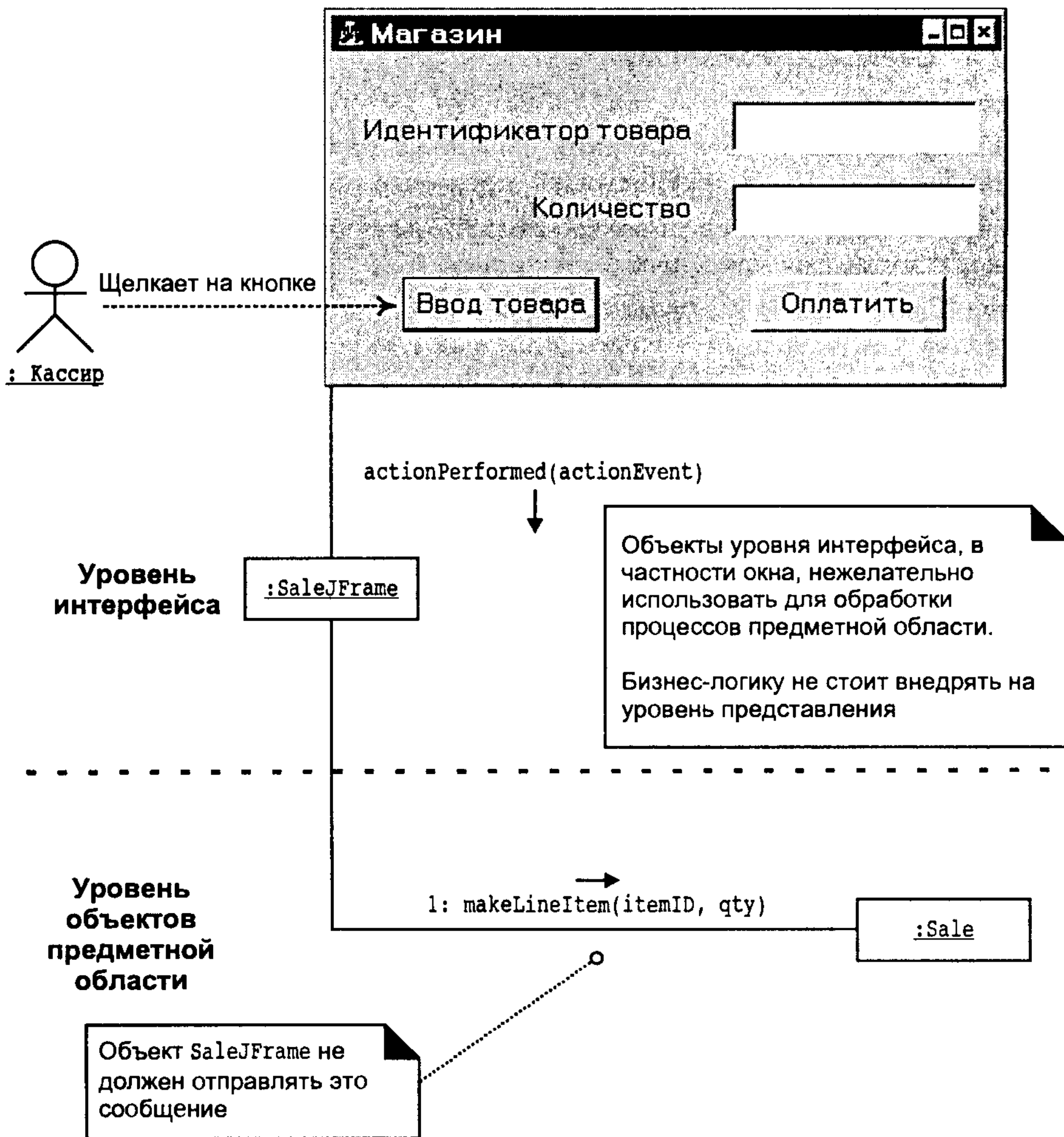



*Рис. 16.18. Нежелательное связывание уровня интерфейса с уровнем предметной области*

Возложение ответственности за выполнение системной операции на специальный контроллер упрощает повторное использование логики программы в последующих приложениях, а также облегчает возможность применения другого интерфейса для данного приложения или использование приложения в неинтерактивном "пакетном" режиме.

### Системы обработки сообщений и шаблоны команд

Многие приложения не содержат интерфейса пользователя, а получают сообщения от других внешних систем. Такие системы называются системами обработки сообщений. Типичным примером такой системы является телекоммуникационный переключатель. В таких системах термины "интерфейс" и "контроллер" имеют иное значение. Более подробно этот вопрос будет изучен в последующих главах, однако

стоит отметить, что стандартным решением является использование шаблона Command [52] и Command Processor [15].

**Похожие шаблоны**

- Command (Команда) — в системах обработки сообщений каждое сообщение может представлять и обрабатывать отдельный объект `Command` [52].
- Facade (Фасад) — выбор объекта, представляющего всю систему или организацию в качестве внешнего контроллера [52].
- Layers (Слои) — шаблон, предложенный в [15]. Согласно шаблону Layers, логика реализации размещается в слое реализации, а не в слое представления.
- Pure Fabrication (Чистая синтетика) — еще один шаблон GRASP, который подразумевает создание искусственного класса, не имеющего аналога в предметной области. Элементом шаблона Pure Fabrication является контроллер прецедента.

## 16.11. Проектирование объектов и карты CRC

Для распределения обязанностей и выявления взаимодействия между объектами иногда используется еще одно средство, формально не являющееся частью языка UML — *карты CRC* (CRC cards — Class-Responsibility-Collaborator cards) [8]. Они были введены Кентом Беком (Kent Beck) и Уордом Каннингхемом (Ward Cunningham), которые внесли значительный вклад в повышение абстрактности мышления разработчиков объектно-ориентированных систем и уделили большое внимание распределению обязанностей и взаимодействию объектов.

Карты CRC — это индексные карты (по одной на каждый класс), на которых кратко описаны обязанности классов и перечислены объекты, взаимодействующие с данным классом при выполнении этих обязанностей. Обычно они разрабатываются в процессе небольших групповых семинаров.

Карты CRC — это лишь один из возможных подходов к записи результатов распределения обязанностей. Следующим шагом на этом пути является создание диаграмм классов и взаимодействий. Реальное значение имеют не сами карты как таковые, а пристальное внимание к процессу распределения обязанностей.

## 16.12. Дополнительная литература

Идея взаимодействия объектов при выполнении своих обязанностей (или проектирование на основе обязанностей) (Responsibility Driven Design) не нова. Она развивалась еще в процессе программирования на языке Smalltalk в компании Tektronix (Портленд) Кентом Беком (Kent Beck), Уордом Каннингемом (Ward Cunningham), Ребеккой Вирфс-Брок (Rebecca Wirfs-Brock) и др. Основополагающим трудом в этой области является [107], который в настоящее время столь же актуален, как и во время его написания.

Рекомендуем еще одну книгу, в которой основное внимание уделяется фундаментальным принципам объектного проектирования. Это книга Коада [30].

Глава

# Модель проектирования: реализация прецедентов на основе шаблонов GRASP

*Чтобы изобрести нечто новое, нужно иметь хорошее воображение и груду заготовок.*

*Томас Эдисон (Thomas Edison)*

Основные задачи

- Спроектировать реализации прецедентов.
- Применить шаблоны GRASP для распределения обязанностей между классами.
- Использовать обозначения диаграммы кооперации для иллюстрации взаимодействия объектов.

## Введение

В этой главе рассказывается о проектировании взаимодействующих классов, выполняющих определенные обязанности. Особое внимание уделяется применению шаблонов GRASP для разработки удачного проектного решения. Стоит заметить, что шаблоны GRASP сами по себе не очень важны, они лишь облегчают общение между разработчиками и позволяют методически проектировать объекты.

В этой главе на примере POS-системы NextGen обсуждаются принципы, на основе которых разработчик распределяет обязанности между объектами и определяет механизмы их взаимодействия. В процессе разработки объектно-ориентированных систем эти вопросы являются наиболее важными.

Распределение обязанностей и проектирование взаимодействия объектов — наиболее важные "творческие" факторы процесса проектирования. Они чрезвычайно важны как для построения диаграмм, так и для программной реализации.

Материал этой главы чрезвычайно детализирован. Мы постараемся подробно показать, что в процессе объектно-ориентированного проектирования нет ничего магического или необъяснимого. Варианты распределения обязанностей и выбор способов взаимодействия между объектами можно рационально объяснить и изучить.

## 17.1. Реализация прецедентов

Приведем цитату: "Реализация прецедента показывает, как определенный прецедент реализуется в рамках модели проектирования в терминах взаимодействующих объектов" [97]. Если сказать более точно, разработчик может описать проектное решение для одного или нескольких *сценариев* (scenario) прецедента, каждый из которых называется реализацией прецедента. Реализация прецедента — это термин процесса UP или концепция, позволяющая сохранить связь между требованиями, выраженными в виде прецедентов, и процессом проектирования объектов, которые этим требованиям удовлетворяют.

Диаграммы взаимодействия UML являются стандартным средством, используемым для иллюстрации реализаций прецедента. Как упоминалось в предыдущей главе, в процессе проектирования объектов можно использовать принципы и шаблоны, такие как Information Expert и Low Coupling.

На рис. 17.20 (в конце этой главы) иллюстрируется взаимосвязь между некоторыми артефактами UP.

- Системные события определяются на основе прецедентов и подробно отображаются на диаграммах последовательностей.
- Результаты системных событий в терминах изменения объектов предметной области могут дополнительно определяться в описаниях системных операций.
- Системные события представляют собой сообщения, с которых начинаются диаграммы взаимодействия, иллюстрирующие взаимодействие объектов для выполнения требуемых задач — реализацию прецедентов.
- Диаграммы взаимодействия включают процессы передачи сообщений между программными объектами, имена которых зачастую определяются именами концептуальных классов модели предметной области, а также другими классами объектов.

## 17.2. Комментарии к артефактам

### *Диаграммы взаимодействия и реализация прецедентов*

На текущей итерации рассматриваются различные сценарии и следующие системные события.

- `Оформление продажи` (Process Sale): `makeNewSale`, `enterItem`, `endSale`, `makePayment`.

Если диаграммы кооперации используются для иллюстрации реализации прецедентов, то для отражения процесса обработки каждого сообщения, соответствующего системному событию, требуется отдельная диаграмма (рис. 17.1).

Однако если для этого используется диаграмма последовательностей, то все сообщения *можно* отобразить на одной и той же диаграмме, как показано на рис. 17.2.

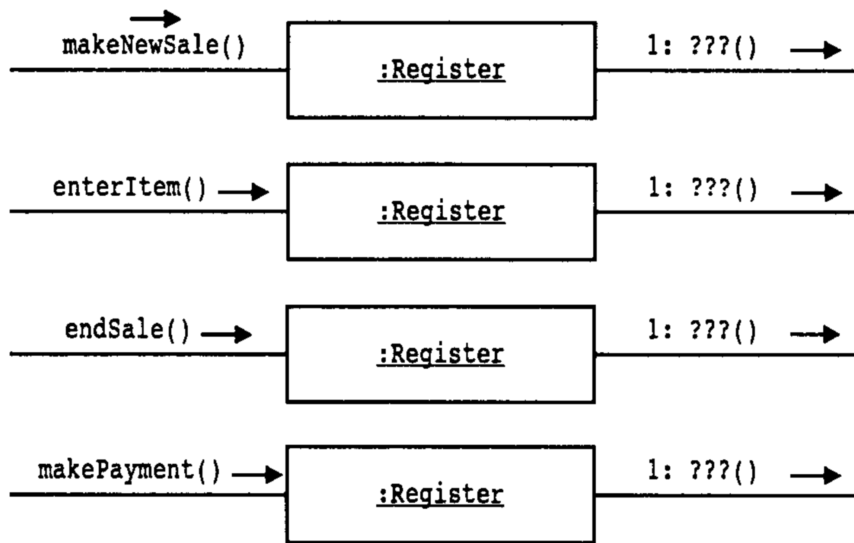



*Рис. 17.1. Диаграммы кооперации и обработка сообщений о системных событиях*

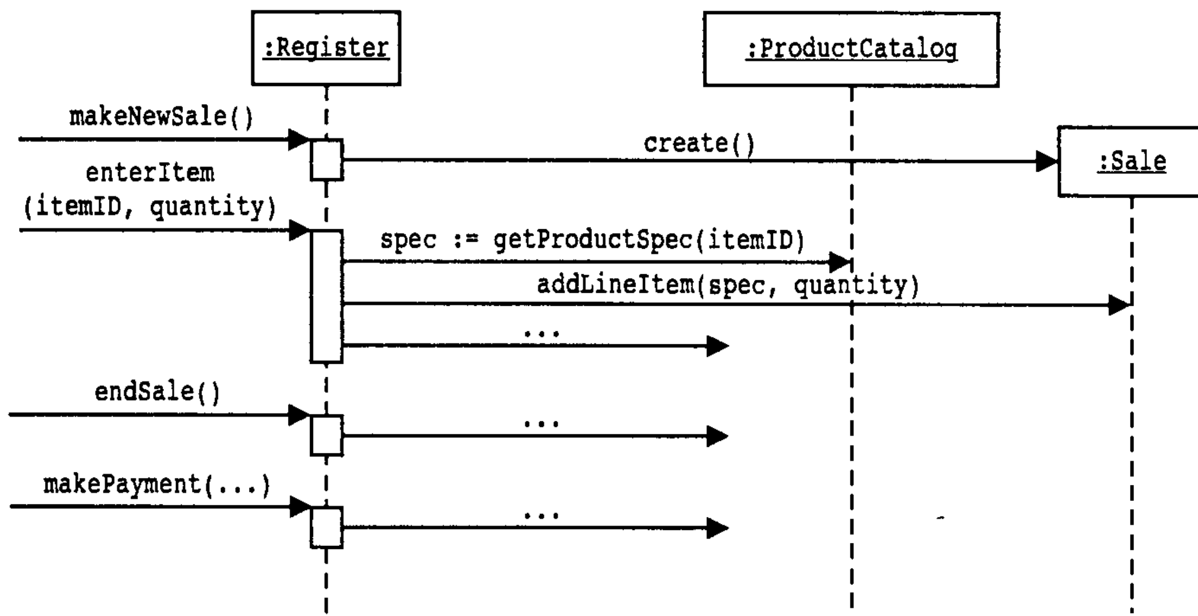



*Рис. 17.2. Одна диаграмма последовательностей и обработка сообщений о системных событиях*

Однако зачастую диаграмма последовательностей оказывается слишком сложной или большой. В этом случае, как и при использовании диаграмм взаимодействия, можно строить отдельную диаграмму последовательностей для каждого сообщения о системном событии, как показано на рис. 17.3.

## Описания системных операций и реализация прецедентов

Стоит еще раз напомнить, что реализацию прецедентов вполне возможно проектировать непосредственно на основе текста из описания прецедента. Кроме того, описание некоторых системных операций может быть более подробным или содержать некоторые специфические особенности, как, например, показано ниже.

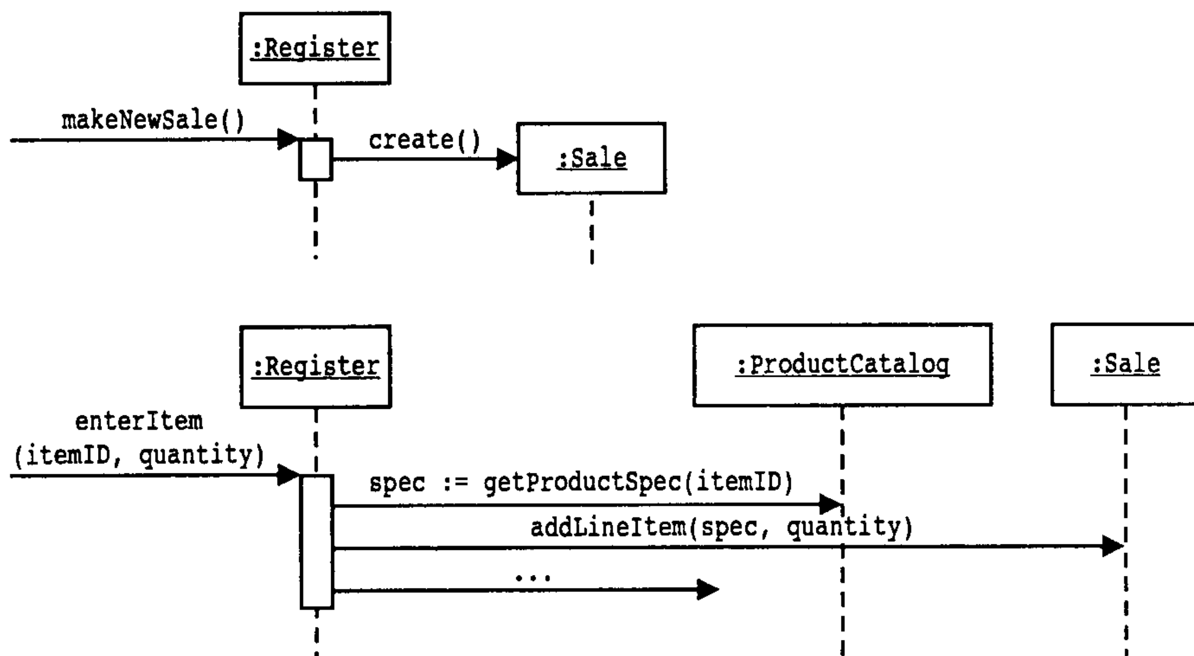



*Рис. 17.3. Несколько диаграмм последовательностей и обработка сообщений о системных событиях*

### Описание операции ОП2: enterItem

| | |
|---|---|
| **Операция** | enterItem(itemID : ItemID, quantity : integer) |
| **Ссылки** | Прецеденты: Оформление продажи (Process Sale) |
| **Предусловия** | Инициирована продажа |
| **Постусловия** | - Создан экземпляр sli класса SalesLineItem (*создание экземпляра*) |
| | - ... |

На основе этой информации анализа описаний прецедентов для каждого описания определяются изменения состояния, приводящие к реализации постусловий, и проектируются виды взаимодействия по передаче сообщений с целью удовлетворения требованиям. Например, на основе приведенного выше фрагмента описания системной операции enterItem можно построить диаграмму взаимодействия, представленную на рис. 17.4 и завершающуюся созданием экземпляра объекта SalesLineItem.

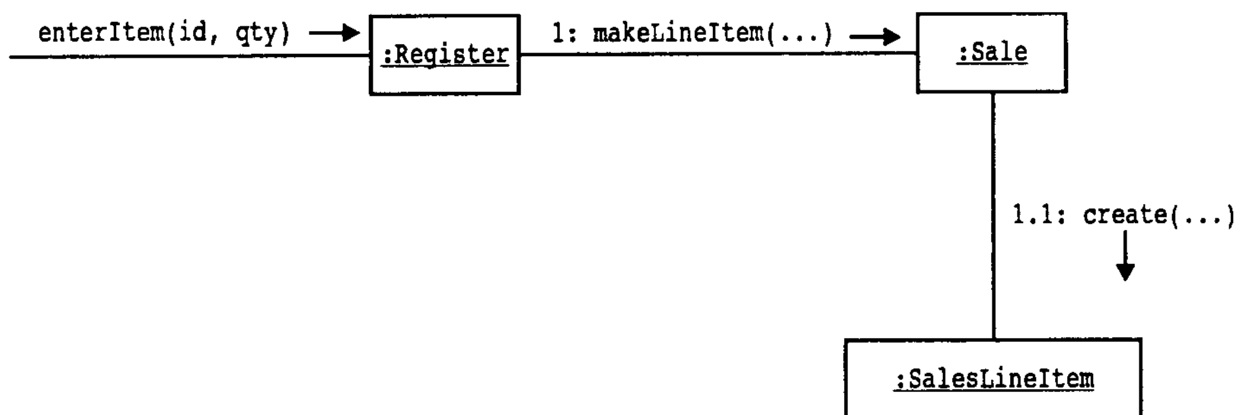



*Рис. 17.4. Фрагмент диаграммы взаимодействия*

### Предупреждение: требования могут измениться

Необходимо помнить, что описанные ранее прецеденты и созданные описания операций — это лишь примерное описание видения будущей системы. История разработки программного обеспечения свидетельствует о том, что требования нельзя считать окончательными, поскольку в процессе разработки они могут изменяться. Это не означает, что можно отказаться от этапа формулировки требований или считать этот этап не очень важным. Однако реальные требования определяются только в процессе получения обратной связи от пользователей и экспертов в предметной области.

Преимуществом итеративной разработки является естественная поддержка многочисленных этапов анализа и проектирования в процессе разработки системы и ее реализации. Идея итеративной разработки состоит в получении некоторого количества информации на этапе анализа требований с последующим ее уточнением в процессе проектирования и реализации.

### Модель предметной области и реализация прецедентов

Некоторые из программных объектов, взаимодействующих между собой посредством передачи сообщений и отображаемых на диаграммах взаимодействия, формируются на основе модели предметной области. Так, концептуальному классу `Sale` соответствует класс `Sale` модели проектирования. Обязанности между ними распределяются с учетом шаблонов GRASP, на основе информации о модели предметной области. Как уже упоминалось, существующая модель предметной области имеет свои недостатки. В ней возможны ошибки и упущения. Вам придется вводить новые понятия, которые были упущены ранее, а также выполнять аналогичные действия с ассоциациями и атрибутами.

### Концептуальные классы и классы модели проектирования

Напомним, что в модели предметной области программные классы не отображаются, однако из нее можно почерпнуть имена некоторых программных классов для модели проектирования. На этапе построения диаграмм взаимодействия или программирования разработчики могут обращаться к модели предметной области и заимствовать из нее имена некоторых программных классов. При этом сокращается разрыв между проектным решением программной системы и понятиями реальной предметной области, для которой эта система создается (рис. 17.5).

Должны ли имена программных классов из модели проектирования в точности соответствовать именам объектов модели предметной области? Вовсе нет. На этапе проектирования могут "всплыть" новые имена концептуальных классов, которые были упущены на ранних стадиях анализа. Кроме того, могут появиться программные классы, имена и назначение которых абсолютно не соответствуют модели предметной области.

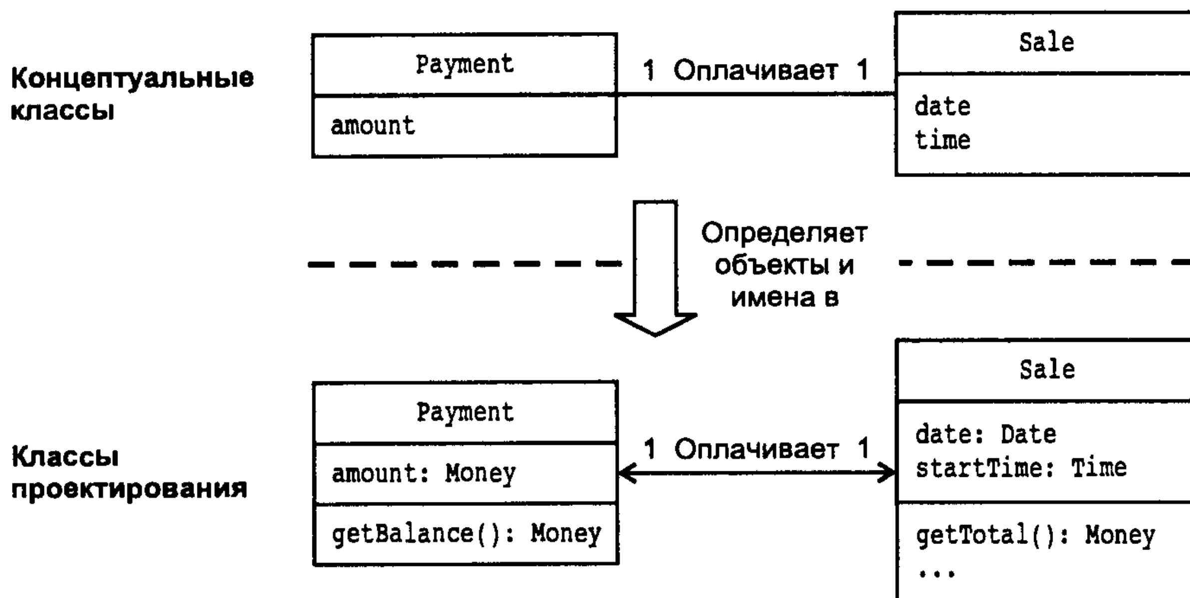



*Рис. 17.5. Сокращение разрыва между именами программных и концептуальных классов*

## 17.3. Реализация прецедентов для данной итерации разработки системы NextGen

Рассмотрим процесс проектирования реализации прецедентов на основе шаблонов GRASP. Это будет сделано достаточно подробно, чтобы проиллюстрировать, что при построении диаграмм кооперации нет ничего сверхъестественного. Они базируются на вполне обоснованных принципах.

Объекты, принимающие участие в обработке каждого системного события, будут показаны на отдельной диаграмме, чтобы сфокусировать внимание на каждом проектном решении. Однако их можно сгруппировать и поместить на общей диаграмме последовательностей.

## 17.4. Проектное решение: `makeNewSale`

Системная операция `makeNewSale` инициируется в тот момент, когда кассир передает запрос о начале новой продажи после того, как покупатель подходит к кассе с выбранными покупками. Необходимую для реализации этой операции информацию можно почерпнуть непосредственно из описания прецедента, однако при рассмотрении POS-системы мы условились составлять описания для всех системных операций.

**Описание операции ОП1: makeNewSale**

| | |
|---|---|
| **Операция** | makeNewSale() |
| **Ссылки** | Прецеденты: Оформление продажи |
| **Предусловия** | Отсутствуют |
| **Постусловия** | - Создан экземпляр s объекта Sale (*создание экземпляра*)<br>- Экземпляр Sale связан с объектом Register (*формирование ассоциации*)<br>- Инициализированы атрибуты экземпляра s |

### *Выбор класса-контроллера*

Сначала необходимо выбрать контроллер для обработки сообщений системной операции enterItem. Согласно шаблону Controller, в качестве контроллера может выступать класс, удовлетворяющий одному из следующих условий.

| | |
|---|---|
| Класс представляет всю систему в целом, устройство или подсистему. | Register, POSSystem |
| Класс является получателем или обработчиком всех системных событий для некоторого сценария прецедента. | ProcessSaleHandler, ProcessSaleSession |

Выбор внешнего контроллера, например класса Register, обоснован в том случае, если в приложении существует лишь несколько системных операций и на внешний контроллер будет возложено не слишком много обязанностей (другими словами, не нарушится зацепление). Контроллеры прецедентов удобно использовать при наличии множества системных операций для распределения обязанностей между различными классами во избежание перегрузки классов-контроллеров (другими словами, для обеспечения зацепления). В данном случае, так как в системе существует лишь несколько системных операций, в качестве контроллера можно выбрать класс Register.

> Важно понимать, что в данном случае Register — это программный объект из модели проектирования, а не физический реестр системы розничной торговли. Он представляет собой программную абстракцию, имя которой способствует сокращению разрыва между именами программных объектов и понятиями предметной области.

Таким образом, представленная на рис. 17.6 диаграмма взаимодействия начинается с отправляемого сообщения makeNewSale программному объекту Register.

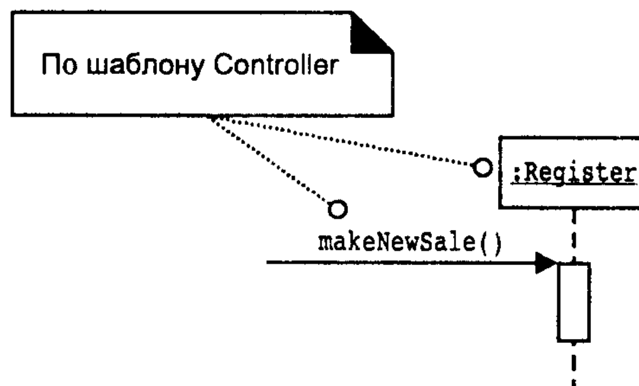



*Рис. 17.6. Применение шаблона Controller*

## *Создание нового экземпляра объекта* `Sale`

Для создания программного объекта `Sale` воспользуемся шаблоном Creator, согласно которому обязанность по созданию новых экземпляров делегируется классу, содержащему, агрегирующему или записывающему информацию о создаваемых классах.

Проанализировав модель предметной области, приходим к выводу, что запись информации о продажах может выполнять объект `Register`. Заметим, что термин "реестр" означает журнал, в который записываются (или регистрируются) сведения о транзакциях, в частности о проданных товарах.

Поэтому объекту `Register` целесообразно поручить создание экземпляров объектов `Sale`. Если экземпляры `Sale` будут создаваться объектом `Register`, то впоследствии объект `Register` будет содержать ссылку на текущий экземпляр `Sale`.

Кроме того, после создания объекта `Sale` необходимо создать пустую коллекцию (контейнер наподобие `List` в Java) для записи всех добавляемых впоследствии экземпляров `SalesLineItem`. Эта коллекция будет поддерживаться экземпляром `Sale`, который, согласно шаблону Creator, является наилучшим кандидатом для ее создания.

Таким образом, объект `Register` создает экземпляры `Sale`, а `Sale`, в свою очередь, создает пустой контейнер, представленный на диаграмме взаимодействия как сложный объект.

Таким образом, получена диаграмма взаимодействия, представленная на рис. 17.7.

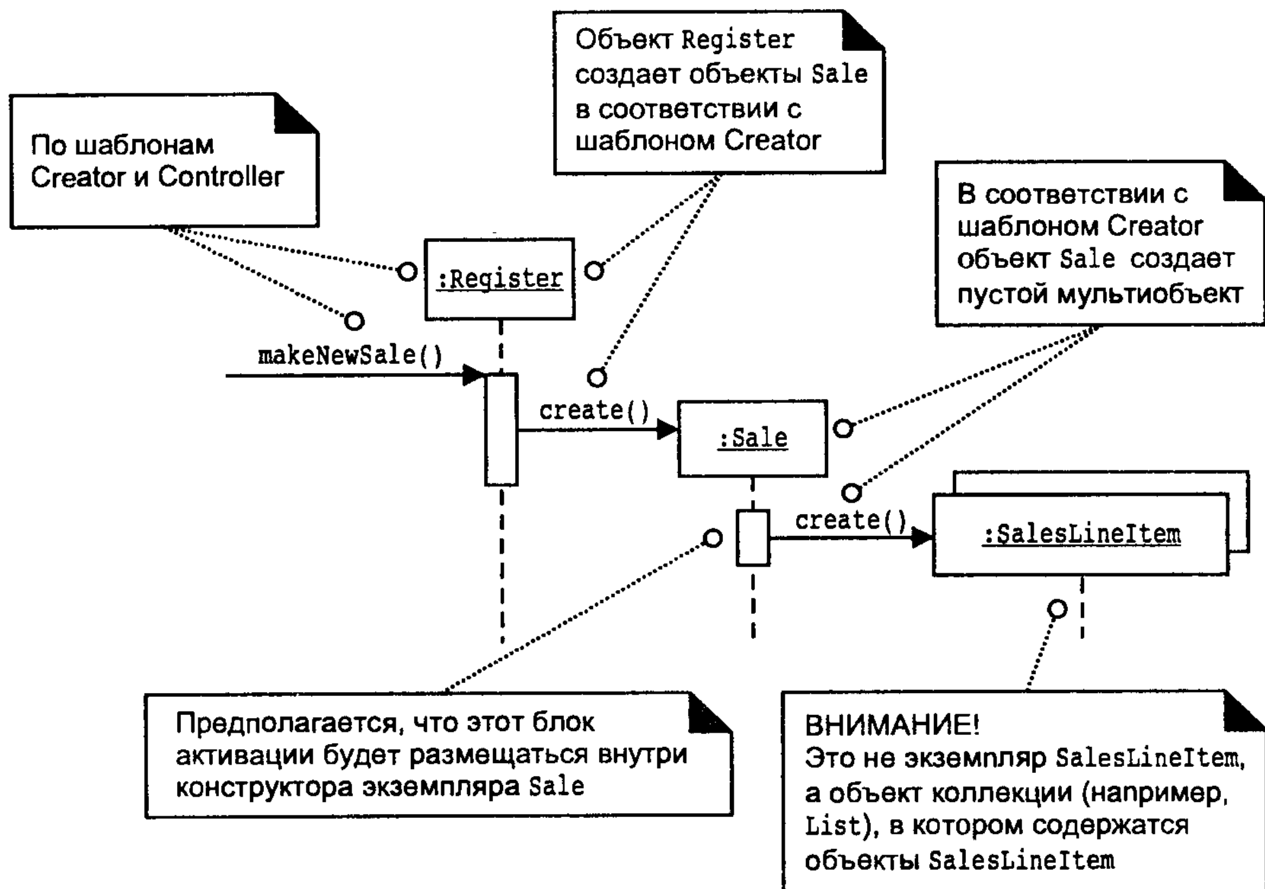



*Рис. 17.7. Создание экземпляров* `Sale` *и сложного объекта*

### *Заключение*

Процесс проектирования оказался несложным, однако на его примере мы смогли продемонстрировать применение шаблонов Controller и Creator и методически пояснить проектное решение в терминах принципов проектирования и шаблонов GRASP.

## 17.5. Проектное решение: enterItem

Системная операция enterItem начинается с ввода кассиром кода товара itemID и количества покупаемых единиц. Приведем полное описание этой операции.

#### Описание операции ОП2: enterItem

| | |
|---|---|
| **Операция** | enterItem(itemID: ItemID, quantity: integer) |
| **Ссылки** | Прецеденты: Оформление продажи |
| **Предусловия** | Инициирована продажа |
| **Постусловия** | - Создан экземпляр sli класса SalesLineItem (*создание экземпляра*)<br>- Экземпляр sli связан с текущим экземпляром класса Sale (*формирование ассоциации*)<br>- Атрибуту sli.quantity присвоено значение quantity (*модификация атрибута*)<br>- Экземпляр sli связан с классом ProductSpecification на основе соответствия идентификатора товара itemID (*формирование ассоциации*) |

Диаграмма взаимодействия должна обеспечивать выполнение постусловий описания системной операции enterItem. Проектное решение принимается на основе шаблонов GRASP.

### *Выбор класса-контроллера*

Класс-контроллер должен обрабатывать сообщение системной операции enterItem. Согласно шаблону Controller, в качестве контроллера будем использовать класс Register, как и для операции makeNewSale.

### *Отображение описания товара и его цены*

В соответствии с шаблоном Model-View Separation, специальные объекты (такие, как Register и Sale) не должны взаимодействовать с элементами уровня интерфейса пользователя (например, графическими окнами). Поэтому на данном этапе отображение описания товара и его цены не рассматривается.

Пока достаточно лишь знать, что эти данные известны.

### *Создание экземпляров SalesLineItem*

Среди постусловий в описании системной операции enterItem содержатся данные о создании экземпляра объекта SalesLineItem, его инициализации и связи с другими объектами. Анализируя модель предметной области, приходим к выводу, что объект Sale содержит объекты SalesLineItem, в связи с чем именно этому классу логично поручить создание экземпляров SalesLineItem.

В таком случае со временем объект Sale будет связан с новым экземпляром коллекции продаваемых товаров. Из постусловий следует, что при создании экземпляра SalesLineItem необходимо указывать количество единиц покупаемого

товара, поэтому данное значение необходимо передавать из объекта Register объекту Sale, который, в свою очередь, передаст его в качестве параметра сообщения create (на языке Java это можно реализовать с помощью вызова конструктора с параметром).

Таким образом, согласно шаблону Creator, для создания объекта SalesLineItem объекту Sale передается сообщение makeLineItem. Объект Sale создает экземпляр SalesLineItem, а затем хранит этот новый экземпляр в своем постоянном контейнере.

Параметрами сообщения makeLineItem являются количество единиц товара quantity и спецификация товара ProductSpecification, соответствующая коду товара itemID.

### Нахождение ProductSpecification

Экземпляр SalesLineItem необходимо связать со спецификацией ProductSpecification, соответствующей коду данного товара itemID. Это означает, что необходимо уметь по коду товара находить значение ProductSpecification.

Прежде чем определять, *как* следует находить значение ProductSpecification, необходимо решить, *кто* это будет делать. Для начала следует воспользоваться полезной рекомендацией.

| Распределение обязанностей необходимо начинать только после их изучения. |
|---|

Переформулируем задачу следующим образом.

Кто должен отвечать за нахождение значения ProductSpecification на основе кода товара itemID?

Эта проблема не является ни задачей создания экземпляра, ни задачей выбора контроллера для системного события. Здесь впервые предоставляется возможность использования шаблона Information Expert.

В большинстве случаев здесь необходимо применить шаблон Expert, согласно которому эта обязанность возлагается на объект, обладающий нужной информацией. Какой класс обладает информацией о спецификациях товаров ProductSpecification?

Проанализировав модель предметной области, приходим к выводу, что класс ProductCatalog логически содержит полную информацию о ProductSpecification. На основе структуры объектов предметной области можно разработать аналогичную организацию программных классов: программный класс ProductCatalog будет содержать программные классы ProductSpecification.

Следовательно, согласно шаблону Expert, класс ProductCatalog является хорошим кандидатом для реализации обязанности поиска, поскольку обладает полной информацией обо всех объектах ProductSpecification.

Это можно реализовать, например, с помощью метода getSpecification.[1]

---

[1] В каждом языке приняты свои соглашения об именовании методов. В Java всегда используется форма object.getFoo(), в C++ принято использовать обозначение object.foo(), а в C# — object.Foo. В данной книге используется стиль Java.

### *Обеспечение видимости класса* `ProductCatalog`

Кто должен отправлять сообщение getSpecification классу ProductCatalog с запросом на получение значения ProductSpecification?

Разумно предположить, что экземпляры объектов Register и ProductCatalog должны быть созданы в процессе реализации прецедента Запуск системы (Start Up), и между этими объектами должна устанавливаться постоянная связь. Согласно такому предположению, отправку сообщения getSpecification для класса ProductCatalog можно поручить классу Register.

Здесь возникает еще один важный вопрос объектно-ориентированного проектирования: вопрос видимости. *Видимость* (visibility) — это способность одного объекта "видеть" другой объект или ссылаться на него.

> Для того чтобы один объект мог отправлять сообщения другому объекту, объект-получатель должен находиться в области видимости объекта-отправителя.

Поскольку мы предположили, что объект Register имеет постоянную связь с объектом ProductCatalog (или ссылку на него), он может отправлять ему сообщения, в том числе сообщение getSpecification.

В следующей главе вопросы обеспечения видимости будут рассмотрены более подробно.

### *Получение значения* `ProductSpecification` *из базы данных*

Очень нежелательно, чтобы в последней версии POS-системы NextGen все значения ProductSpecification хранились в оперативной памяти. Скорее всего, они будут храниться в реляционной или объектной базе данных и извлекаться из нее по требованию. Однако пока мы не будем рассматривать вопросы извлечения информации из базы данных, а предположим, что все значения ProductSpecification находятся в оперативной памяти.

Вопросы обеспечения доступа к базе данных представляют особый интерес, требуют привлечения таких технологий, как J2EE, .NET. Они будут рассмотрены в главе 34.

### *Диаграмма взаимодействия для системной операции* `enterItem`

Согласно приведенным выше рассуждениям, можно построить диаграмму взаимодействия, отражающую распределение обязанностей и способы взаимодействия между объектами (рис. 17.8). Заметим, что эта диаграмма построена в результате взвешенного применения шаблонов GRASP.

### *Сообщения сложным объектам*

Заметим, что отправка сообщения сложному объекту в UML интерпретируется как передача сообщения объекту-контейнеру, а не каждому элементу набора объектов. Это особенно очевидно для таких операций, как find и add.

Например, из диаграммы взаимодействия для системной операции enterItem можно узнать следующее.

- Сообщение find, передаваемое сложному объекту ProductSpecification, — это сообщение, которое отправляется один раз целому набору данных, представленному в виде сложного объекта (такому, как Map в Java).

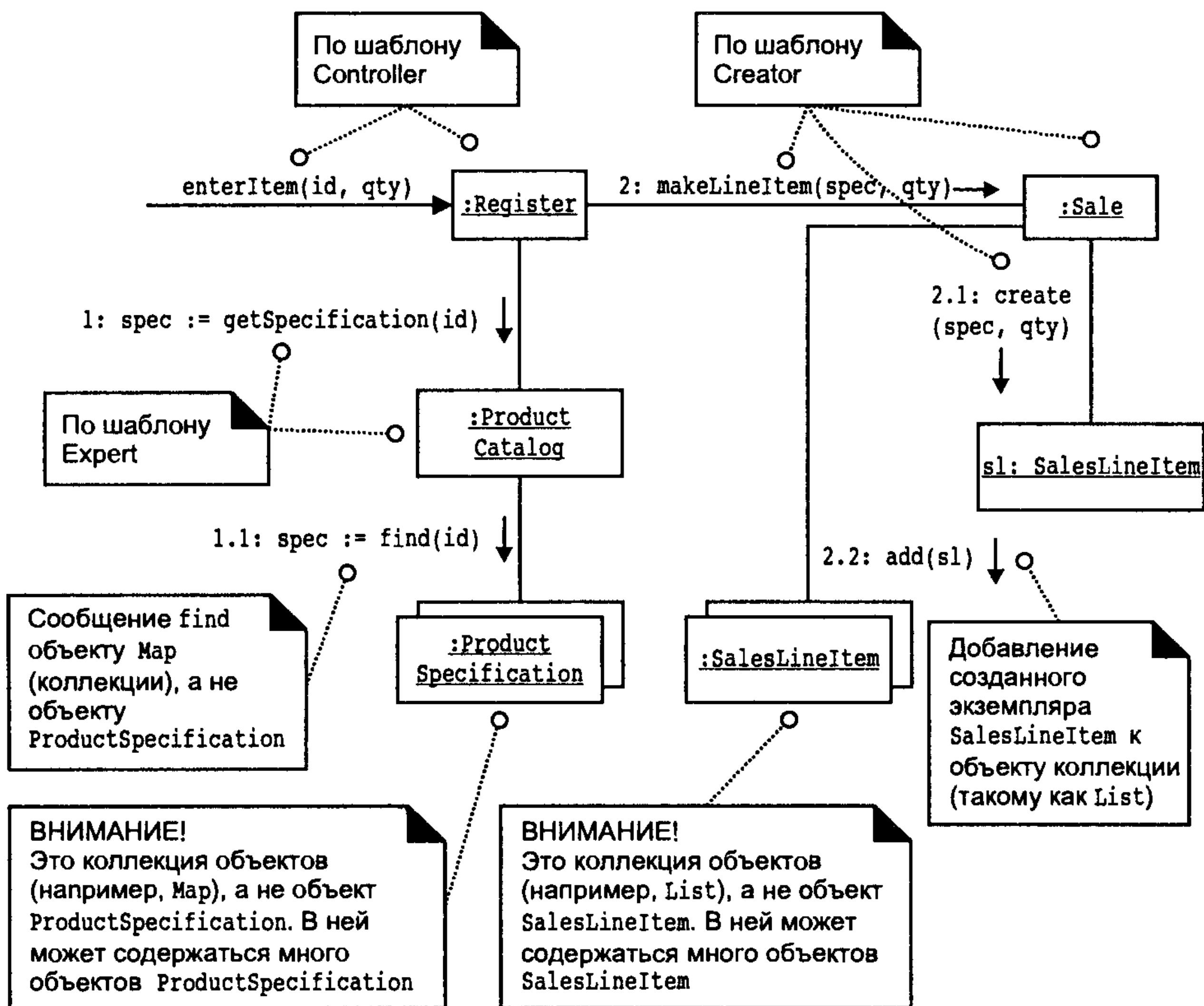



*Рис. 17.8. Диаграмма взаимодействия для системной операции* `enterItem`

- Независимое от языка общее сообщение `find` в процессе программирования будет трансформировано в реальное название метода конкретной библиотеки или языка программирования. Возможно, на языке Java оно превратится в метод `Map.get`. На диаграмме также можно было использовать сообщение `get`. Однако мы выбрали имя `find`, чтобы продемонстрировать необходимость перехода от проектных решений к конкретным библиотекам и языкам программирования.

- Сообщение `add` передается сложному объекту `SalesLineItem` для добавления элемента в контейнер, представленный сложным объектом (таким как `List` в Java).

## 17.6. Проектное решение: endSale

Системная операция `endSale` выполняется при нажатии кассиром кнопки, свидетельствующем о завершении продажи. Вот полное описание этой операции.

### Описание операции ОПЗ: endSale

| | |
|---|---|
| **Операция** | `endSale()` |
| **Ссылки** | Прецеденты: `Оформление продажи` |
| **Предусловия** | Инициирована продажа |
| **Постусловия** | - Атрибут `Sale.isComplete` принял значение `true` (*модификация атрибута*) |

### Выбор класса-контроллера

Для начала необходимо назначить обязанность, связанную с обработкой сообщения системной операции `endSale`. В соответствии с шаблоном Controller, в качестве контроллера будем использовать класс `Register`, как и для системной операции `enterItem`.

### Установка значения атрибута `Sale.isComplete`

Единственным постусловием в описании системной операции `endSale` является следующее.

- Атрибут `Sale.isComplete` принял значение `true` (*модификация атрибута*).

Если задача не сводится к созданию объекта или выбору контроллера (а в данном случае так оно и есть), то первым шаблоном, который необходимо применить для ее решения, является шаблон Expert.

Какой класс должен отвечать за установку атрибута `isComplete` объекта `Sale` равным значению `true`?

Согласно шаблону Expert, эту обязанность следует возложить на класс `Sale`, поскольку он владеет атрибутом `isComplete`. Тогда класс `Register` будет отправлять объекту `Sale` сообщение `becomeComplete`, требующее установки этого атрибута в значение `true`.

### Обозначения UML для ограничений, комментариев и описаний алгоритмов

Заметим, что на рис. 17.9 показано сообщение `becomeComplete` без детализации работы метода `becomeComplete` (в данном случае выполняются тривиальные действия). Иногда в UML для описания алгоритма или метода используются дополнительные текстовые пояснения.

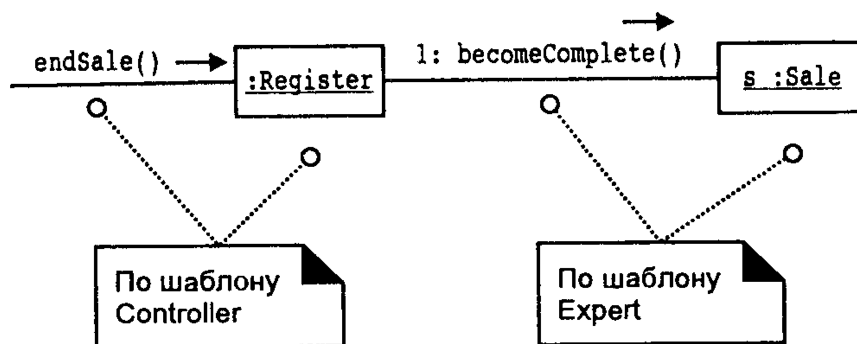



*Рис. 17.9. Завершение продажи*

Для подобных целей в UML предусмотрены обозначения *ограничений* (constraints) и *примечаний* (notes). Ограничение в UML — это некоторая семантически значимая информация, относящаяся к элементу модели. Ограничения в UML представляются в виде текстовых комментариев, заключенных в фигурные скобки, например `{x>20}`. В тексте ограничений можно использовать любой неформальный или формальный язык. Кроме того, для описания ограничений можно применять специально включенный в UML язык OCL (Object Constraint Language) [106].

Примечание в UML — это комментарий, не имеющий семантического значения, например, указывающий дату создания или автора.

Примечания всегда отображаются в *блоках примечаний* (note box) — блоках с "завернутым уголком".

Ограничения можно изображать в виде текстовых фрагментов, заключенных в фигурные скобки. Однако длинные тексты ограничений можно размещать в "блоках ограничений", аналогичных блокам примечаний. Чтобы отличить блок ограничения от блока примечания, его текст помещают в фигурные скобки.

На рис. 17.10 показаны оба типа обозначений, принятых для ограничений. Заметим, что простые ограничения (заключенные в фигурные скобки без помещения в блок) содержат логическое выражение, которое должно выполняться (классическое понятие ограничения из логики). А в блоке ограничений приводится реализация метода на языке Java. В UML для обозначения ограничений можно использовать оба стиля.

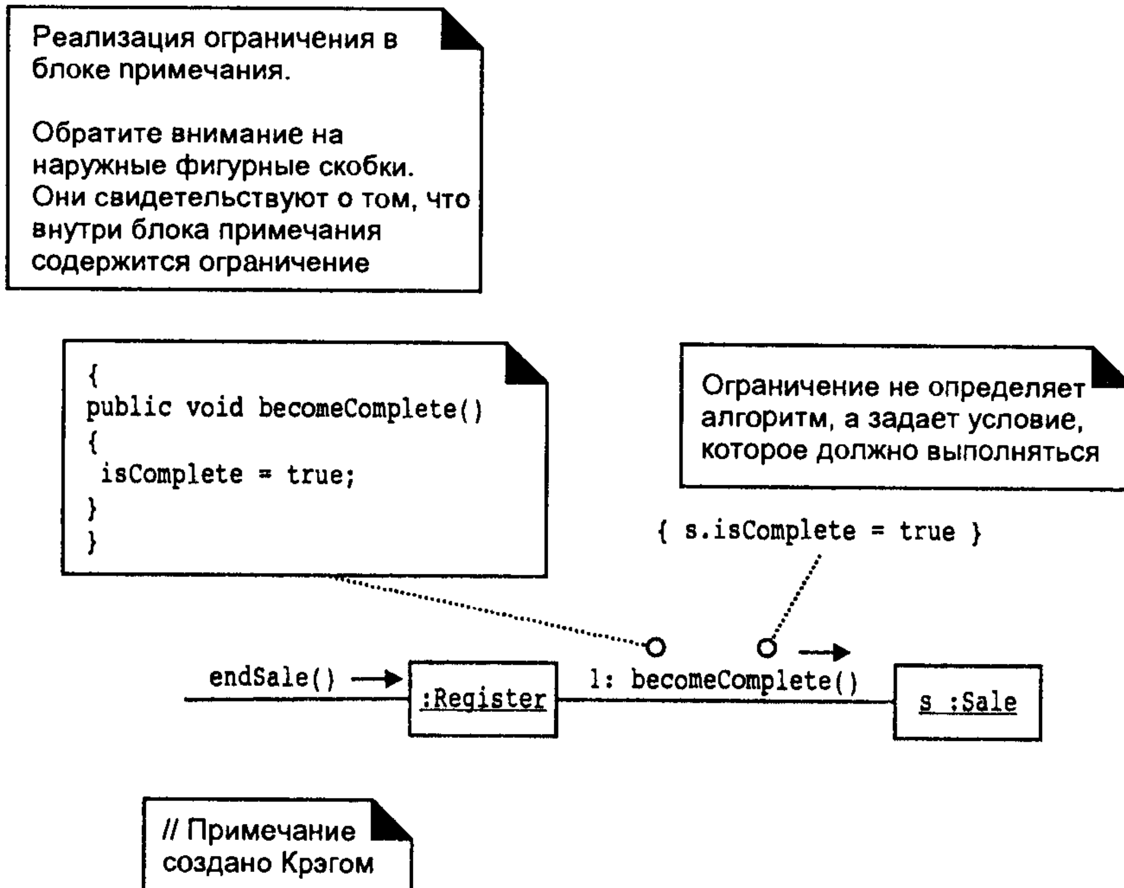



*Рис. 17.10. Ограничения и примечания*

## Вычисление общей стоимости покупки

Рассмотрим фрагмент прецедента Оформление продажи.

**Основной успешный сценарий**

1. Покупатель подходит к кассовому аппарату POS-системы с выбранными товарами.
2. Кассир открывает новую продажу.
3. Кассир вводит идентификатор товара.
4. Система записывает наименование товара и выдает его описание, цену и общую стоимость. Цена вычисляется на основе набора правил.

*Кассир повторяет действия, описанные в пп. 3–4, для каждого наименования товара.*

5. Система вычисляет общую стоимость покупки с налогом.

В соответствии с этим описанием, в п. 5 вычисляется общая стоимость товара. Согласно шаблону Model-View Separation, на данном этапе не следует заботиться о том, как будет отображаться общая стоимость покупки. Необходимо лишь удостовериться, что это значение известно. Заметим, что нельзя указать класс, которому значение известно в данный момент, поэтому необходимо предусмотреть взаимодействие между объектами, обеспечивающее выполнение данного требования, и построить соответствующую диаграмму.

Как обычно, если не ставится задача выбора контроллера и создания новых объектов (а в данном случае это именно так), следует применить шаблон Expert.

Достаточно очевидно, что сам объект Sale должен отвечать за вычисление общей стоимости покупки, однако для чистоты эксперимента проведем следующий анализ.

1. Сформулируем обязанность.
   - Какой класс должен отвечать за вычисление общей стоимости покупки?
2. Подытожим требуемую информацию.
   - Общая стоимость покупки равна сумме стоимостей каждого из покупаемых товаров.
   - Стоимость каждого из покупаемых товаров равна произведению количества единиц покупаемого товара и цены этого товара.
3. Выпишем информацию, необходимую для выполнения данной процедуры, и классы, обладающие этой информацией.

| Информация, необходимая для вычисления общей стоимости покупки | Классы-эксперты |
|---|---|
| ProductSpecification.price | ProductSpecification |
| SalesLineItem.quantity | SalesLineItem |
| Все значения SalesLineItem для текущей продажи Sale | Sale |

Продолжим детальный анализ.

- Какой класс должен отвечать за вычисление общей стоимости продажи Sale? Согласно шаблону Expert, это должен быть класс Sale, поскольку он обладает информацией обо всех экземплярах SalesLineItem, стоимость которых суммируется при вычислении общей стоимости. Следовательно, класс Sale будет отвечать за вычисление общей стоимости, и эта обязанность будет реализована в виде метода getTotal.
- Для вычисления общей стоимости покупки Sale необходимо знать стоимость каждого покупаемого товара SalesLineItem. Какой класс должен отвечать за вычисление стоимости SalesLineItem? Согласно шаблону Expert, это должен быть класс SalesLineItem, поскольку ему известно количество покупаемых единиц товара и спецификация этого товара ProductSpecification. Следовательно, обязанность вычисления стоимости каждого товара, реализованная в виде метода getSubtotal, должна возлагаться на класс SalesLineItem.
- Для вычисления стоимости каждого товара SalesLineItem необходимо знать цену товара со спецификацией ProductSpecification. Кто должен отвечать за предоставление цены товара со спецификацией ProductSpecification? Согласно шаблону Expert, это должен быть класс ProductSpecification, по-

скольку он инкапсулирует цену товара в качестве своего атрибута. Следовательно, эта обязанность, реализованная в виде операции getPrice, возлагается на класс ProductSpecification.

> Хотя в данном конкретном случае проведенный анализ выглядит тривиальным, а степень детализации избыточной, такую же стратегию следует применять и в более сложных ситуациях. Освоив эти принципы, вы сможете легко распределить обязанности.

### *Проектное решение Sale--getTotal*

Основываясь на приведенных рассуждениях, можно построить диаграмму взаимодействия, иллюстрирующую процесс передачи сообщения getTotal объекту Sale. Первым на этой диаграмме является сообщение getTotal, однако это не системное событие.

Отсюда следует вывод.

> Не каждая диаграмма взаимодействия начинается с сообщения о системном событии. Она может начинаться с любого сообщения, для которого необходимо показать виды взаимодействия между объектами.

Диаграмма взаимодействия представлена на рис. 17.11. Сначала объекту Sale передается сообщение getTotal. Затем объект Sale отправляет сообщение getSubtotal каждому связанному с ним экземпляру SalesLineItem. В свою очередь, каждый экземпляр SalesLineItem отправляет сообщение getPrice связанному с ним объекту ProductSpecification.

Поскольку арифметические действия обычно не изображаются с помощью сообщений, их можно проиллюстрировать путем добавления на диаграмму дополнительной информации в виде ограничений или описаний алгоритма.

Какой класс должен передавать сообщение getTotal объекту Sale? Скорее всего, это должен быть объект уровня представления, например объект JFrame Java.

Рис. 17.12 иллюстрирует использование примечаний и ограничений при описании операций getTotal и getSubtotal.

## 17.7. Проектное решение: makePayment

Системная операция makePayment выполняется при вводе кассиром внесенной за покупку суммы. Вот полное описание этой операции.

#### Описание операции ОП4: makePayment

| | |
|---|---|
| **Операция** | makePayment(amount: Money) |
| **Ссылки** | Прецеденты: Оформление продажи |
| **Предусловия** | Инициирована продажа |
| **Постусловия** | - Создан экземпляр p объекта Payment (*создание экземпляра*)<br>- Атрибут p.amountTendered принял значение amount (*модификация атрибута*)<br>- Экземпляр p связан с текущим экземпляром класса Sale (*формирование ассоциации*)<br>- Текущий экземпляр Sale связан с экземпляром класса Store для его добавления в журнал регистрации продаж (*формирование ассоциации*) |

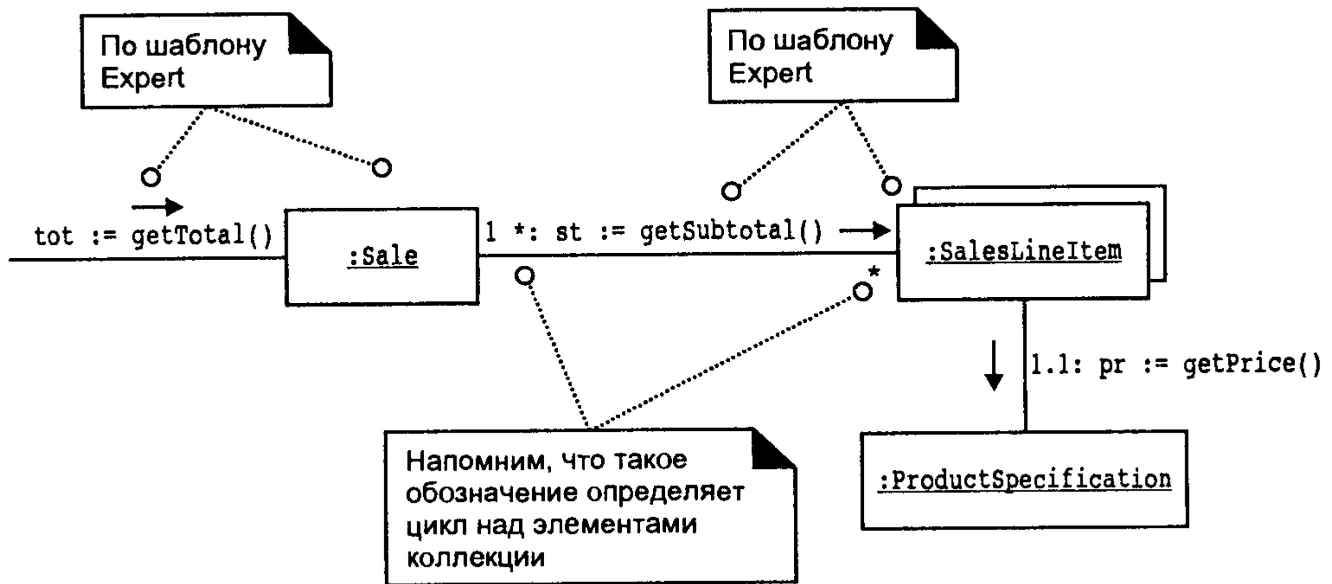



*Рис. 17.11. Диаграмма взаимодействия для метода* `Sale--getTotal`

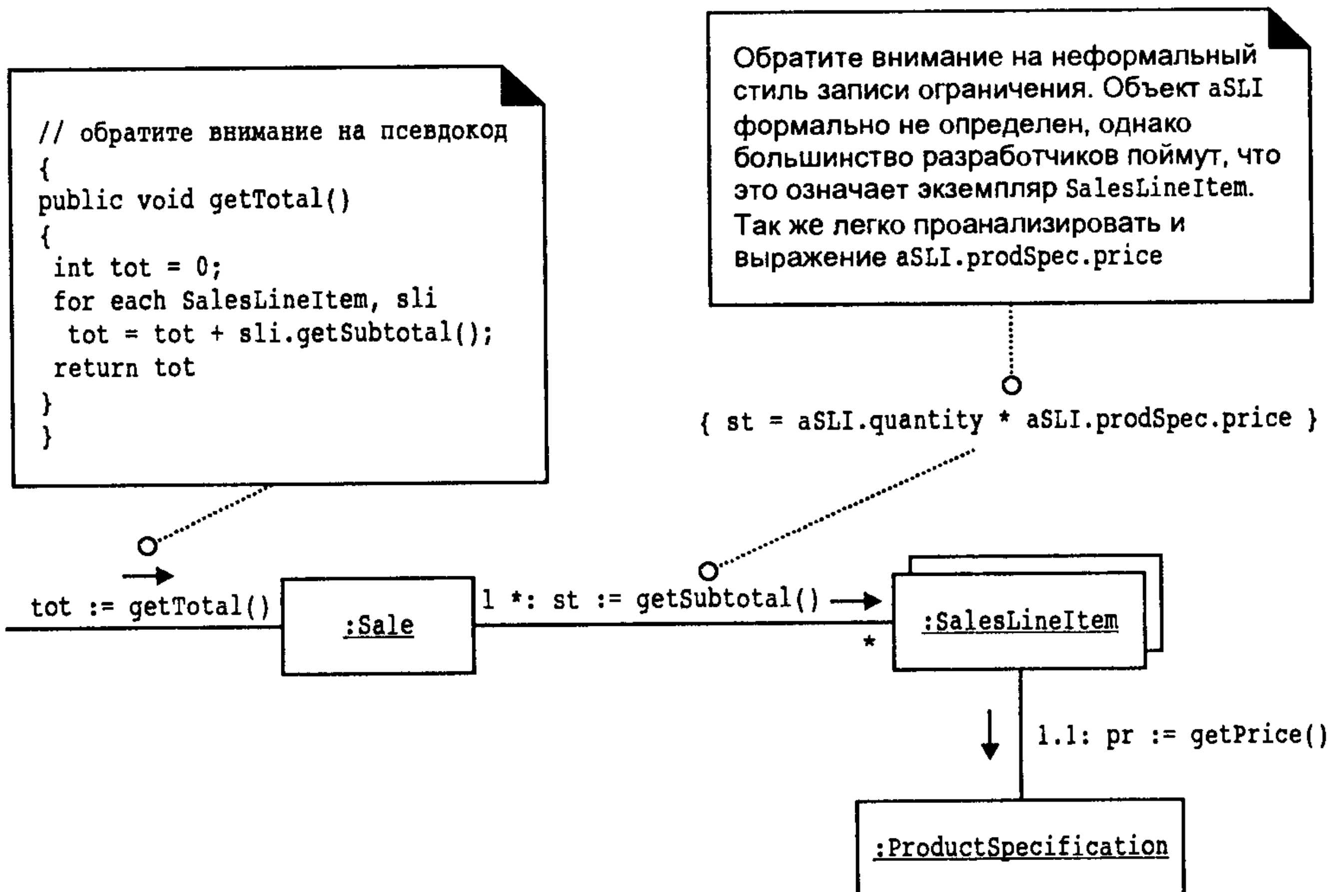



*Рис. 17.12. Использование примечаний и ограничений*

Проектное решение должно обеспечить выполнение постусловий системной операции `makePayment`.

## Создание экземпляра Payment

Одним из постусловий описания операции является следующее.

- Создан экземпляр p объекта `Payment` (*создание экземпляра*).

При распределении обязанностей, связанных с созданием новых экземпляров, необходимо применять шаблон Creator.

Какой класс записывает, агрегирует, наиболее активно использует или содержит объекты Payment? Одним из кандидатов на выполнение этой обязанности может быть класс Register, поскольку он по логике записывает сведения о платежах. При таком проектном решении сокращается разрыв между терминологией предметной области и именами программных классов. Еще одним кандидатом может выступать класс Sale, поскольку он наиболее активно использует информацию объекта Payment.

При определении класса, который должен создавать новые экземпляры, можно также воспользоваться шаблоном Expert. При этом необходимо ответить на вопрос: какой из классов является экспертом в смысле инициализации данных (в данном случа се — внесенной суммы)? Класс Register, выступающий в роли контроллера, получает сообщение о системной операции makePayment, поэтому он изначально обладает информацией о внесенной сумме. Следовательно, класс Register снова является кандидатом на выполнение этой обязанности.

Таким образом, имеем двух кандидатов.

- Register
- Sale

Это приводит к необходимости применения главной идеи проектирования.

> При наличии двух альтернативных вариантов проектирования их следует рассматривать с точки зрения связывания и зацепления, а также, по возможности, будущих изменений. Выбор целесообразно остановить на варианте с хорошими показателями в области связывания и зацепления, обладающем высокой устойчивостью к возможным изменениям в будущем.

Рассмотрим каждый из этих вариантов "с точки зрения" шаблонов High Cohesion и Low Coupling. Если экземпляр объекта Payment создается классом Sale, то работа (обязанности) класса Register облегчается. Кроме того, классу Register не нужно знать о существовании экземпляра Payment, поскольку он может записывать его опосредованно через объект Sale. Это обеспечит низкий уровень связывания объекта Register. Полученная диаграмма представлена на рис. 17.13.

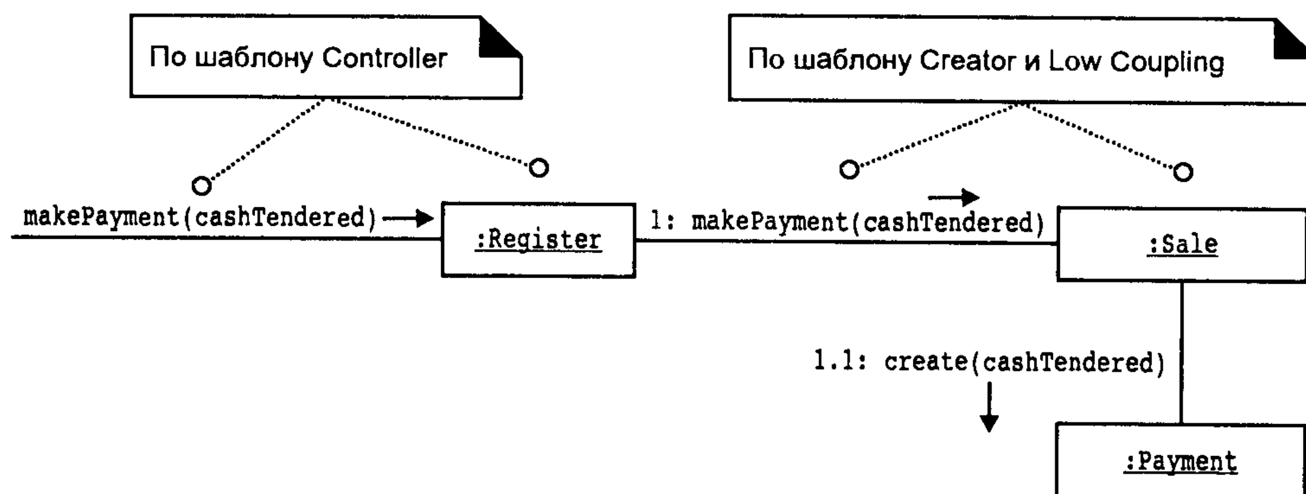



*Рис. 17.13. Диаграмма взаимодействия для операции Register--makePayment*

Эта диаграмма взаимодействия удовлетворяет постусловиям, приведенным в описании системной операции: создан экземпляр объекта Payment, связанный с объектом Sale, и для данного платежа установлено значение amountTendered.

## Регистрация покупки

Сведения о приобретенных покупках должны заноситься в журнал регистрации. Как обычно, если задача не сводится к выбору контроллера или созданию новых объектов (а в данном случае это именно так), при распределении обязанностей необходимо воспользоваться шаблоном Expert и ответить на следующий вопрос.

Какой из классов обладает всей информацией о продажах и может выполнять регистрацию?

С точки зрения сокращения разрыва между терминологией предметной области и именами программных классов лучше всего, чтобы всей необходимой информацией обладал объект `Store`, поскольку он связан с финансовой деятельностью предприятия. В качестве второго варианта можно выбрать класс, реализующий традиционное бухгалтерское понятие, например гроссбух `SalesLedger`. Однако вводить дополнительные объекты имеет смысл только при достаточном разрастании системы, когда класс `Store` перестанет обладать свойством зацепления (рис. 17.14).

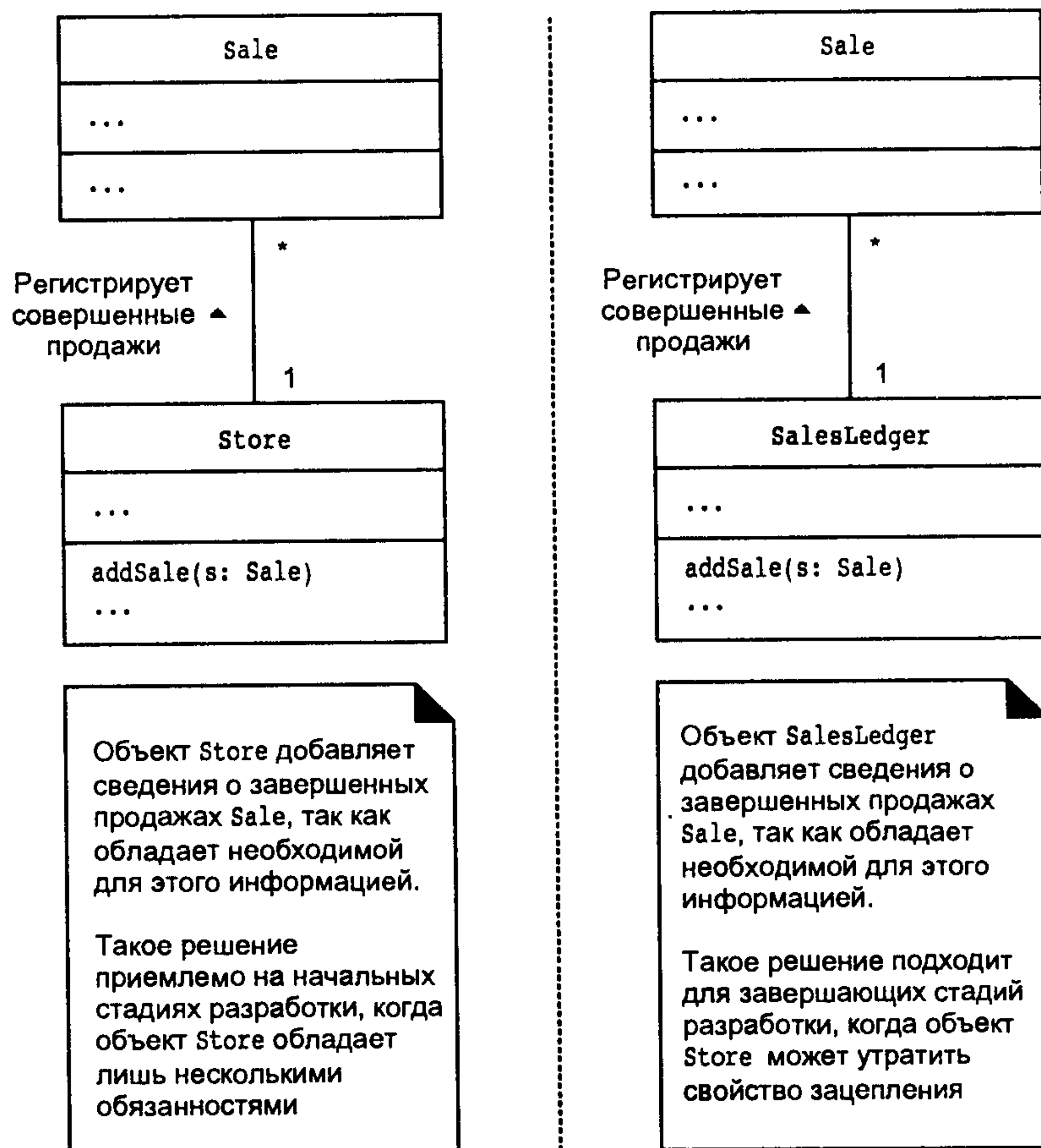



*Рис. 17.14. Какой класс должен отвечать за информацию о совершенных покупках?*

Заметим, что среди постусловий в описании системной операции упоминается связь объектов Sale и Store. Это пример ситуации, когда постусловие не обязательно нужно реализовывать в проекте. На ранних этапах разработки речь не шла об объекте SalesLedger, однако сейчас стало ясно, что именно его можно использовать вместо Store. После принятия такого решения этот объект нужно будет также добавить в модель предметной области, поскольку его имя соответствует понятию реального мира. Это пример дополнительных исследований и изменений модели предметной области в процессе проектирования, предполагаемый в рамках итеративного подхода к разработке системы.

В данном случае мы будем придерживаться исходного плана и использовать объект Store (рис. 17.15).

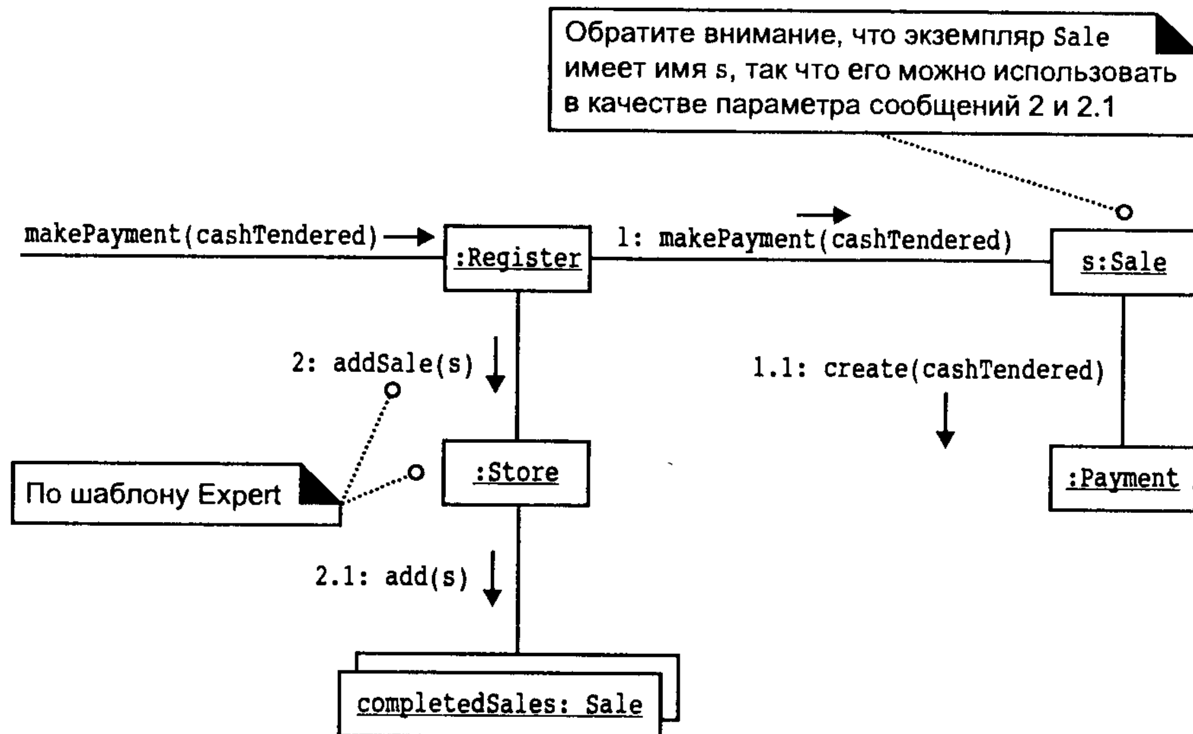



*Рис. 17.15. Регистрация совершенной покупки*

## Вычисление причитающейся сдачи

Теперь необходимо реализовать процесс вычисления причитающейся сдачи, сумма которой должна отображаться на экране и печататься на чеке. Эта операция выполняется в рамках прецедента Оформление продажи.

Согласно основному принципу шаблона Model-View Separation, сейчас не стоит акцентировать внимание на способе отображения суммы сдачи, однако необходимо убедиться, что эта сумма известна. Заметим, что в настоящее время ни один из классов не обладает такой информацией, поэтому необходимо разработать диаграмму взаимодействия, которая будет обеспечивать выполнение данного требования.

Как обычно, если задача не сводится к выбору контроллера или к созданию новых объектов (а в данном случае это именно так), при распределении обязанностей необходимо воспользоваться шаблоном Information Expert и ответить на следующий вопрос.

**Какой из классов отвечает за вычисление суммы сдачи (баланса торговой операции)?**

Для вычисления баланса необходимо знать общую стоимость покупки и внесенную покупателем сумму. Следовательно, при решении этой проблемы в качестве частичных экспертов должны выступать классы `Sale` и `Payment`.

Если основная обязанность по вычислению баланса возлагается на класс `Payment`, то для него следует обеспечить видимость класса `Sale`, с помощью которого необходимо получить общую стоимость покупки. Такой подход повышает уровень связывания классов в проекте и не соответствует основному принципу шаблона Low Coupling.

Если же основная обязанность по вычислению баланса возлагается на класс `Sale`, то для него следует обеспечить видимость класса `Payment`, от которого необходимо получить внесенную покупателем сумму. Поскольку класс `Sale` является создателем экземпляров `Payment`, такая видимость уже обеспечена, и данный подход не повышает уровень связывания классов в проекте, что соответствует основному принципу шаблона Low Coupling и, следовательно, является более предпочтительным.

Таким образом, диаграмма взаимодействия приобретает следующий вид (рис. 17.16).

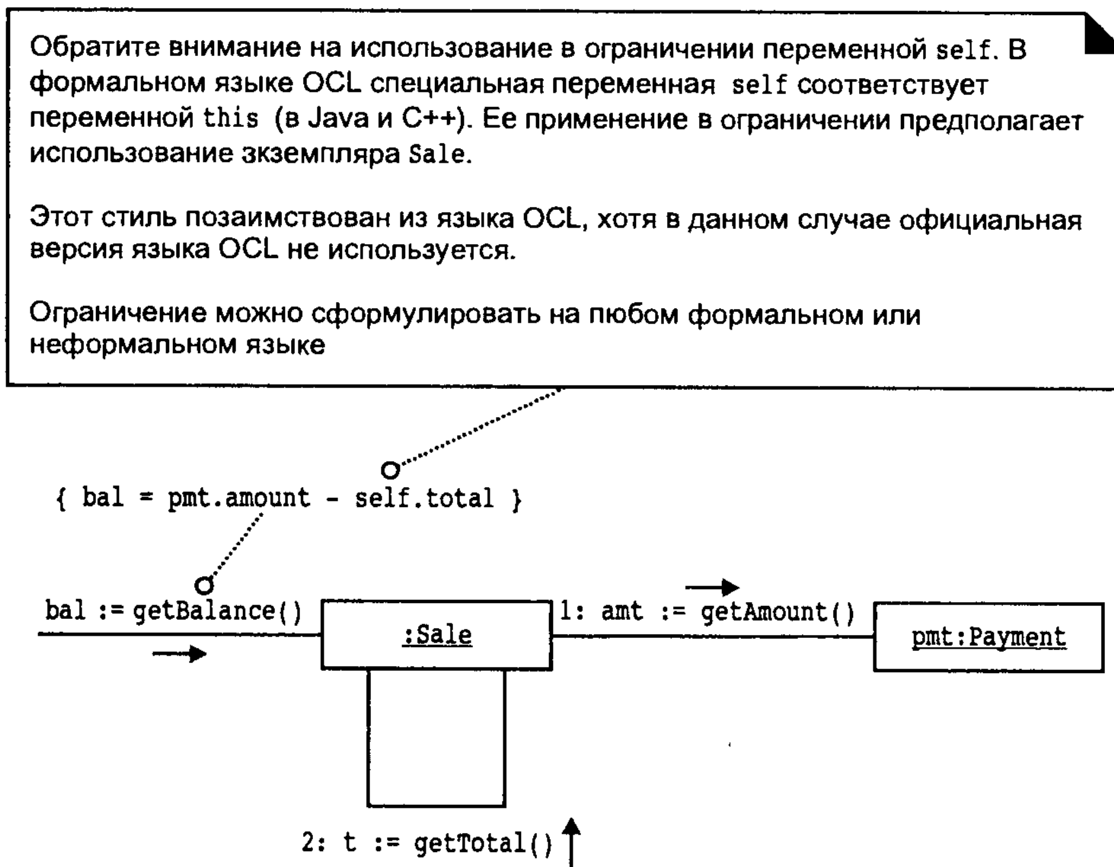



*Рис. 17.16. Диаграмма взаимодействия для операции `Sale--getBalance`*

## 17.8. Проектное решение: `startUp`

### *Когда создавать диаграмму взаимодействия для системной операции `startUp`*

Практически все системы включают прецедент `Запуск системы` (Start Up) и некоторую исходную системную операцию, связанную с запуском приложения.

Хотя системная операция startUp выполняется одной из первых, ее диаграмму взаимодействия разрабатывают после рассмотрения всех остальных системных операций. При таком подходе гарантируется, что к моменту разработки системной операции startUp известна вся существенная информация, связанная с инициализацией системы и необходимая для поддержки диаграмм взаимодействия последующих системных операций.

| Диаграмма взаимодействия для системной операции startUp разрабатывается последней. |
|---|

### *Как запускаются приложения*

Операция startUp — это абстрактное представление этапа инициализации при запуске приложения. Чтобы разобраться, как разрабатывать диаграмму взаимодействия для такой операции, необходимо понять основные принципы запуска приложения. Способ запуска приложения и инициализации данных зависит от операционной системы и выбранного языка программирования.

В любом случае, как правило, разрабатывается *исходный объект предметной области* (initial domain object), который создается первым среди других объектов предметной области.

Обратите внимание на терминологию. Как будет видно дальше, приложение делится на логические уровни, выполняющие основные функции приложения. К ним относятся уровень пользовательского интерфейса (для общения пользователя с системой) и уровень предметной области (для реализации логики предметной области приложения). В модели проектирования к уровню предметной области относятся программные классы, имена которых соответствуют понятиям предметной области. Все рассмотренные до сих пор объекты, такие как Sale и Register, относятся к уровню предметной области в модели проектирования.

После создания исходного объекта предметной области ему в обязанность вменяется создание дочерних объектов предметной области. Например, если в качестве исходного объекта предметной области выбран объект Store, то он может отвечать за создание объекта Register.

Способ создания исходного специального объекта зависит от выбранной объектной технологии. Например, в приложении Java для его создания можно использовать метод main или делегировать обязанность создания этого объекта, в соответствии с шаблоном Factory, объекту-фабрике.

```
public class Main
{
public static void main(String[] args)
{
   // Store - исходный объект предметной области.
   //Он отвечает за создание
   //других объектов предметной области.

   Store store = new Store();

   Register register = store.getRegister();
```

```
    ProcessSaleJFrame frame = new ProcessSaleJFrame(register);
    ...
}
}
```

### *Интерпретация системной операции `startUp`*

Из приведенных рассуждений следует, что системная операция `startUp` — это зависимая от языка программирования абстракция. На этапе проектирования определяется, где создается исходный объект и нужно ли ему передавать управление процессом. Обычно при наличии графического интерфейса пользователя исходный объект предметной области не получает управления процессом. Ему передается управление лишь при отсутствии интерфейса пользователя.

Диаграмма взаимодействия для операции `startUp` представляет события, происходящие после создания исходного объекта предметной области, а также (не обязательно) события, происходящие при передаче управления этому объекту. Она не включает никаких предварительных или последующих видов деятельности, связанных с объектами уровня графического интерфейса пользователя, если таковой существует.

> Итак, операцию `startUp` можно интерпретировать следующим образом.
>
> 1. На одной диаграмме взаимодействия отображается передача сообщения `create()` для создания исходного объекта предметной области.
> 2. (Дополнительно.) Если исходному объекту передается управление процессом, то на второй диаграмме взаимодействия отображается передача сообщения `run()` (или эквивалентного ему сообщения) исходному объекту.

### *Операция `startUp` POS-приложения*

Системная операция `startUp` выполняется при включении менеджером питания POS-системы и загрузке программной части системы. Предположим, исходному объекту предметной области управление *не* передается. Оно передается на уровень графического интерфейса пользователя (например, `JFrame` Java) после создания исходного объекта предметной области. Тогда диаграмма взаимодействия для системной операции `startUp` должна содержать лишь передачу сообщения `create()` для создания исходного объекта.

### *Выбор исходного объекта предметной области*

Какой класс следует выбрать в качестве исходного объекта предметной области?

> В качестве исходного объекта предметной области выбирается класс, максимально приближенный к корню иерархии агрегации объектов предметной области. В качестве такого объекта можно выбрать внешний контроллер, такой как `Register`, или любой другой объект, содержащий все или большую часть других объектов предметной области, например `Store`.

Выбор между этими объектами осуществляется на основе шаблонов High Cohesion и Low Coupling. В рассматриваемом приложении выберем в качестве исходного объекта класс Store.

### Объекты из базы данных: ProductSpecification

Экземпляры ProductSpecification должны храниться в постоянной памяти, например в реляционной или объектной базе данных. В процессе операции startUp они могут загружаться в оперативную память компьютера (если их не очень много). Однако если таких объектов достаточно много, то их загрузка в оперативную память потребует слишком большого объема памяти и длительного времени. Поэтому лучше при необходимости загружать в память отдельные экземпляры этих объектов.

Процесс динамической загрузки объектов из базы данных в оперативную память упрощается при использовании объектной базы данных и затрудняется при использовании реляционной базы данных. Однако на данном этапе мы не будем рассматривать эту проблему, а предположим, что экземпляры ProductSpecification каким-то чудом помещаются в память объектом ProductCatalog.

Вопросы работы с объектами из базы данных и извлечения их в оперативную память рассматриваются в главе 34.

### Проектное решение: Store--create()

Задачи создания и инициализации возникают в процессе проектирования приложения, в частности, при разработке проектного решения для операции enterItem. Из рассмотренных выше диаграмм взаимодействия следует необходимость инициализации для следующих объектов.

- Необходимо создать экземпляры объектов Store, Register, ProductCatalog и ProductSpecification.
- Объект ProductCatalog необходимо связать с объектом ProductSpecification.
- Объект Store необходимо связать с объектом ProductCatalog.
- Объект Store необходимо связать с объектом Register.
- Объект Register необходимо связать с объектом ProductCatalog.

На рис. 17.17 показано соответствующее проектное решение. Для создания объектов ProductCatalog и Register, согласно шаблону Creator, был выбран объект Store. В процессе аналогичных рассуждений для создания объектов ProductSpecification был выбран объект ProductCatalog. Напомним, что такой подход к созданию спецификации можно рассматривать лишь как временный. При окончательном проектировании эти объекты при необходимости будут материализовываться из базы данных.

*Обозначение UML*: обратите внимание, что создание всех экземпляров ProductSpecification и их добавление в контейнер обозначено символом *, следующим за порядковым номером операции.

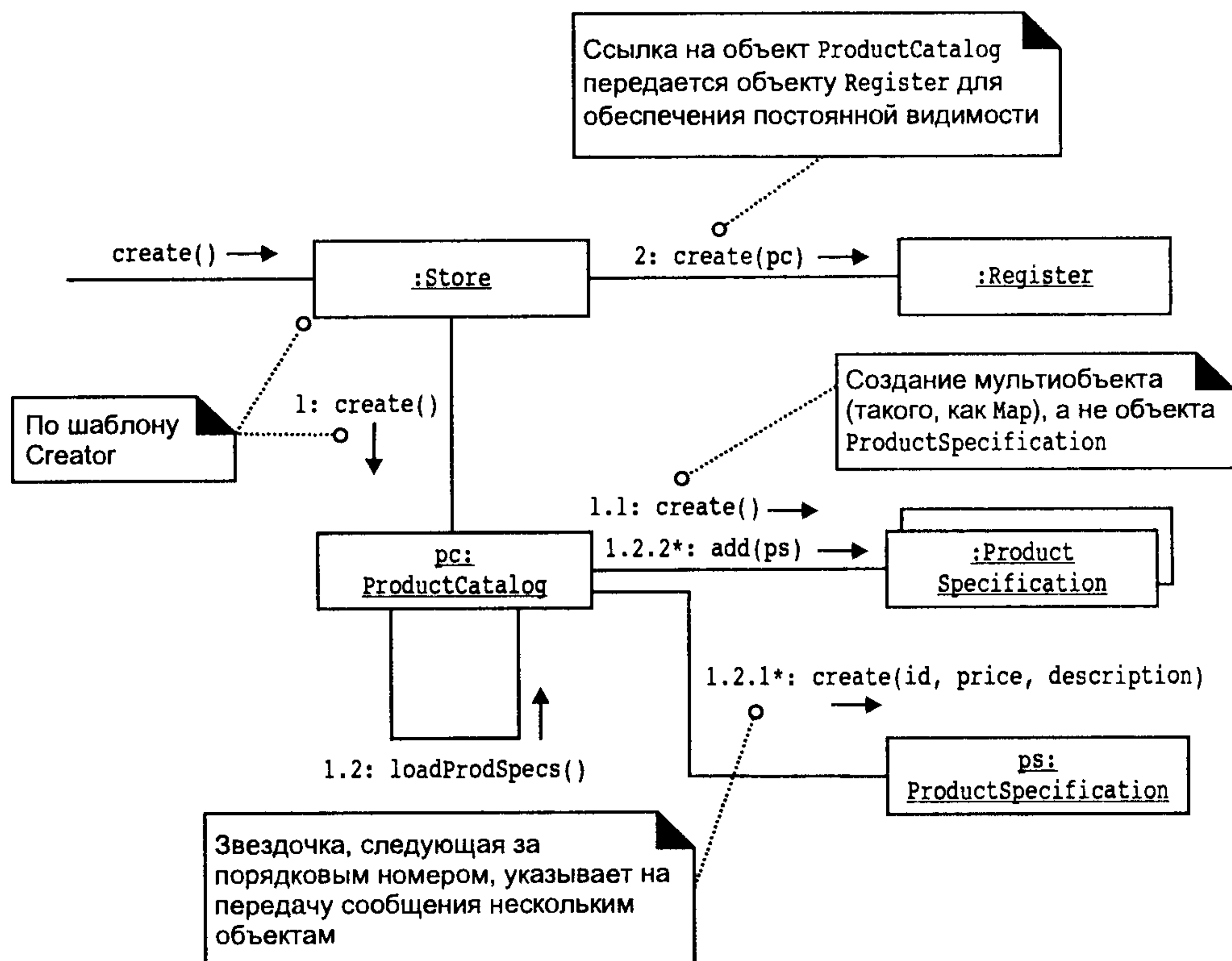



*Рис. 17.17. Создание исходного объекта предметной области и последующих объектов*

Интересное различие между этапами анализа и проектирования состоит в том, что объект `Store` создает лишь *один* объект `Register`. В реальной жизни магазин может содержать множество терминалов POS-системы. Однако мы рассматриваем программный продукт. Поэтому согласно требованиям к системе, программный объект `Store` должен создать единственный экземпляр программного объекта `Register`.

| Количество классов и объектов в моделях предметной области и проектирования может различаться. |
|---|

## 17.9. Подключение уровня интерфейса пользователя к уровню предметной области

Как отмечалось выше, приложения делятся на логические уровни, представляющие различные аспекты функционирования приложения, в том числе уровень графического интерфейса пользователя и уровень объектов предметной области (для реализации логики предметной области).

Основные проектные решения, обеспечивающие видимость объектов уровня предметной области для объектов уровня пользовательского интерфейса, сводятся к следующему.

- Инициализирующая программа (например, метод main Java) создает и объекты уровня пользовательского интерфейса, и объект предметной области, а затем передает этот объект на уровень интерфейса пользователя.
- Объект пользовательского интерфейса получает объект предметной области из стандартного источника, например, от объекта-фабрики, отвечающего за создание объектов предметной области.

Следующий фрагмент кода демонстрирует пример использования первого подхода.

```
public class Main
{
public static void main(String[] args)
{
   Store store = new Store();
   Register register = store.getRegister();
   ProcessSaleJFrame frame = new ProcessSaleJFrame(register);
   ...
}
}
```

Будучи связанным с экземпляром объекта Register (выступающим в данном приложении в роли внешнего контроллера), объект уровня пользовательского интерфейса может направить ему сообщение о системном событии, такое как enterItem или endSale (рис. 17.18).

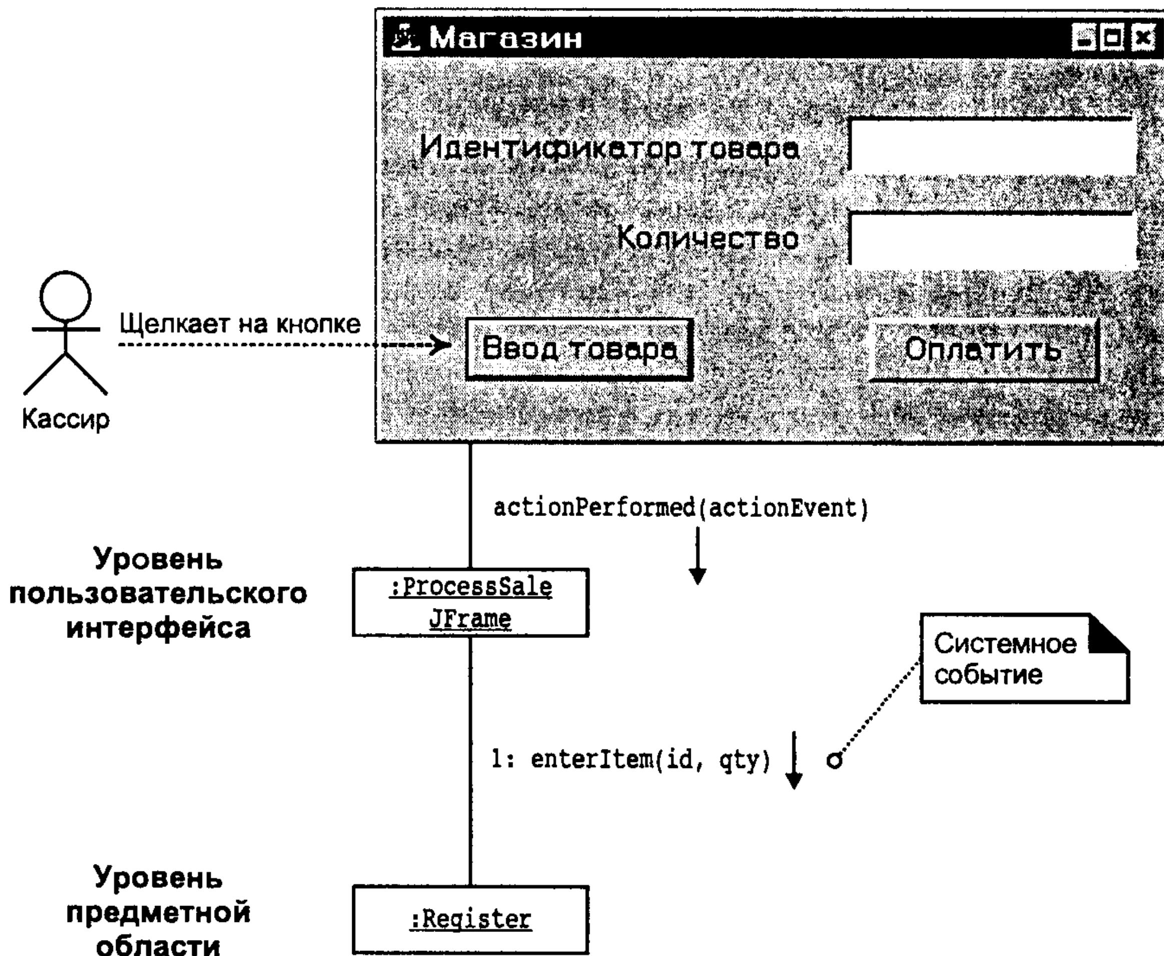



*Рис. 17.18. Связь уровней предметной области и пользовательского интерфейса*

Для передачи сообщения enterItem необходимо иметь диалоговое окно, в котором будет отображаться общая стоимость каждого товара. При этом можно использовать несколько проектных решений.

- Добавить объекту Register метод getTotal. Тогда с уровня пользовательского интерфейса объекту Register будет передаваться сообщение getTotal, а этот объект, в свою очередь, перенаправит это сообщение объекту Sale. Преимуществом такого подхода является низкая степень связывания между уровнями пользовательского интерфейса и предметной области — пользовательскому интерфейсу известен лишь один объект Register. Однако при этом "раздувается" интерфейс объекта Register, что приводит к слабой степени зацепления этого объекта.
- Интерфейс пользователя запрашивает ссылку на текущий объект Sale, а затем при необходимости вычисления общей стоимости (или для получения любой другой информации о продаже) он передает сообщение напрямую объекту Sale. Такое проектное решение повышает степень связывания между уровнями пользовательского интерфейса и предметной области. Однако, как указывалось при рассмотрении шаблона GRASP Low Coupling, высокая степень связывания сама по себе не является проблемой. Проблема возникает при связывании с неустойчивыми объектами. Объект Sale можно считать устойчивым, поскольку он является неотъемлемой частью проектного решения. Следовательно, связывание с этим объектом не составляет проблемы.

Как видно из рис. 17.19, в проектном решении для данного приложения выбран второй подход.

Заметим, что, согласно этим диаграммам, окно Java (ProcessSaleJFrame), составляющее часть уровня пользовательского интерфейса, не отвечает за реализацию логики приложения. Оно лишь направляет запросы для выполнения системных операций объектам уровня предметной области через объект Register. Отсюда следует один из важных принципов проектирования.

*Обязанности объектов уровней пользовательского интерфейса и предметной области*

Уровень пользовательского интерфейса не должен отвечать за реализацию логики приложения. Его обязанностью является лишь выполнение задач пользовательского интерфейса, например, обновление информации в диалоговых окнах.

Объекты уровня пользовательского интерфейса должны направлять запросы на выполнение всех задач предметной области на уровень объектов предметной области, который и отвечает за их выполнение.

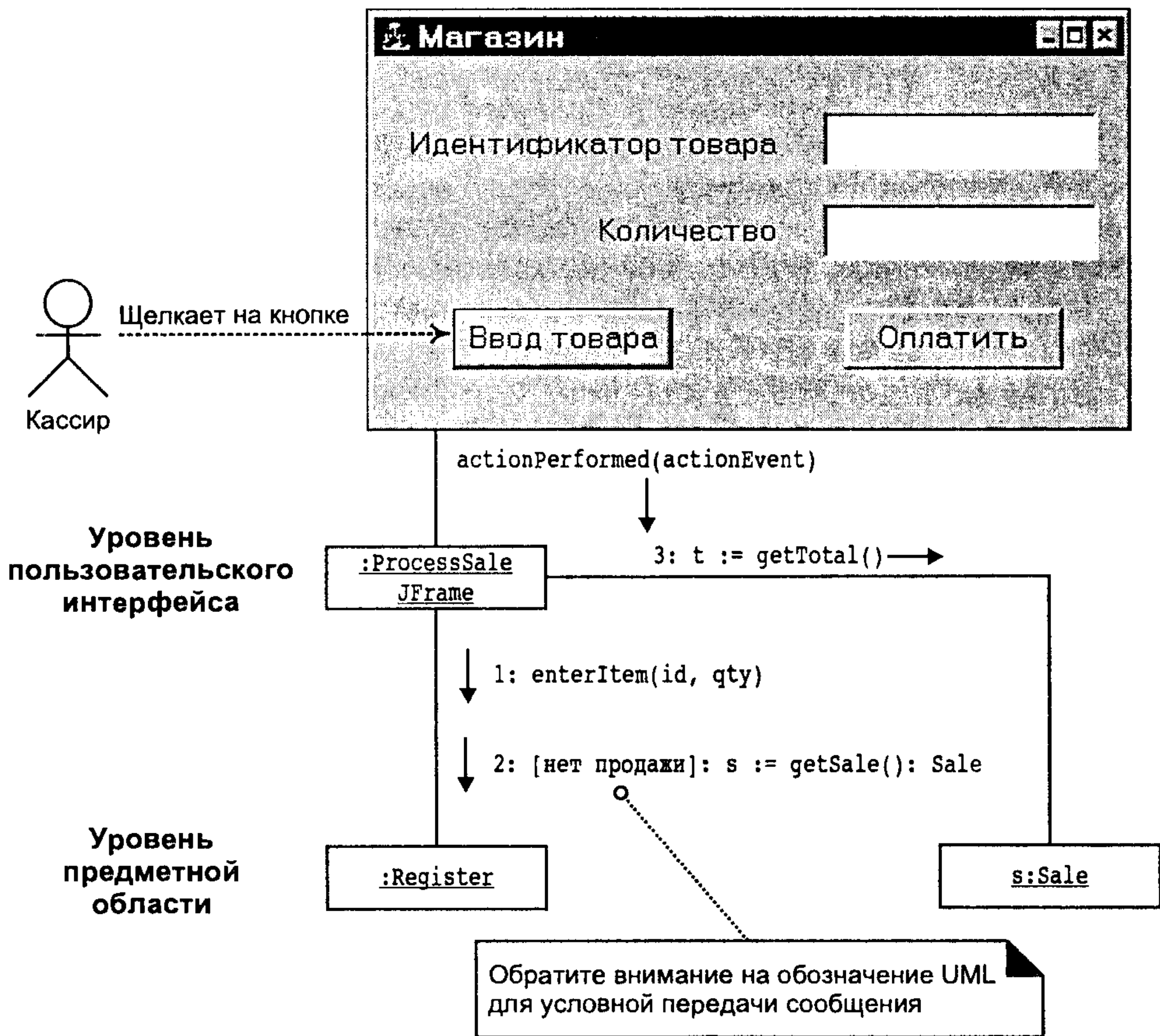



*Рис. 17.19. Взаимодействие уровней пользовательского интерфейса и предметной области*

## 17.10. Реализация прецедентов в рамках UP

Реализация прецедентов — составная часть процесса создания модели проектирования в рамках UP (табл. 17.1). В этой главе основное внимание сосредоточено на построении диаграмм взаимодействия, однако зачастую рекомендуется также параллельно строить и диаграммы классов, которые будут обсуждаться в главе 19.

**Таблица 17.1. Пример планирования сроков реализации артефактов UP (н — начало, р — развитие)**

| Дисциплина | Артефакт<br>Итерация→ | Начало<br>I1 | Развитие<br>E1..En | Конструирование<br>C1..Cn | Передача<br>T1..Tn |
|---|---|---|---|---|---|
| Бизнес-моделирование | Модель предметной области | | н | | |
| Требования | Модель прецедентов | н | р | | |
| | Видение системы | н | р | | |
| | Дополнительная спецификация | н | р | | |

Окончание табл. 17.1

| Дисциплина | Артефакт<br>Итерация→ | Начало<br>I1 | Развитие<br>E1..En | Конструирование<br>C1..Cn | Передача<br>T1..Tn |
|---|---|---|---|---|---|
| | Словарь терминов | н | р | | |
| Проектировани е | **Модель проектирования** | | н | р | |
| | Описание архитектуры | | н | | |
| | Модель данных | | н | р | |
| Реализация | Модель реализации | | н | р | р |
| Управление проектом | План разработки | н | р | р | р |
| Тестирование | Модель тестирования | | н | р | |
| Окружение | Набор документов | н | р | | |

## *Фазы*

**Начало** — разработка модели проектирования и реализация прецедентов обычно начинаются на стадии развития, поскольку эти виды деятельности требуют принятия подробных проектных решений, для которых на начальной стадии отсутствует необходимая информация.

**Развитие** — в течение этой фазы обеспечивается реализация прецедентов для большинства архитектурно значимых или рискованных сценариев. Однако диаграммы UML строятся не для всех сценариев и необязательно прорабатываются в деталях. Главное построить диаграммы прецедентов для основных сценариев, сконцентрировав внимание на важнейших проектных решениях.

**Конструирование** — выполняется реализация остальных прецедентов.

## *Артефакты UP в контексте унифицированного процесса*

Взаимосвязи между артефактами UP показаны на рис. 17.20.

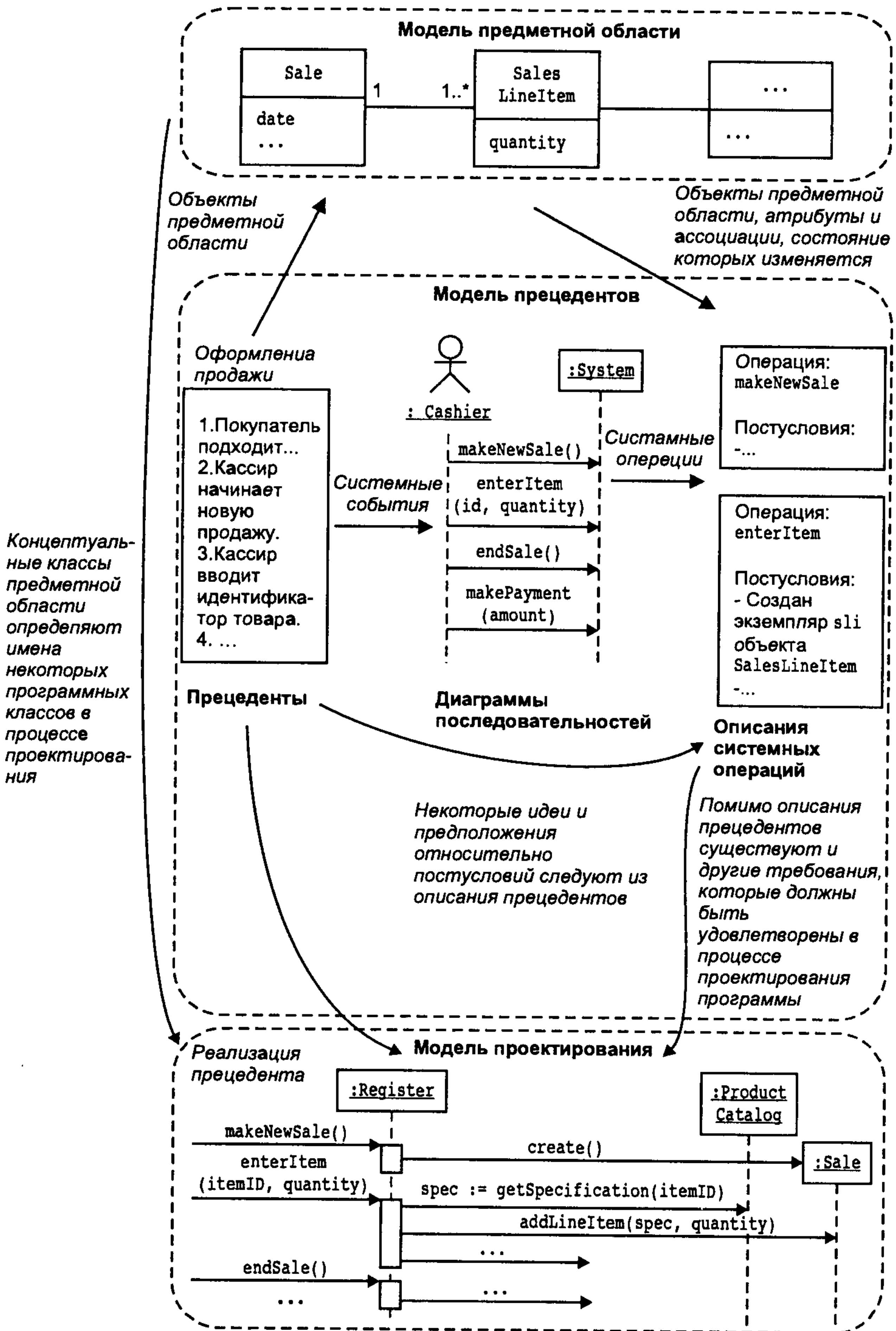



*Рис. 17.20. Пример взаимосвязи артефактов UP*

В контексте UP реализация прецедентов относится к проектированию системы. На рис. 17.21 приводятся рекомендации по организации этой деятельности.

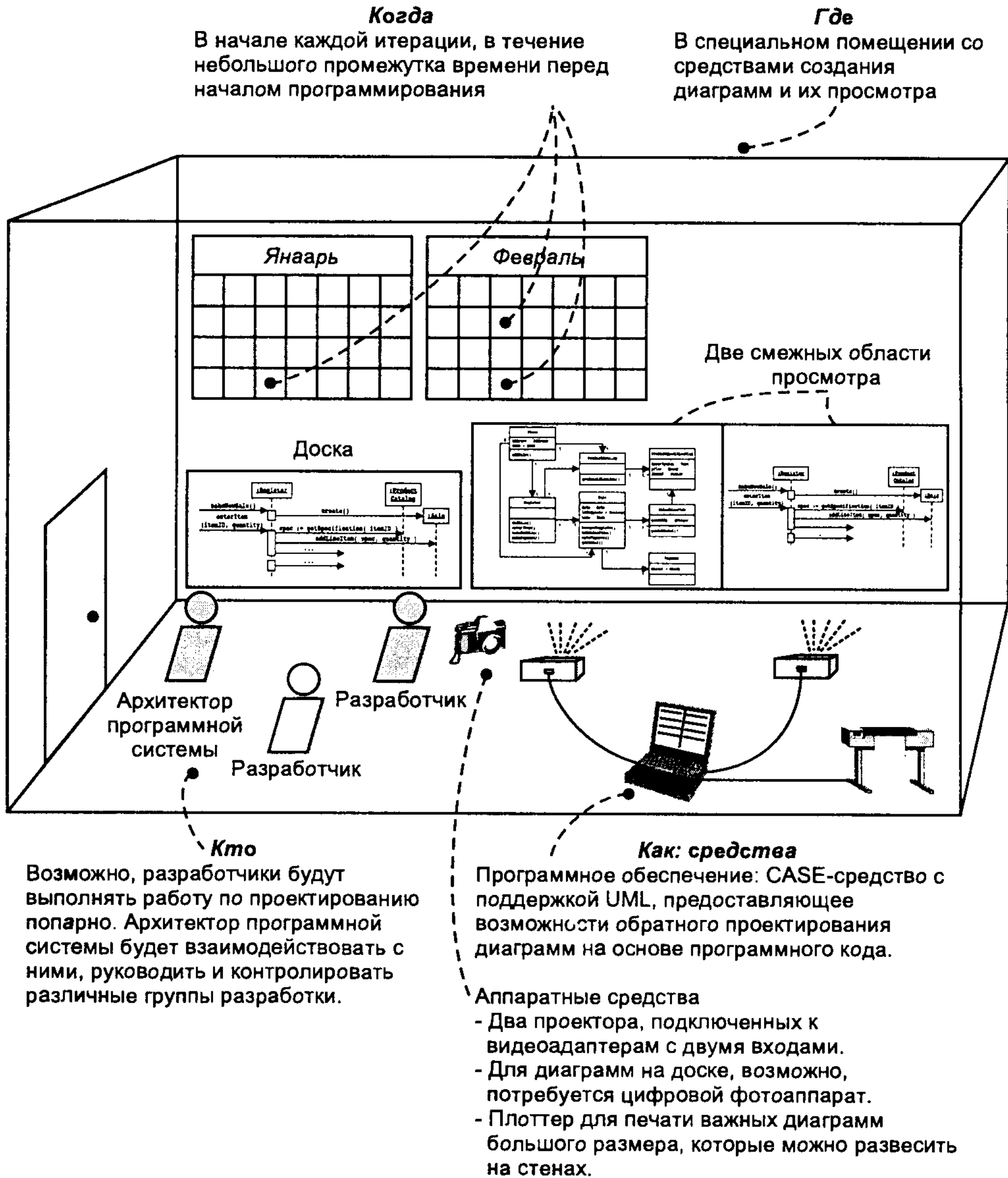



*Рис. 17.21. Рекомендации по организации деятельности в рамках реализации прецедентов*

## 17.11. Резюме

Разработка способов взаимодействия между объектами для обработки сообщений и распределение обязанностей — основные процессы объектно-ориентированного проектирования. Решение этих задач оказывает существенное влияние на расширяемость, прозрачность объектно-ориентированной системы, а также на качество, возможность поддержки и повторного использования разрабатываемых компонентов. Существуют строгие принципы распределения обязанностей. Наиболее важные и типичные из них сформулированы в виде шаблонов GRASP.

Глава

# Модель проектирования: области видимости

*Математик — это устройство для преобразования кофе в теоремы.*

*Поль Эрдос (Paul Erdos)*

Основные задачи

- Ознакомиться с четырьмя типами видимости.
- Установить области видимости.
- Проиллюстрировать области видимости в системе обозначений языка UML.

## Введение

Области видимости обеспечивают способность одного объекта видеть другой объект или ссылаться на него. В этой главе рассматриваются вопросы, связанные с обеспечением видимости объектов в процессе проектирования.

## 18.1. Видимость объектов

Диаграммы взаимодействия для системных событий (`enterItem` и т.д.) иллюстрируют процесс передачи сообщений между объектами. Но чтобы некоторый объект-отправитель мог отослать сообщение объекту-получателю, получатель должен быть виден отправителю, т.е. отправитель должен содержать некую ссылку или указатель на объект-получатель.

Например, для отправки сообщения `getSpecification` от объекта `Register` к объекту `ProductCatalog` необходимо обеспечить видимость экземпляра объекта `ProductCatalog` для экземпляра объекта `Register` (рис. 18.1).

При разработке взаимодействующих между собой объектов следует обеспечить необходимый для передачи сообщений уровень видимости.

В языке UML для иллюстрации областей видимости существуют специальные обозначения. В этой главе рассматриваются различные типы областей видимости.

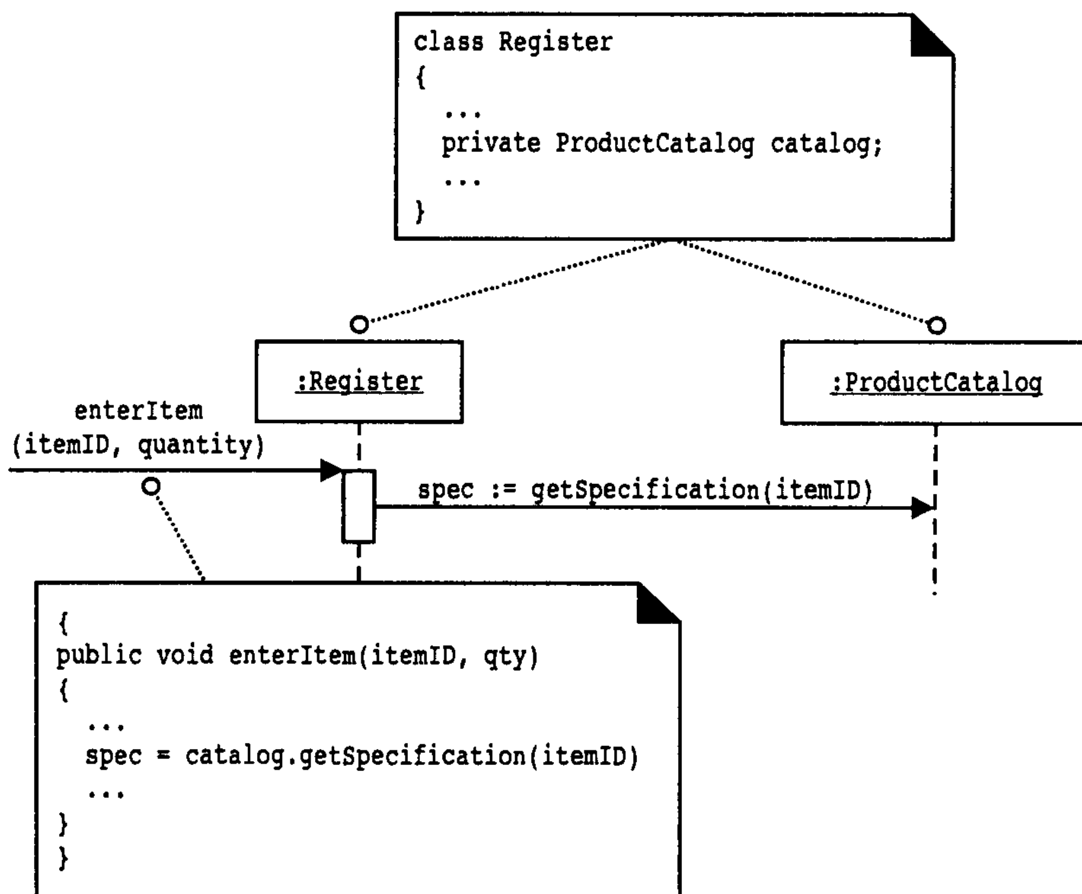



*Рис. 18.1. Необходимость обеспечения видимости объекта* `ProductCatalog` *для объекта* `Register`[1]

## 18.2. Области видимости

В обычном понимании *область видимости* (visibility) обеспечивает способность одного объекта видеть другой объект или ссылаться на него. В более общем смысле это понятие связано с понятием контекста (или области действия): попадает ли один ресурс (например, экземпляр объекта) в контекст другого? Существует четыре основных способа обеспечения видимости объекта В объектом А.

1. *Посредством атрибутов.* Объект В является атрибутом объекта А.
2. *Посредством параметров.* Объект В является параметром метода объекта А.
3. *Локальная видимость.* Объект В объявлен как локальная переменная в методе объекта А.
4. *Глобальная видимость.* Объект В каким-то образом виден глобально.

При рассмотрении вопросов видимости объектов используется следующая мотивация.

| Для отправки сообщения от объекта А к объекту В необходимо, чтобы объект В был виден объекту А. |
|---|

[1] В этом и последующих примерах фрагментов кода для обеспечения ясности изложения может применяться упрощенный синтаксис языка программирования.

Например, для создания диаграммы взаимодействия, на которой сообщение передается от экземпляра объекта Register к экземпляру ProductCatalog, необходимо, чтобы объект ProductCatalog был виден объекту Register. Типичным решением такой проблемы является использование ссылки на экземпляр ProductCatalog в качестве атрибута объекта Register.

## Обеспечение видимости посредством атрибутов

*Видимость посредством атрибутов* (attribute visibility) между объектами A и B существует в том случае, когда B является атрибутом A. Такой тип видимости можно назвать постоянным, поскольку видимость между объектами поддерживается до тех пор, пока существуют сами объекты. В объектно-ориентированных системах это очень популярная форма обеспечения видимости.

В качестве примера рассмотрим определение класса Register на языке Java, в котором обеспечивается видимость объекта ProductCatalog посредством атрибутов (переменных-членов Java).

```
public class Register
{
...
private ProductCatalog catalog;
...
}
```

Такая видимость между объектами необходима, поскольку на диаграмме взаимодействия для события enterItem, представленной на рис. 18.2, объект Register отправляет сообщение getSpecification объекту ProductCatalog.

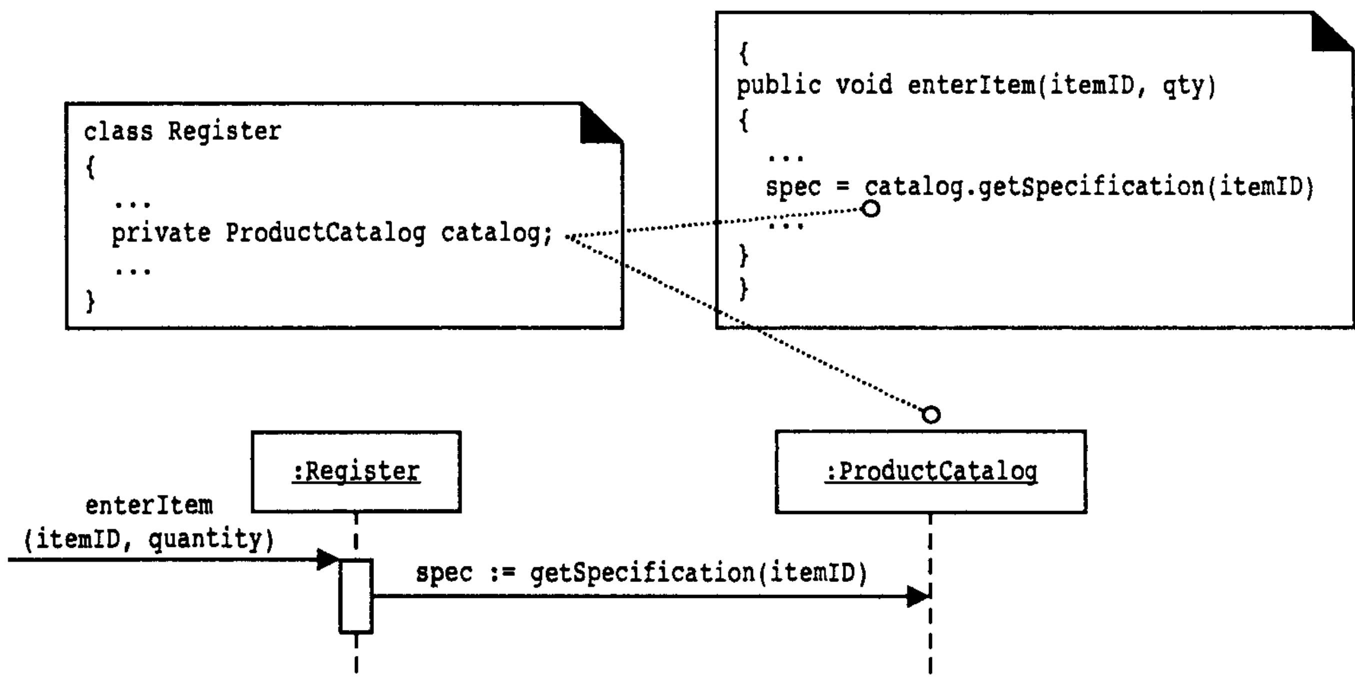



*Рис. 18.2. Видимость, обеспеченная с помощью атрибутов*

## Обеспечение видимости посредством параметров

Видимость между объектами A и B *посредством параметров* (parameter visibility) существует в том случае, если объект B передается в качестве параметра метода объекту A. Это "временная" видимость, поскольку она существует

только в контексте данного метода. После видимости, обеспечиваемой с помощью атрибутов, это второй по распространенности тип обеспечения видимости в объектно-ориентированных системах.

В качестве примера можно привести процесс передачи сообщения makeLineItem экземпляру объекта Sale, когда экземпляр объекта ProductSpecification передается в качестве параметра. В контексте метода makeLineItem между объектами Sale и ProductSpecification существует видимость, обеспечиваемая с помощью параметров (рис. 18.3).

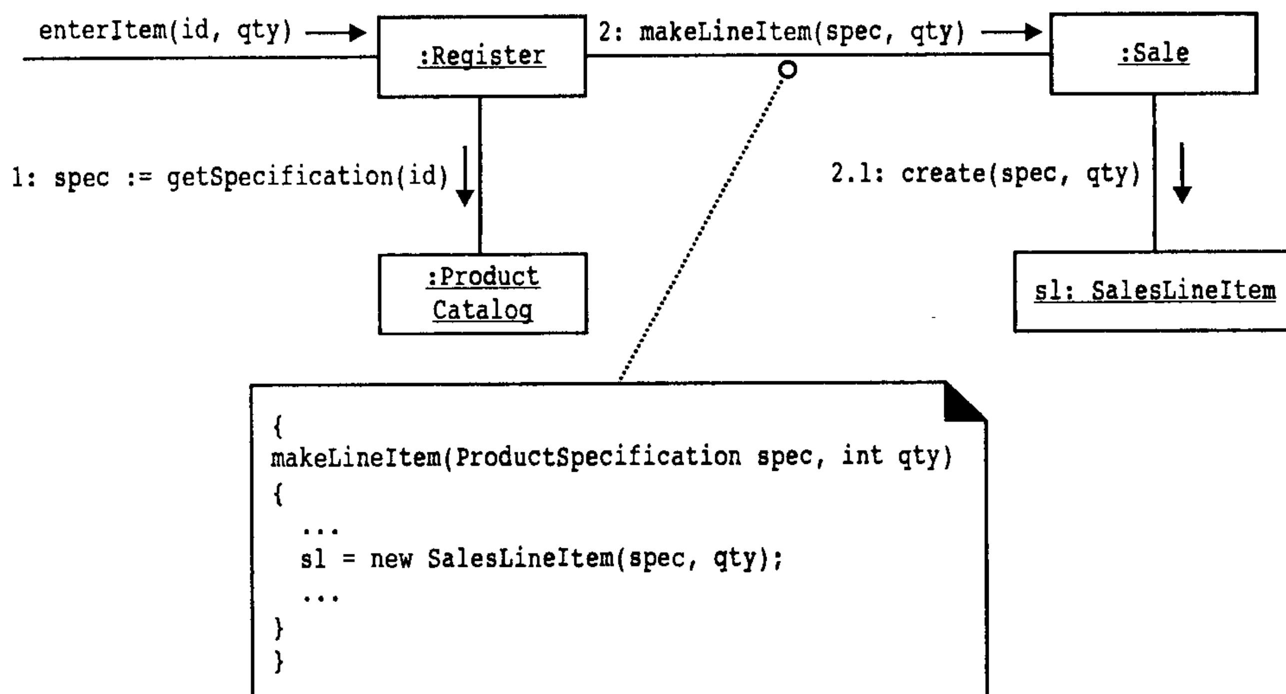


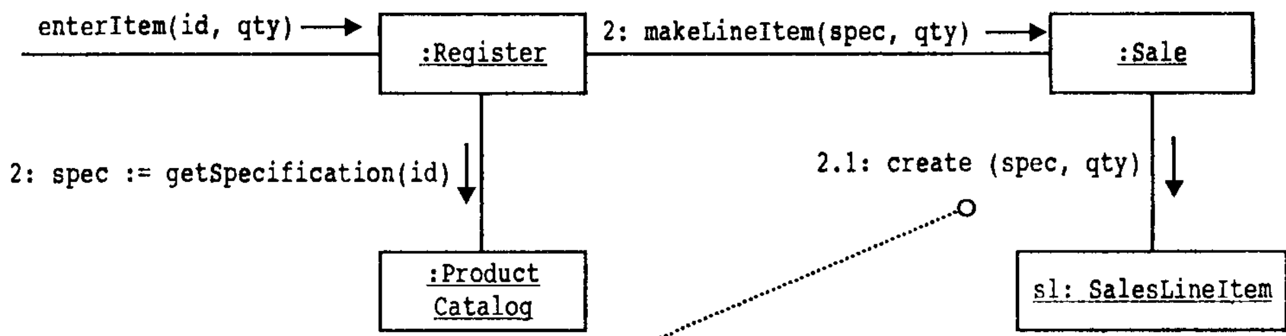

*Рис. 18.3. Видимость, обеспечиваемая с помощью параметров*

Зачастую видимость, обеспечиваемую посредством параметров, преобразуют в видимость, обеспечиваемую посредством атрибутов. Например, при создании нового экземпляра SalesLineItem объектом Sale методу инициализации (в языках C++ и Java это *конструктор* (constructor)) в качестве параметра передается экземпляр ProductSpecification. В методе инициализации значение этого параметра присваивается атрибуту и видимость устанавливается посредством атрибутов (рис. 18.4).

## Локальная видимость

*Локальная видимость* (locally declared visibility) между объектами A и B существует в том случае, если объект B объявлен в качестве локальной переменной в методе объекта A. Это относительно "временная" видимость, поскольку она существует только в контексте одного метода. Это третий по распространенности способ обеспечения видимости между объектами в объектно-ориентированных системах. Такая форма видимости между объектами достигается двумя следующими способами.

- Создается новый локальный экземпляр и присваивается в качестве значения локальной переменной.
- Локальной переменной присваивается объект, возвращаемый другим методом.




```
// метод инициализации (например, конструктор Java)
{
SalesLineItem(ProductSpecification spec, int qty)
{
...
productSpec = spec; // переход к видимости посредством атрибутов
...
}
}
```

*Рис. 18.4. Преобразование видимости посредством параметров в видимость посредством атрибутов*

Как и в случае с обеспечением видимости посредством параметров, зачастую локальная видимость преобразуется в видимость посредством атрибутов.

В качестве примера второго варианта можно рассматривать метод `enterItem` класса `Register` (рис. 18.5).

Существует вариант второго способа, когда переменная явно не объявляется в методе, но неявно существует как возвращаемое значение метода. Например,

```
//неявная локальная видимость объекта,
//возвращаемого при вызове метода getFoo()
anObject.getFoo().doBar();
```

```
{
enterItem(id, qty)
{
...
// локальная видимость обеспечивается путем
// присваивания возвращаемого объекта
ProductSpecification spec = catalog.getSpecification(id);
...
}
}
```

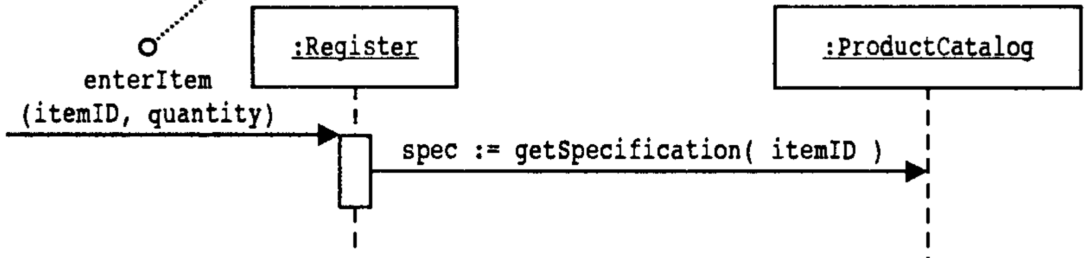



*Рис. 18.5. Локальная видимость*

### Глобальная видимость

*Глобальная видимость* (global visibility) между объектами A и B существует в том случае, когда объект B является глобальным по отношению к A. Это относительно постоянная видимость, поскольку она поддерживается до тех пор, пока существуют объекты A и B. Это наиболее редкий способ обеспечения видимости между объектами в объектно-ориентированном программировании.

Один из способов достижения глобальной видимости состоит в присваивании экземпляра объекта глобальной переменной. Однако такой подход возможен не во всех языках (например, возможен в C++, но не в Java).

Более предпочтительный способ состоит в использовании шаблона Singleton [52], описанного в последующих главах.

## 18.3. Иллюстрация видимости между объектами средствами языка UML

В языке UML существуют обозначения для иллюстрации различных типов видимости на диаграммах кооперации (рис. 18.6). Однако они являются необязательными и зачастую не используются. Их применяют только в том случае, когда требуется уточнить способ обеспечения видимости.

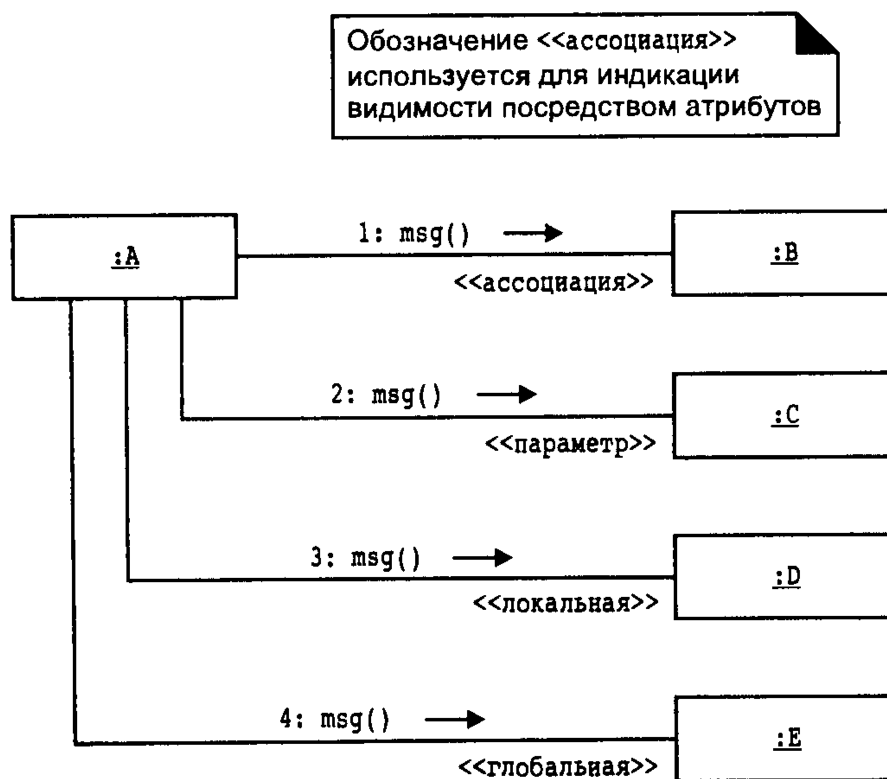



*Рис. 18.6. Реализация различных способов обеспечения видимости*

Глава

# Модель проектирования: создание диаграммы классов

*Итеративность присуща деятельности человека, а рекурсивность — деятельности высших существ.*

*Автор неизвестен*

Основные задачи

- Создать диаграммы классов проектирования.
- Определить классы, методы и ассоциации и отобразить их на диаграмме классов.

## Введение

Завершая построение диаграмм взаимодействия для реализации прецедента, рассматриваемого на текущей итерации создания POS-приложения NextGen, можно написать спецификацию для программных классов и интерфейсов, которые принимают участие в программном решении, указав некоторые детали проектирования, в частности — методы.

Детали проектного решения можно отобразить средствами UML на диаграммах классов, которые и рассматриваются в этой главе.

## 19.1. Когда следует создавать диаграммы классов

Хотя в этой книге диаграммы классов создаются *после* диаграмм взаимодействия, на самом деле они зачастую разрабатываются параллельно. Имена многих классов, методов и типы отношений с помощью шаблонов распределения обязанностей можно определить уже на начальной стадии проектирования, до построения диаграмм взаимодействия. Желательно выполнять проектирование следующим образом. Сначала стоит построить схематичные диаграммы взаимодействия, затем создать диаграммы классов, после чего детализировать диаграммы взаимодействия и т.д.

Диаграммы классов можно рассматривать как альтернативу картам CRC, обеспечивающую более удобное графическое представление информации при распределении обязанностей между классами.

## 19.2. Пример диаграммы классов

Диаграмма классов, представленная на рис. 19.1, иллюстрирует фрагмент программного решения для классов `Register` и `Sale`.

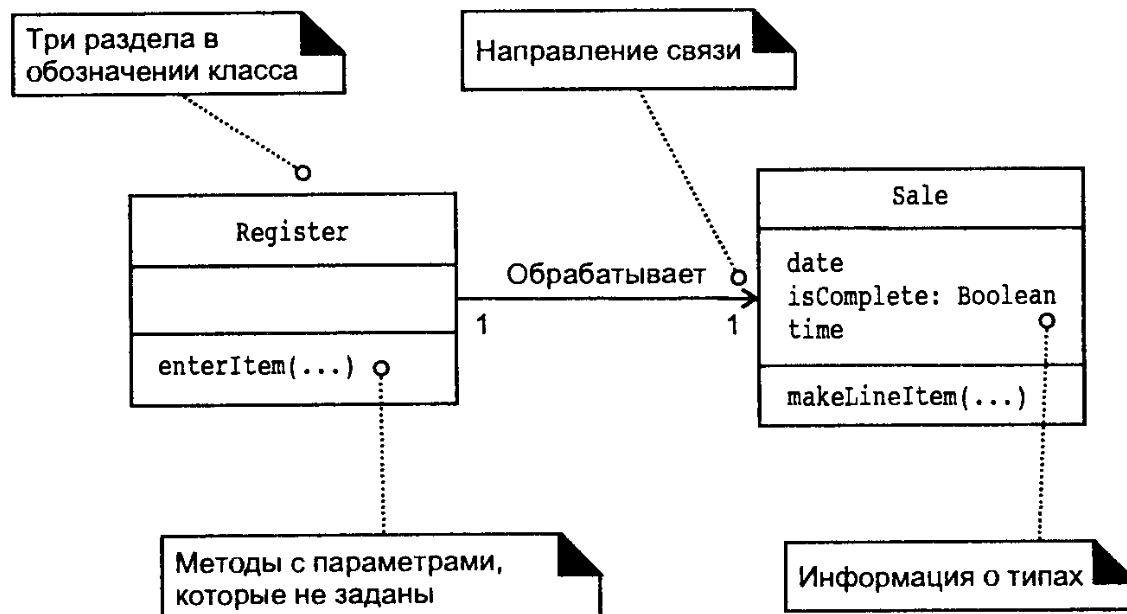



*Рис. 19.1. Пример диаграммы программных классов*

Помимо основных ассоциаций и атрибутов, на диаграмме отображаются методы каждого класса, информация о типах атрибутов, а также способы обеспечения видимости и навигации между объектами.

## 19.3. Диаграммы классов и терминология UP

*Диаграмма классов* (design class diagram) иллюстрирует спецификации программных классов и интерфейсов (например, интерфейсов Java) в приложении. Обычно на такую диаграмму выносится следующая информация.

- Классы, ассоциации и атрибуты
- Интерфейсы со своими операциями и константами
- Методы
- Информация о типах атрибутов
- Способы навигации
- Зависимости

В отличие от диаграммы классов из модели предметной области, диаграммы классов проектирования отображают определения программных сущностей, а не понятия предметной области.

В UP не определен отдельный артефакт под названием "диаграмма классов проектирования". Там определена лишь модель проектирования, которая может включать несколько типов диаграмм, в том числе диаграммы классов, взаимодействия и

пакетов. Диаграммы классов из модели проектирования UP иллюстрируют “классы проектирования” в терминах UP. Поэтому зачастую диаграммы классов из модели проектирования называют просто диаграммами классов проектирования.

## 19.4. Классы из модели предметной области и модели проектирования

Напомним, что в модели предметной области объект `Sale` не представляет программную сущность, а определяет абстракцию понятия реального мира, состояние которой необходимо знать в системе. В отличие от модели предметной области, на диаграмме классов проектирования отображаются программные компоненты. Здесь элемент `Sale` представляет собой программный класс (рис. 19.2).

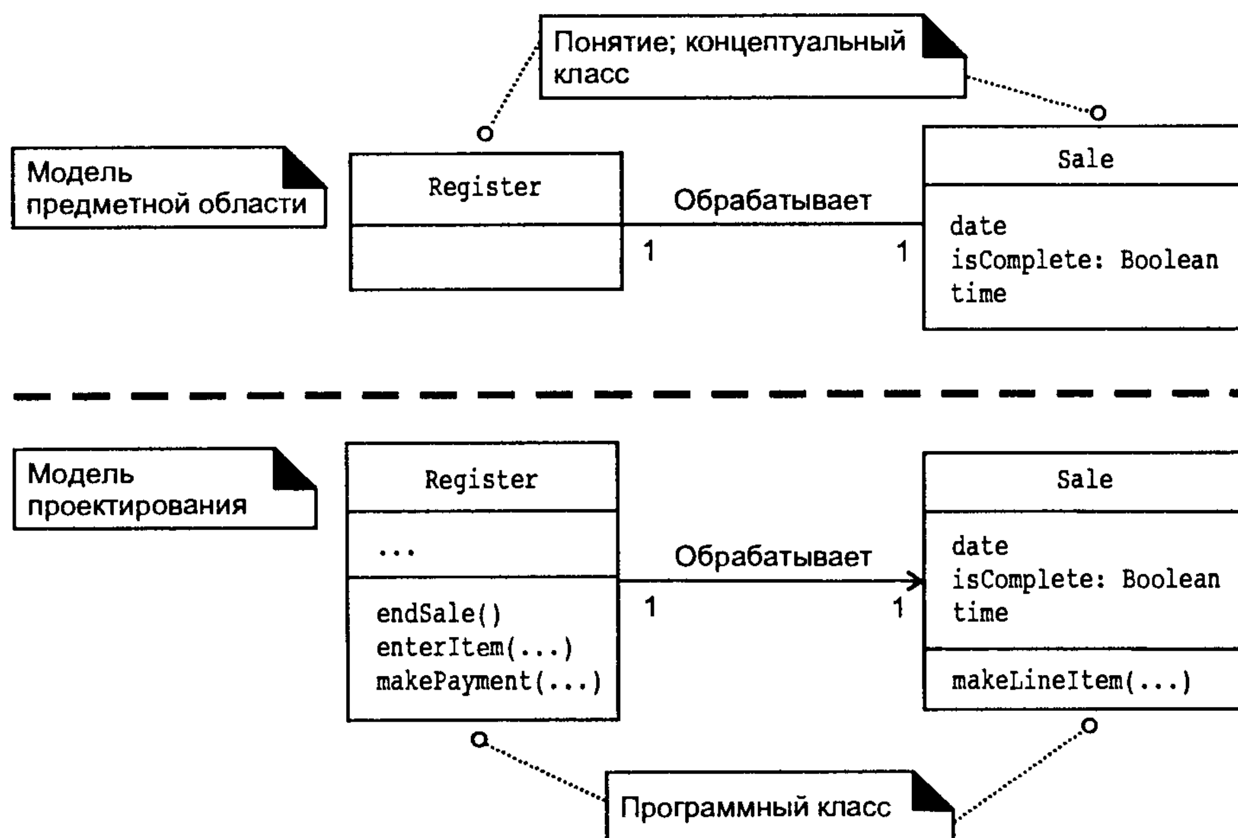



*Рис. 19.2. Классы модели предметной области и модели проектирования*

## 19.5. Создание диаграммы классов для POS-системы NextGen

### *Идентификация программных классов и их отображение*

Первым шагом на пути создания диаграммы классов является идентификация классов, которые должны участвовать в программном решении. Эту задачу можно решить, внимательно изучив все диаграммы взаимодействия и выбрав упомянутые на них классы.

Для POS-приложения к числу таких классов относятся следующие.

```
Register                    Sale
ProductCatalog              ProductSpecification
Store                       SalesLineItem
Payment
```

Следующим шагом является нанесение этих классов на диаграмму и добавление атрибутов, определенных с использованием модели предметной области (рис. 19.3).

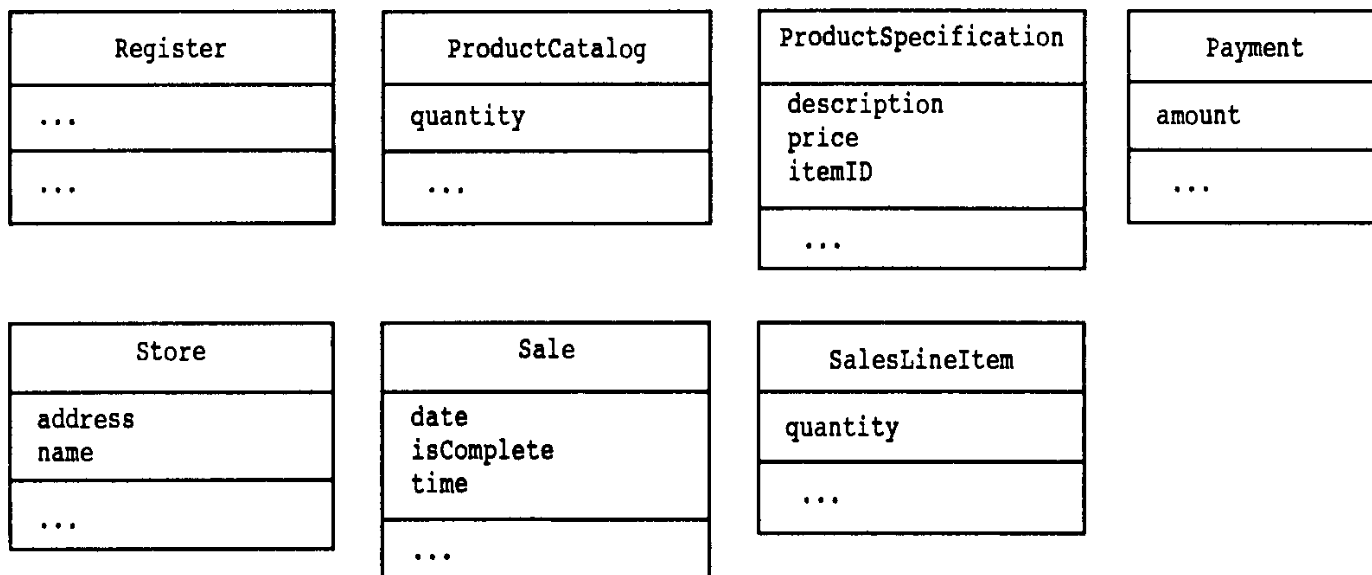



*Рис. 19.3. Программные классы приложения*

Заметим, что многие понятия модели предметной области, такие как Cashier, на диаграмме отсутствуют. Дело в том, что на данной итерации нет необходимости разрабатывать их программное представление. Однако на последующих итерациях, при появлении новых требований и разработке новых прецедентов, они, возможно, будут внесены в диаграмму. Например, при реализации требований к безопасности и процессу регистрации желательно ввести программный класс Cashier.

## Добавление имен методов

Методы каждого класса можно определить путем анализа диаграмм взаимодействия. Например, если сообщение makeLineItem передается экземпляру класса Sale, то в классе Sale должен быть определен метод makeLineItem (рис. 19.4).

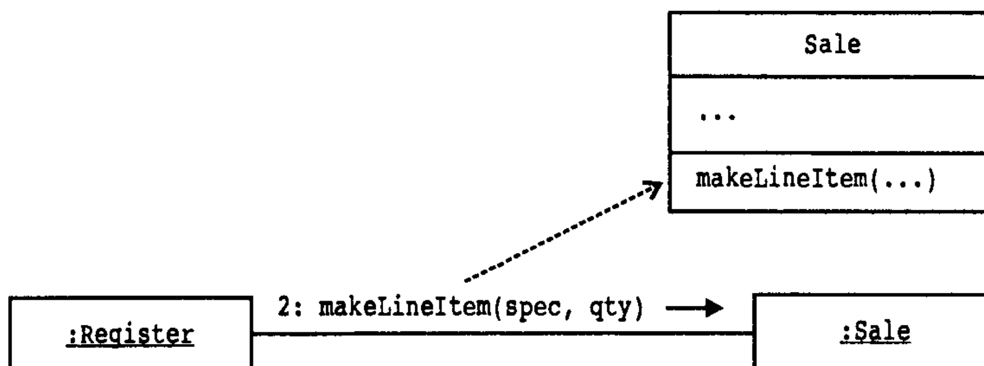



*Рис. 19.4. Имена методов из диаграмм взаимодействия*

Таким образом, имена всех сообщений, передаваемых классу X, отображенные на всех диаграммах взаимодействия, определяют большую часть методов этого класса.

Изучив все диаграммы взаимодействия для POS-приложения, получим имена методов, представленные на рис. 19.5.

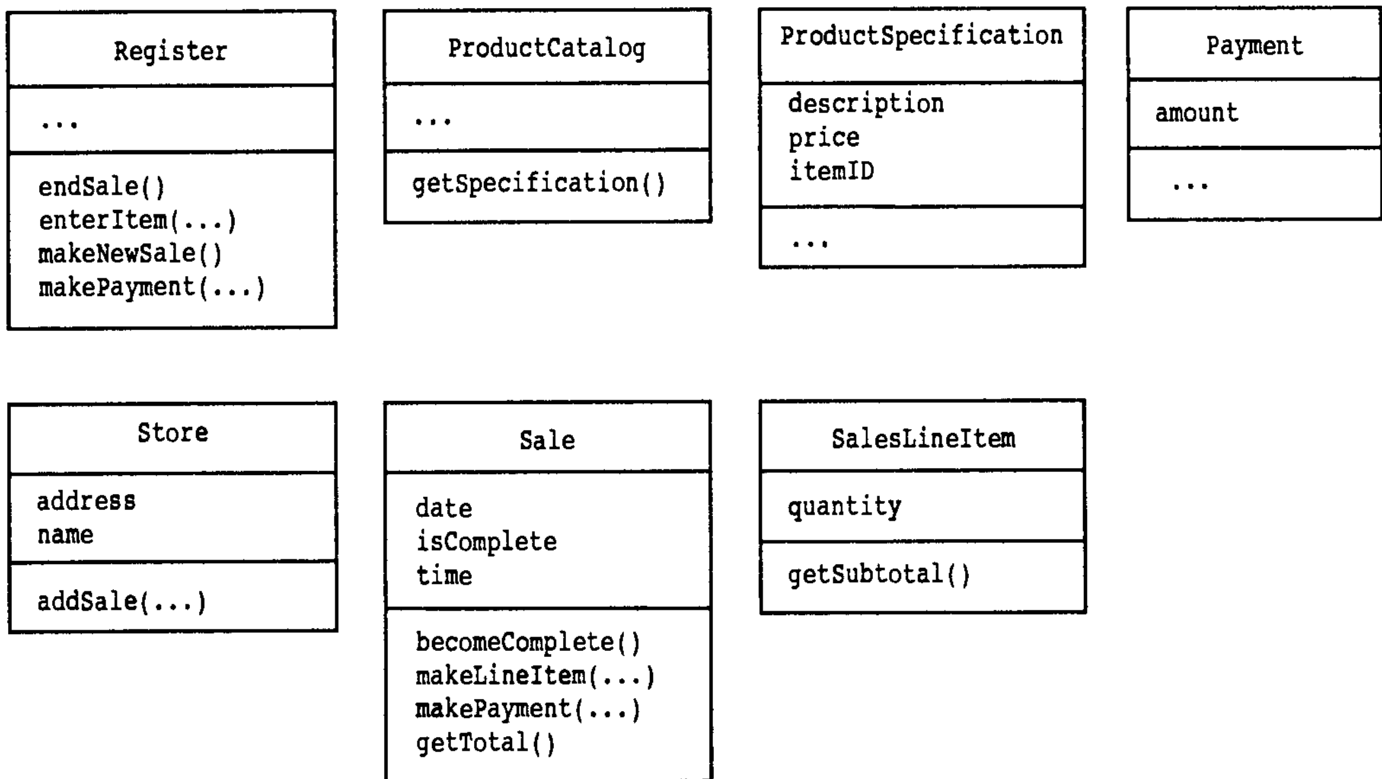



*Рис. 19.5. Методы приложения*

### Выбор имен методов

При выборе имен методов необходимо руководствоваться следующими соображениями.

- Интерпретировать сообщение `create()`
- Описывать методы доступа
- Интерпретировать сообщения сложным объектам
- Использовать синтаксис языка программирования

### Имена методов: `create`

Сообщение `create` на языке UML представляет собой абстрактную форму инициализации или инстанцирования. При переходе к реализации системы на объектно-ориентированном языке программирования процесс передачи сообщения необходимо выразить средствами языка программирования с использованием его идиом для инстанцирования и инициализации. В языке C++, Java или Smalltalk не существует реального метода `create`. Например, на языке C++ память выделяется с помощью оператора `new`, после которого следует имя конструктора.

Поскольку инициализация — это очень типичный вид деятельности, который по-разному выражается в различных языках программирования, методы создания объектов и конструкторы на диаграммах классов зачастую не отображаются.

### Имена методов: методы доступа

*Методы доступа* (accessing methods) — это методы получения или установки значений атрибутов. В некоторых языках программирования для каждого атрибута принято создавать свои методы получения и установки значений, а са-

ми атрибуты объявлять в закрытой области доступа (для обеспечения инкапсуляции). Эти методы обычно не отображаются на диаграмме классов, чтобы не загромождать ее лишней информацией, поскольку при наличии N атрибутов у класса появляется 2N стандартных метода. Например, метод getPrice (или price) класса ProductSpecification не показан на диаграмме классов, поскольку это простой метод доступа.

### Имена методов: сложные объекты

Сообщения сложному объекту интерпретируются как сообщения контейнеру или объекту-коллекции. Например, следующее сообщение find сложному объекту можно интерпретировать как сообщение объекту-контейнеру, такому как Map в Java, map в C++ или Dictionary в Smalltalk (рис. 19.6).

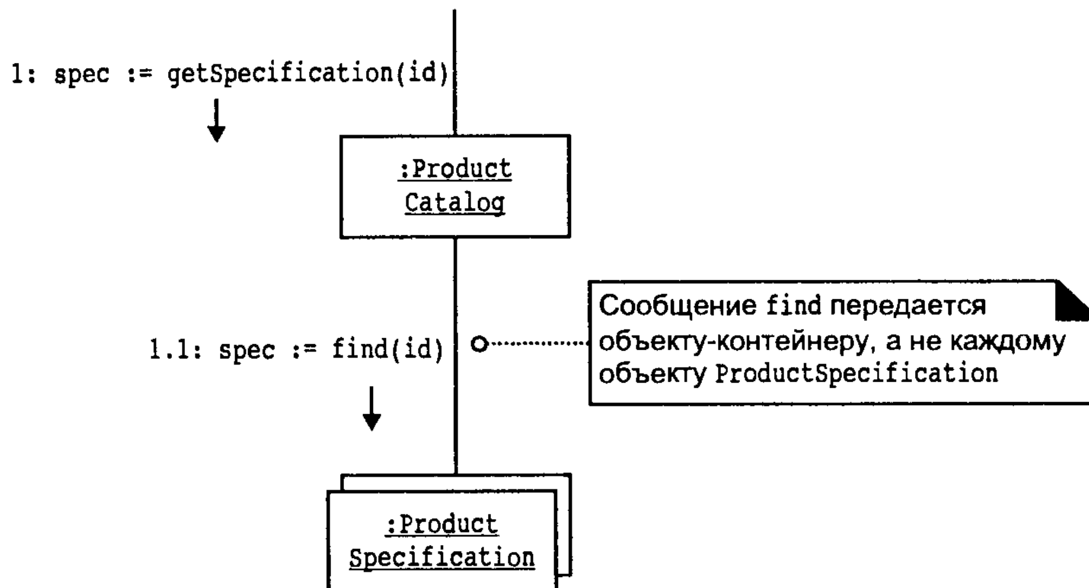



*Рис. 19.6. Сообщение сложному объекту*

Таким образом, метод find не является частью класса ProductSpecification, а относится к интерфейсу сложного объекта. Поэтому некорректно добавлять метод find в спецификацию класса ProductSpecification.

Классы или интерфейсы-контейнеры (такие как java.util.Map) — это элементы предопределенной библиотеки, которые не отображаются на диаграмме классов, поскольку они лишь загромождают диаграмму и не несут новой информации.

### Имена методов: синтаксис с учетом языка

В некоторых языках, таких как Smalltalk, синтаксическая форма описания метода отличается от базового формата UML вида *имяМетода(списокПараметров)*. На диаграммах классов рекомендуется использовать базовый формат языка UML, даже если планируется реализовывать систему на языке с другим синтаксисом. Переход к синтаксису конкретного языка целесообразно выполнять на этапе генерации кода, а не в процессе создания диаграмм классов. Однако UML допускает применение различных синтаксисов для спецификации методов.

### Добавление дополнительной информации о типах

На диаграмме классов можно отображать информацию о типах атрибутов, параметрах методов и возвращаемых значениях. Принимая решение о том, следует ли отображать эту информацию, необходимо учитывать следующие соображения.

Диаграмма классов создается для ее пользователей.

- Если генерация кода будет выполняться автоматически с использованием CASE-средств, то на диаграмме классов необходимо отобразить полную и исчерпывающую информацию.
- Если диаграмма классов создается для программистов, то избыточные детали могут загромождать диаграмму.

Например, всегда ли необходимо отображать на диаграмме все параметры и информацию об их типах? Это зависит от того, насколько информация очевидна для пользователей диаграммы.

На диаграмме классов, представленной на рис. 19.7, показана дополнительная информация о типах параметров.

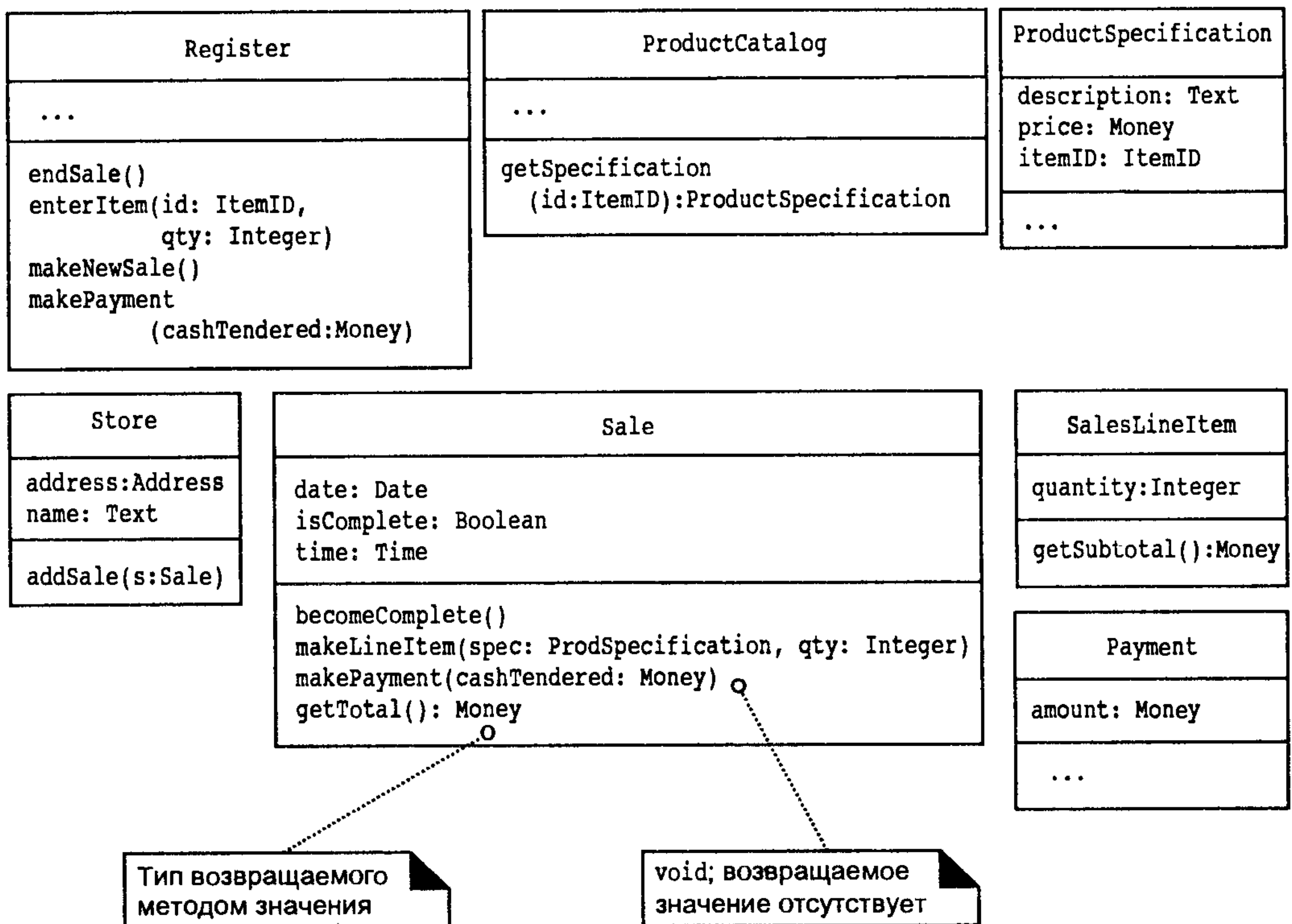



*Рис. 19.7. Добавление информации о типах*

## Добавление ассоциаций и информации о навигации

Каждый конец линии ассоциации называется *ролью* (role). На диаграмме классов роль может отмечаться стрелкой (стрелкой навигации), указывающей направление связи. *Информация о навигации* (navigability) — это свойство роли, указывающее возможное направление передачи информации от объекта-источника к целевому классу. Информация о навигации связана с видимостью объектов, обычно — с видимостью, обеспечиваемой посредством атрибутов (рис. 19.8).

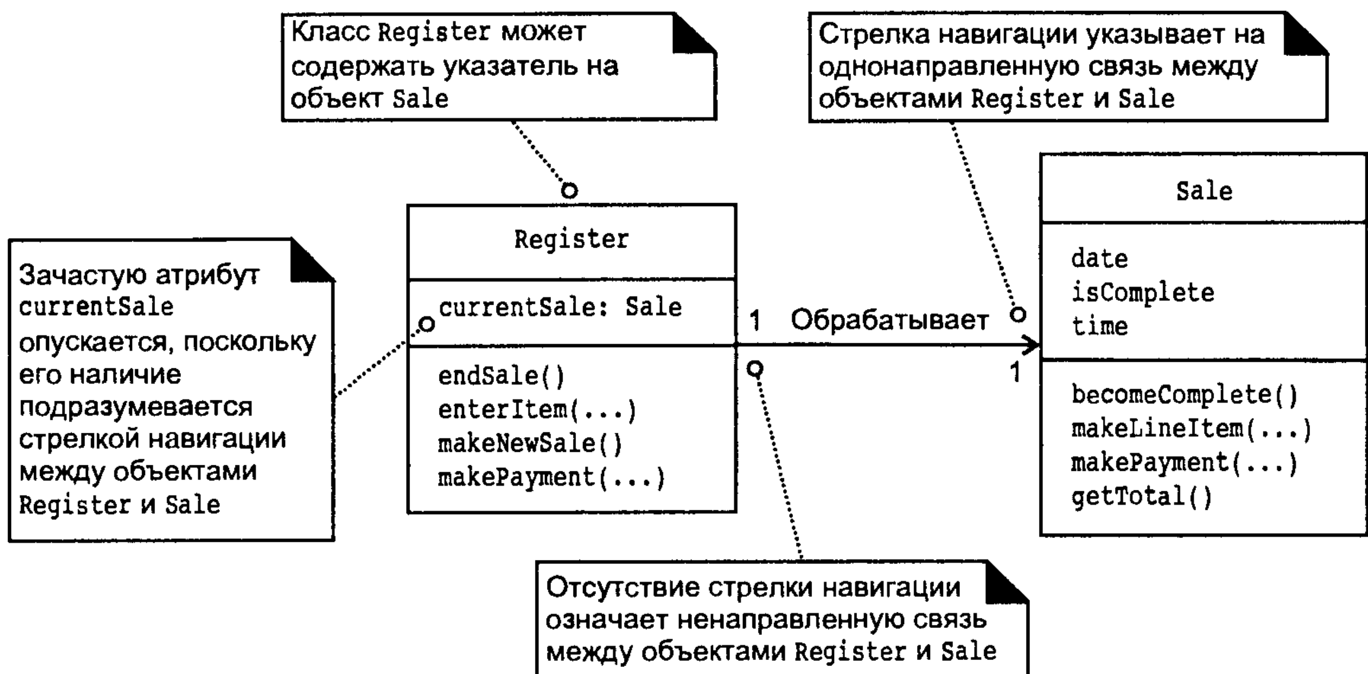



*Рис. 19.8. Информация о навигации или видимости, обеспечиваемой посредством атрибутов*

Линия ассоциации со стрелкой навигации обычно интерпретируется как видимость целевого класса для класса-источника, обеспечиваемая с помощью атрибутов. В процессе реализации на объектно-ориентированном языке программирования она обычно выражается в том, что один из атрибутов класса-источника является ссылкой на экземпляр целевого класса. В частности, один из атрибутов класса `Register` должен являться ссылкой на экземпляр класса `Sale`.

| Большинство (если не все) ассоциаций на диаграммах классов должно быть снабжено необходимыми стрелками навигации. |
|---|

На диаграммах классов ассоциации выбираются по жесткому программно-ориентированному критерию: отображаются те ассоциации, которые необходимы для удовлетворения требований видимости и памяти, вытекающих из диаграмм взаимодействий. Такой подход отличается от подхода к отображению ассоциаций в модели предметной области, где ассоциации отображают зависимости объектов предметной области. Здесь снова проявляются различия между моделями проектирования и предметной области: первая описывает программные компоненты, а вторая — результаты анализа предметной области.

Требуемые свойства видимости и ассоциации между классами определяются на основе диаграмм взаимодействия. Вот типичные ситуации, требующие определения ассоциаций с указанием направления связи от объекта `A` к объекту `B`.

- Объект `A` отправляет сообщение объекту `B`
- Объект `A` создает экземпляр объекта `B`
- Объект `A` должен поддерживать связь с объектом `B`

Например, из диаграммы взаимодействия, представленной на рис. 19.9, видно, что объект `Store` должен быть связан с создаваемыми им экземплярами объектов `Register` и `ProductCatalog`, причем эта связь должна быть направлена от объекта `Store`. Целесообразно также установить связь объекта `ProductCatalog` с создаваемой им коллекцией объектов `ProductSpecification`. На самом деле

практически всегда направление связи соответствует направлению от объекта-создателя к создаваемым им объектам. На диаграмме классов такие связи представляются в виде ассоциаций.

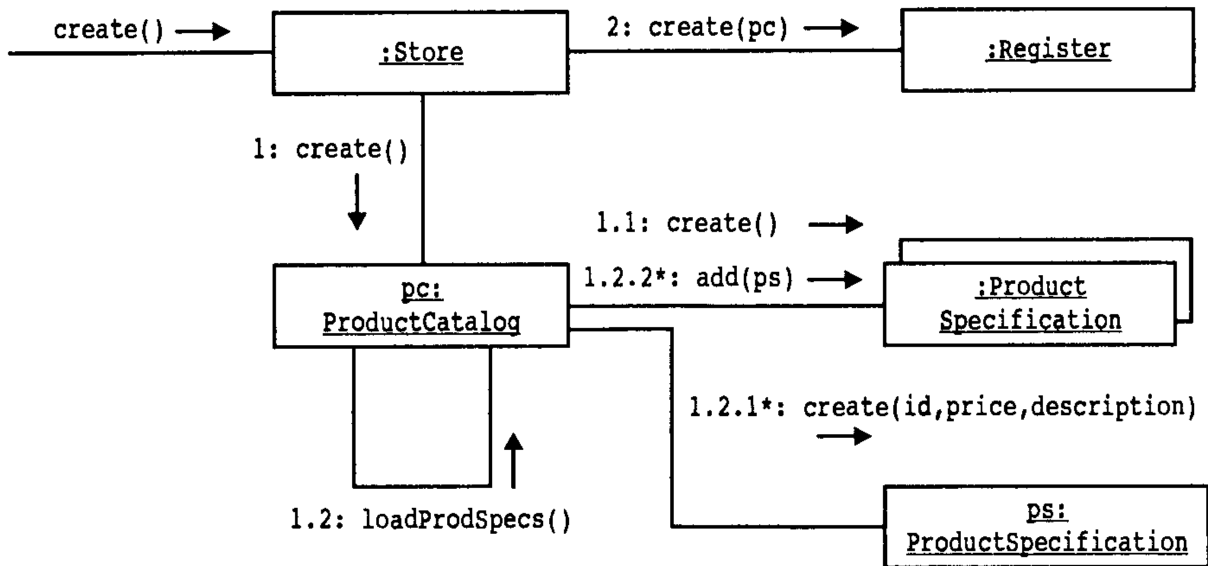



*Рис. 19.9. Направление связи определяется из диаграмм взаимодействия*

Пользуясь приведенными выше критериями для определения ассоциаций и направления связей на основе анализа всех диаграмм взаимодействия, для POS-приложения NextGen можно построить диаграмму классов, представленную на рис. 19.10. (Для большей ясности дополнительная информация о типах на этой диаграмме не отображается.)

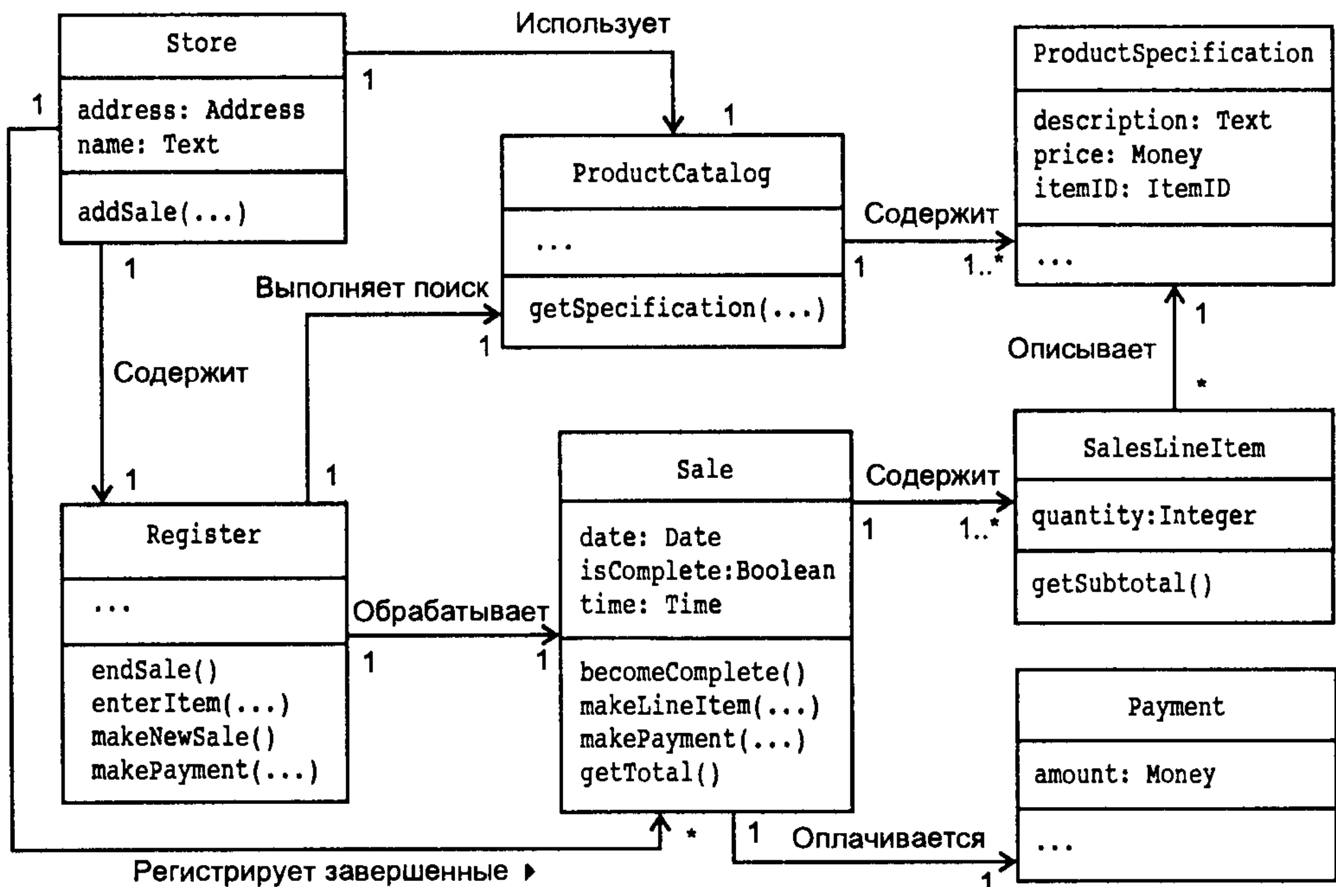



*Рис. 19.10. Ассоциации с указанием направления связей*

Заметим, что эти ассоциации не совпадают с набором ассоциаций, сгенерированным для диаграммы классов из модели предметной области. Например, в модели предметной области между классами Register и ProductCatalog отсутствует ассоциация Находит в, поскольку на этапе создания модели предметной области эта взаимосвязь не казалась важной. Однако в процессе создания диаграмм взаимодействия было решено, что программный объект Register должен быть связан с программным объектом ProductCatalog с целью нахождения спецификации ProductSpecification.

## Добавление зависимостей

В языке UML существует обозначение для *отношения зависимости* (dependency relationship), указывающего, что один элемент (любого типа, включая классы, прецеденты и т.д.) знает о другом элементе. Такое отношение отображается пунктирной линией со стрелкой. На диаграмме классов отношение зависимости отображает видимость между классами, отличную от обеспечиваемой посредством атрибутов, т.е. глобальную, локальную видимость или видимость, обеспечиваемую с помощью параметров. Видимость, обеспечиваемая посредством атрибутов, отображается сплошной линией ассоциации со стрелкой, которая указывает направление связи. Например, программный объект Register получает возвращаемый объект типа ProductSpecification из сообщения, отправляемого им объекту ProductCatalog. Таким образом, для объекта Register обеспечивается кратковременная локальная видимость объекта ProductSpecification. Объект Sale получает ProductSpecification в качестве параметра метода makeLineItem (видимость, обеспечиваемая посредством параметров).

Такие способы обеспечения видимости отображаются пунктирными линиями со стрелками, определяющими отношение зависимости (рис. 19.11). Линии зависимости не обязательно должны быть изогнутыми; просто так было удобно графически отобразить их в данном примере.

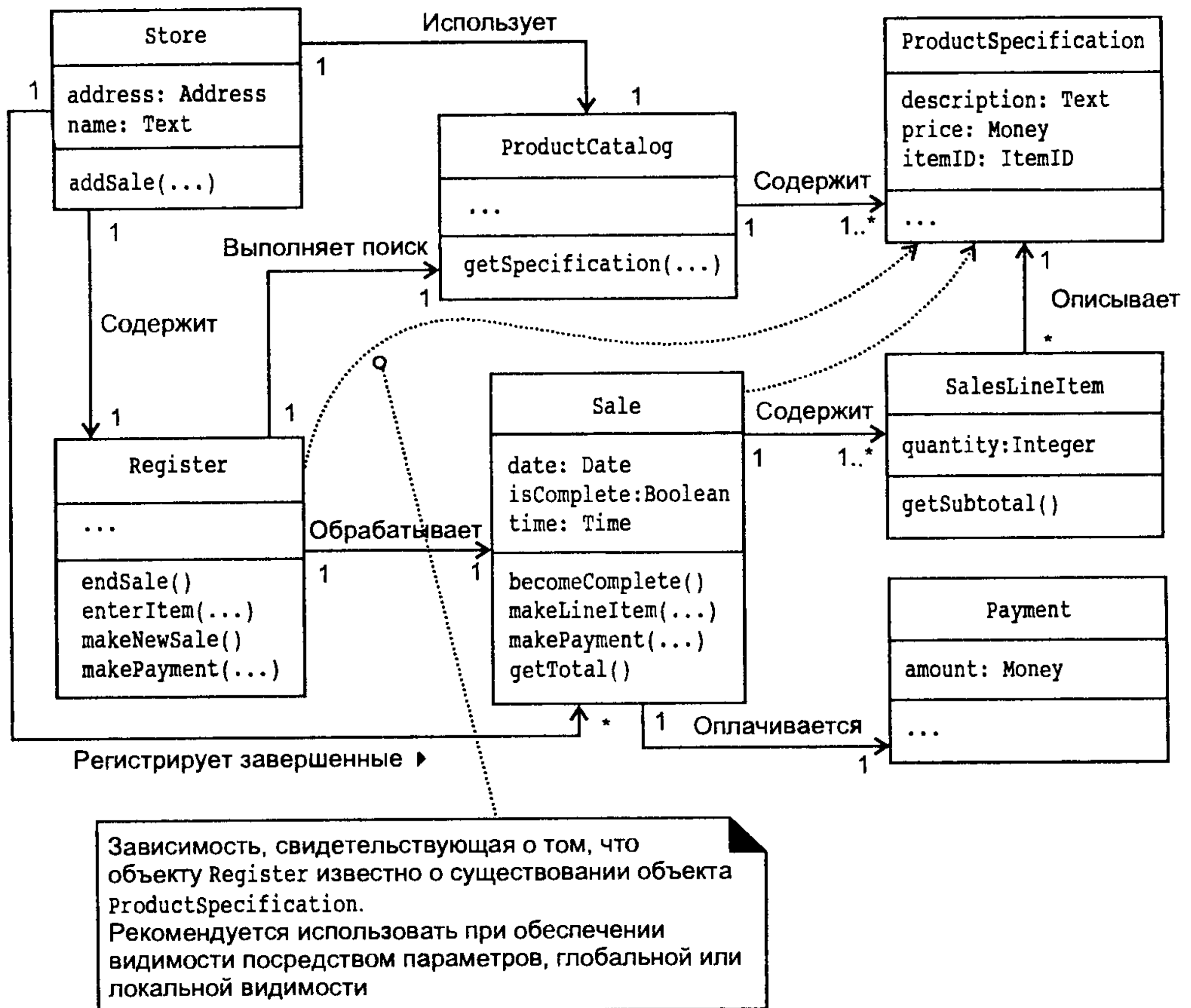



*Рис. 19.11. Отношения зависимости, задающие видимость, отличную от видимости, обеспечиваемой посредством атрибутов*

## 19.6. Обозначения для детальной информации о членах класса

В языке UML содержится обширный набор обозначений для описания свойств членов классов и интерфейсов, таких как видимость, начальные значения и т.д. Примеры таких обозначений представлены на рис. 19.12.

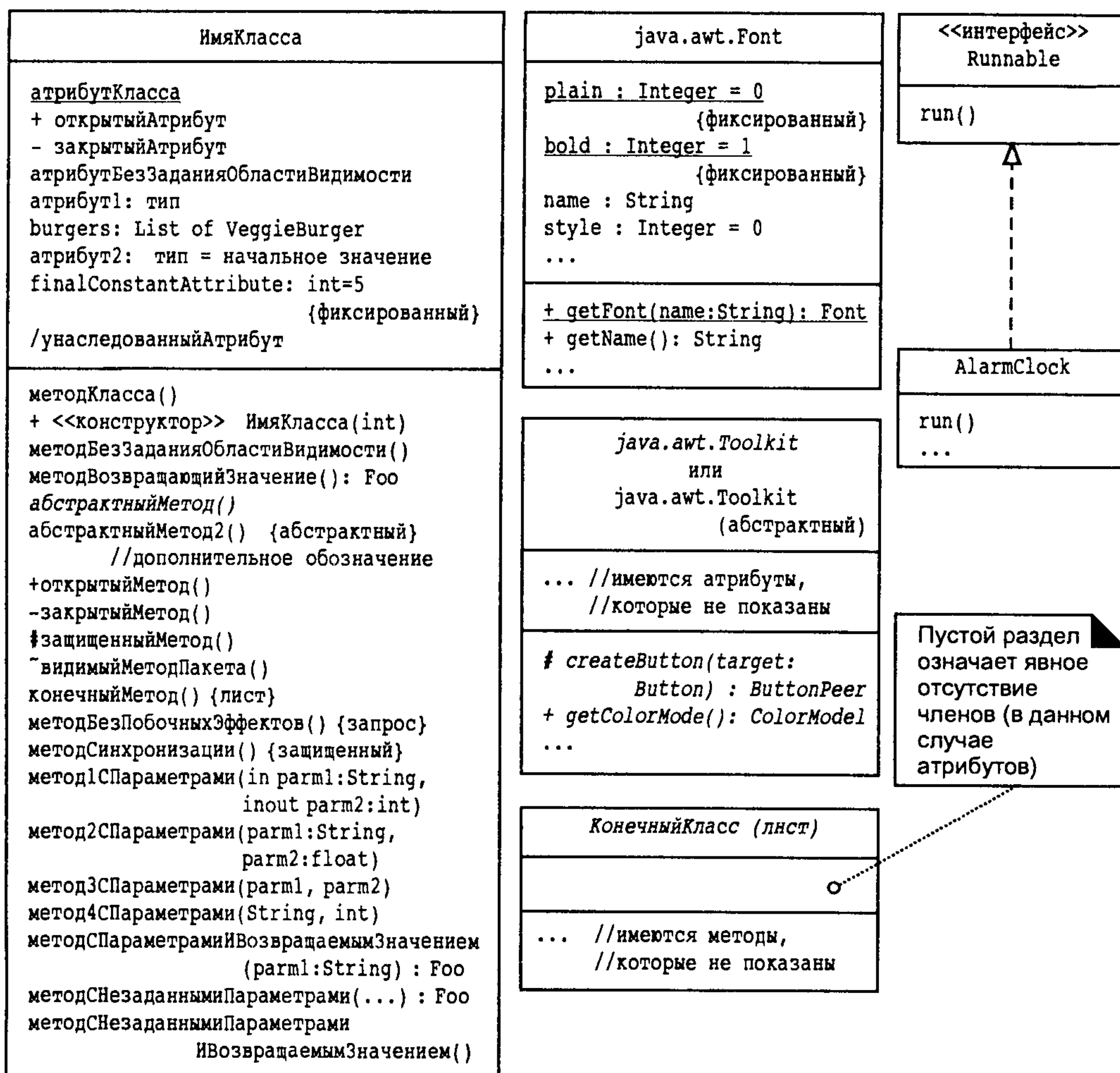



*Рис. 19.12. Обозначения UML для детальной информации о членах класса*

### Параметры видимости, используемые по умолчанию

Существуют ли задаваемые по умолчанию параметры видимости, если на диаграмме классов закрытые и открытые члены классов не отмечены явно? Нет, в языке UML не существует правил определения значений по умолчанию. Однако существует общепринятое соглашение о том, что атрибуты классов должны быть закрытыми, а методы — открытыми.

На текущей итерации разработки диаграммы классов POS-приложения NextGen (рис. 19.13) многие интересные детали, относящиеся к членам классов, отсутствуют. Из диаграммы классов видно, что все атрибуты являются закрытыми, а методы — открытыми.

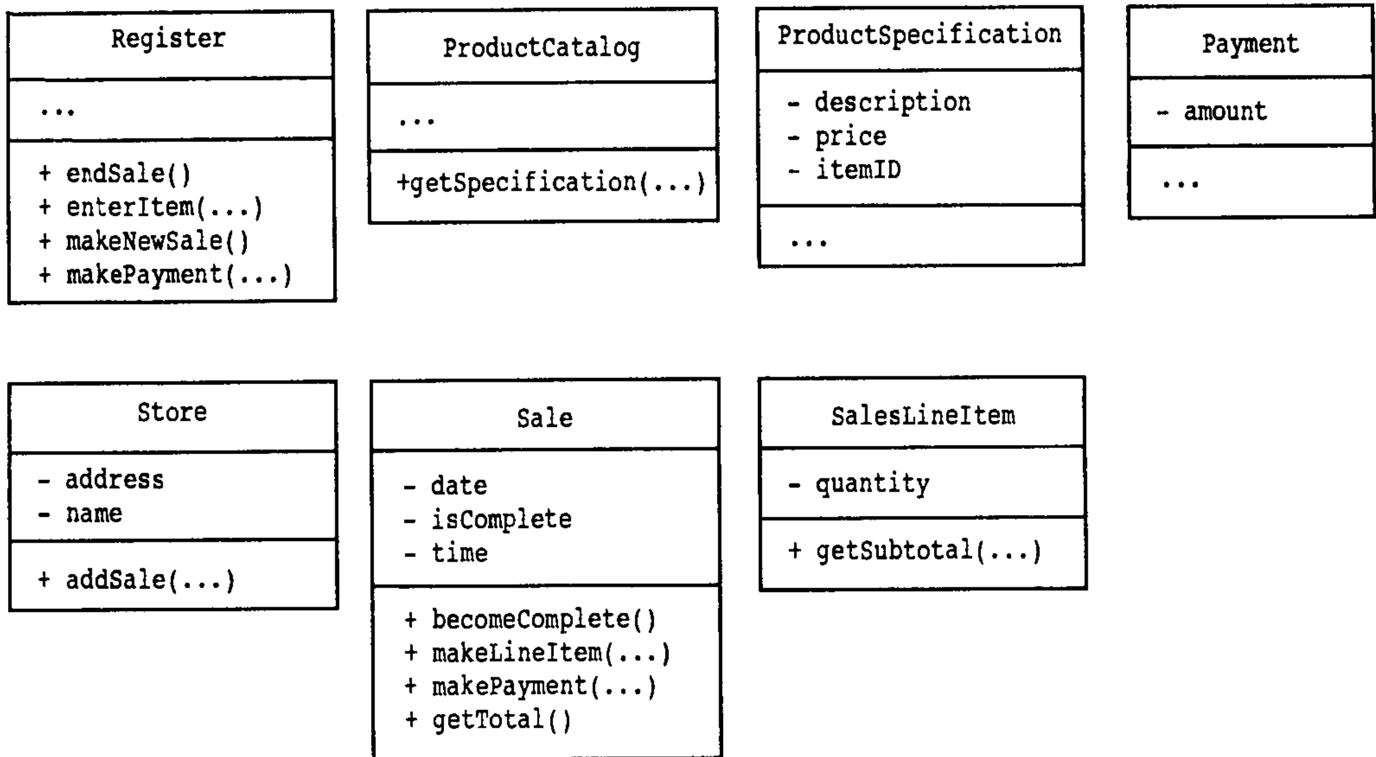



*Рис. 19.13. Детальная информация о членах классов на диаграмме классов приложения розничной торговли*

## Отображение тела метода на диаграмме классов

На диаграмме классов или взаимодействия можно отображать тело метода (рис. 19.14).

*Обозначения UML*
Тело метода может быть показано в блоке примечания. При этом его нужно поместить в скобки, чтобы указать на особый семантический смысл (это больше, чем просто комментарий).

Можно использовать синтаксис псевдокода или любого языка.

Строку заголовка метода указывать не принято (public void...), однако при необходимости ее можно определить

```
{
  ProductSpecification spec = catalog.getSpecification(id);
  sale.makeLineItem(spec, qty);
}
```

Register
...
endSale()
enterItem(id, qty)
makeNewSale()
makePayment(cashTendered)

```
{
public void enterItem(id, qty)
{
  ProductSpecification spec = catalog.getSpecification(id);
  sale.makeLineItem(spec, qty);
}
}
```

*Рис. 19.14. Изображение тела метода*

## 19.7. Построение диаграмм классов и CASE-средства

CASE-средства позволяют выполнять обратное проектирование (генерирование) диаграмм классов на основе исходного кода программы. В главе 35, посвященной CASE-средствам, затрагиваются вопросы практического построения диаграмм классов.

## 19.8. Диаграммы классов в контексте UP

Диаграммы классов, как составная часть реализации прецедентов, относятся к модели проектирования в рамках UP (табл. 19.1).

**Таблица 19.1. Пример планирования сроков реализации артефактов UP (н — начало, р — развитие)**

| Дисциплина | Артефакт<br>Итерация→ | Начало<br>I1 | Развитие<br>E1..En | Конструирование<br>C1..Cn | Передача<br>T1..Tn |
|---|---|---|---|---|---|
| Бизнес-моделирование | Модель предметной области | | н | | |
| Требования | Модель прецедентов | н | р | | |
| | Видение системы | н | р | | |
| | Дополнительная спецификация | н | р | | |
| | Словарь терминов | н | р | | |
| Проектирование | **Модель проектирования** | | н | р | |
| | Описание архитектуры | | н | | |
| | Модель данных | | н | р | |
| Реализация | Модель реализации | | н | р | р |
| Управление проектом | План разработки | н | р | р | р |
| Тестирование | Модель тестирования | | н | р | |
| Окружение | Набор документов | н | р | | |

### *Фазы*

**Начало** — разработка модели проектирования и диаграмм классов обычно начинается на стадии развития, поскольку этот процесс требует принятия подробных проектных решений, для которых на начальной стадии отсутствует необходимая информация.

**Развитие** — в течение этой фазы в рамках реализации прецедентов строятся диаграммы классов и взаимодействия. Такие диаграммы обычно создаются для большинства архитектурно значимых программных классов.

Напомним, что CASE-средства позволяют выполнять обратное проектирование и генерировать диаграммы классов на основе исходного кода. Рекомендуется делать это регулярно с целью визуализации статической структуры системы.

**Конструирование** — продолжается генерирование диаграмм классов на основе исходного кода с целью визуализации статической структуры системы.

## 19.9. Артефакты в контексте унифицированного процесса

Взаимосвязи между артефактами UP показаны на рис. 19.15. При этом основное внимание уделяется диаграммам классов.

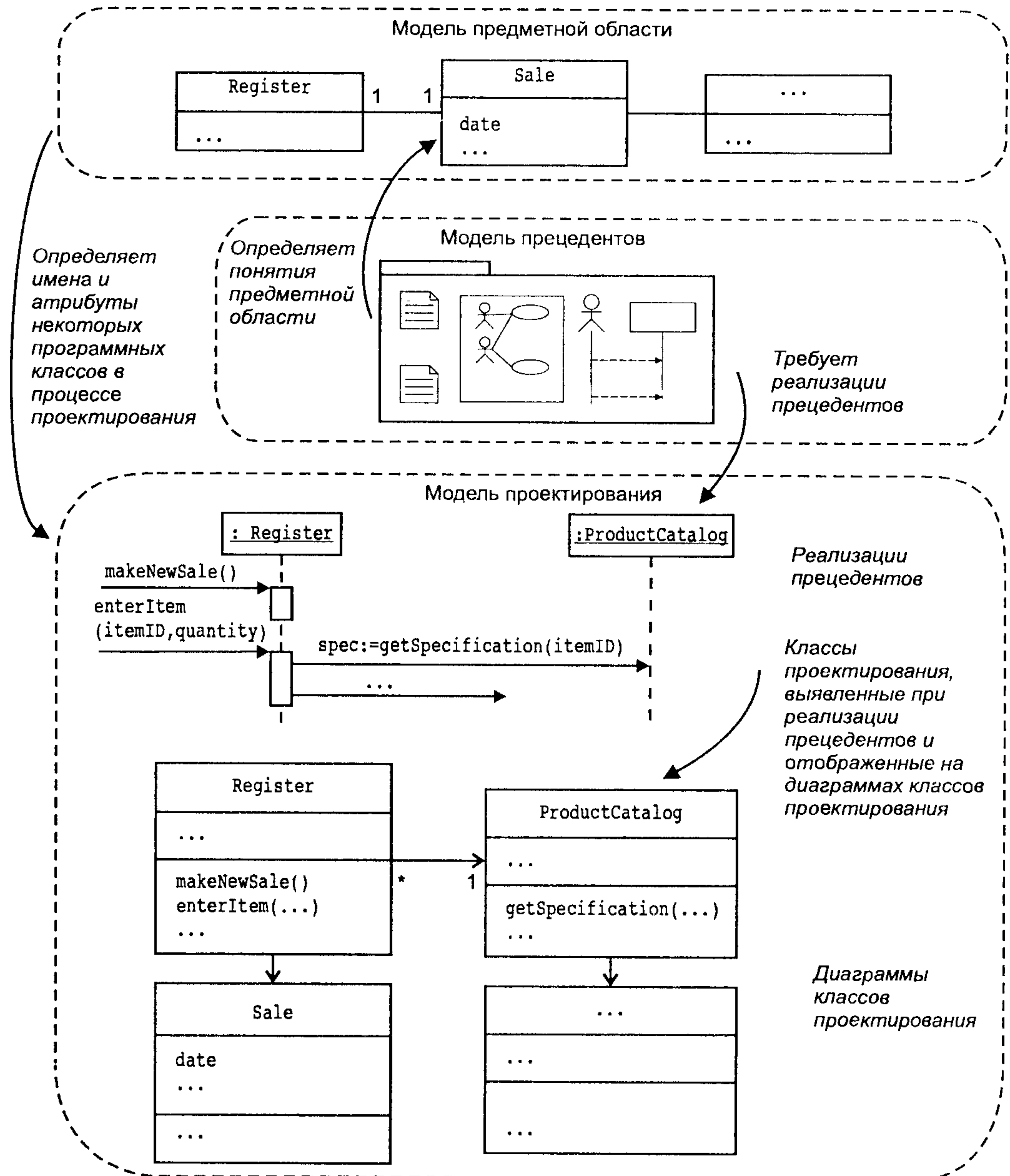



*Рис. 19.15. Пример взаимосвязи артефактов UP*

Глава 20

# Модель реализации: преобразование результатов проектирования в программный код

*Следите за ошибками в приведенном коде: я лишь доказал его корректность, но не протестировал.*

*Дональд Кнут (Donald Knuth)*

**Основная задача**

- **Преобразовать артефакты проектирования в код на объектно-ориентированном языке программирования.**

## Введение

После завершения проектирования диаграмм классов и взаимодействия на текущей итерации разработки приложения NextGen остался еще один существенный момент — генерация кода для объектов уровня предметной области.

При этом в процессе генерации кода в качестве входной информации будут использоваться созданные на стадии проектирования артефакты языка UML — диаграммы взаимодействия и диаграммы классов.

В контексте UP существует понятие "модель реализации". К этой модели относятся такие артефакты реализации, как исходный код, структура баз данных, страницы JSP/XML/HTML и т.п. Код, который будет рассмотрен в этой главе, тоже составляет часть модели реализации.

### *Выбор языка программирования*

Рассматриваемые примеры написаны на языке Java. Такой выбор объясняется популярностью и широким распространением этого языка. Однако автор не выделяет язык Java среди других языков. Языки C#, Visual Basic, C++, Smalltalk, Python и

многие другие тоже удовлетворяют принципам объектного проектирования и могут использоваться для преобразования разработанной модели в исходный код.

## 20.1. Программирование и процесс разработки

Выполнение предварительного проектирования совсем не означает, что в процессе программирования нельзя выполнять макетирование и проектирование. Современные средства разработки предоставляют прекрасную среду для быстрого изучения альтернативных подходов, а некоторые из них (или даже многие) позволяют сочетать процессы проектирования и программирования.

Однако некоторые разработчики считают, что до начала этапа программирования желательно разработать базовые визуальные модели. Это особенно полезно тем разработчикам, которые обладают "визуальным стилем мышления" и лучше воспринимают информацию, представленную в виде диаграмм.

*Совет*

Если одна итерация разработки занимает две недели, то, как минимум, полдня на начальном этапе этой итерации нужно посвятить визуальному моделированию и проектированию, а только потом следует переходить к программированию. Используйте простые средства построения диаграмм, в том числе применяйте такие "инструменты", как доска и цифровая камера. Если вы знаете компьютерное CASE-средство, удовлетворяющее требованиям простоты, удобства и быстроты освоения, — тем лучше.

Написание кода на объектно-ориентированном языке программирования, наподобие Java или C#, не относится к процессу анализа или проектирования системы — это конечная цель проектирования. Артефакты, создаваемые в контексте UP в рамках модели проектирования, предоставляют часть информации, необходимой для генерирования кода.

Преимущество объектно-ориентированного подхода к анализу, проектированию и программированию в рамках UP состоит в том, что он обеспечивает полный цикл разработки системы — от формулировки требований до программной реализации. Артефакты последовательно трансформируются в артефакты следующей стадии разработки, постепенно обеспечивая превращение системы в работающее приложение. Не стоит ожидать, что этот процесс окажется гладким или механическим — он достаточно творческий и неоднозначный. Однако предлагаемый подход обеспечивает отправную точку для экспериментирования и обсуждения.

### Внесение изменений на стадии реализации

Значительная часть усилий и творческого потенциала была задействована на стадии проектирования. Как станет видно из последующего обсуждения, генерация программного кода является относительно механическим процессом преобразования.

Тем не менее, процесс программирования не сводится к примитивной генерации кода. Совсем наоборот: результаты, полученные на стадии проектирования, оказываются далеко не совершенными. В процессе программирования и тестирования наверняка потребуется внести многочисленные изменения, а также выявить и разрешить возникшие проблемы.

Артефакты проектирования будут составлять эластичное ядро, которое можно масштабировать, сохраняя при этом изящность и устойчивость, а также обеспечивая решение новых возникающих в процессе программирования проблем. Поэтому на стадии построения и тестирования будьте готовы к изменению проектных решений.

## Модификация кода и итеративный процесс

Преимущество итеративного и инкрементального процесса разработки заключается в том, что результаты предыдущей итерации могут служить начальными данными для последующей итерации (рис. 20.1). Таким образом, результаты последовательного анализа и проектирования непрерывно совершенствуются на последующих итерациях разработки. Например, если код, созданный на итерации N, отклоняется от результатов проектирования этой итерации (что почти наверняка произойдет), то проектные решения, основанные на этой реализации, могут использоваться в качестве входных данных для моделей анализа и проектирования итерации N+1.

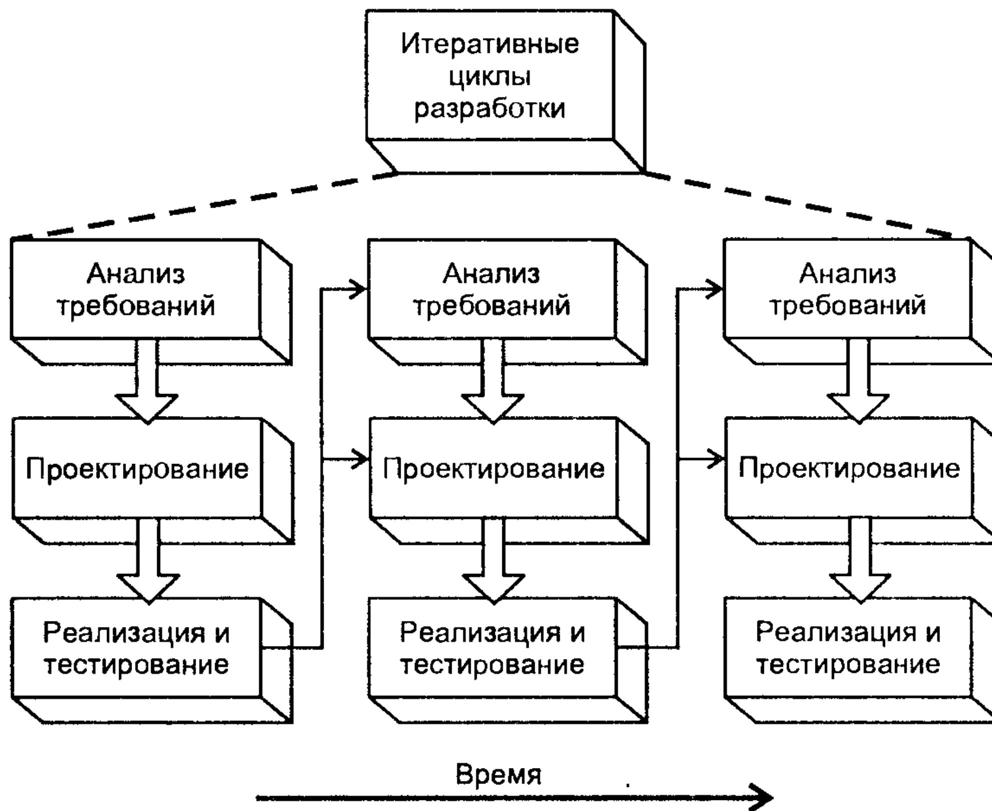



*Рис. 20.1. Влияние программной реализации на процесс проектирования последующих итераций*

Именно поэтому одним из самых ранних видов деятельности на каждой итерации разработки является синхронизация артефактов. Диаграммы итерации N могут не соответствовать результирующему программному коду итерации N. Прежде чем переходить к новому этапу анализа и проектирования, нужно выполнить синхронизацию.

## Изменение кода, CASE-средства и обратное проектирование

Очень желательно, чтобы диаграммы, полученные на стадии проектирования, были полуавтоматически обновлены и отражали изменения, внесенные на

соответствующей стадии кодирования. В идеале это делается с использованием CASE-средств, с помощью которых можно считывать исходный код и автоматически генерировать, например, диаграммы пакетов, классов и последовательностей. Это пример *обратного проектирования* (reverse engineering), когда логические модели генерируются на основе исходного (или даже исполняемого) кода.

## 20.2. Преобразование результатов проектирования в программный код

Реализация на объектно-ориентированном языке программирования требует написания исходного кода для:

- определений классов и интерфейсов;
- методов.

В последующих разделах обсуждается генерация таких определений на языке Java (как типичный случай).

## 20.3. Создание определений классов на основе диаграмм классов

Коротко можно сказать, что на диаграммах классов отображаются имена классов и интерфейсов, суперклассов, сигнатуры методов и простые атрибуты классов. Этого вполне достаточно для создания базового определения класса на объектно-ориентированном языке программирования. Ниже будет рассмотрен процесс добавления информации об интерфейсе и пространстве имен (или пакете), а также других данных.

### *Определение класса с методами и простыми атрибутами*

Как видно из рис. 20.2, преобразование диаграммы классов в базовые определения атрибутов (простые экземпляры переменных-членов на Java) и сигнатуры методов для определения класса `SalesLineItem` на языке Java выполняется очень просто.

Обратите внимание на добавление конструктора `SalesLineItem(...)`. Это сделано на основании того, что в диаграмме взаимодействия для системной операции `enterItem` объекту `SalesLineItem` передается сообщение `create(spec, qty)`. При переходе к языку Java это означает необходимость использования конструктора с двумя указанными параметрами. Метод `create` зачастую исключается из диаграммы классов, поскольку он является стандартным и имеет несколько интерпретаций, зависящих от используемого языка программирования.

### *Добавление атрибутов-ссылок*

*Атрибут-ссылка* (reference attribute) — это атрибут, ссылающийся на другой сложный объект, а не на простой тип, такой как `String`, `Number` и т.д.

| На диаграмме классов атрибуты-ссылки представлены ассоциациями и связанным с ними направлением перемещения. |
|---|

Например, класс SalesLineItem имеет ассоциацию, направленную к классу ProductSpecification. Обычно эта ассоциация интерпретируется как атрибут-ссылка класса SalesLineItem, ссылающаяся на экземпляр ProductSpecification (рис. 20.3).

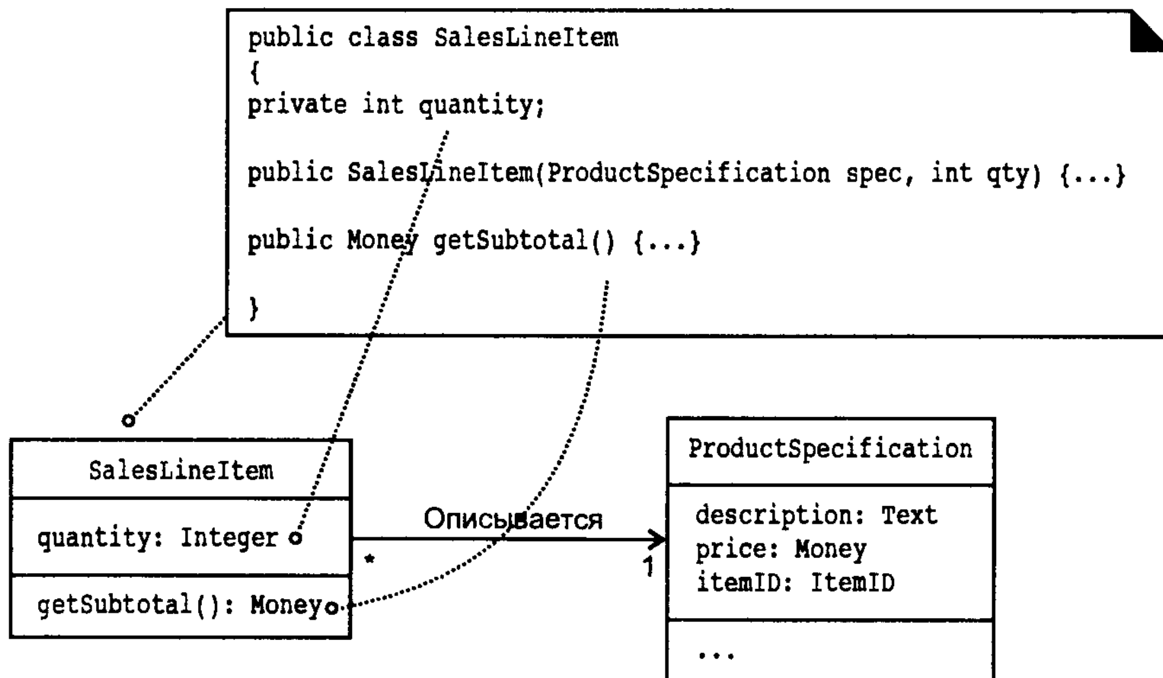



*Рис. 20.2. Класс SalesLineItem на языке Java*

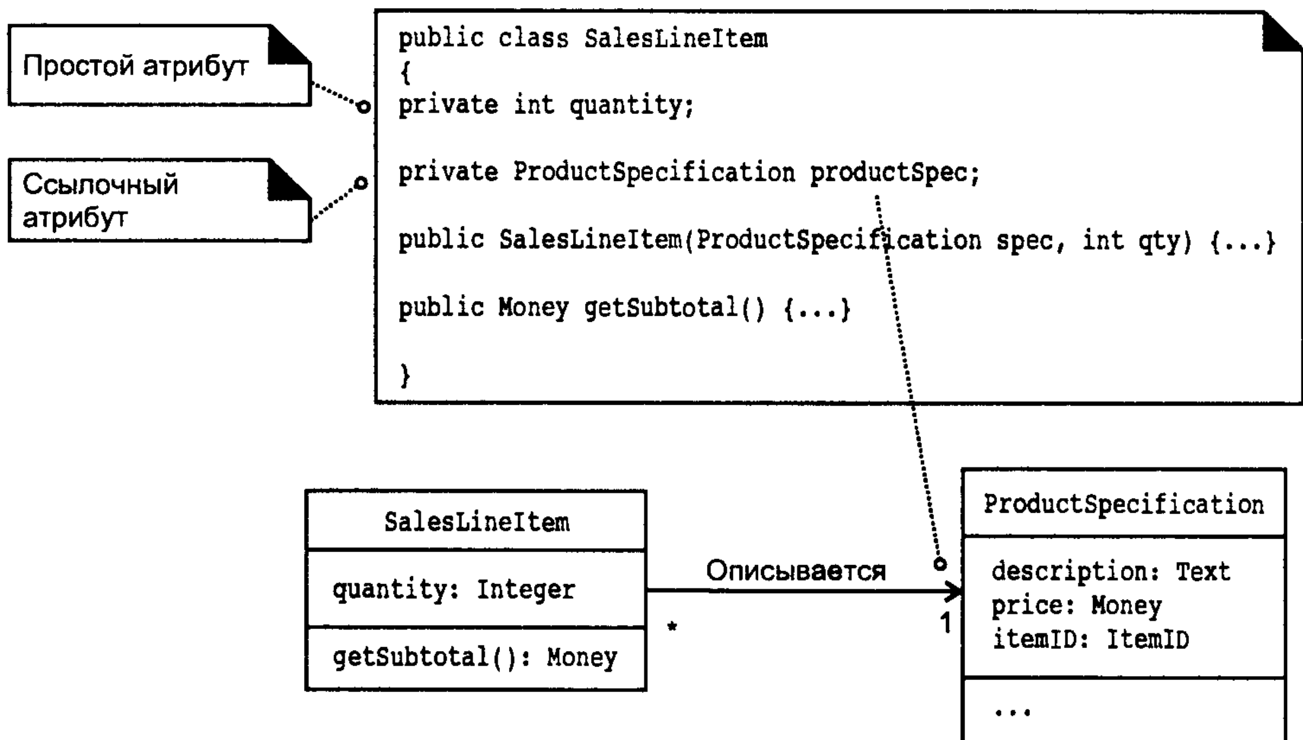



*Рис. 20.3. Добавление атрибутов-ссылок*

На языке Java это означает добавление поля, ссылающегося на экземпляр ProductSpecification.

| Обратите внимание, что атрибуты-ссылки классов зачастую косвенно присутствуют, а не явно определяются на диаграмме классов. |
|---|

Например, хотя в определение класса SalesLineItem на языке Java был добавлен экземпляр переменной, указывающей на экземпляр ProductSpecification, на диаграмме классов в разделе атрибутов явно объявленный атрибут отсутствует. Это объясняется предполагаемой видимостью атрибута, задаваемой ассоциацией и ее направлением. На стадии генерации кода эта ассоциация явно определяется как атрибут.

### Атрибуты-ссылки и имена ролей

Рассмотрим имена ролей на статической структурной диаграмме. Каждый конец линии ассоциации называется ролью. Если говорить кратко, *имя роли* (role name) — это имя, идентифицирующее роль и обеспечивающее некоторый семантический контекст, которой иллюстрирует его природу.

Если имя роли присутствует на диаграмме классов, то при генерации кода его нужно использовать в качестве основы имени атрибута-ссылки (рис. 20.4).

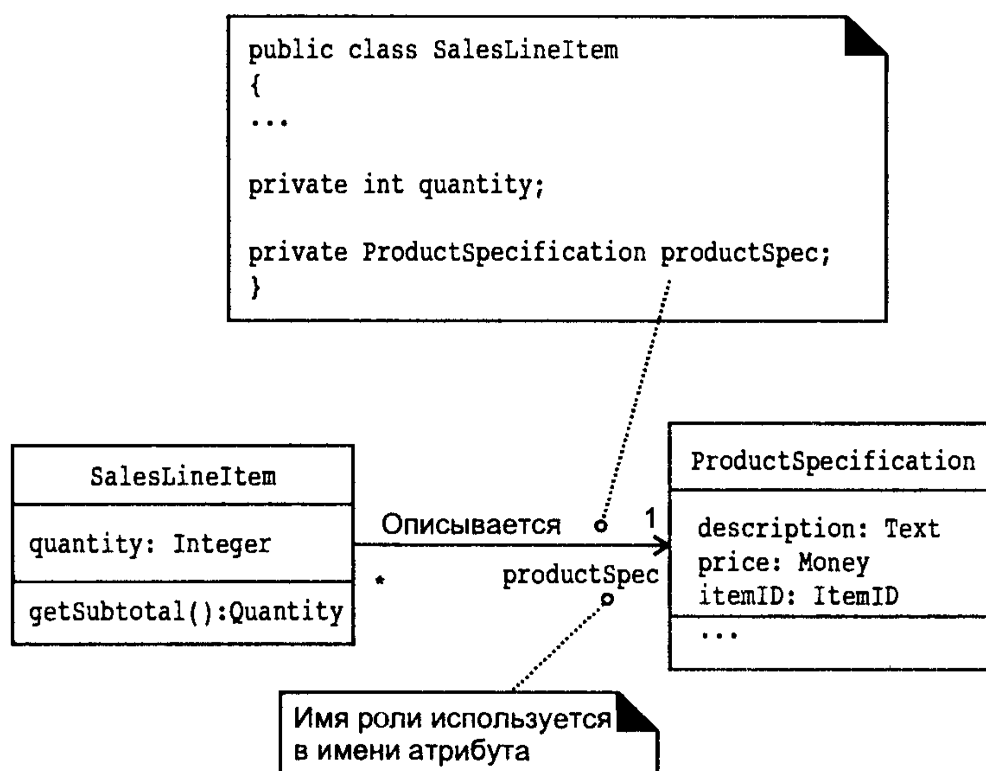



*Рис. 20.4. Имена ролей можно использовать для генерации имен экземпляров переменных*

### Отображение атрибутов

На примере класса Sale можно удостовериться в неоднозначности отображения атрибутов диаграммы проектирования в исходный код. На рис. 20.5 показаны возможные проблемы, связанные с таким преобразованием.

## 20.4. Создание методов на основе диаграмм взаимодействия

На диаграммах взаимодействия представлены сообщения, которые передаются в ответ на вызов метода. Последовательность этих сообщений преобразуется в серию операторов в определении метода. Для иллюстрации определения метода

enterItem на языке Java можно использовать диаграмму взаимодействия системной операции enterItem (рис. 20.6).

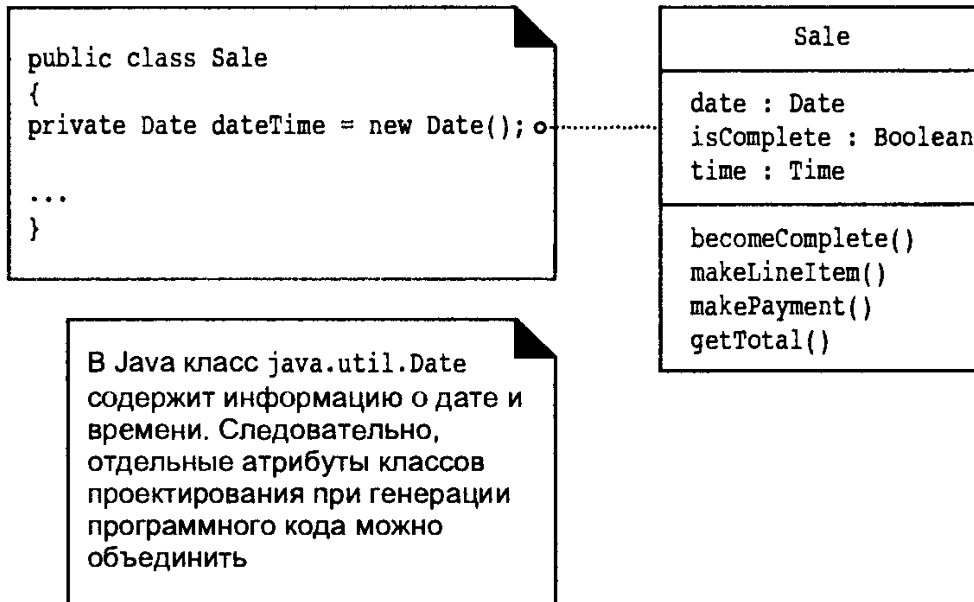



*Рис. 20.5. Преобразование в исходный код на языке Java атрибутов даты и времени*

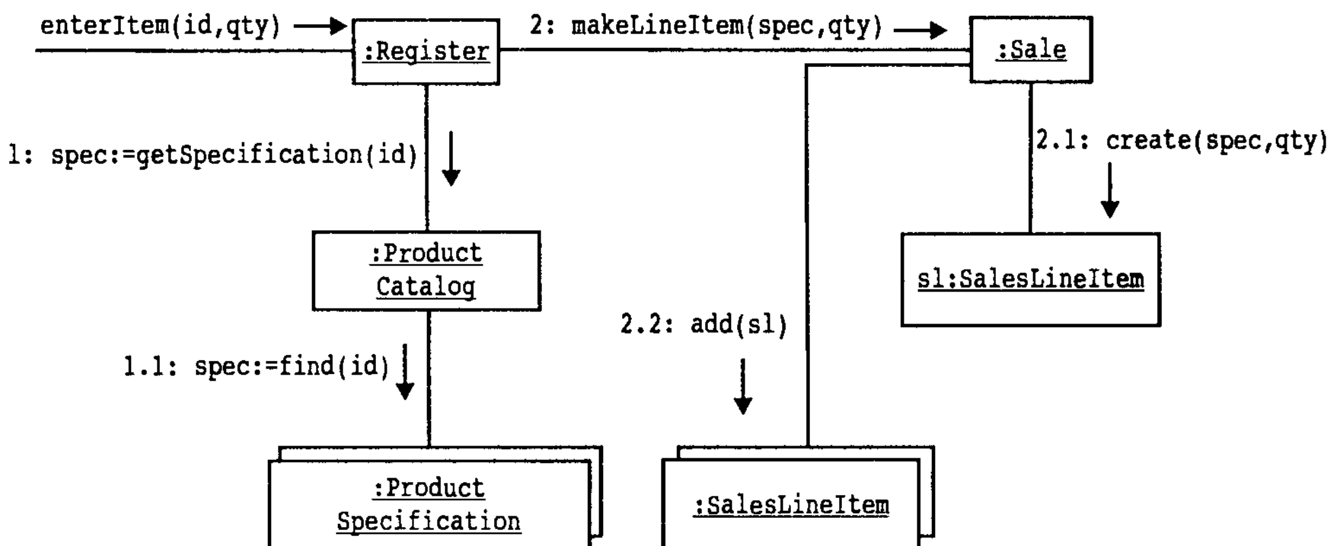



*Рис. 20.6. Диаграмма взаимодействия для системной операции enterItem*

В приведенном примере будет использован класс Register. Его определение на языке Java представлено на рис. 20.7.

## Метод Register--enterItem

Сообщение enterItem передается экземпляру объекта Register. Следовательно, метод enterItem определяется в этом классе.

```
public void enterItem(ItemID itemID, int qty)
```

**Сообщение 1.** Для получения объекта ProductSpecification объекту ProductCatalog передается сообщение getSpecification.

```
ProductSpecification spec =
        catalog.getSpecification(itemID);
```

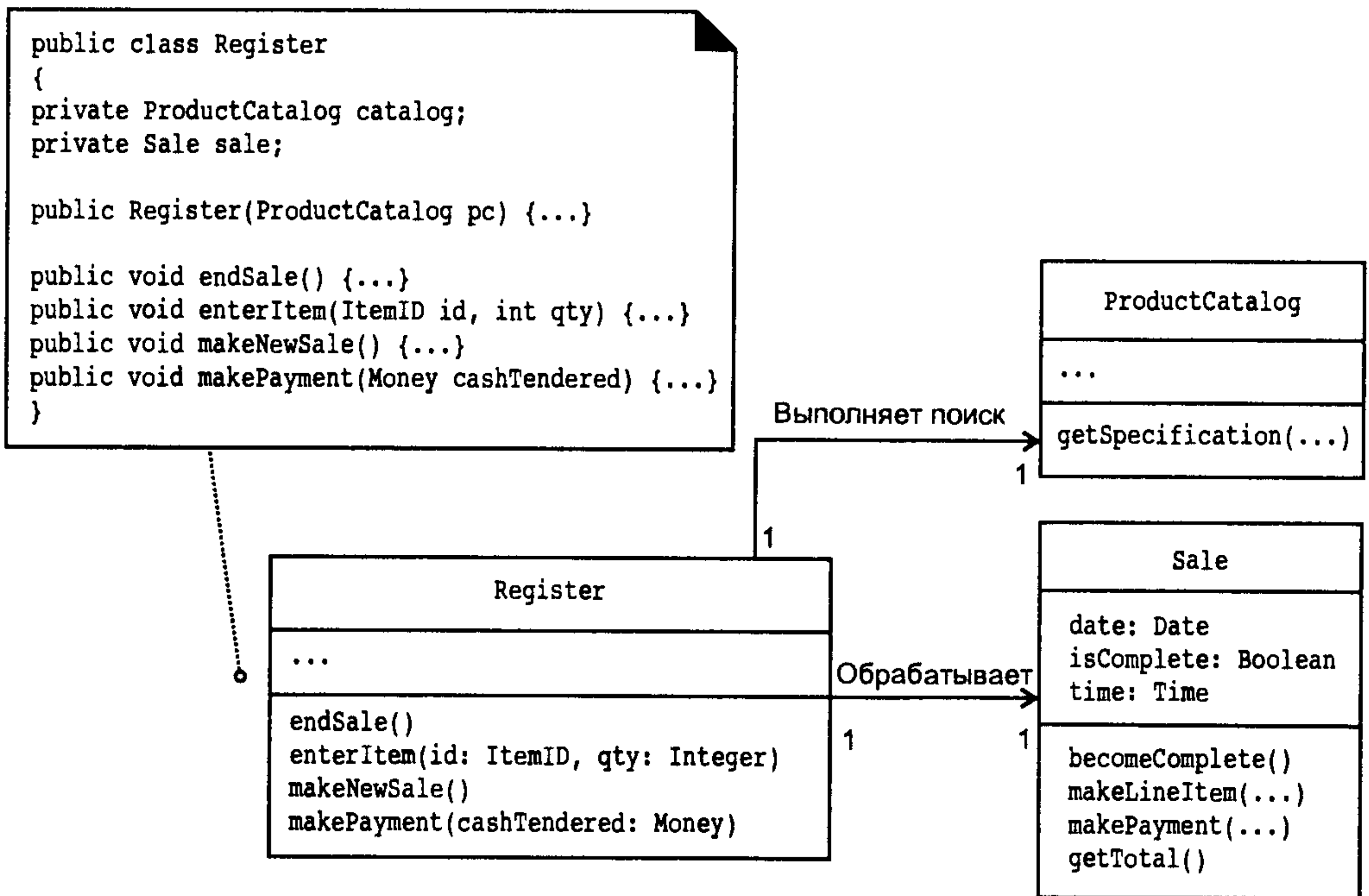



*Рис. 20.7. Класс* `Register`

**Сообщение 2.** Объекту `Sale` передается сообщение `makeLineItem`.

```
sale.makeLineItem(spec, qty);
```

Подводя итоги, можно еще раз повторить, что каждое сообщение из последовательности внутри метода, как показано на диаграмме взаимодействия, преобразуется в оператор метода на языке Java.

Полученный метод `enterItem` и его связь с диаграммой взаимодействия представлены на рис. 20.8.

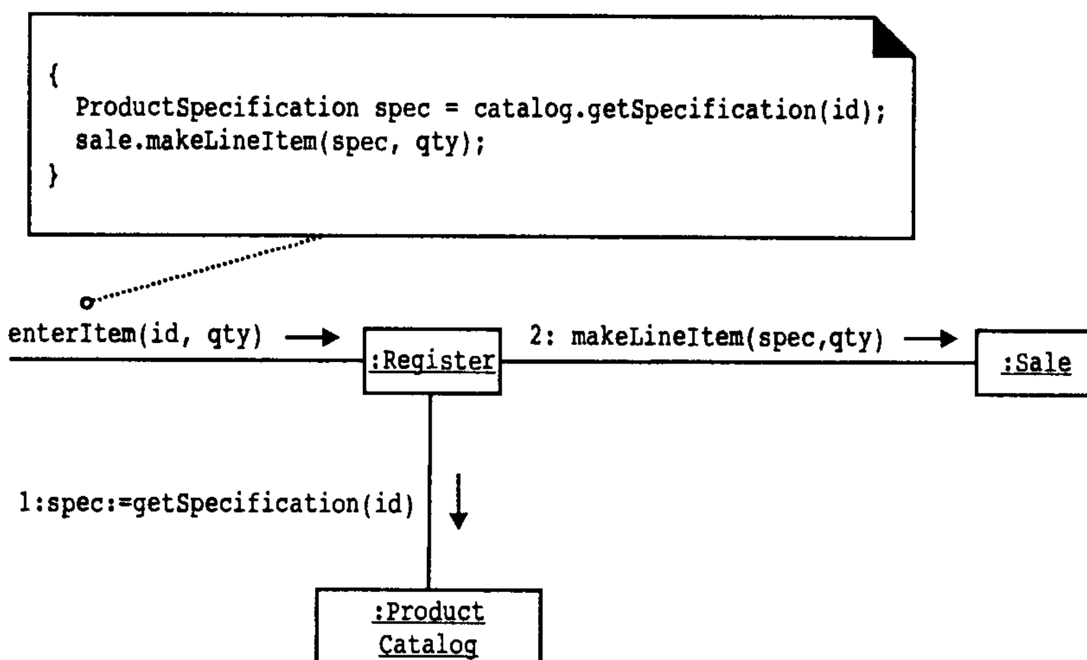



*Рис. 20.8. Метод* `enterItem`

## 20.5. Классы-контейнеры в программном коде

Зачастую для объекта необходимо обеспечить видимость группы других объектов. Обычно это следует непосредственно из значения кратности, указанного на диаграмме классов. Оно может превышать единицу. Например, для объекта Sale необходимо обеспечить видимость группы экземпляров класса SalesLineItem (рис. 20.9).

В объектно-ориентированных языках программирования такие взаимосвязи в основном реализуются с помощью промежуточного контейнера или коллекции объектов. В классе, с которым связано единичное значение кратности, определяется атрибут-ссылка, содержащий указатель на экземпляр контейнера/коллекции. А в самом контейнере содержатся экземпляры класса, для которого значение кратности превышает единицу.

Например, в библиотеках Java имеются такие контейнерные классы, как ArrayList и HashMap, реализующие интерфейсы List и Map, соответственно. При использовании класса ArrayList в классе Sale можно определить атрибут, указывающий на упорядоченный список экземпляров SalesLineItem.

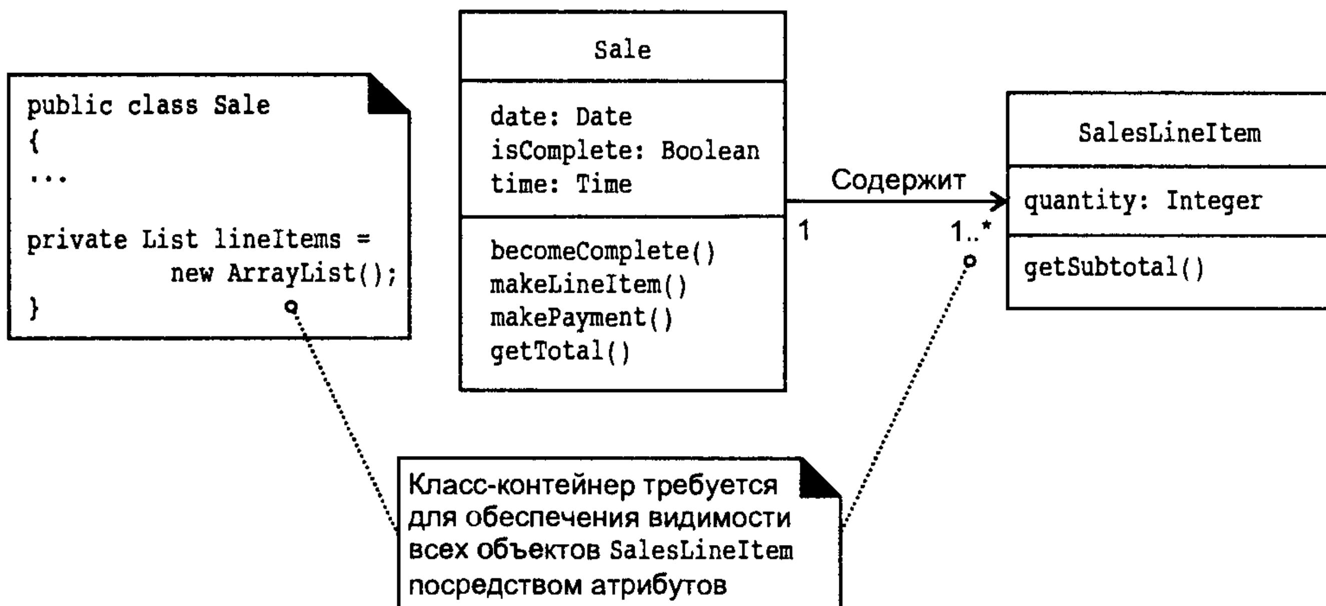



*Рис. 20.9. Добавление контейнера*

Выбор подходящего контейнерного класса определяется реальными требованиями. Если планируется применять поиск по ключу, то лучше воспользоваться классом Map; при организации расширяемого упорядоченного списка потребуется класс List и т.д.

## 20.6. Исключения и обработка ошибок

До сих пор обработка ошибок в процессе разработки игнорировалась, поскольку мы рассматривали основные вопросы распределения обязанностей и объектно-ориентированного проектирования. Однако при разработке реального приложения на стадии проектирования, а особенно реализации, не лишним будет рассмотреть также вопросы обработки ошибок.

В языке UML для обозначения исключений используются асинхронные сообщения на диаграммах взаимодействия. Эти вопросы более подробно рассматриваются в главе 33.

## 20.7. Определение метода Sale--makeLineItem

В качестве заключительного примера рассмотрим метод makeLineItem класса Sale, который может быть описан на основе анализа диаграммы взаимодействия системной операции enterItem. Фрагмент диаграммы взаимодействия и соответствующий метод на языке Java представлены на рис. 20.10.

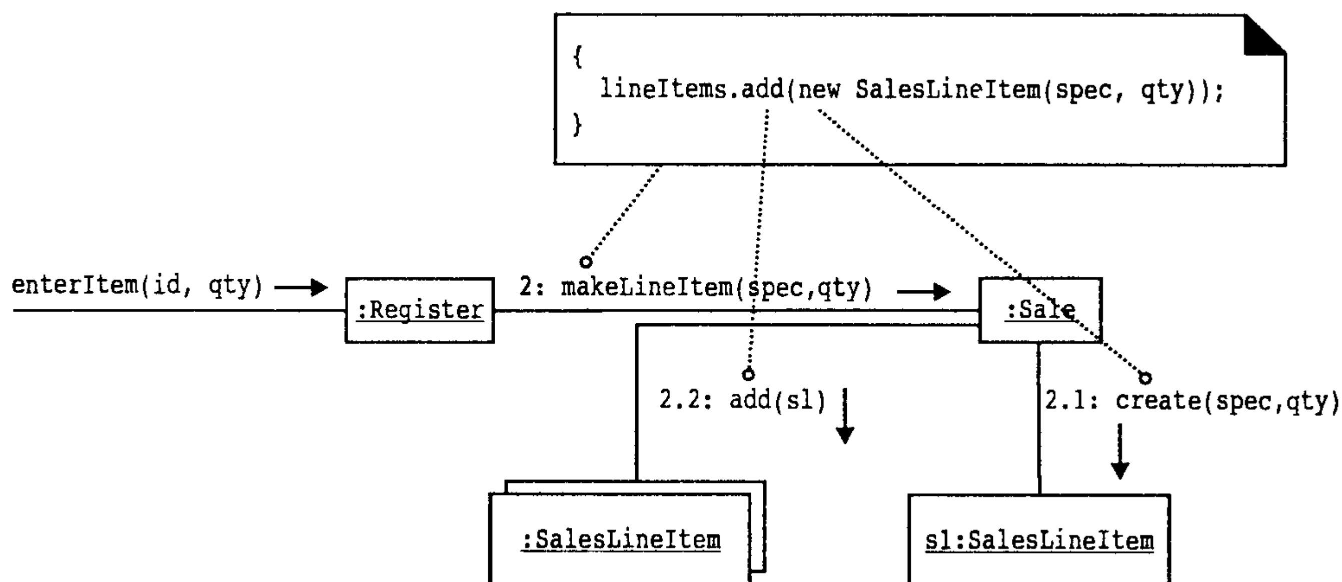



*Рис. 20.10. Метод Sale--makeLineItem*

## 20.8. Порядок реализации

Классы нужно реализовывать (и, в идеале, полностью тестировать в рамках модулей), начиная от минимально связанных с другими классами до максимально связанных (рис. 20.11). Например, первым можно реализовать класс Payment либо ProductSpecification. Следующими должны реализовываться те классы, которые зависят от уже реализованных классов: класс ProductCatalog либо SaleLineItem.

## 20.9. Программирование на основе тестирования

В рамках метода XP (Extreme Programming) [11] предложен применимый к унифицированному процессу разработки подход к *программированию на основе тестирования* (test-first programming). В рамках этого подхода код тестов для каждого модуля пишется раньше, чем сам тестируемый код. При этом разработчик разрабатывает код тестов для *всех* модулей приложения. При этом процесс реализации происходит следующим образом. Разработчик пишет код для небольшого теста, затем реализует часть приложения, тестирует его с помощью созданного теста, пишет код для следующего теста и т.д.

Такой подход имеет следующие преимущества.

- **Тесты для модулей действительно разрабатываются.** Разработчики (особенно программисты) стремятся избежать необходимости программирования тестов или оставить эту работу на более поздний срок.
- **Программа – результат человеческих усилий.** Если разработчик реализует код приложения, неформально его тестирует, а после этого вынужден разрабаты-

вать формальные тесты — он не получает удовлетворения от своей деятельности. Если же сначала создаются тесты, затем реализуется код приложения, который должен удовлетворить написанным ранее тестам, то разработчик после прохождения теста получает "некоторое удовлетворение". Этот философский аспект не стоит сбрасывать со счетов, поскольку программа — это результат человеческих усилий.

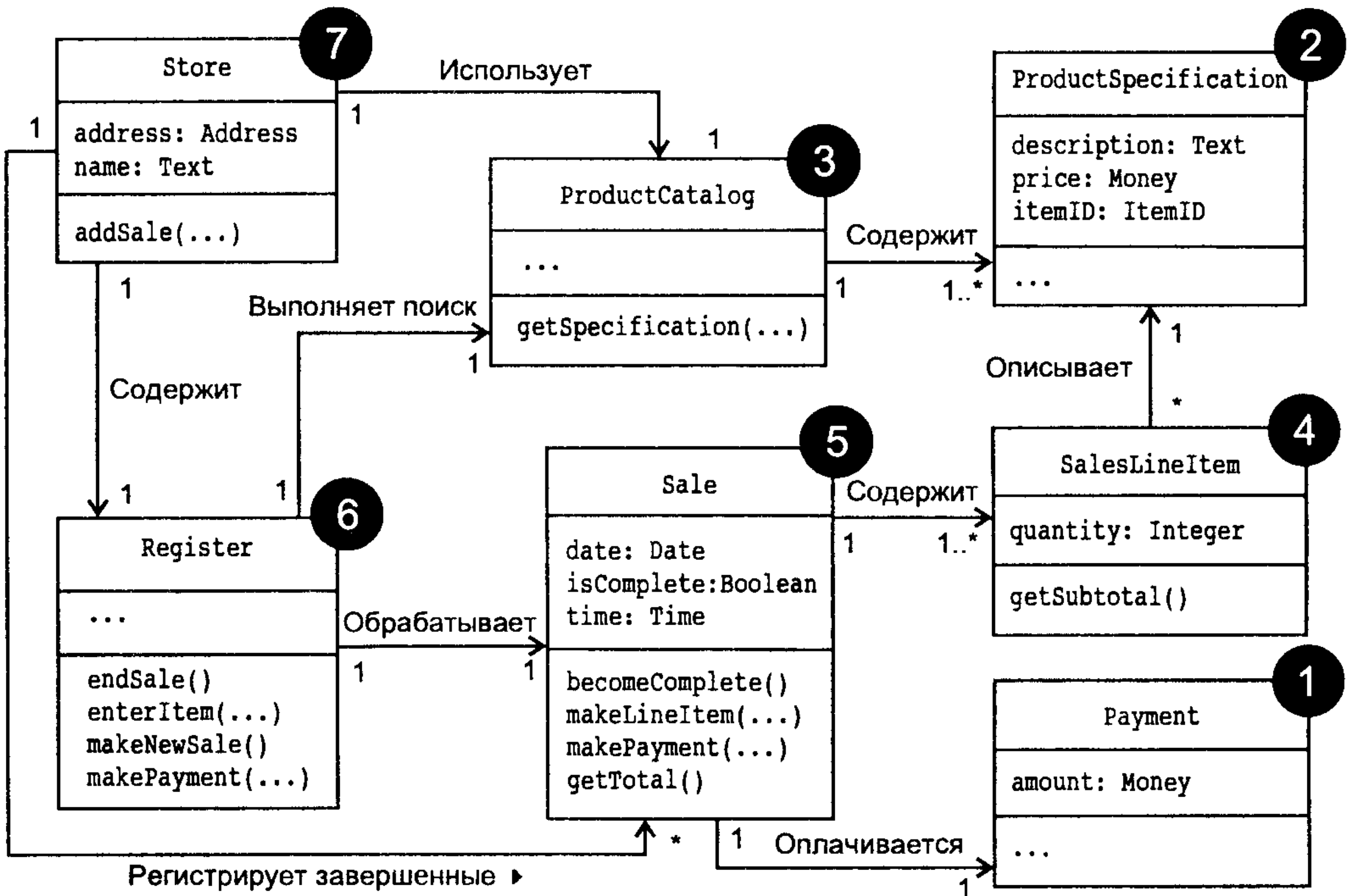



*Рис. 20.11. Одна из возможных последовательностей реализации классов*

- **Определение интерфейса и поведения.** Зачастую точный открытый интерфейс и поведение классов трудно определить окончательно до завершения программной реализации. Если сначала разрабатывать код для тестов, то в процессе его создания прояснится проектное решение класса.
- **Обоснованная верификация.** Очевидно, наличие сотен и тысяч модульных тестов обеспечивает некоторую гарантию корректности кода.
- **Верификация кода при возможных изменениях.** Если сначала разрабатываются сотни или тысячи модульных тестов, то в итоге для каждого класса приложения существует свой отдельный тест. Если разработчик пожелает внести изменения в существующий код, то модифицированный код он сможет проверить с помощью готового теста. При этом сразу же будут выявлены все несоответствия и привнесенные ошибки.

В качестве примера можно привести популярное, простое и бесплатное средство JUnit (`www.junit.org`) для модульного тестирования кода на Java. Воспользуемся этим средством для программирования класса `Sale` на основе тестирования. Тогда до начала программной реализации класса `Sale` нужно раз-

работать модульный тест, а именно написать соответствующий метод класса `SaleTest`, выполняющий следующие действия.

1. Создание нового экземпляра продажи.
2. Добавление к нему некоторого выбранного товара.
3. Запрос на вычисление общей стоимости покупки и проверку корректности ее вычисления.

Модульный тест может выглядеть следующим образом.

```
public class SaleTest extends TestCase
{
//...
    public void testTotal()
    {
        //создание экземпляра новой продажи
        Money total = new Money( 7.5 );
        Money price = new Money( 2.5 );
        ItemID id = new ItemID(1);
        ProductSpecification spec;
        spec = new ProductSpecification(id, price, "товар 1");
        Sale sale = new Sale();

        // добавление выбранных товаров
        sale.makeLineItem( spec, 1);
        sale.makeLineItem( spec, 2);

        // проверка правильности вычисления общей стоимости покупки
        assertEquals(sale.getTotal(), total);
    }
}
```

Только после разработки класса `SaleTest` можно приступать к реализации класса `Sale`, который должен удовлетворять созданному тесту. Однако не все методы тестирования нужно реализовывать заранее. Разработчик сначала должен написать один метод тестирования, затем реализовать для него соответствующий код приложения, после этого приступать к написанию следующего теста и т.д.

## 20.10. Заключительные замечания по поводу преобразования проектного решения в код

Процесс преобразования объектно-ориентированных диаграмм классов в их определения и диаграмм взаимодействия в методы является относительно простым. При этом на стадии программирования разработчик по-прежнему имеет достаточную свободу действий, чтобы принимать дополнительные решения и изменять уже принятые. Однако общая архитектура и основные решения должны быть приняты до перехода к кодированию.

## 20.11. Основное программное решение

В этом разделе приводится пример программного описания объектов уровня предметной области на языке Java для данной итерации. Генерация кода во многом определяется диаграммами классов и диаграммами взаимодействия, раз-

работанными ранее на стадии проектирования, и основывается на принципах их отображения в исходный код.

> Основной смысл приведенного примера заключается в том, что существует относительно простой способ преобразования артефактов проектирования в "скелет" программного кода. Этот код определяет контрольный пример, который не может претендовать на устойчивую, полноценную программу на Java с синхронизацией, обработкой исключений и т.д.

### Класс `Payment`

```
public class Payment
{
    private Money amount;
    public Payment( Money cashTendered ) { amount = cashTendered; }
    public Money getAmount() { return amount; }
}
```

### Класс `ProductCatalog`

```
public class ProductCatalog
{
    private Map productSpecifications = new HashMap();

    public ProductCatalog()
    {
        // тестовые данные
        ItemID id1 = new ItemID( 100 );
        ItemID id2 = new ItemID( 200 );
        Money price = new Money( 3 );

        ProductSpecification ps;
        ps = new ProductSpecification( id1, price, "товар 1" );
        productSpecifications.put( id1, ps );
        ps = new ProductSpecification( id2, price, "товар 2" );
        productSpecifications.put( id2, ps );
    }

    public ProductSpecification getSpecification( ItemID id )
    {
        return (ProductSpecification)
        productSpecifications.get( id );
    }
}
```

### Класс `Register`

```
public class Register
{
    private ProductCatalog catalog;
    private Sale sale;
```

```
    public Register( ProductCatalog catalog )
    {
        this.catalog = catalog;
    }

    public void endSale()
    {
        sale.becomeComplete();
    }

    public void enterItem( ItemID id, int quantity )
    {
        ProductSpecification spec =
            catalog.getSpecification( id );
        sale.makeLineItem( spec, quantity );
    }

    public void makeNewSale()
    {
        sale = new Sale();
    }

    public void makePayment( Money cashTendered )
    {
        sale.makePayment( cashTendered );
    }
}
```

**Класс ProductSpecification**

```
public class ProductSpecification
{
    private ItemID id;
    private Money price;
    private String description;

    public ProductSpecification
        ( ItemID id, Money price, String description )
    {
        this.id = id;
        this.price = price;
        this.description = description;
    }

    public ItemID getItemID() { return id; }

    public Money getPrice() { return price; }

    public String getDescription() { return description; }
}
```

**Класс Sale**

```
public class Sale
{
    private List lineItems = new ArrayList();
    private Date date = new Date();
    private boolean isComplete = false;
    private Payment payment;

    public Money getBalance()
    {
        return payment.getAmount().minus( getTotal() );
    }

    public void becomeComplete() { isComplete = true; }

    public boolean isComplete() { return isComplete; }

    public void makeLineItem
        ( ProductSpecification spec, int quantity )
    {
        lineItems.add( new SalesLineItem(spec, quantity) );
    }

    public Money getTotal()
    {
        Money total = new Money();
        Iterator i = lineItems.iterator();
        while( i.hasNext() )
        {
            SalesLineItem sli = (SalesLineItem) i.next();
            total.add( sli.getSubtotal() );
        }
        return total;
    }

    public void makePayment(Money cashTendered)
    {
        payment = new Payment(cashTendered);
    }
}
```

**Класс SalesLineItem**

```
public class SalesLineItem
{
    private int quantity;
    private ProductSpecification productSpec;

    public SalesLineItem
    ( ProductSpecification spec, int quantity )
    {
        this.productSpec = spec;
```

```
        this.quantity = quantity;
    }

    public Money getSubtotal()
    {
    return productSpec.getPrice().times( quantity );
  }
}
```

**Класс Store**

```
public class Store
{
    private ProductCatalog catalog =
        new ProductCatalog();
    private Register register = new Register( catalog );

    public Register getRegister() { return register; }
}
```

# Часть IV

# Вторая итерация фазы развития

Глава

# Вторая итерация и требования к ней

Основная задача

- Определить требования для второй итерации фазы развития.

## 21.1. Вторая итерация: объектное проектирование и шаблоны

В главах, посвященных начальной фазе и первой итерации фазы развития, рассматривался ряд фундаментальных вопросов анализа и объектного проектирования, на которых базируется построение объектных систем.

При описании данной итерации основное внимание будет уделяться следующим вопросам.

- Базовое объектное проектирование
- Использование шаблонов для получения единого проектного решения
- Применение UML для визуализации моделей

Эти вопросы позволяют решить основные задачи данной книги и получить навыки грамотного проектирования.

Сначала кратко будет описан этап анализа требований и построения модели предметной области, затем более пристальное внимание будет уделено вопросам проектирования. При описании первой итерации были изложены основные идеи и принципы проектирования.

Конечно, на данной итерации выполняется и множество других функций, связанных с анализом, проектированием и реализацией системы, однако они не будут затронуты в данной книге, поскольку не оказывают решающего влияния на процесс проектирования объектов.

## 21.2. От первой ко второй итерации

К моменту завершения первой итерации получены следующие результаты.

- Вся существующая на данный момент программа протестирована с помощью модульного теста, теста загрузки, с точки зрения удобства использова-

ния и т.д. Одна из основных идей UP состоит в ранней, реалистичной и непрерывной верификации качества и корректности кода. Ранняя обратная связь обеспечивает возможность адаптации и модификации системы и нахождения "пути истинного".

- Пользователи регулярно привлекались к оцениванию разработанной части системы, обеспечению обратной связи с целью определения требований и адаптации системы. Таким образом пользователи знают о текущем состоянии разработки системы.
- Все подсистемы интегрированы между собой и стабилизированы.

Многие процессы в рамках первой итерации и перехода ко второй итерации не будут рассмотрены детально, поскольку основное внимание в этой книге уделяется вопросам ООА/П. Кратко остановимся на некоторых из них.

- В начале новой итерации с помощью CASE-средства выполняется обратное проектирование диаграмм UML на основе исходного кода, написанного на предыдущей итерации (в результате обновляется модель проектирования UP). Полученную модель можно распечатать с высоким разрешением на плоттере и повесить на стене для иллюстрации отправной точки для логического проектирования на следующей итерации.
- Продолжается анализ удобства использования и разработка пользовательского интерфейса. Это чрезвычайно важный этап, определяющий успех многих приложений. Однако разработка интерфейса пользователя — это отдельный и нетривиальный вопрос, изложение которого не входит в тему данной книги.
- Продолжается моделирование и разработка базы данных.
- К концу предыдущей итерации были сформулированы требования для следующей итерации.
- Проводится очередной двухдневный семинар, в течение которого выполняется развернутое описание следующих прецедентов. В течение фазы развития должны быть реализованы 10% наиболее важных прецедентов с высокой степенью риска. Одновременно происходит более глубокое изучение и определение 80% прецедентов системы, большая часть которых будет реализована на стадии конструирования.
  - К участию в семинаре привлекаются несколько разработчиков (в том числе архитектор программной системы), выполнявших реализацию системы на первой итерации. Тогда на семинаре будет учитываться реальное состояние дел. Ключевая идея UP и итеративной разработки состоит в том, что для прояснения требований не разрабатывается никакое дополнительное программное обеспечение.

### Упрощения, принятые при рассмотрении примера

В реальных проектах в рамках UP требования к первым итерациям разрабатываются с учетом риска и реального значения каждого прецедента в системе. Сначала реализуются наиболее важные и критичные из них. Однако при рассмотрении основного примера данной книги будет сделано отступление от этого принципа. Оно объясняется необходимостью проиллюстрировать фундаментальные идеи и принципы ООА/П на начальных итерациях. Поэтому для реализации на начальных итера-

циях будут выбраны те требования, которые максимально способствуют решению задачи обучения читателей, а не достижению целей проекта.

## 21.3. Требования для второй итерации

На второй итерации разработки POS-приложения NextGen будут реализованы следующие требования.

1. Поддержка различных внешних служб. Например, необходимо обеспечить возможность подключения к системе различных программ для вычисления налоговых платежей с различными интерфейсами, различных бухгалтерских программ и т.д. Все внешние программы имеют разные программные интерфейсы, и их базовые функции реализуются по различным протоколам.
2. Сложные правила вычисления стоимости.
3. Подключаемые бизнес-правила.
4. Обновление окна интерфейса пользователя при изменении общей стоимости покупки.

Эти требования будут рассмотрены в рамках реализации сценариев прецедента `Оформление продажи`.

Заметим, что эти требования не новы. Они были определены на начальной стадии реализации проекта. Например, проблема вычисления общей стоимости упоминалась в исходном описании прецедента `Оформление продажи`.

**Основной успешный сценарий**

1. Покупатель подходит к кассовому аппарату POS-системы с выбранными товарами.
2. Кассир открывает новую продажу.
3. Кассир вводит идентификатор товара.
4. Система записывает наименование товара и выдает его описание, цену и общую стоимость. **Цена вычисляется на основе набора правил.**

...

Более того, в дополнительной спецификации содержится раздел, в котором описаны правила для вычисления стоимости и указана необходимость поддержки различных внешних систем.

**Дополнительная спецификация**

...

***Программные интерфейсы***

Для большинства внешних систем (вычисления налоговых платежей, системы складского учета, бухгалтерской системы и т.д.) необходимо обеспечить возможность подключения через различные интерфейсы.

...

**Бизнес-правила**

| Название | Правило | Возможность изменения | Источник |
|---|---|---|---|
| ПРАВ4 | Правила вычисления скидок (примеры)<br>Работник компании — скидка 20%<br>Привилегированный покупатель — 10% | Высокая вероятность изменения. Каждая торговая организация устанавливает свои скидки | Политика торговых организаций |
| ... | ... | | |

### Информация из предметной области

***Ценовая политика***

Помимо описанных выше правил ценообразования, каждому товару могут соответствовать две цены: *исходная* и *постоянная сниженная цена*. Указанная цена товара (без вычисления положенных скидок) соответствует постоянной сниженной цене, если таковая имеется. Однако организации сохраняют исходную цену даже при наличии постоянной сниженной для бухгалтерской отчетности.

---

### Инкрементальная разработка одного и того же прецедента в течение нескольких итераций

На основе этих требований на второй итерации продолжается реализация прецедента `Оформление продажи`. Однако теперь реализуются новые сценарии. Таким образом система постепенно разрастается. Ситуация, когда различные сценарии одного и того же прецедента рассматриваются в течение нескольких итераций, встречается довольно часто. Система постепенно расширяется до тех пор, пока не будут реализованы все требования. В то же время, простые прецеденты могут быть полностью реализованы в течение одной итерации.

На первой итерации были приняты некоторые упрощения, чтобы проблема не оказалась слишком сложной для решения в течение одной итерации. По этой же причине для второй итерации было определено лишь несколько требований.

В реальном проекте для второй итерации могли быть выбраны совсем другие требования, относящиеся к обновлению информации в системе складского учета, реализации платежей по кредитной карточке или новым прецедентам. Однако выбор требований в данной книге определялся задачей обучения читателя.

## 21.4. Уточнение артефактов стадии анализа

### Модель прецедентов: прецеденты

Никакая модификация описания прецедентов с точки зрения требований для данной итерации не выполняется.

Однако на этой итерации, помимо проектирования и реализации, проводится семинар по определению новых требований к системе и развернутому описанию новых прецедентов. Описанные ранее прецеденты (например, `Оформление продажи`), возможно, тоже будут пересмотрены на основе полученного на первой итерации опыта разработки. Некоторые из них будут рассмотрены на последующих итерациях фазы развития, а большинство будет реализовано на стадии конструирования (поскольку они не являются критичными и архитектурно значимыми).

### Модель прецедентов: диаграммы последовательностей

На этой итерации будет добавлена поддержка внешних систем с различными интерфейсами, в том числе системы вычисления налоговых платежей. POS-система NextGen будет удаленно взаимодействовать с внешними системами. Следовательно, необходимо обновить диаграммы последовательностей и отразить на них межсистемное взаимодействие с целью прояснения специфики новых системных событий.

На рис. 21.1 показана диаграмма последовательностей для одного сценария оплаты по кредитной карточке, в рамках которого происходит взаимодействие с несколькими внешними системами. И хотя вопросы оплаты по кредитной карточке не рассматриваются на данной итерации, разработчик (автор книги) по-

строил для этого сценария диаграмму последовательностей (а может быть и несколько диаграмм), чтобы лучше понять специфику межсистемного взаимодействия и обеспечить поддержку внешних систем с разными интерфейсами.

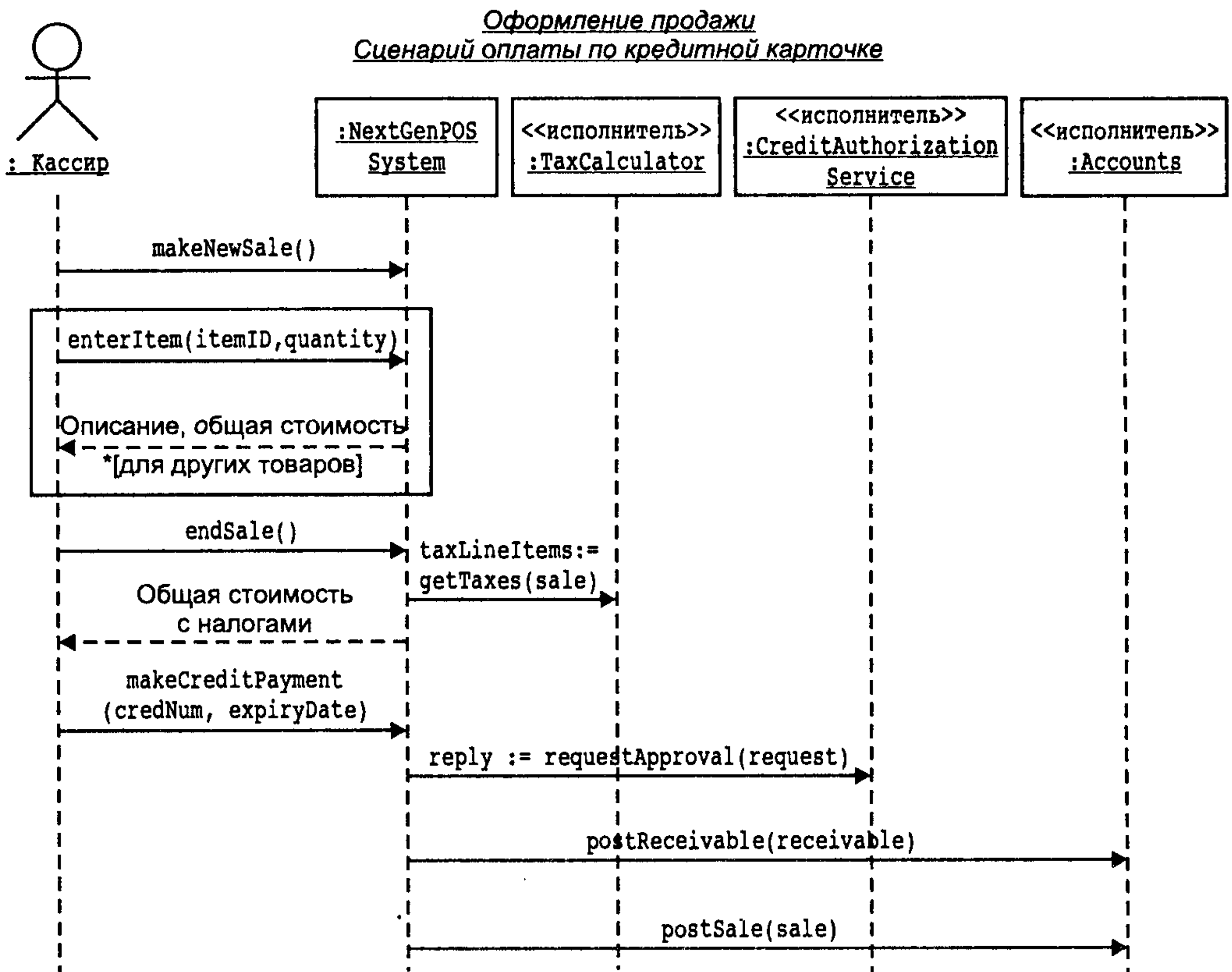



*Рис. 21.1. Диаграмма последовательностей, иллюстрирующая взаимодействие с внешними системами*

## Модель предметной области

Имея минимальный опыт моделирования предметной области, разработчик может оценить степень влияния новых требований на модель предметной области с точки зрения появления новых понятий, ассоциаций и атрибутов. В отличие от предыдущей итерации, новые требования не приводят к необходимости рассмотрения множества новых понятий предметной области. Краткий анализ новых требований показывает, что в модель предметной области следует добавить лишь несколько понятий, в том числе `PriceRule` (правило вычисления стоимости).

Поэтому можно не останавливаться на модификации модели предметной области, а сразу перейти к проектированию, в процессе которого разработчики и введут новые понятия предметной области. В контексте UP главное определить, добавляет ли артефакт новое качество. Если создание артефакта сводится к выполнению механической работы, то его лучше пропустить.

Такая гибкость имеет обратную сторону. Очень часто разработчикам начинает казаться, что можно пропустить все предшествующие программированию

этапы разработки и сразу же приступить к созданию кода. Это, конечно, можно, поскольку самым важным этапом является именно программирование, а не создание модели предметной области. Однако большинство разработчиков может привести примеры из своего опыта, когда небольшое исследование проблемы до начала программирования значительно упрощало процесс разработки системы и способствовало устранению множества проблем.

### Модель прецедентов: описание системных операций

На этой итерации новые операции не рассматриваются, поэтому ничего описывать не нужно. Описание любой системной операции — это лишь возможность ее детального рассмотрения, более точного, чем при описании прецедентов.

Глава 22

# Дополнительные шаблоны GRASP для распределения обязанностей

*Удача — это основа проектирования.*

*Бранч Рики (Branch Rickey)*

**Основная задача**

- **Научиться применять остальные шаблоны GRASP.**

## Введение

Ранее были рассмотрены основные пять шаблонов GRASP.

- Information Expert (Информационный эксперт), Creator (Создатель), High Cohesion (Высокое зацепление), Low Coupling (Слабое связывание) и Controller (Контроллер)

  В состав шаблонов GRASP входят еще четыре шаблона.
- Polymorphism (Полиморфизм)
- Indirection (Перенаправление)
- Pure Fabrication (Чистая синтетика)
- Protected variations (Защищенные вариации)

На основе этих шаблонов можно значительно обогатить свой опыт проектирования программных систем. В последующих главах будут также рассмотрены шаблоны из набора GoF, разработанные так называемым "союзом четырех". К шаблонам GoF относятся Strategy (Стратегия) и Factory (Фабрика). После знакомства с этими шаблонами фраза о том, что нужно применить шаблон Strategy на основе шаблона Factory для удовлетворения требований шаблона Protected Variations и обеспечения низкой степени связывания обретет новый смысл и ин-

формационную емкость, поскольку за обычными именами шаблонов стоят сложные концепции проектирования.

В этой главе кратко рассматриваются остальные шаблоны GRASP, представляющие наиболее общие принципы распределения обязанностей между объектами на этапе проектирования.

В следующей главе вводятся другие полезные шаблоны, применяемые на второй итерации разработки POS-приложения NextGen.

## 22.1. Шаблон Polymorphism

**Решение.** Если поведение объектов одного типа (класса) может изменяться, обязанности распределяются для различных вариантов поведения с использованием полиморфных операций для этого класса[1].

*Совет.* Используйте не проверку для типа объекта, а условную логику для реализации различных альтернативных вариантов поведения на основе типа.

**Проблема.** Как обрабатывать альтернативные варианты поведения на основе типа? Как создавать подключаемые программные компоненты?

*Альтернативные варианты поведения на основе типа.* Условная передача управления — это основная отличительная особенность любой программы. Если программа разработана с использованием условных операторов типа `if-then-else` (если-то-иначе) или операторов условного перехода по ключу, то при добавлении новых вариантов поведения приходится модифицировать логику условных операторов. Такой подход затрудняет процесс модификации программ в соответствии с новыми вариантами поведения, поскольку изменения приходится вносить сразу в нескольких местах программного кода — там, где используются условные операторы.

*Подключаемые программные компоненты.* Если рассматривать компоненты с точки зрения отношения клиент/сервер, то как можно заменить один серверный компонент другим, не затрагивая при этом клиентские компоненты?

**Пример.** POS-система NextGen должна поддерживать работу различных внешних систем вычисления налоговых платежей (в том числе Tax-Master и Good-As-Gold TaxPro) и интеграцию с другими внешними системами. Каждая система вычисления налоговых платежей имеет свой интерфейс и обладает собственным поведением (хотя и аналогичным поведению других систем). Один продукт может поддерживать протокол TCP, другой — интерфейс SOAP, а третий — интерфейс Java RMI.

Какие объекты должны отвечать за обработку различных внешних интерфейсов систем вычисления налоговых платежей?

Поскольку поведение службы адаптации внешней системы вычисления налоговых платежей зависит от типа используемой программы вычисления (условно говоря, калькулятора), согласно шаблону Polymorphism, необходимо распределить обязанности по адаптации к различным типам калькуляторов. Для этого можно использовать полиморфную операцию `getTaxes` (рис. 22.1).

---

[1] Термин *полиморфизм* (polymorphism) имеет несколько взаимосвязанных значений. В данном контексте он означает "присваивание одинаковых имен службам различных объектов" [30], если эти службы однотипны или взаимосвязаны. Различные типы объектов зачастую реализуют общий интерфейс или связываются между собой посредством иерархии и наследования свойств общего суперкласса, однако такая реализация зависит от выбранного языка программирования. Например, в языках с динамическим связыванием, таких как Smalltalk, этого не требуется.

Объекты, обеспечивающие адаптацию, — это не внешние калькуляторы, а локальные программные объекты, представляющие внешние системы вычисления налоговых платежей. При отправке сообщения локальному объекту выполняется обращение к внешней системе с использованием ее собственного программного интерфейса.

Каждому методу `getTaxes` в качестве параметра передается объект `Sale`, чтобы система вычисления налоговых платежей могла проанализировать продажу. Реализации каждого такого метода отличаются. Например, объект `TaxMasterAdapter` может адаптировать запросы к системе Tax-Master и т.п.

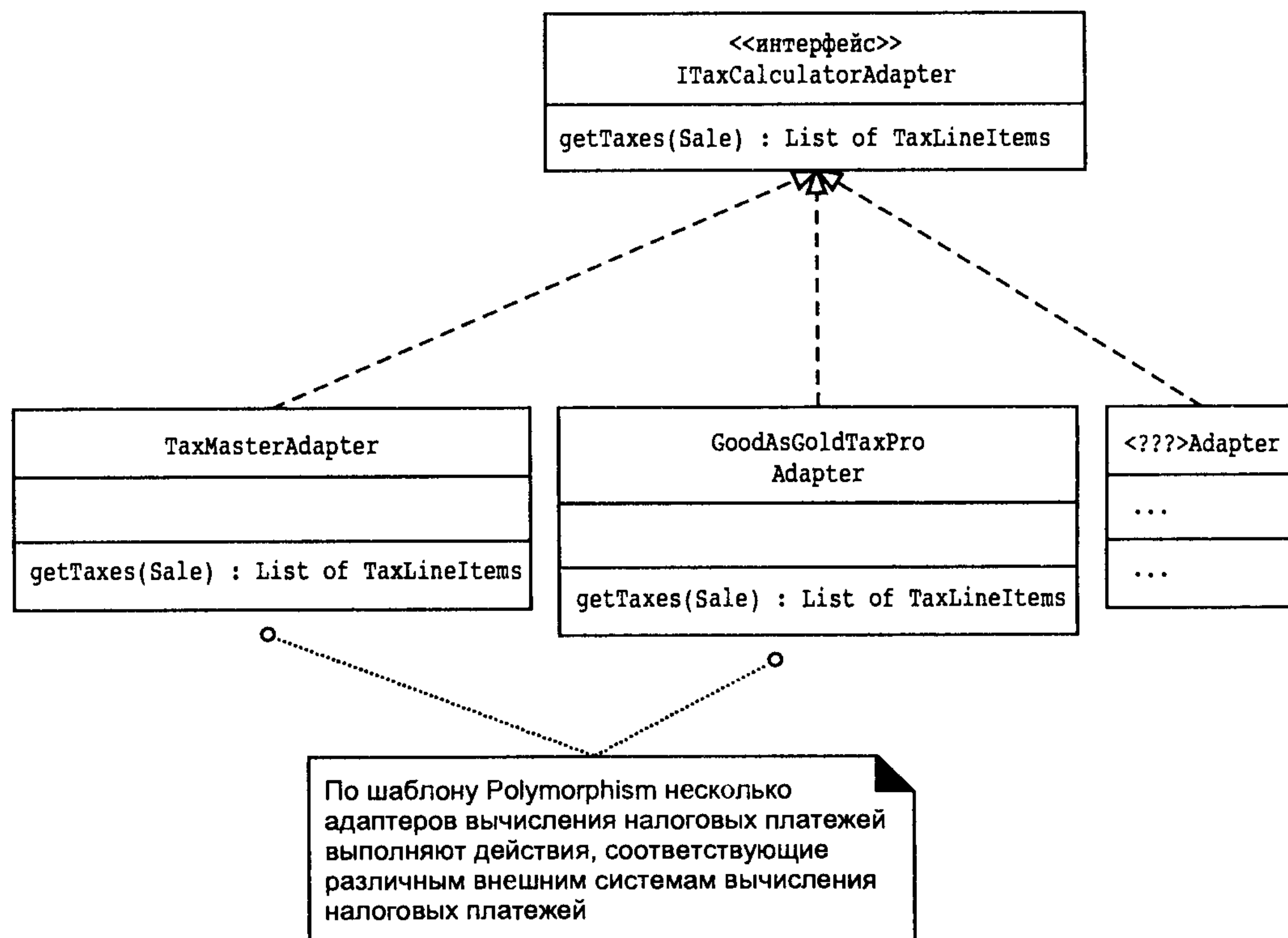



*Рис. 22.1. Использование полиморфизма при адаптации к различным внешним системам вычисления налоговых платежей*

**Система обозначений UML.** На рис. 22.2 показаны некоторые обозначения UML для задания *интерфейсов* (interface) (дескрипторов операций без реализации), их реализации и возвращаемого значения типа контейнера или коллекции объектов. Для обозначения интерфейса в UML используются *стереотипы*. Стереотип — это механизм категоризации элементов. Имя стереотипа заключается в двойные угловые скобки, например <<интерфейс>>. Однако Румбах отмечает, что прямые кавычки в типографиях при необходимости можно заменять на двойные угловые скобки [93].

**Обсуждение.** Полиморфизм — это основной принцип проектирования поведения системы в рамках обработки аналогичных ситуаций. Если обязанности в системе распределены на основе шаблона Polymorphism, то такую систему можно легко модифицировать, добавляя в нее новые вариации. Например, добавив

класс для адаптации новой системы вычисления налоговых платежей со своим полиморфным методом getTaxes, разработчик никак не повлияет на дееспособность уже существующей части системы.

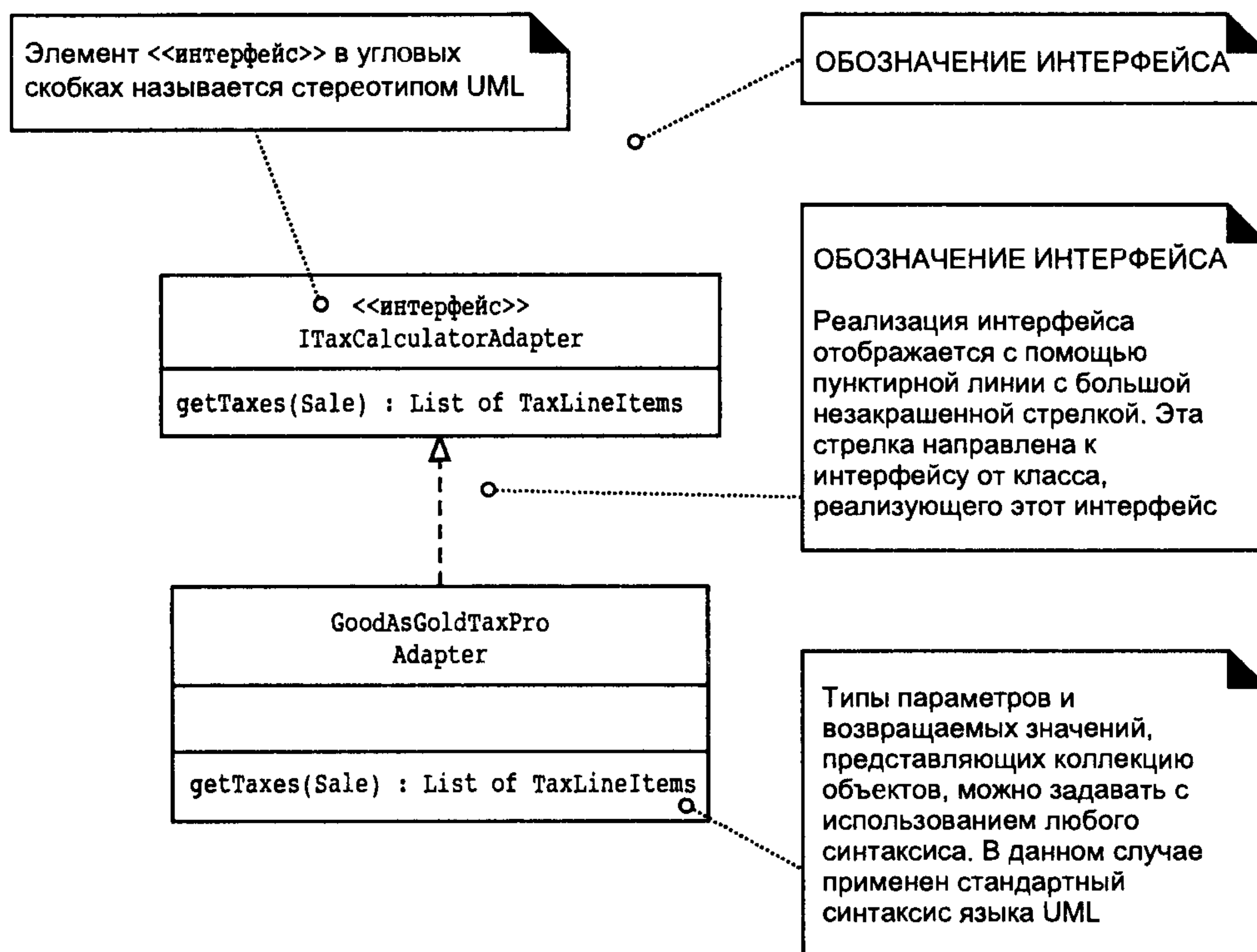



*Рис. 22.2. Обозначения UML для интерфейсов и типов возвращаемых значений*

**Когда не следует применять шаблоны.** Иногда разработчики систем злоупотребляют добавлением интерфейсов и применением принципа полиморфизма с целью обеспечения дееспособности системы в неизвестных заранее новых ситуациях. Если точка вариаций известна и обоснована, если существует высокая вероятность вариантного поведения, то такие усилия по обеспечению гибкости системы и применению принципа полиморфизма вполне оправданы. Однако к применению этого принципа нужно подходить критически, поскольку зачастую усилия по обеспечению вариантов поведения "на все случаи жизни" затрачиваются впустую. Реально оценивайте вероятность вариантного поведения.

### Преимущества

- С помощью шаблона впоследствии можно легко расширять систему, добавляя в нее новые вариации.
- Новые реализации можно вводить без модификации клиентской части приложения.

### Связанные шаблоны

- Protected Variations (Защищенные вариации).

- Множество популярных шаблонов GoF [52], которые будут описаны в последующих главах, основаны на применении принципа полиморфизма. К числу таких шаблонов относятся Adapter (Адаптер), Command (Команда), Composite (Композит), Proxy (Прокси), State (Состояние) и Strategy (Стратегия).

**Также известен как; аналогичен.** Choosing Message (Выбор сообщения), Don't Ask "What Kind?" (Не спрашивай "Что это?").

## 22.2. Шаблон Pure Fabrication

**Решение.** Присвоить группу обязанностей с высокой степенью зацепления искусственному классу, не представляющему конкретного понятия из предметной области, т.е. синтезировать искусственную сущность для поддержки высокого зацепления, слабого связывания и повторного использования.

Такой класс является продуктом нашего воображения и представляет собой *синтетику* (fabrication). В идеале, присвоенные этому классу обязанности поддерживают высокую степень зацепления и низкое связывание, так что структура этого синтетического класса является очень прозрачной или *чистой* (pure). Отсюда и название: Pure Fabrication ("чистая синтетика").

И наконец, словосочетание "чистая синтетика" подразумевает создание некоторой сущности в тот момент, когда разработчик "близок к отчаянию".

**Проблема.** Какой класс должен обеспечить реализацию шаблонов High Cohesion и Low Coupling или других принципов проектирования, если шаблон Expert (например) не обеспечивает подходящего решения?

Объектно-ориентированные системы отличаются тем, что программные классы реализуют понятия предметной области, как, например, классы `Sale` и `Customer`. Однако существует множество ситуаций, когда распределение обязанностей только между такими классами приводит к проблемам с зацеплением и связыванием, т.е. с невозможностью повторного использования кода.

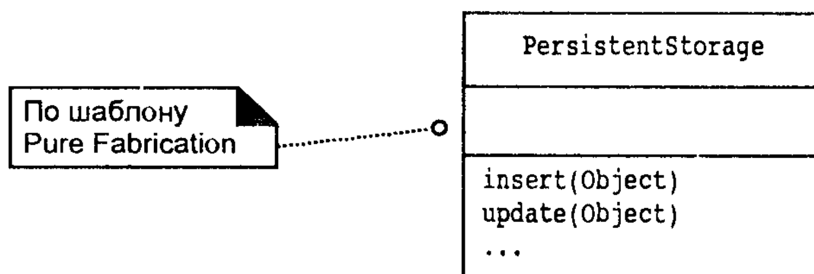
**Пример.** Предположим, необходимо сохранять экземпляры класса `Sale` в реляционной базе данных. Согласно шаблону Information Expert, эту обязанность можно присвоить самому классу `Sale`. Однако следует принять во внимание следующие моменты.

- Данная задача требует достаточно большого числа специализированных операций, связанных со взаимодействием с базой данных и никак не связанных с понятием самой продажи. Поэтому класс `Sale` получает низкую степень зацепления.
- Класс `Sale` должен быть связан с интерфейсом реляционной базы данных (таким как JDBC для технологии Java). Поэтому возрастает степень связывания, причем даже не с другим объектом предметной области, а с интерфейсом базы данных.
- Хранение объектов в реляционной базе данных — это достаточно общая задача, решение которой необходимо для многих классов. Возлагая эту обязанность на класс `Sale`, разработчик не обеспечивает возможности повторного использования этих операций и вынужден дублировать их в других классах.

Поэтому, несмотря на то, что по логике вещей класс `Sale` является хорошим кандидатом для выполнения обязанности самосохранения в базе данных, согласно шаблону Information Expert, такое распределение обязанностей приво-

дит к низкой степени зацепления, сильному связыванию и невозможности повторного использования кода. Это именно та безвыходная ситуация, в которой необходимо "синтезировать" нечто новое.

Естественным решением данной проблемы является создание нового класса, ответственного за сохранение объектов некоторого вида на постоянном носителе, например в реляционной базе данных. Его можно назвать PersistentStorage[2] (Постоянное хранилище). Этот класс является продуктом нашего воображения, что полностью соответствует шаблону Pure Fabrication.




Обратите внимание на имя класса PersistentStorage, оно вполне понятно, хотя объект с аналогичным названием отсутствует в модели предметной области. Если разработчик спросит сотрудников магазина, работают ли они с объектами из постоянного хранилища, то его, скорее всего, не поймут. Им известны понятия "продажа" или "платеж". PersistentStorage — это не понятие из предметной области, а нечто, синтезированное для удобства разработчика программы.

Этот чисто синтетический объект решает следующие задачи.

- Класс Sale сохраняет высокую степень зацепления и слабое связывание.
- Класс PersistentStorage также обладает этими свойствами, выполняя единственную задачу: сохранение объектов на постоянном носителе.
- Класс PersistentStorage является достаточно общим и допускает повторное использование.

Этот пример — типичная ситуация для создания "чисто синтетического" класса. Он позволяет избежать проблем с зацеплением и связыванием и обеспечивает новые возможности для повторного использования.

Обратите внимание, что, подобно остальным шаблонам GRASP, в данном шаблоне основное внимание уделяется распределению обязанностей. В нашем примере обязанности передаются от класса Sale (выбранного согласно шаблону Expert) новому "чисто синтетическому" классу.

**Обсуждение**. Проектируемые объекты делятся на две следующие группы.

1. Создаваемые на основе *декомпозиции представления.*
2. Создаваемые на основе *декомпозиции поведения.*

Например, программный класс Sale был создан на основе декомпозиции представления, поскольку он представляет понятие предметной области (или связан с ним). Декомпозиция представления — это типичная стратегия объектного проектирования, позволяющая сократить разрыв между реальными объек-

[2] В реальных задачах, как правило, требуется несколько "чисто синтетических" классов. Такие объекты обеспечивают интерфейсы взаимодействия для множества вспомогательных объектов.

тами и их программным представлением. Однако иногда возникает потребность распределить обязанности на основе однотипного поведения или реализации некоторого алгоритма, не привязываясь к понятиям реального мира.

Хорошим примером объекта-алгоритма является генератор оглавления `TableOfContentsGenerator`. Это вспомогательный класс, не имеющий аналога в словаре терминов из предметной области книг или документов. Он предназначен для удобства разработчиков, поскольку в нем сосредоточены методы, реализующие некоторое однотипное поведение. То есть этот класс создан на основе *декомпозиции поведения.*

В отличие от рассмотренного примера, класс `TableOfContents` основан на декомпозиции представления, поскольку он реализует конкретное понятие из реального мира (список наименований глав).

Вспомогательный класс не обязательно должен быть "чисто синтетическим". Это название служит скорее целям обучения, поскольку отражает общую идею распределения обязанностей между программными классами: некоторые классы представляют понятия предметной области, а другие создаются просто для удобства разработчиков. Эти вспомогательные классы зачастую объединяются в группы на основе общности поведения и соответствуют принципу декомпозиции поведения, а не представления.

Другими словами, согласно шаблону Pure Fabrication, классы проектируются на основе общей функциональности и представляют собой поведенческие или функционально-ориентированные объекты.

Многие существующие шаблоны объектно-ориентированного проектирования являются примерами использования шаблона Pure Fabrication. К ним можно отнести шаблоны Adapter, Strategy, Command и т.д. [52].

Напоследок вспомним основное назначение шаблона Pure Fabrication. Иногда решение, соответствующее шаблону Information Expert, не подходит для других шаблонов проектирования. С одной стороны, объект является хорошим кандидатом на выполнение некоторой обязанности, поскольку обладает всей необходимой для этого информацией, а с другой — такое распределение обязанностей приводит к нарушению принципов зацепления и связывания.

**Преимущества**

- При использовании шаблона Pure Fabrication реализуется шаблон High Cohesion, поскольку обязанности передаются отдельному классу, сконцентрированному на решении специфического набора взаимосвязанных задач.
- Повышается потенциал повторного использования, поскольку чисто синтетические классы можно применять в других приложениях.

**Когда не нужно использовать шаблон.** Новички в области объектно-ориентированного проектирования, имеющие опыт разработки программ в рамках структурного или функционального подхода, зачастую "злоупотребляют" использованием шаблона Pure Fabrication. В их трактовке функции просто превращаются в объекты. Нет ничего предосудительного в создании функционально-ориентированных или алгоритмически-ориентированных объектов. Однако необходимо соблюдать пропорции между количеством таких объектов и объектов, созданных на основе декомпозиции представления, согласно шаблону Information Expert. Класс `Sale` тоже имеет право на существование и должен выполнять свои обязанности. Созданные на основе шаблона Information Expert объекты обладают информацией, необходимой для вы-

полнения своих обязанностей и соответствуют принципу слабого связывания объектов. При "злоупотреблении" применением шаблона Pure Fabrication нарушаются требования к слабому связыванию объектов. Типичным симптомом такой ситуации является необходимость передачи данных одного объекта другим объектам для выполнения действий над ними.

**Связанные шаблоны**

- Low Coupling.
- High Cohesion.
- Классы, созданные согласно шаблону Pure Fabrication, обычно имеют обязанности, которые, в соответствии с шаблоном Expert, должны быть возложены на классы предметной области.
- Многие шаблоны проектирования GoF являются примерами использования шаблона Pure Fabrication (к их числу относятся Adapter (Адаптер), Command (Команда), Strategy (Стратегия) и т.д. [52]).
- Все остальные шаблоны проектирования также реализуют принципы шаблона Pure Fabrication.

## 22.3. Шаблон Indirection

**Решение**. Присвоить обязанности промежуточному объекту для обеспечения связи между другими компонентами или службами, которые не связаны между собой напрямую.

При таком подходе связи *перенаправляются* между другими компонентами или службами.

**Проблема**. Как распределить обязанности, чтобы обеспечить отсутствие прямого связывания? Как снизить уровень связывания объектов, согласно шаблону Low Coupling, и сохранить высокий потенциал повторного использования?

### *Примеры*

#### Класс `TaxCalculatorAdapter`

Эти объекты обеспечивают связь с внешними службами вычисления налоговых платежей. Согласно шаблону Polymorphism, они обеспечивают единый интерфейс взаимодействия с внутренними объектами и скрывают вариации программных интерфейсов внешних систем. Добавляя возможность перенаправления и обеспечивая реализацию принципов полиморфизма, объекты-адаптеры защищают внутренние объекты системы от возможного изменения интерфейсов внешних систем (рис. 22.3).

#### Класс `PersistentStorage`

В примере, приведенном при рассмотрении шаблона Pure Fabrication, класс `Sale` отделяется от служб взаимодействия с реляционной базой данных посредством введения класса `PersistentStorage`. Этот же пример соответствует реализации шаблона Indirection. Класс `PersistentStorage` выступает в роли промежуточного звена между классом `Sale` и базой данных.

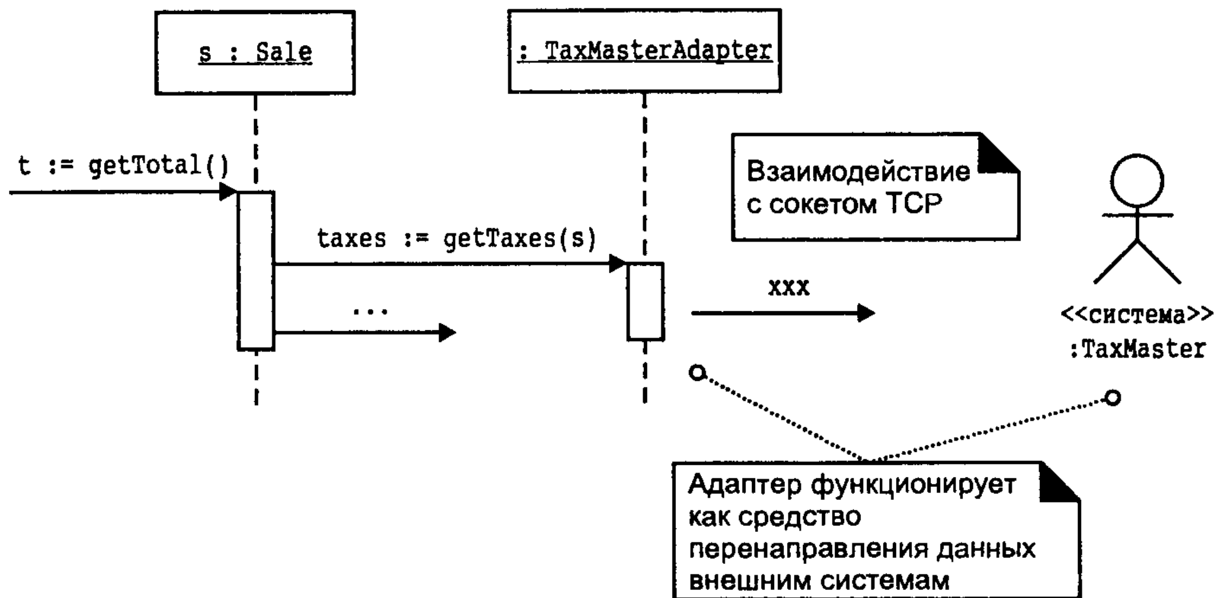



*Рис. 22.3. Перенаправление связей с помощью адаптеров*

**Обсуждение.** *Большинство проблем, связанных с компьютерными науками, можно решить с помощью нового уровня перенаправления* — это старая истина, применимая и к объектно-ориентированному проектированию[3].

Точно также, как многие шаблоны проектирования являются частным случаем шаблона Pure Fabrication, существует множество специализированных вариантов шаблона Indirection. Примерами таких шаблонов являются Adapter, Facade и Observer [52]. Кроме того, многие вариации шаблона Pure Fabrication зачастую базируются на основном принципе перенаправления связи по шаблону Indirection. Целью такого перенаправления обычно является слабое связывание, обеспечиваемое за счет отделения друг от друга различных компонентов или служб.

**Преимущества.** Слабое связывание между компонентами.

**Связанные шаблоны и принципы**

- Protected Variations.
- Low Coupling.
- Многие шаблоны GoF, в том числе Adapter, Bridge, Observer и Mediator [52].
- Различные частные случаи использования шаблона Pure Fabrication.

## 22.4. Шаблон Protected Variations

**Решение.** Идентифицировать точки возможных вариаций или неустойчивости; распределить обязанности таким образом, чтобы обеспечить устойчивый интерфейс.

Заметим, что термин "интерфейс" в данном случае используется для обозначения способа обеспечения доступа и не сводится к понятиям интерфейса Java или COM.

**Проблема.** Как спроектировать объекты, подсистемы и систему, чтобы изменение этих элементов не оказывало нежелательного влияния на другие элементы?

---

[3] Если какую-либо истину, относящуюся к компьютерным наукам, можно считать старой! Правда, я запамятовал автора этого высказывания. Возможно, это Парнас (Parnas)? Заметим, что известно и другое высказывание: "Большинство проблем, связанных с производительностью, можно решить, удалив новый уровень перенаправления!"

**Пример.** Вернемся к рассмотренной выше проблеме взаимодействия с различными системами вычисления налоговых платежей. Решение этой проблемы на основе шаблона Polymorphism в то же время иллюстрирует применение шаблона Protected Variations (см. рис. 22.1). В данном случае точкой вариации или неустойчивости являются различные интерфейсы или API внешних систем вычисления налоговых платежей. POS-система должна обеспечить интеграцию с различными известными (а также еще не существующими) системами вычисления налоговых платежей.

Добавив уровень (или интерфейс) перенаправления на основе принципа полиморфизма, можно разработать разные реализации интерфейса `ITaxCalculatorAdapter`, защитив систему от возможного изменения внешних интерфейсов. Тогда внутренние объекты смогут взаимодействовать с устойчивым интерфейсом, а детали взаимодействия с внешними системами будут скрыты в конкретных реализациях адаптеров.

**Обсуждение.** Шаблон Protected Variations (PV) впервые был описан Кокбурном в [104], хотя этот фундаментальный принцип проектирования известен уже несколько десятилетий.

### Механизмы, реализуемые с помощью шаблона PV

Шаблон PV описывает ключевой принцип, на основе которого реализуются механизмы и шаблоны программирования и проектирования с целью обеспечения гибкости и защиты системы от влияния изменений внешних систем.

В процессе реализации шаблона PV совершенствуется уровень разработчиков и архитекторов, поскольку они должны реализовать подходящее решение для этого шаблона. На начальных стадиях обучения программированию человек знакомится с инкапсуляцией данных, интерфейсами и принципом полиморфизма — базовыми механизмами шаблона PV. Затем он осваивает новые приемы и области, такие как интерпретаторы правил, проектирование на основе метаданных, виртуальные машины — средства защиты системы от возможных вариаций.

Рассмотрим некоторые примеры.

### Базовые механизмы защиты от вариаций

Инкапсуляция данных, интерфейсы, полиморфизм, перенаправление — все эти принципы реализуются в рамках шаблона PV. Заметим, что компоненты, в том числе брокеры и виртуальные машины, — это сложные примеры перенаправления с целью реализации принципа PV.

### Проектирование на основе данных

Это широкий спектр методов, включая чтение кодов, значений, путей к файлам классов, имен классов из внешнего источника с целью обеспечения возможности изменения поведения или "параметризации" системы на этапе выполнения. К другим вариациям этих методов относятся листы стилей, метаданные для отображения объектов в реляционное представление, файлы свойств, чтение макетов и т.д. Защита системы от изменения данных, метаданных или других вариаций обеспечивается за счет внешнего расположения изменчивой информации и ее чтения на этапе выполнения программы.

### Поиск служб

Поиск служб подразумевает использование служб именования (например, JNDI для Java) или средств получения доступа к службе (например, Jini для

Java или UDDI для служб Web). Клиенты в этом случае защищены от изменения расположения служб: они могут использовать стандартный интерфейс поиска служб. Это специальный случай проектирования на основе данных.

### Проектирование на основе интерпретаторов

Проектирование на основе интерпретаторов предполагает включение интерпретаторов правил, выполняющих правила и считывающих их из внешнего источника, интерпретаторов сценариев или языков программирования, считывающих информацию и запускающих программы, виртуальных машин, нейронных сетей, механизмов реализации логики ограничений и т.д. Такой подход позволяет изменять или параметризировать поведение системы с помощью внешних логических выражений. Система защищена от изменения логики за счет внешнего расположения описания этой логики, считывания его и использования интерпретатора.

### Рефлексивное проектирование или проектирование на метауровне

Примером реализации этого подхода является использование `java.beans.Introspector` для получения объекта `BeanInfo`, запрашивающего имя метода, реализующего свойство компонента X и вызывающего *Метод*`.invoke`. При этом система остается защищенной от влияния логики и изменения кода внешних компонентов за счет использования служб метаязыка. Это проектное решение является частным случаем проектирования на основе данных.

### Унифицированный доступ

Некоторые языки, такие как Ada, Eiffel и C#, поддерживают синтаксические конструкции, при которых доступ к полям и методам классов осуществляется одинаково. Например, запись `aCircle.radius` может означать и вызов метода `radius():float`, и прямое обращение к открытому полю, в зависимости от определения класса. Тогда открытые поля можно преобразовывать в метод доступа без изменения кода клиентских объектов.

### Принцип подстановки Лискова

Принцип подстановки Лискова [76] — это формализация защиты от влияния изменений в различных реализациях интерфейса или расширениях суперкласса.

Приведем цитату.

> "Здесь требуется использовать следующее свойство подстановки: если для каждого объекта o1 типа S существует объект o2 типа T, такой что для всех программ P, определенных в терминах T, поведение P не изменяется при замене o2 на o1, то S является подтипом T" [76].

Неформально это можно выразить так. Программные элементы (методы, классы и т.д.), ссылающиеся на тип `T` (некоторый интерфейс или суперкласс), должны корректно работать при подстановке вместо `T` любой его реализации или подкласса `S`. Вот пример.

```
public void addTaxes( ITaxCalculatorAdapter calculator, Sale sale )
{
        List taxLineItems = calculator.getTaxes( sale );
        //...
}
```

Не имеет значения, какая из реализаций класса `ITaxCalculatorAdapter` передается в качестве реального параметра метода `addTaxes`, метод продолжает работать "как и ожидалось". Идея принципа подстановки Лискова проста и интуитивно понятна многим разработчикам объектных программ.

### Сокрытие структуры

В первой редакции этой книги упоминался важный принцип объектного проектирования, получивший название Don't Talk to Strangers (Не разговаривайте с незнакомцами) или Law of Demeter (Закон Деметры) [75]. Этот принцип был назван одним из девяти шаблонов GRASP. Вкратце он сводится к необходимости избегать проектных решений, предполагающих передачу сообщений (или разговоры) с удаленными непрямыми объектами (или незнакомцами). Непрямыми считаются объекты, известные другим объектам, но не самому клиенту. Такое проектное решение обеспечивает устойчивость системы к изменению структуры объектов. А это довольно типичная точка неустойчивости. Однако во втором издании книги вместо шаблона Don't Talk to Strangers рассматривается более общий шаблон PV, для которого Don't Talk to Strangers является частным случаем. Таким образом, чтобы обеспечить защиту от влияния структурных изменений, применяется шаблон Don't Talk to Strangers.

Шаблон Don't Talk to Strangers налагает ограничения на перечень объектов, которым определенный метод должен отправлять сообщения. Этот "закон" гласит, что в рамках метода сообщения должны отправляться только следующим объектам.

1. Объекту `this` (или `self`).
2. Параметру этого метода.
3. Атрибуту объекта `this`.
4. Элементу коллекции, являющемуся атрибутом объекта `self`.
5. Объекту, созданному внутри метода.

Основная цель этого шаблона — избежать связывания клиентского объекта с непрямыми объектами.

Прямые объекты — это "знакомые" клиента, а непрямые — "незнакомцы". Клиент должен общаться только со "знакомыми" и не разговаривать с "незнакомцами".

Для выполнения этих требований прямым объектам могут понадобиться новые операции, которые выступают в роли дополнительных операций, позволяющих избежать "разговоров с незнакомцами".

Рассмотрим пример, демонстрирующий нарушение принципа Don't Talk to Strangers. После кода приводятся его комментарии и объяснения по поводу нарушения принципа шаблона.

```
class Register
{
private Sale sale;

public void slightlyFragileMethod()
{
    // sale.getPayment() отправляет сообщение "знакомому" (вариант 3)
    // однако в выражении sale.getPayment().getTenderedAmount()
```

```
    // сообщение getTenderedAmount() передается "незнакомцу" Payment

    Money amount = sale.getPayment().getTenderedAmount();
    //...
}
    //...
}
```

В этом коде устанавливается соединение объекта `Sale` с "незнакомцем" `Payment`, а затем этому объекту передается сообщение. Это довольно неустойчивое решение, поскольку для передачи сообщения сначала необходимо установить связь между объектами `Sale` и `Payment`. Однако на самом деле здесь нет никакой проблемы.

Рассмотрим еще один фрагмент.

```
public void moreFragileMethod()
{
    AccountHolder holder =
        sale.getPayment().getAccount().getAccountHolder();

    //...
}
```

Это искусственный пример, однако с его помощью можно проиллюстрировать передачу сообщения непрямому объекту путем установки цепочки связей. Это проектное решение зависит от конкретной структуры связей между объектами. Причем чем дальше по цепочке передается сообщение, тем неустойчивее становится связь между начальным и конечным объектами.

Карл Либерхерр (Karl Lieberherr) со своими коллегами исследовал принципы хорошего проектирования объектов в ракурсе закона Деметры. Шаблон Don't Talk to Strangers (или закон Деметры) был сформулирован потому, что неустойчивость объектной структуры довольно часто приводит к необходимости модификации кода, построенного на знании типа связи между объектами.

Как будет видно из следующего раздела, этому шаблону нужно следовать не всегда. Все зависит от самой структуры объектов. В стандартных библиотеках (например, в библиотеках Java) структурные связи между классами объектов довольно устойчивы. В почти готовых системах структура объектов тоже довольно устойчива. Однако на ранних итерациях выполнения проекта объектная структура еще не стабильна.

В целом, чем длиннее цепочка передачи сообщения, тем более неустойчивой является связь между исходным и конечным объектами, поэтому в такой ситуации нужно следовать шаблону Don't Talk to Strangers.

Строгое следование принципу этого шаблона, призванного защитить систему от влияния структурных изменений, требует добавления новых открытых операций для классов-знакомых. Эти операции обеспечивают получение требуемой информации, оставляя "за кадром" способ ее получения. Например, для реализации принципов шаблона Don't Talk to Strangers в двух приведенных выше случаях требуется определить следующие операции.

```
// случай 1
Money amount = sale.getTenderedAmountOfPayment();
```

```
// случай 2
AccountHolder holder = sale.getAccountHolderOfPayment();
```

**Когда не следует применять шаблоны.** Следует определить два типа особых точек.

- **Точка вариации** (variation point) — точка ветвления в существующей на данный момент системе или в требованиях к ней, например, необходимость поддержки нескольких интерфейсов для систем вычисления налоговых платежей.
- **Точка эволюции** (evolution point) — предполагаемая точка ветвления, которая может возникнуть в будущем, однако не определяемая существующими требованиями.[4]

Шаблон PV применяется и к точкам вариации, и к точкам эволюции.

Следует сделать оговорку. Иногда для реализации точек эволюции требуется гораздо больше усилий, чем для реализации простого проектного решения без учета этих точек. В таком случае лучше остановиться на неробастном решении и модифицировать его по мере необходимости (в случае реальных, а не предполагаемых изменений).

Например, автору пришлось участвовать в создании системы обработки "бумажных" сообщений, к которой архитектор добавил язык сценариев и интерпретатор для этого языка с целью повышения гибкости системы и поддержки точек эволюции. Однако впоследствии от сложного (и неэффективного) языка сценариев пришлось отказаться, поскольку он был просто не нужен. В начале своей карьеры программиста объектно-ориентированных систем (в начале 80-х годов) автор старался создавать максимально обобщенный код и тратил много времени на создание суперклассов для действительно нужных ему классов. Он стремился добиться максимальной общности кода и его гибкости (хотел защитить его от вариаций) в расчете на последующие изменения, которые так никогда и не понадобились. В то время автор не знал, когда действительно нужно закладывать в систему робастные решения.

Эти примеры приведены не с целью пропаганды неробастных решений и последующей переработки системы. Если необходимость защиты от влияния возможных вариаций действительно существует, нужно придерживаться шаблона PV. Однако если в системе зафиксированы лишь точки эволюции, вероятность реализации которых очень низка, то, принимая робастное проектное решение, следует хорошо взвесить все "за" и "против".

Неопытные разработчики зачастую тяготеют к неробастным проектным решениям, более опытные уделяют слишком большое внимание гибкости системы и стремятся к обобщениям (даже если они никогда не понадобятся). Самые опытные специалисты выбирают промежуточное решение, тщательно оценивая вероятность возможных модификаций и усилия, необходимые для обеспечения робастного решения.

#### Преимущества

- Легкость добавления новых расширений и вариаций.
- Возможность добавления новых реализаций, не затрагивая клиента.

---

[4] В рамках UP точки эволюции могут быть формально зафиксированы в разделах "Возможные изменения" (Change cases). В каждом таком разделе описываются соответствующие аспекты возможных точек эволюции в разрабатываемой архитектуре.

- Слабое связывание.
- Минимизация влияния изменений.

**Связанные шаблоны и принципы**

- Большинство принципов и шаблонов проектирования обеспечивают механизмы для реализации шаблона PV. К их числу относятся принципы полиморфизма, перенаправления, инкапсуляции данных, шаблоны GoF, Indirection и т.д.
- В [90] точки вариации и эволюции названы "горячими точками".

**Также известен под именем.** Шаблон PV соответствует принципам сокрытия информации и OCP (Open-Closed Principle). Свое официальное название Protected Variations он получил в 1996 году в [104].

### *Сокрытие информации*

Известная статья Дэвида Парнаса [88] — это пример классического произведения, которое часто цитируют, но редко читают. В этой статье автор вводит концепцию *сокрытия информации*. Поскольку этот термин, на первый взгляд, близок понятию инкапсуляции данных, то его неправильно интерпретировали и в некоторых книгах определяют как синоним этого понятия. Однако идея Парнаса заключалась в сокрытии информации о проектном решении, относящемся к точкам ветвления или возможных изменений, от других модулей. Пропагандируемый автором статьи принцип проектирования сводится к следующему.

> Мы предлагаем сначала составить список сложных проектных решений, или проектных решений, подлежащих возможному изменению. Затем при разработке каждого модуля нужно скрыть детали реализации этого решения от других модулей.

Таким образом, принцип сокрытия информации Парнаса соответствует шаблону PV и не сводится к простой инкапсуляции данных. Однако этот термин настолько часто неправильно интерпретировали как синоним инкапсуляции, что его уже невозможно использовать в изначальном смысле без дополнительных пояснений.

### *Принцип OCP*

Принцип *OCP* (*Open-Closed Principle*), описанный Бертрандом Мейером (Bertrand Meyer) по существу эквивалентен шаблону PV и принципу сокрытия информации. Приведем его определение.

> Модули должны быть одновременно и открыты (для расширения или адаптации) и закрыты (для модификации, затрагивающей клиентов).

Принципы OCP и PV являются по существу различными выражениями одной и той же идеи — защиты точек вариации и эволюции. В контексте OCP "модуль" включает все отдельные программные элементы, в том числе методы, классы, подсистемы, приложения и т.д.

В контексте OCP выражение "закрыт в смысле X" означает, что изменения X не затрагивают клиентов. Например, "класс закрыт в смысле определений полей экземпляров" означает применение механизма инкапсуляции для определения закрытых полей и открытых методов доступа. В то же время такой класс открыт для модификации определений закрытых данных, поскольку внешние клиенты не взаимодействуют напрямую с этими полями.

Рассмотрим еще один пример. Адаптеры систем вычисления налоговых платежей "закрыты в смысле их открытого интерфейса", что обеспечивает устойчивость интерфейса `ITaxCalculatorAdapter`. Однако эти адаптеры открыты для расширения, поскольку их можно модифицировать при изменении внешнего интерфейса систем вычисления налоговых платежей, не затрагивая клиентов.

Глава

# Реализация прецедентов с использованием шаблонов GoF

Основная задача

- Применить шаблоны проектирования GRASP и GoF для проектирования приложения NextGen.

## Введение

В этой главе рассматривается процесс проектирования в рамках реализации прецедентов на второй итерации разработки приложения NextGen. На этой итерации необходимо реализовать взаимодействие с внешними службами, интерфейсы которых могут изменяться, обеспечить выполнение более сложных правил вычисления стоимости товаров и подключение новых бизнес-правил на этапе функционирования системы.

В контексте обсуждаемых проблем проектирования будут введены новые обозначения языка UML.

Основное внимание здесь уделяется применению шаблонов GoF и GRASP, а также полезных шаблонов, опубликованных другими разработчиками. Мы попытаемся проиллюстрировать тот факт, что объектно-ориентированное проектирование и распределение обязанностей можно построить на применении шаблонов — "перечня" принципов и идиом, помогающих грамотно проектировать объекты.

### *Шаблоны Gang-of-Four*

Дополнительные шаблоны, представленные в этой главе, взяты из [52]. В ней описаны 23 шаблона, предназначенных для использования на этапе объектно-ориентированного проектирования. Поскольку эта книга написана четырьмя авторами, представленные в ней шаблоны получили название шаблонов Gang-of-Four (Союз четырех) или GoF[1].

[1] Тонкий намек на китайскую политику.

В данной главе кратко рассматриваются лишь некоторые популярные шаблоны GoF. Остальные шаблоны представлены в последующих главах.[2] При этом читателям настоятельно рекомендуется внимательно изучить книгу [52] для более полного изучения принципов объектного проектирования. Правда, авторы книги предполагают, что читатели уже имеют некоторый опыт проектирования. Наша книга рассчитана на менее подготовленных читателей.

### *Словарь терминов*

Помимо перечня графических обозначений языка UML, к концу этой главы будет создан словарь, включающий имена шаблонов проектирования. Это позволит обсуждать идеи проектирования программных систем с использованием диаграмм UML, указывая на этих диаграммах названия применяемых шаблонов.

## 23.1. Шаблон Adapter (GoF)

Проблема, описанная в предыдущей главе при рассмотрении шаблона Polymorphism, более точно соответствует шаблону Adapter из набора GoF.

*Шаблон Adapter*

*Контекст/Проблема*

Как обеспечить взаимодействие несовместимых интерфейсов или как создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами?

*Решение*

Конвертировать исходный интерфейс компонента к другому виду с помощью промежуточного объекта-адаптера.

Напомним, что POS-система NextGen должна поддерживать несколько разных внешних служб, включая систему вычисления налоговых платежей, службы авторизации платежей по кредитной карточке, системы бухгалтерского и складского учета и т.д. Каждая из этих систем обладает отдельным интерфейсом приложения, который нельзя изменить.

Решение этой проблемы состоит в добавлении специального объекта с общим интерфейсом в рамках данного приложения и перенаправлении связей от внешних объектов к этому объекту-адаптеру. Решение проблемы, обеспечиваемое посредством шаблона Adapter, показано на рис. 23.1.

Как видно из рис. 23.2, для каждой внешней службы создается отдельный экземпляр объекта-адаптера,[3] в частности SAP для системы бухгалтерского учета. Этот объект преобразует запрос `postSale` с учетом требований внешнего интерфейса, а именно интерфейса XML протокола SOAP поверх HTTPS для Web-службы в рамках корпоративной сети.

---

[2] На самом деле широко используются примерно 15 из существующих 23 шаблонов.

[3] В архитектуре J2EE такие адаптеры для внешних служб более точно называются *адаптерами ресурсов* (resource adapter).

Адаптеры используют интерфейсы и принципы полиморфизма для добавления уровня перенаправления для различных программных интерфейсов других компонентов

<<интерфейс>>
ITaxCalculatorAdapter

getTaxes(Sale):List of TaxLineItems

TaxMasterAdapter

getTaxes(Sale):List of TaxLineItems

GoodAsGoldTaxProAdapter

getTaxes(Sale):List of TaxLineItems

<<интерфейс>>
IAccountingAdapter

postReceivable(CreditPayment)
postSale(Sale)
...

<<интерфейс>>
ICreditAuthorizationServiceAdapter

requestApproval(CreditPayment, TerminalID, MerchantID)
...

<<интерфейс>>
IInventoryAdapter

...

SAPAccountingAdapter

postReceivable(CreditPayment)
postSale(Sale)
...

GreatNorthernAccountingAdapter

postReceivable(CreditPayment)
postSale(Sale)
...

*Рис. 23.1. Шаблон Adapter*

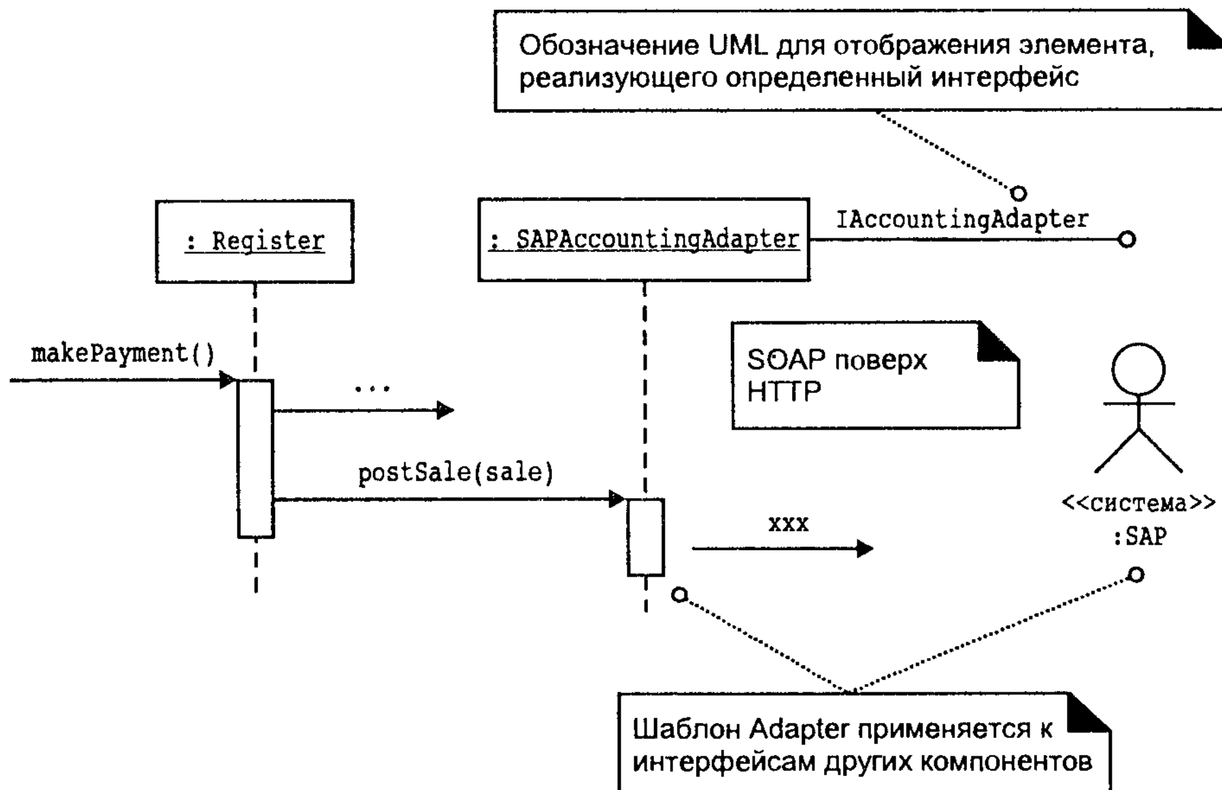



*Рис. 23.2. Использование шаблона Adapter*

Чтобы подчеркнуть значение интерфейса, реализуемого экземпляром `SAPAccountingAdapter`, на рис. 23.2 использовано специальное обозначение UML, напоминающее "леденец на палочке".

### *Шаблоны Polymorphism, Indirection и Protected Variations (GRASP)*

В предыдущем примере шаблон Adapter реализует принципы сразу нескольких шаблонов GRASP. В соответствии с шаблоном Protected Variations обеспечивается защита от влияния изменений интерфейсов внешних пакетов. Это достигается за счет применения принципа перенаправления, декларируемого шаблоном Indirection на основе идеи полиморфизма.

Заметим, что даже самые сложные или специализированные шаблоны можно анализировать в терминах базовых шаблонов семейства GRASP. Существуют сотни шаблонов проектирования. И хотя эти шаблоны полезно изучить с целью более быстрого овладения принципами проектирования, необходимо понимать, что все они базируются на применении основных принципов (обеспечиваемых шаблонами GRASP Protected Variations, Low Coupling, Polymorphism, Indirection и т.д.). Это поможет абстрагироваться от множества избыточных деталей и овладеть базовыми принципами объектного проектирования.

### *Соглашение об именовании: включать ли имя шаблона в имя типа*

Обратите внимание, что имена типов включают название шаблона Adapter. Это достаточно распространенный стиль, преимущество которого состоит в облегчении понимания диаграмм или кода посторонними читателями, которые сразу могут определить, на основе какого шаблона построено то или другое проектное решение.

## 23.2. Анализ на этапе проектирования: модель предметной области

Обратите внимание, что на диаграмме, представленной на рис. 23.1 и демонстрирующей применение шаблона Adapter, операция `getTaxes` возвращает список объектов `TaxLineItems`. Это означает, что в процессе глубокого анализа проблемы вычисления налоговых платежей разработчик (т.е. автор) обнаружил существование множества различных типов налогов на продажи, в том числе региональных, государственных и т.д. (причем всегда существует вероятность введения новых налогов!). Многообразие типов налоговых отчислений привело к необходимости определения целого списка налогов `TaxLineItems`.

Эти объекты не только относятся к вновь созданному программному классу `TaxLineItems` в модели проектирования, но и представляют реальное понятие из предметной области. Выявление новых понятий предметной области и модификация соответствующих диаграмм на этапе проектирования и программирования — довольно распространенное и нормальное явление. Итеративный процесс разработки поддерживает такую форму поэтапного исследования.

Нужно ли отражать эти изменения в модели предметной области (или в словаре)? Если в будущем модель предметной области будет использована в качестве базовой для проектирования новых элементов системы, то такая модификация имеет смысл. На рис. 23.3 показана модифицированная модель предметной области.

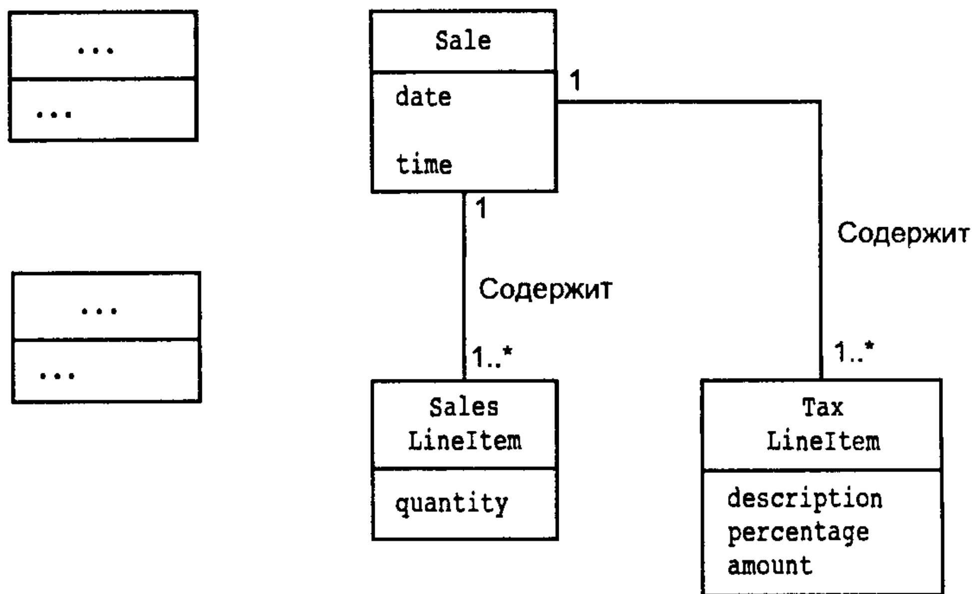



*Рис. 23.3. Обновленный фрагмент модели предметной области*

## Поддержка модели предметной области

В завершение обсуждения вопроса модификации модели предметной области отметим, что модель проектирования обычно делится на уровни (этот аспект более детально обсуждается в следующей главе). Один из этих уровней классов проектирования называется уровнем классов предметной области. Он содержит программные классы, имена и структура которых определяются словарем понятий предметной области (`Sale`, `TaxLineItem` и т.д.).

> *Совет*
>
> После нескольких итераций разработки модель предметной области, изначально выступающая основой для создания классов уровня предметной области в модели проектирования, может утратить свою актуальность и устареть. Если с практической точки зрения ее обновление не имеет смысла, от нее стоит избавиться.
>
> В этом случае лучше просто выполнить обратное проектирование диаграммы классов уровня предметной области из модели проектирования (с помощью CASE-средства, поддерживающего UML). И хотя эти классы являются уже программными и не точно соответствуют концептуальным классам, они отражают важные термины из словаря предметной области. Таким образом, диаграмма UML классов проектирования может стать полезной основой для воссоздания истинной модели предметной области.
>
> Однако не поймите превратно: здесь предлагается не полностью отказаться от модели предметной области, а лишь оценить ее значение для дальнейшей разработки.

**Связанные шаблоны.** Адаптер ресурсов, скрывающий детали интерфейса внешних систем, можно рассматривать как некий "фасадный" объект (шаблон Facade тоже относится к группе шаблонов GoF и будет описан ниже в этой главе), обеспечивающий общий доступ к подсистемам или системам (в этом и состоит основная идея шаблона Facade). Однако название "адаптер ресурсов" более точно отражает ситуацию адаптации к различным внешним интерфейсам.

# 23.3. Шаблон Factory (GoF)

Использование шаблона Adapter порождает новую проблему в проектировании: какой класс должен отвечать за создание объектов-адаптеров при использовании шаблона Adapter? Кто при этом определяет тип создаваемого объекта-адаптера, к примеру `TaxMasterAdapter` или `GoodAsGoldTaxProAdapter`?

Если эти объекты создаются неким объектом уровня предметной области, то обязанности объектов уровня предметной области выходят за рамки обеспечения логики приложения (например, вычисления общей стоимости покупки) в плоскость взаимосвязи с внешними программными компонентами.

Это приводит к нарушению еще одного фундаментального принципа проектирования — разделения различных аспектов функционирования системы (разделения обязанностей). Суть этого принципа сводится к разделению системы на модули, в соответствии с разными аспектами функционирования и задачами системы. Например, программные объекты уровня предметной области должны отвечать только за реализацию логики приложения, а взаимодействие с внешними службами должны обеспечивать отдельные группы объектов.

Следовательно, выбор объекта уровня предметной области (например, `Register`) на роль создателя объектов-адаптеров не соответствует принципу разделения обязанностей и низкого связывания.

Проектное решение, обеспечиваемое шаблоном Factory, показано на рис. 23.4.

Обратите внимание на диаграмму, представленную на рис. 23.4. На ней приводится детальный псевдокод для метода `getTaxCalculatorAdapter`. В статические диаграммы классов языка UML можно включать детали реализации динамических алгоритмов, что позволит снизить информационную нагрузку на диаграммы взаимодействия.

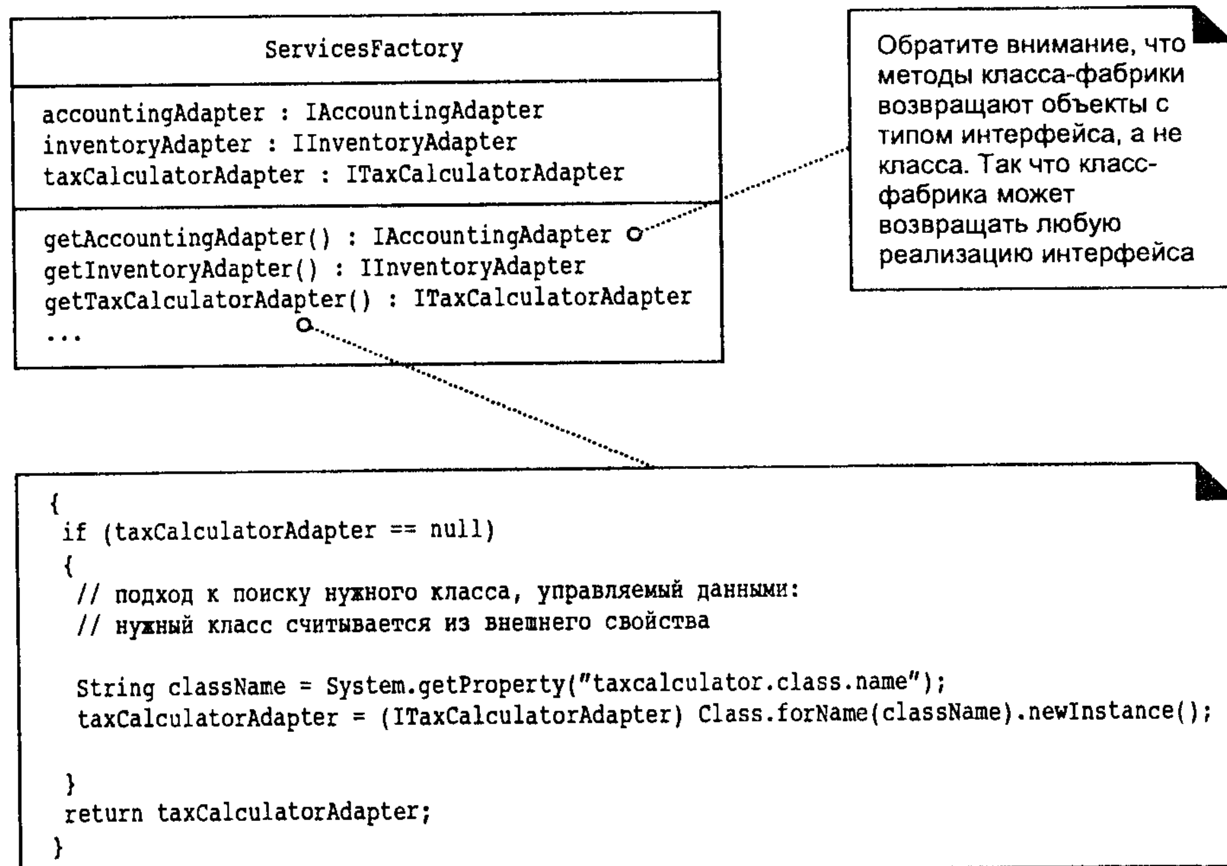



*Рис. 23.4. Шаблон Factory*

Выход из сложившейся ситуации обеспечивает шаблон Factory (или Concrete Factory), в соответствии с которым для построения объектов создается чисто синтетический класс-фабрика.

Такой подход имеет несколько преимуществ.

- Позволяет выделить обязанность создания сложных объектов и делегировать ее отдельному вспомогательному классу.
- Скрывает сложную логику создания объектов.
- Позволяет реализовать стратегии оптимального управления памятью и повышения производительности, в частности кэширование объектов.

*Шаблон (Concrete) Factory*

*Контекст/Проблема*

Что нужно сделать, чтобы узнать, кто должен отвечать за создание сложных объектов в особых условиях, предполагающих сложную логику создания, необходимость выделения обязанностей создания объектов и т.д.?

*Решение*

Создать искусственный объект, соответствующий шаблону Pure Fabrication, и возложить на него обязанность по созданию других объектов в соответствии с шаблоном Factory.

Из рис. 23.4 видно, что объект `ServicesFactory` определяет тип создаваемого объекта путем получения имени класса из внешнего источника (например, через системное свойство при использовании Java), а затем динамически загружает класс. Это пример проектирования на основе данных (data-driven design). Такой подход соответствует шаблону Protected Variations в смысле независимости от изменения реализации класса-адаптера. Новые экземпляры новых типов адаптеров можно создавать без модификации исходного кода класса-фабрики путем изменения значения свойства и обеспечения видимости нового класса.

**Связанные шаблоны.** Доступ к объектам-фабрикам зачастую осуществляется на основе шаблона Singleton.

## 23.4. Шаблон Singleton (GoF)

Использование объекта-фабрики `ServicesFactory` порождает новую проблему проектирования — кто должен создавать саму фабрику и как получить к ней доступ?

Отметим, во-первых, что в рамках приложения необходим лишь один экземпляр объекта-фабрики. Во-вторых, методы объекта могут понадобиться разным объектам и вызываться из различных частей кода, подобно тому, как для обращения к внешним службам из разных частей программы может понадобиться доступ к объектам-адаптерам. Таким образом возникает проблема получения доступа к единственному экземпляру `ServicesFactory` (проблема видимости).

Одно из решений этой проблемы состоит в том, чтобы передать объект `ServicesFactory` в качестве параметра для обеспечения видимости или инициализировать объекты, требующие такой видимости, с постоянной ссылкой на этот объект. Такой путь возможен, но достаточно неудобен. Альтернативный способ решения этой проблемы предоставляет шаблон Singleton.

Кстати, для одного экземпляра класса желательно поддерживать глобальную видимость или единственную точку доступа и не использовать другие формы обеспечения видимости. Это касается и экземпляра объекта `ServicesFactory`.

> *Шаблон Singleton*
>
> *Контекст/Проблема*
>
> Допускается использование только одного экземпляра класса. Различные объекты должны обращаться к этому экземпляру через одну точку доступа.
>
> *Решение*
>
> Определить статический метод класса, возвращающий этот единственный объект.

На рис. 23.5 показан пример реализации шаблона Singleton.

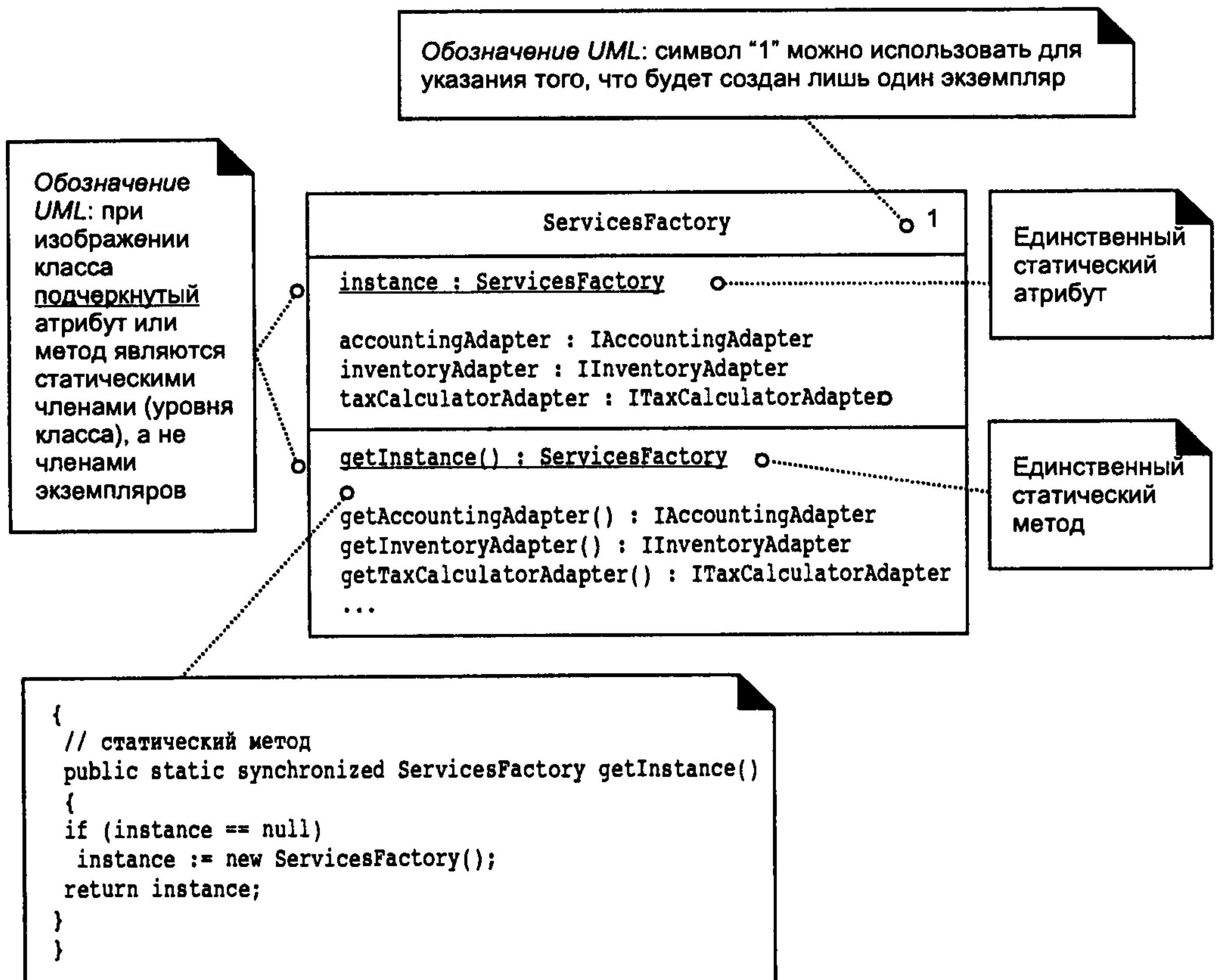



*Рис. 23.5. Использование шаблона Singleton при реализации класса `ServicesFactory`*

Основная идея состоит в определении статического метода `getInstance` класса `X`, обеспечивающего создание единственного экземпляра этого класса.

При таком подходе разработчик обеспечивает глобальную видимость для единственного экземпляра с помощью статического метода этого класса `getInstance`.

```
public class Register
{

public void initialize()
{
    ... выполняем необходимые действия ...
    // доступ к единственному экземпляру объекта-фабрики
    осуществляется через вызов метода getInstance
        ServicesFactory.getInstance().getAccountingAdapter();

    ... выполняем необходимые действия ...
}

// другие методы...

}
```

Поскольку открытые классы чаще всего относятся к глобальной области видимости, то при вызове любого метода любого класса из каждой точки программы можно обратиться к методу `SingletonClass.getInstance()` и получить доступ к единственному экземпляру объекта-фабрики. Тогда этому экземпляру можно передать сообщение `SingletonClass.getInstance().doFoo()` и выполнить любые действия. Сложно переоценить возможность выполнить любые действия из любой точки программы.

### Обозначение UML для доступа к единственному экземпляру на диаграмме взаимодействия

Система обозначений языка UML предполагает использование стереотипа `<<единственный экземпляр>>` (singleton) для единственного экземпляра при отображении (неявном) передачи сообщения `getInstance` (рис. 23.6). При таком подходе для передачи некоторого сообщения единственному объекту очевидное сообщение `getInstance` явно не изображается на диаграмме.

### Вопросы реализации и проектирования

Метод `getInstance`, используемый при реализации шаблона Singleton, вызывается довольно часто. В многопоточных приложениях для создания экземпляров используется *пассивная инициализация* (lazy initialization) на основе управления параллелизмом потоков. Например, при использовании пассивной инициализации в Java управление параллелизмом осуществляется следующим образом.

```
public static synchronized ServicesFactory getInstance()
{
    if ( instance == null )
        // важный раздел для многопоточных приложений
        instance = new ServicesFactory();
    }
    return instance;
}
```

Возникает вопрос, почему не используется *активная инициализация* (eager initialization), как в следующем примере?

```
public class ServicesFactory
{

// активная инициализация
private static ServicesFactory instance =
       new ServicesFactory();

public static ServicesFactory getInstance()
{
    return instance;
}

// другие методы...

}
```

Первый подход на основе пассивной инициализации обычно предпочтительнее хотя бы по следующим причинам.

- Процесс создания экземпляра (или обращения к "дорогим ресурсам") не инициируется, пока реально не потребуется доступ к экземпляру.
- Пассивная инициализация иногда выполняется на основе сложной логики.

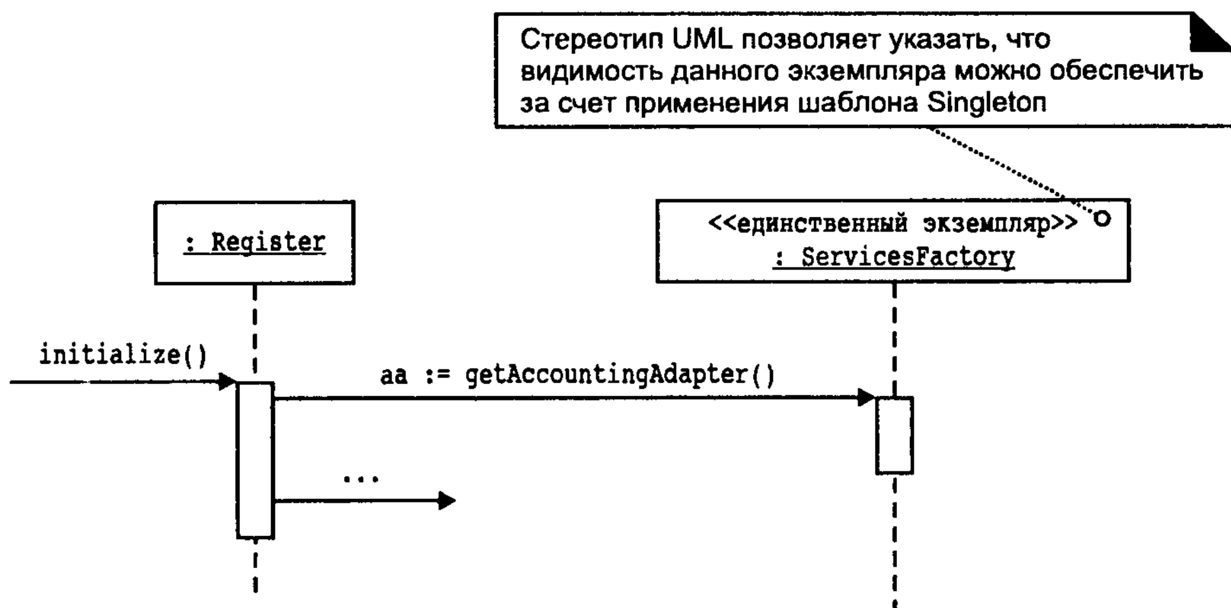



*Рис. 23.6. Неявная передача сообщения* `getInstance`, *согласно шаблону Singleton, обозначается в UML с помощью стереотипов*

При реализации шаблона Singleton напрашивается еще один вопрос: почему все служебные методы не объявить статическими методами самого класса? Зачем использовать экземпляр объекта и методы экземпляра? Например, можно создать статический метод `getAccountingAdapter` для класса `ServicesFactory`. Однако создание экземпляра с его методами обычно предпочтительнее по следующим причинам.

- При использовании методов экземпляра можно использовать механизм наследования и создавать подклассы. Статические методы в большинстве языков программирования (за исключением SmallTalk) не полиморфны (не могут быть виртуальными) и не допускают перекрытия в производных классах.

- Большинство механизмов удаленного взаимодействия на основе объектно-ориентированного подхода (например, RMI в Java) удаленно поддерживают только методы экземпляров, а не статические методы. Поэтому к единственному экземпляру, созданному согласно шаблону Singleton, можно получить удаленный доступ, хотя это и требуется крайне редко.
- Класс может обеспечивать выполнение шаблона Singleton не во всех приложениях. К примеру, в приложении X его объект может выступать в роли единственного экземпляра, а в приложении Y — нет. Кроме того, нередко возникают ситуации, когда на начальных стадиях проектирования предполагается использование лишь единственного экземпляра объекта, а впоследствии требуется несколько его экземпляров. Поэтому решение на основе создания экземпляра является более гибким.

**Связанные шаблоны.** Шаблон Singleton зачастую одновременно используется с шаблонами Factory и Facade, которые будут описаны ниже.

## 23.5. Еще несколько слов о внешних службах с разными интерфейсами

Шаблоны Adapter, Factory и Singleton применялись для реализации основного принципа взаимодействия с внешними системами с различными интерфейсами в соответствии с шаблоном Protected Variations. На рис. 23.7 показана укрупненная схема применения этих шаблонов при реализации прецедентов.

Возможно, это проектное решение не идеально — нет предела совершенству. Задача этого примера — проиллюстрировать проектное решение, основанное на применении набора принципов и шаблонов. Главное понять методологию проектирования на основе шаблонов. Автор надеется, что проектное решение, представленное рис. 23.7, с очевидностью демонстрирует применение шаблонов Controller, Creator, Protected Variations, Low Coupling, High Cohesion, Indirection, Polymorphism, Adapter, Factory и Singleton.

Обратите внимание, насколько кратко разработчик может сформулировать и описать свои идеи с помощью шаблонов. Можно сказать, что проблему различных интерфейсов с внешними службами будем решать на основе шаблона Adapter, а адаптеры ресурсов будем строить по принципу шаблонов Singleton и Factory. Разработчики программного обеспечения действительно общаются между собой подобным образом. Имена и идеи шаблонов повышают уровень абстракции при обсуждении проектного решения.

## 23.6. Шаблон Strategy (GoF)

Следующая проблема — обеспечение более сложной логики вычисления стоимости товаров с учетом сезонных скидок, скидок постоянным клиентам и т.д.

Стратегия вычисления стоимости (которую можно назвать правилом, политикой или алгоритмом) может изменяться. В один период она может составлять 10% от стоимости каждой покупки, затем — 10 долларов для каждой покупки, превышающей $200 и т.п. Как спроектировать эти изменяемые алгоритмы?

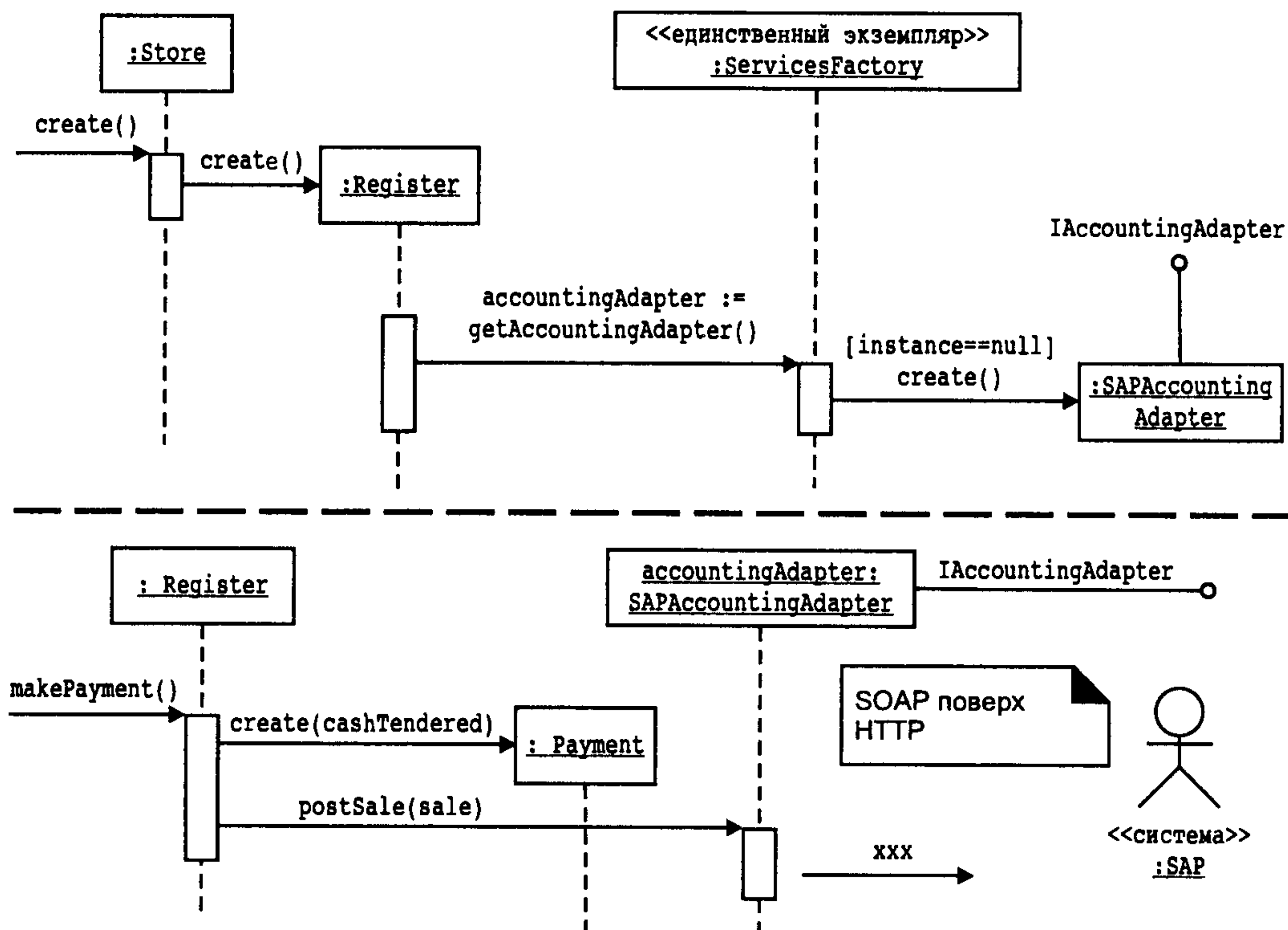



*Рис. 23.7. Проектирование на основе шаблонов Adapter, Factory и Singleton*

> *Шаблон Strategy*
>
> *Контекст/Проблема*
>
> Как спроектировать изменяемые, но надежные алгоритмы или стратегии? Как спроектировать возможность изменения этих алгоритмов или политик?
>
> *Решение*
>
> Определить для каждого алгоритма (политики, стратегии) отдельный класс со стандартным интерфейсом.

Поскольку ценовая политика может изменяться по некоторому алгоритму, можно создать несколько классов `SalePricingStrategy`, каждый из которых содержит полиморфный метод `getTotal` (рис. 23.8). Каждому методу `getTotal` в качестве параметра нужно передавать объект `Sale`, чтобы объект ценовой политики мог найти и применить соответствующее правило вычисления скидок. Реализация метода `getTotal` может быть различной: для объекта `PercentDiscountPricingStrategy` скидки можно вычислять в процентном соотношении и т.п.

Объект стратегии связывается с *контекстным объектом* (context object) — объектом, к которому применяется алгоритм. В рассматриваемом примере контекстным объектом является `Sale`. При отправке объекту `Sale` сообщения `getTotal` он делегирует часть своих задач объекту стратегии, как показано на

рис. 23.9. Имена сообщений, передаваемых контекстному объекту и объекту стратегии, могут не совпадать, но в данном примере (и это довольно типично) выбрано одно и то же имя getTotal. Обычно контекстный объект передает объекту стратегии ссылку на самого себя (this) (это даже необходимо) для обеспечения параметрической видимости.

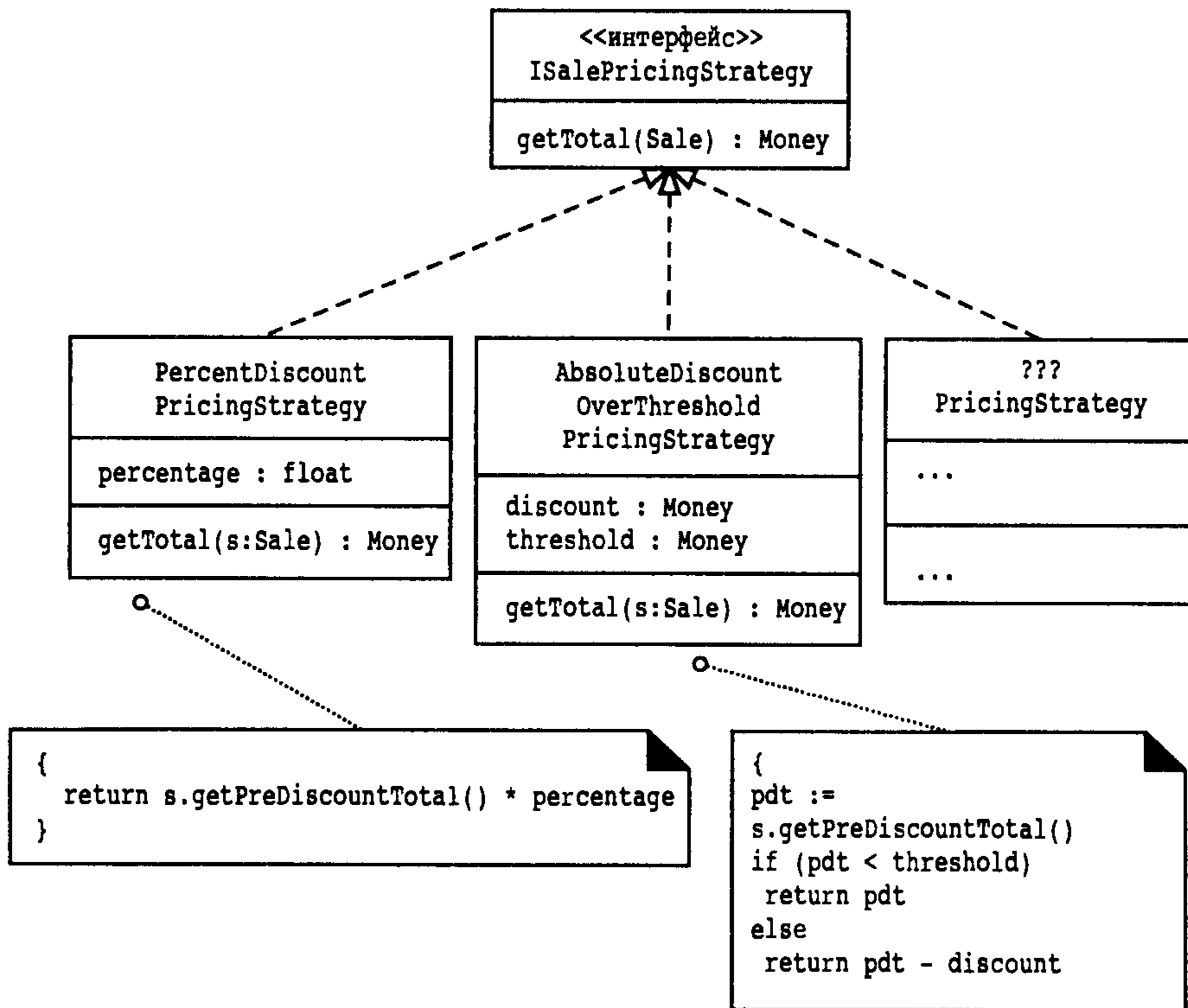



*Рис. 23.8. Классы для описания стратегий вычисления стоимости*

Обратите внимание, что контекстный объект Sale должен обеспечивать для своей стратегии видимость посредством атрибутов. Это отражено на диаграмме классов, представленной на рис. 23.10.

## Создание объекта стратегии на основе шаблона Factory

Существуют различные алгоритмы ценообразования, и они могут изменяться со временем. Кто должен разрабатывать стратегию? Наиболее логично использовать для решения этой проблемы шаблон Factory. Тогда объект PricingStrategyFactory будет отвечать за разработку всех стратегий (всех подключаемых или изменяемых алгоритмов или политик) в данном приложении. Как и объект ServicesFactory, он будет считывать имя класса реализации для стратегии ценообразования из системного свойства (или некоторого внешнего источника данных), а затем создавать экземпляр стратегии. При таком рефлексивном проектном решении политику ценообразования можно изменять в любое время в процессе работы приложения NextGen. При этом нужно лишь задавать новое имя класса создаваемой стратегии.

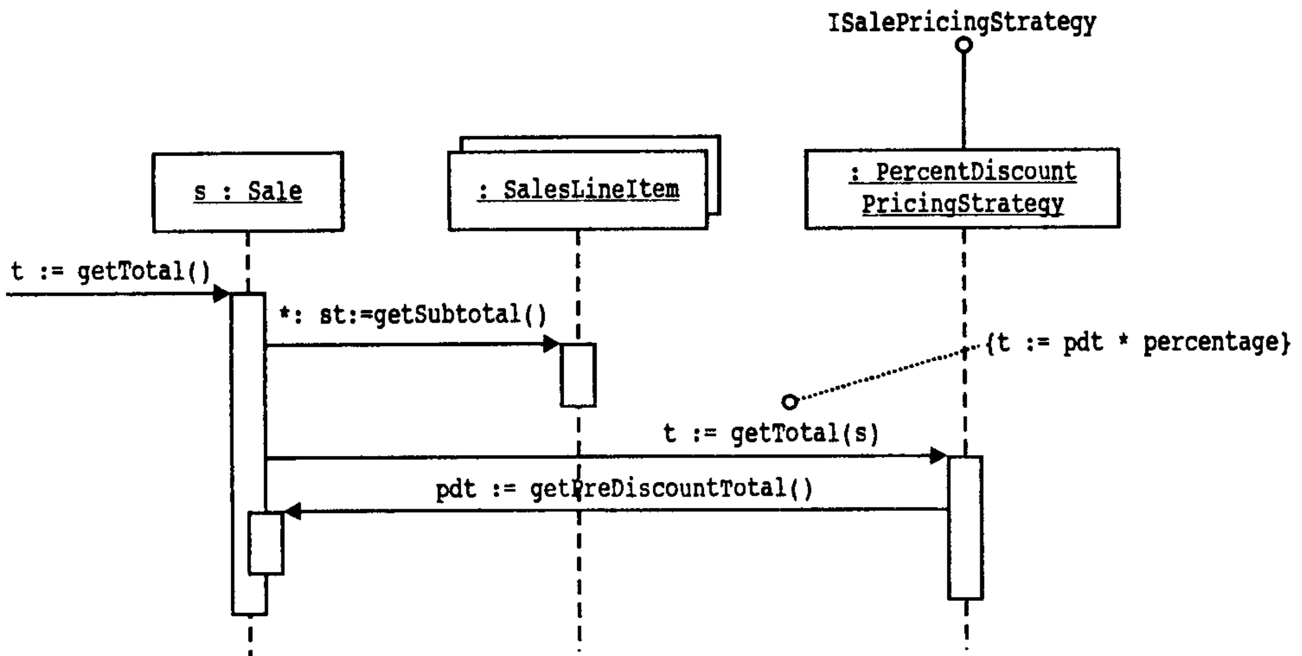



*Рис. 23.9. Стратегия в действии*

Заметим, что для создания объектов стратегий используется новый объект-фабрика, отличный от ServicesFactory. Такой подход обеспечивает высокую степень зацепления: каждая "фабрика" отвечает за создание семейства взаимосвязанных объектов.

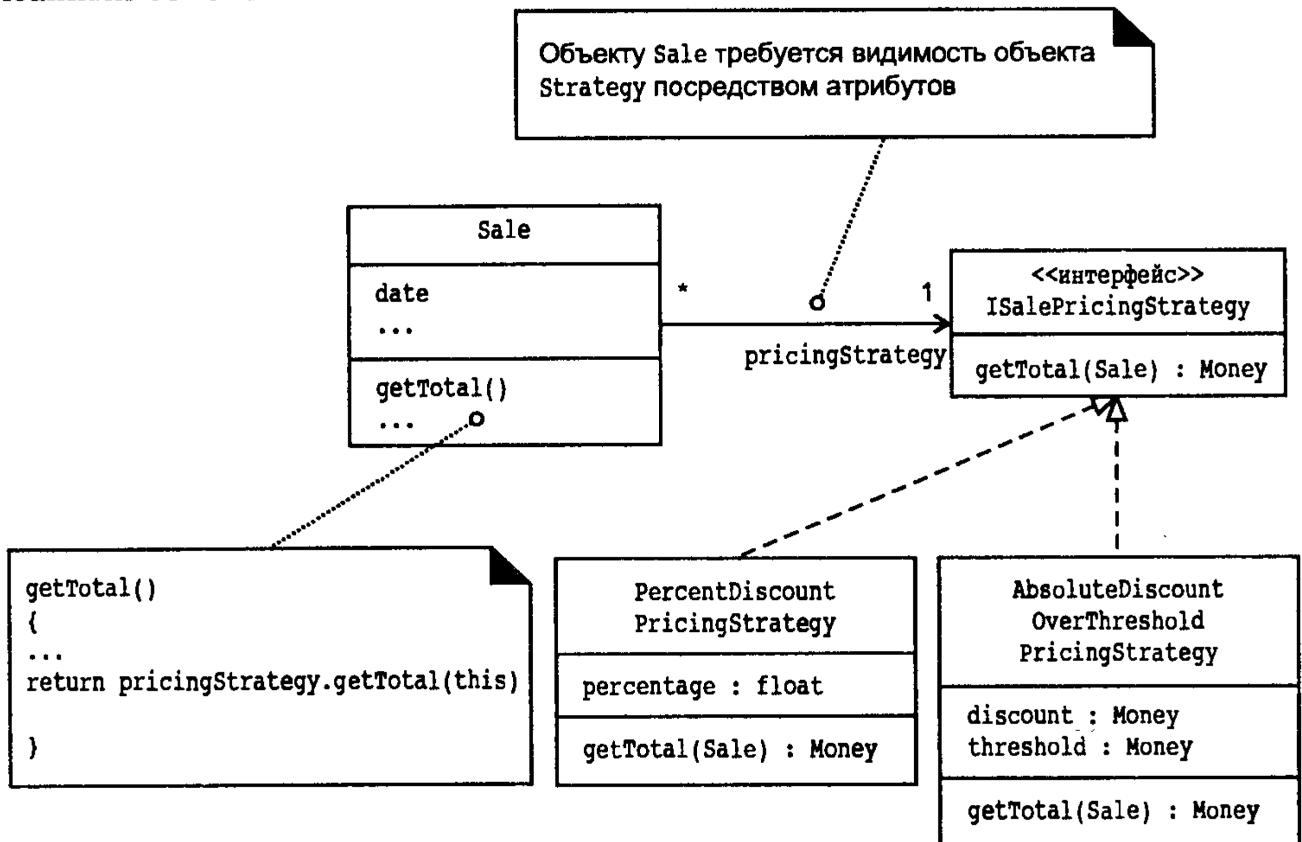



*Рис. 23.10. Контекстные объекты должны обеспечивать видимость своих стратегий посредством атрибутов*

Обратите внимание, что на рис. 23.10 ссылка (направленная ассоциация) указывает на интерфейс ISalePricingStrategy, а не на конкретный класс. Это означает, что ссылочный атрибут объекта Sale будет объявлен в терминах интерфейса, а не класса, и этому атрибуту будет соответствовать любая реализация этого интерфейса.

Поскольку политика ценообразования изменяется довольно часто (возможно даже ежечасно), то созданный экземпляр стратегии *не* желательно кэшировать в поле объекта PricingStrategyFactory. Лучше заново создавать этот объект при каждом использовании, считывая имя класса из внешнего свойства и инстанцируя объект стратегии.

Как и большинство объектов-фабрик, PricingStrategyFactory должен удовлетворять шаблону Singleton (иметь единственный экземпляр) (рис. 23.11).

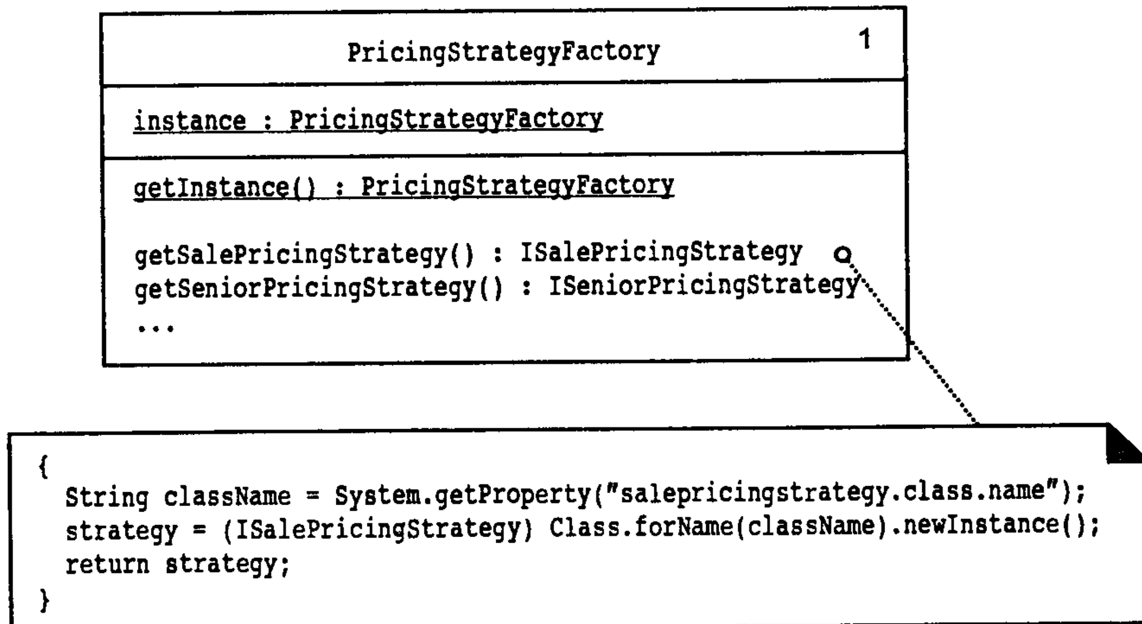



*Рис. 23.11. "Фабрика" стратегий*

При создании экземпляра объекта Sale он может обратиться к объекту-фабрике для определения стратегии ценообразования, как показано на рис. 23.12.

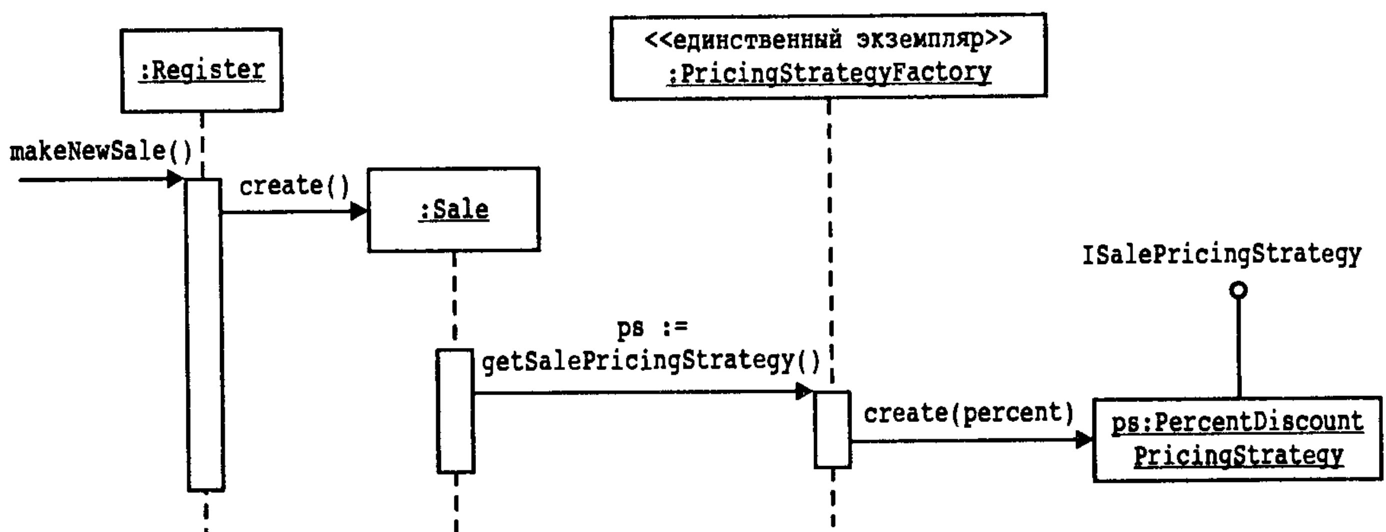



*Рис. 23.12. Создание объекта стратегии*

### Чтение и инициализация процентного соотношения

До сих пор не рассматривалась еще одна проблема проектирования, связанная с определением числовых значений скидок, вычисляемых в процентах или абсолютных значениях. Например, в понедельник значение скидки в процентах, задаваемое объектом `PersentageDiscountPricingStrategy`, должно составлять 10%, а во вторник — 20%.

Процентное соотношение скидки может зависеть от типа покупателя (например, для почетных граждан), а не только от времени.

Такие значения должны храниться на внешнем носителе, например в реляционной базе данных, чтобы их было легко изменить. Какие же объекты должны считывать эти значения и назначать стратегии? Логично поручить эту обязанность самому объекту `StrategyFactory`, поскольку именно он создает экземпляры стратегий ценообразования и может "знать", какое значение нужно считать ("текущей скидки", "скидки для почетных граждан" и т.д.)

Способы считывания этих значений из внешнего хранилища могут варьироваться от самых простых до очень сложных, таких как обычный запрос SQL через JDBC (при использовании Java-технологий) или взаимодействие с объектами путем перенаправления для сокрытия конкретного местоположения, языка запросов или типа хранилища данных. Анализируя точки вариации и эволюции в контексте хранения данных, следует оценить необходимость применения шаблона Protected Variations. Например, можно задать вопрос о том, насколько долговременным будет хранение информации в реляционной базе данных, взаимодействие с которой осуществляется с помощью SQL? Если тип хранилища данных не будет изменяться, то можно ограничиться использованием драйвера JDBC из объекта `StrategyFactory`.

### *Вывод*

Принципы шаблона Protected Variations при реализации динамически изменяемой стратегии ценообразования обеспечиваются применением шаблонов Strategy и Factory. Объекты стратегии строятся на основе принципа полиморфизма и обеспечивают механизмы подключения через общий интерфейс в рамках объектного проектного решения.

**Связанные шаблоны.** Шаблон Strategy основан на шаблоне Polymorphism и обеспечивает реализацию шаблона Protected Variations для изменяемых алгоритмов. Объекты стратегий зачастую создаются с помощью шаблона Factory.

## 23.7. Шаблон Composite (GoF) и другие принципы проектирования

Рассмотрим еще одну интересную проблему проектирования и формулировки требований. Как обрабатывать многочисленные противоречивые политики ценообразования? Допустим, к примеру, что на сегодняшний день в магазине действуют следующие политики.

- 20%-ная скидка для почетных граждан
- Скидка в размере 15% для постоянных клиентов при покупке товаров на сумму свыше $400

- По понедельникам действует скидка в размере $50 при покупке товаров на сумму свыше $500
- При покупке 1 ящика чая Darjeeling клиент получает скидку на все остальные товары в размере 15%

Предположим, почетный гражданин, который является также постоянным покупателем, приобретает 1 ящик чая Darjeeling и на $600 овощных котлет (поистине фанатичный вегетарианец и любитель чая!). Какую политику ценообразования применить в данном случае?

Уточним ситуацию. Действующие на данный момент стратегии ценообразования определяются тремя факторами:

- временем (понедельник);
- типом покупателя (почетный гражданин);
- типом выбранного продукта (чай Darjeeling).

Еще одно уточнение: три из четырех примеров политик по существу являются процентными скидками.

Для решения этой проблемы нужно определить *стратегию разрешения конфликтов* (conflict resolution strategy) для данного магазина. Обычно магазины выбирают лучшую для покупателя (обеспечивающую самую низкую стоимость) стратегию, однако это не обязательное требование. Например, в трудный период магазин может выбрать стратегию "наивысшей стоимости".

Первый важный момент: одновременно существует множество стратегий, причем некоторые из них могут применяться к одной продаже. Второй: политика ценообразования может быть связана с типом покупателя (например, постоянный). Отсюда вывод — тип покупателя должен быть известен объекту `StrategyFactory` на момент создания стратегии ценообразования для данного продукта.

Существует ли способ изменения проектного решения таким образом, чтобы объект `Sale` не обладал информацией о количестве применяемых к нему стратегий, но оставалась возможность разрешения конфликтов? Да, это решение обеспечивается шаблоном Composite.

> *Шаблон Composite*
>
> *Контекст/Проблема*
>
> Как обрабатывать группу или композицию структур объектов одновременно (полиморфно) как не композитный (атомарный) объект?
>
> *Решение*
>
> Определить классы для композитных и атомарных объектов таким образом, чтобы они реализовали один и тот же интерфейс.

Например, можно создать новый класс
`CompositeBestForCustomerPricingStrategy`
(во всяком случае, это имя содержательно), реализующий интерфейс `ISalePricingStrategy` и содержащий другие объекты `ISalePricingStrategy`. Идея этого решения представлена на рис. 23.13.

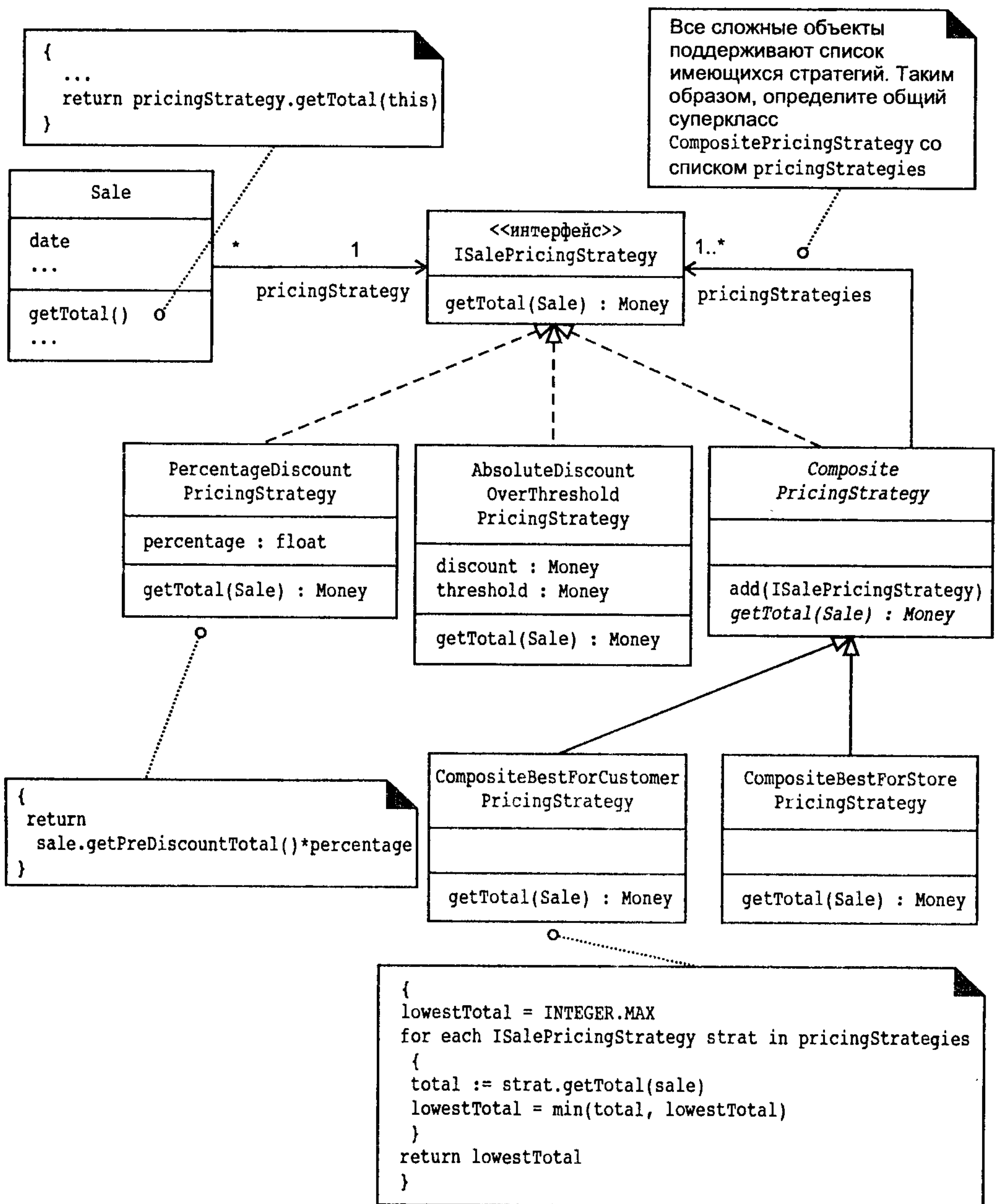



*Рис. 23.13. Применение шаблона Composite*

Обратите внимание, что в этом проектном решении композитный класс CompositeBestForCustomerPricingStrategy наследует атрибут pricingStrategies, содержащий список других объектов ISalePricingStrategy. Это отличительная особенность композитных объектов — внешний композитный объект содержит список внутренних объектов, реализующих

тот же интерфейс, что и внешний объект. Таким образом, композитный класс сам реализует интерфейс ISalePricingStrategy.

Теперь с объектом Sale можно связать либо композитный объект CompositeBestForCustomerPricingStrategy (включающий в себя другие стратегии), либо атомарный объект PercentDiscountPricingStrategy. Причем объект Sale даже не будет знать о том, какой тип стратегии (композитная или атомарная) с ним связан, поскольку для объекта Sale это просто объект, реализующий интерфейс ISalePricingStrategy и "понимающий" сообщение getTotal (рис. 23.14).

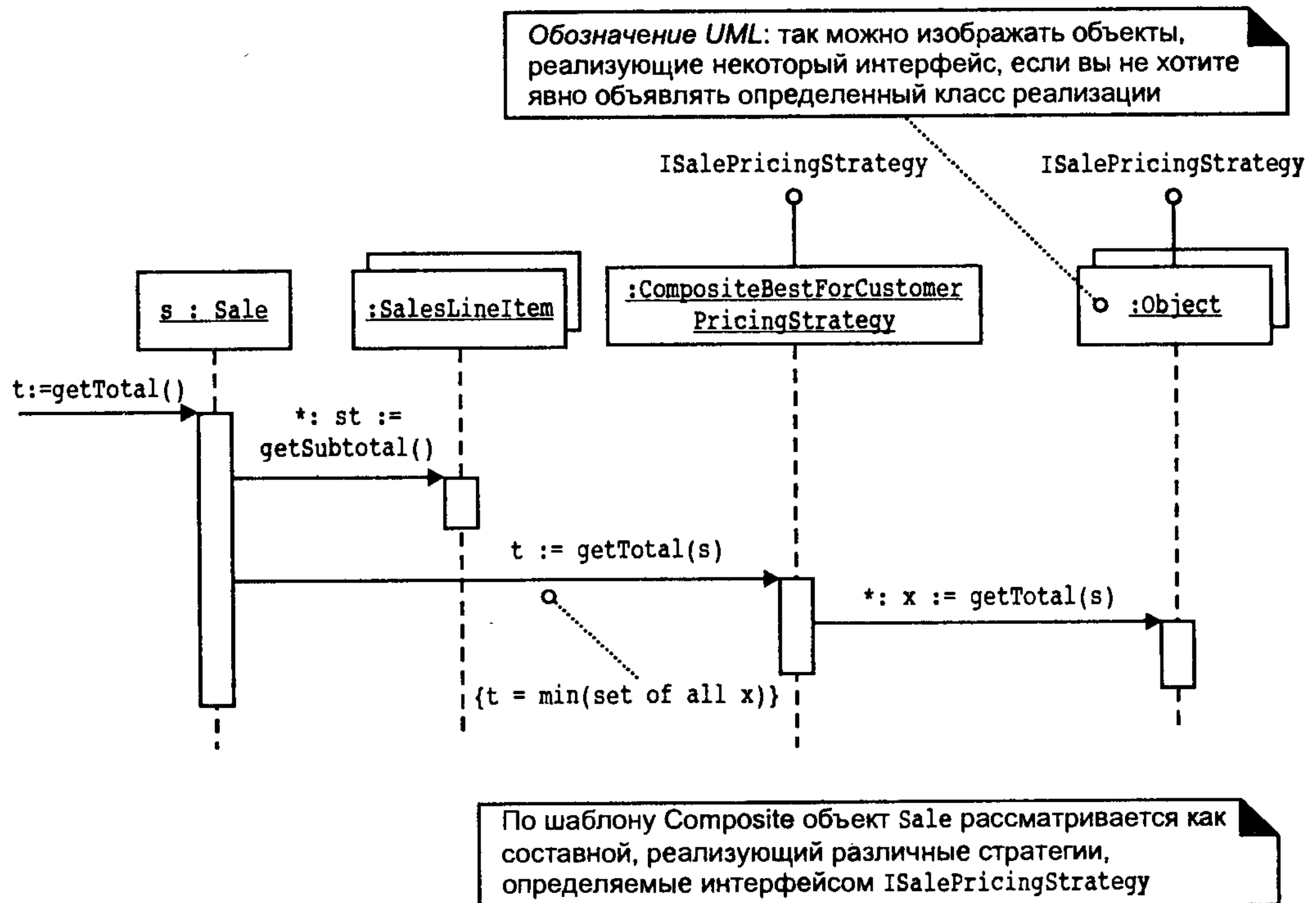



*Рис. 23.14. Взаимодействие на основе шаблона Composite*

Обратите внимание, как на рис. 23.14 показаны объекты, реализующие интерфейс, если не нужна точная реализация класса. Конкретный класс просто показан как объект Object безо всяких комментариев. Это типичное графическое представление.

Для большей ясности рассмотрим реализацию класса CompositePricingStrategy у одного из его подклассов на языке Java.

```
//суперкласс, от которого все производные классы
//могут наследовать список стратегий
public abstract class CompositePricingStrategy
    implements ISalePricingStrategy
{

protected List pricingStrategies = new ArrayList();
```

```
public add(ISalePricingStrategy s)
{
    pricingStrategies.add(s);
}

public abstract Money getTotal(Sale sale);

}// конец класса


// композитная стратегия, возвращающая самую низкую стоимость
// среди всех стоимостей, определяемых внутренними стратегиями
public class CompositeBestForCustomerPricingStrategy
extends CompositePricingStrategy
{

public Money getTotal(Sale sale)
{
    Money lowestTotal new Money (Integer.MAX_VALUE);
    // цикл для всех внутренних стратегий
    for (Iterator i = pricingStrategies.iterator(); i.hasNext();)
    {
        ISalePricingStrategy strategy =
            (ISalePricingStrategy)i.next();
        Money total = strategy.getTotal(sale);
        lowestTotal = total.min(lowestTotal);
    }
return lowestTotal;
}

} // конец класса
```

На рис. 23.13 введены некоторые новые обозначения UML для иерархии классов и наследования. Эти обозначения поясняются на рис. 23.15.

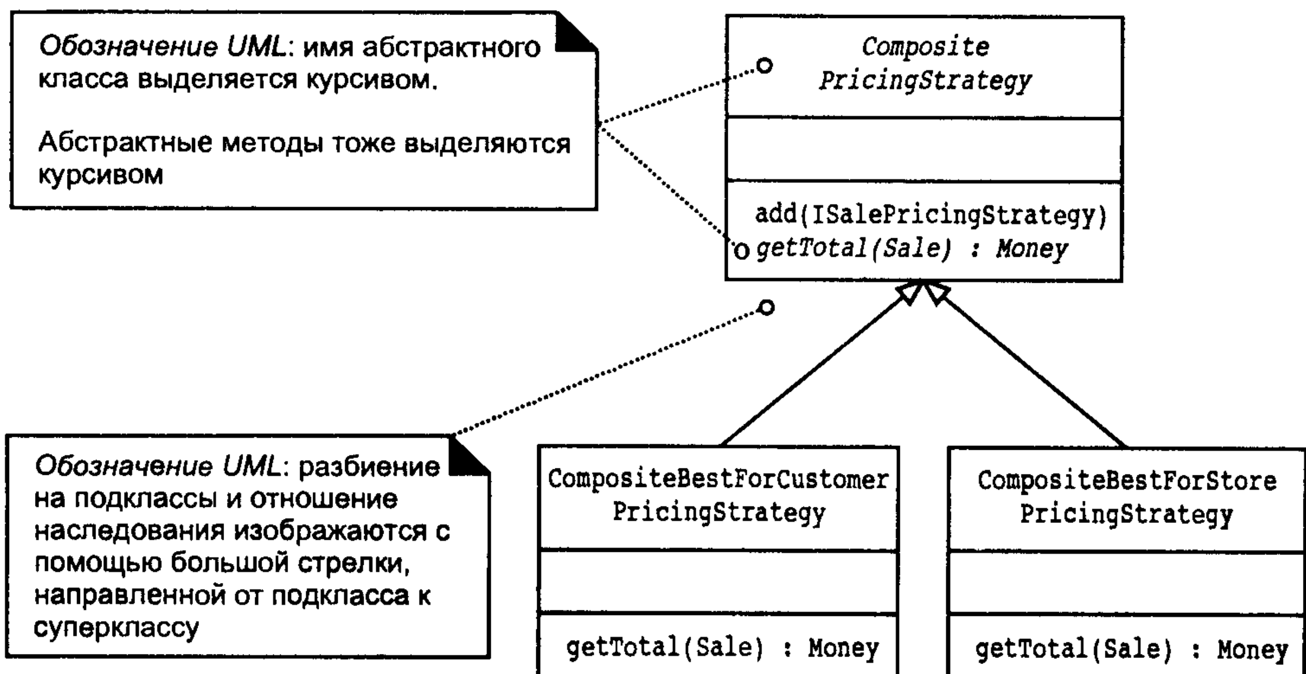



*Рис. 23.15. Абстрактные суперклассы, абстрактные методы и наследование в UML*

### Создание нескольких стратегий SalePricingStrategy

На основе шаблона Composite была создана группа из нескольких (противоречивых) стратегий ценообразования, которые с точки зрения объекта Sale выглядели как одна стратегия. Композитный объект, содержащий эту группу, тоже реализует интерфейс ISalePricingStrategy. Теперь возникает новая (еще более интересная) проблема: когда создавать эти стратегии?

Сначала нужно создать композитный объект, содержащий действующие на данный момент политики скидок (если никакие скидки не действуют, то процентная скидка будет равна 0%), некий PersentageDiscountPricingStrategy. Затем, если в некоторый момент понадобится применить новую стратегию ценообразования (скажем, для постоянных клиентов), то ее можно легко добавить к композитному объекту с помощью унаследованного метода CompositePricingStrategy.add().

Вот три возможных стратегии с учетом времени их добавления.

1. Текущая скидка для данного магазина, добавляемая при создании экземпляра Sale.
2. Скидка для определенных типов покупателей, добавляемая при вводе типа покупателя в POS-систему.
3. Скидка для определенного типа товара (при покупке чая Darjeeling — 15% от общей стоимости покупки), добавляемая при выборе этого типа товара.

Проектное решение для первого случая показано на рис. 23.16. Как и в описанном выше исходном проектном решении, имя класса стратегии считывается как системное свойство, а процентное соотношение скидки поступает из внешнего источника данных.

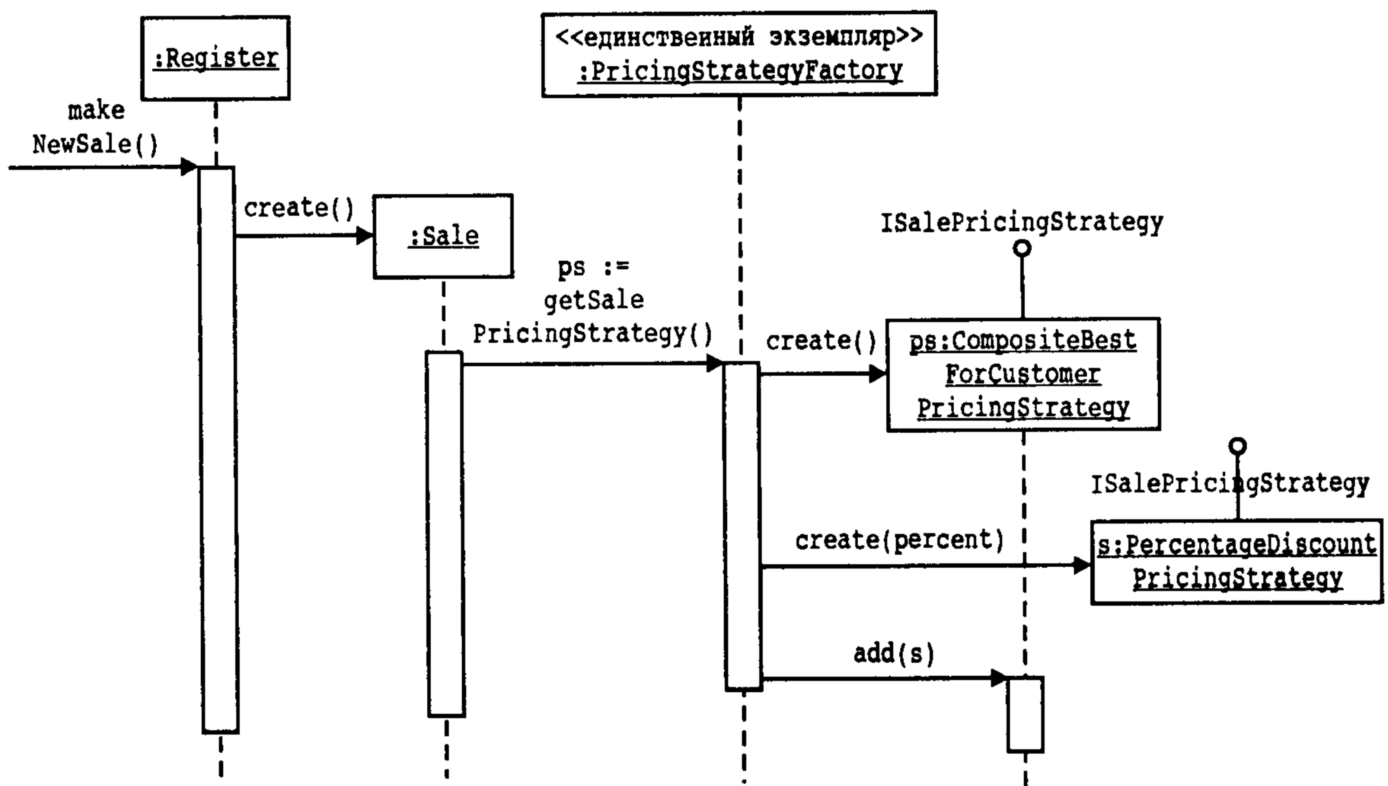



*Рис. 23.16. Создание композитной стратегии*

При рассмотрении второго вида скидок, определяемого типом покупателя, обратимся к расширению описания прецедента, в котором впервые упоминается это требование.

**Прецедент ПР1:** Оформление продажи

...

**Расширения (альтернативные сценарии)**

5б. Покупатель сообщает о положенной ему скидке (например, для сотрудников данного предприятия или постоянных покупателей).

1. Кассир отправляет запрос на скидку.
2. Кассир вводит идентификационные данные покупателя.
3. Система предоставляет сумму скидки, вычисленную на основе дисконтных правил.

Из этого описания видна необходимость новой системной операции для POS-приложения. Помимо существующих операций makeNewSale, enterItem, endSale и makePayment, требуется пятая системная операция enterCustomerForDiscount. Она может дополнительно выполняться после операции endSale. Для ее выполнения необходимо иметь идентификаторы покупателей customerID. Их можно считывать с карточки или вводить с клавиатуры.

Проектное решение для второго случая показано на рис. 23.17 и 23.18. Не удивительно, что объект-фабрика отвечает за создание дополнительных стратегий ценообразования. Он может создать новую стратегию PercentageDiscountPricingStrategy, предоставляющую, например, скидки постоянным клиентам. Как и в исходном проектном решении, имя класса считывается в качестве системного свойства, обеспечивая выполнение шаблона Protected Variations. Благодаря применению шаблона Composite, с объектом Sale можно связать несколько противоречивых стратегий ценообразования, но для самого объекта они будут выглядеть как единая стратегия.

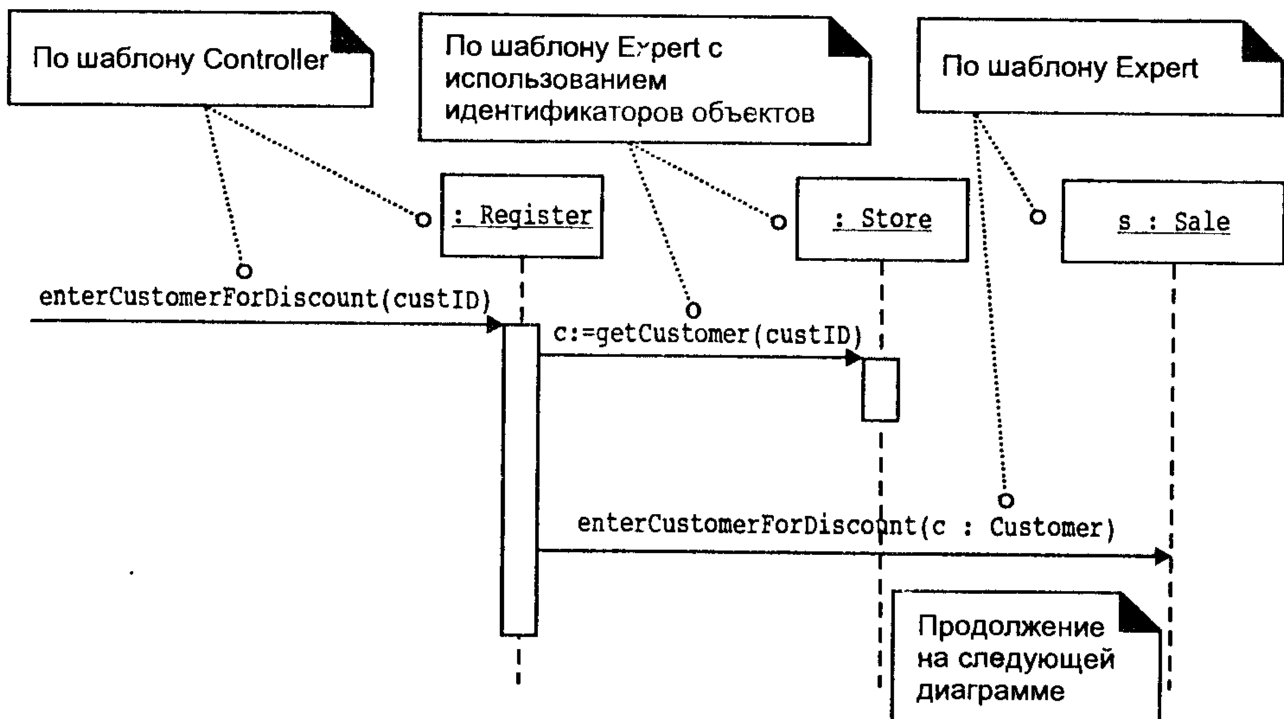



*Рис. 23.17. Создание стратегии ценообразования для конкретного типа покупателя (часть I)*

Рис. 23.17. и 23.18 иллюстрируют важную идею UML, относящуюся к диаграммам взаимодействия, — разбиение одной диаграммы на две части для удобства чтения.

## Шаблоны GRASP и другие принципы проектирования

Рассуждая в терминах шаблонов GRASP, можно задать вопрос: почему во втором случае объект `Register` не отправляет сообщение объекту `PricingStrategyFactory` о создании новой стратегии, а затем не передает ее объекту `Sale`? Одна из причин состоит в соответствии шаблону Low Coupling. Объект `Sale` уже связан с фабрикой. Если с объектом-фабрикой будет взаимодействовать и объект `Register`, то повысится уровень связывания объектов приложения. Более того, объект `Sale` является информационным экспертом, обладающим информацией о текущей стратегии ценообразования (которая может изменяться). Поэтому согласно шаблону Expert, эту обязанность нужно делегировать объекту `Sale`.

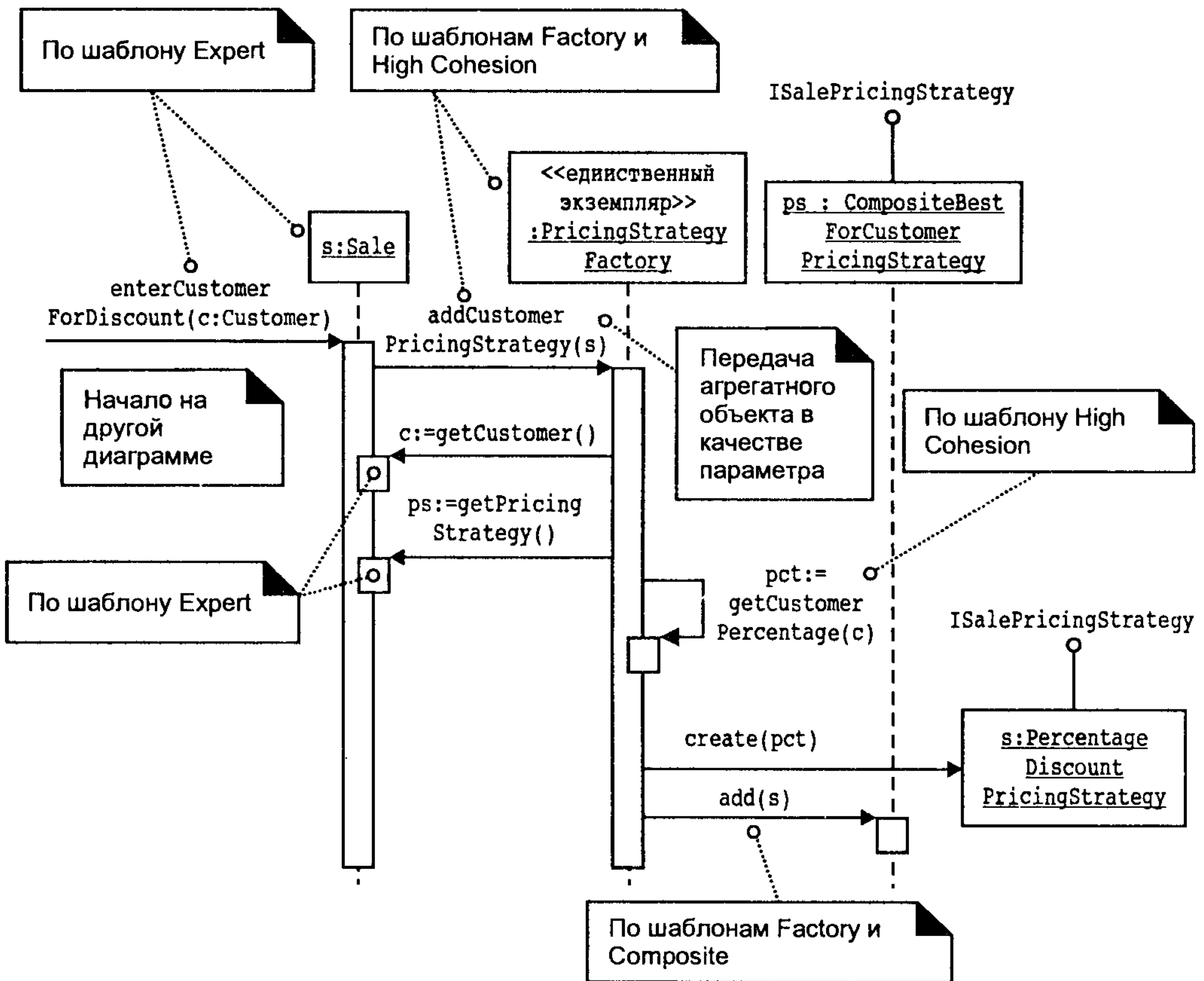



*Рис.23.18. Создание стратегии ценообразования для конкретного типа покупателя (часть II)*

Обратите внимание, что в этом проектном решении идентификатор `customerID` трансформируется в объект `Customer`, когда объект `Register` от-

правляет запрос объекту Store на получение объекта Customer по его идентификатору. Обязанность по предоставлению типа покупателя getCustomer возлагается на объект Store, согласно шаблону Expert. Запрос отправляется объектом Register, поскольку для этого объекта обеспечивается видимость объекта Store посредством атрибутов. Если бы запрос направлял объект Sale, то он должен был бы иметь ссылку на объект Store, что, в свою очередь, повысило бы уровень связывания в разрез с шаблоном Low Coupling.

#### *Преобразование идентификаторов в объекты*

Зачем преобразовывать идентификатор customerID (возможно, числовой) в объект Customer? Это типичная ситуация для объектного проектирования. В рамках этого подхода ключи и идентификаторы зачастую преобразуются в реальные объекты. Такое преобразование обычно выполняется вскоре после передачи идентификатора с уровня интерфейса пользователя на уровень логики приложения. Такой прием не оформлен в виде именованного шаблона, но является хорошим кандидатом в шаблоны, поскольку встречается довольно часто. Его можно назвать шаблоном ID to Objects. Даже если на ранних стадиях проектирования разработчики планируют обойтись числовым идентификатором, впоследствии может возникнуть необходимость во введении реального объекта Customer, обладающего собственным поведением (например, согласно шаблону Expert) и инкапсулирующего всю информацию о покупателе. Такое решение повышает гибкость разработки. Заметим, что еще одним примером применения шаблона IDs to Objects является преобразование идентификатора itemID в объект ProductSpecification.

#### *Передача агрегатного объекта в качестве параметра*

При передаче сообщения addCustomerPricingStrategy(s:Sale) объекту-фабрике в качестве параметра выступает объект Sale, а затем объект-фабрика запрашивает объекты Customer и PricingStrategy у объекта Sale.

Почему не извлечь эти объекты из объекта Sale и не передать их объекту-фабрике? Ответ на этот вопрос заключается в еще одной идиоме объектного проектирования. Дочерние объекты обычно не извлекаются из агрегатных объектов и не передаются отдельно. Как правило, передается сам агрегатный объект.

Этот принцип повышает гибкость проектного решения, поскольку объект-фабрика может взаимодействовать с объектом Sale, в результате чего повышается производительность и устраняется необходимость передачи конкретных объектов, необходимых объекту-фабрике. Эта идиома тоже не имеет имени, но связана с шаблонами Low Coupling и Protected Variations. Ее можно было бы назвать Pass Aggregate Object as Parameter.

### *Резюме*

Рассмотренная проблема решена в рамках нескольких шаблонов и советов объектного проектирования. Опытный разработчик хранит в памяти множество шаблонов и принципов проектирования, в том числе шаблоны семейства GRASP.

Несмотря на то, что шаблон Composite введен для объектов, соответствующих шаблону Strategy, его можно применять не только для стратегий, но и для любых других объектов. Например, зачастую создают макрокоманды — коман-

ды, включающие другие команды. Это тоже делается на основе шаблона Composite. Шаблон Command будет рассмотрен в последующих главах.

*Упражнения*

Упражнение 1.

При покупке некоторого товара дается скидка на всю покупку. Например, при покупке 1 ящика чая Darjeeling обеспечивается скидка в размере 15% на все купленные товары.

Упражнение 2.

Все рассмотренные выше политики ценообразования применялись ко всей покупке сразу. Наиболее интересно рассмотреть скидки на отдельные товары. Например:

- покупая два костюма, третий получаете бесплатно;
- при покупке трех компьютеров типа X, получаете скидку в размере 50% на принтер типа Y.

Можно ли предложить "элегантное" решение этих проблем на основе шаблона Strategy?

**Связанные шаблоны.** Шаблон Composite зачастую применяется вместе с шаблонами Strategy и Command. Он основан на принципах шаблона Polymorphism и обеспечивает реализацию Protected Variations для клиента, поскольку клиенту не нужно знать, с чем он связан — с композитным или атомарным объектом.

## 23.8. Шаблон Facade (GoF)

Еще одним требованием для этой итерации является реализация подключаемых бизнес-правил. Это значит, что в некоторых фиксированных точках сценария, например, при выполнении операций `makeNewSale` или `enterItem` в рамках прецедента `Оформление продажи`, различные пользователи могут слегка модифицировать поведение системы NextGen.

Уточним сказанное. Предположим, требуется определить правила, отменяющие некоторые действия. Рассмотрим следующие примеры.

- Допустим, при создании нового экземпляра продажи можно учесть наличие подарочного сертификата. В магазине можно ввести правило, ограничивающее количество покупаемых товаров при наличии подарочного сертификата одной единицей. И тогда все операции `enterItem`, кроме первой, должны быть отменены.
- Если покупка оплачивается с помощью подарочного сертификата, необходимо отменить все типы сдачи, кроме других подарочных сертификатов. Например, если кассир вводит запрос на вычисление сдачи наличными, то этот запрос необходимо отменить.
- При создании новой продажи можно оформить ее как благотворительную акцию магазина. Для подобных акций можно использовать товары, стоимость которых не превышает $250, при этом в качестве кассира должен зарегистрироваться менеджер магазина.

В терминах анализа требований необходимо идентифицировать конкретные точки реализации сценария (операции `enterItem`, `chooseCashChange` и т.д.). Далее будем рассматривать только операцию `enterItem`, хотя предложенное решение подходит и для других операций.

Допустим, архитектор системы хочет использовать программное решение, оказывающее минимальное влияние на существующие программные компоненты. То есть он хочет выработать общее решение для различных ситуаций, детали реализации которого для каждой из этих ситуаций различны. Допустим, он не знает, какую реализацию использовать для данных подключаемых правил, и хочет поэкспериментировать с разными решениями для представления, загрузки и обработки правил. Например, правила можно реализовать с помощью шаблона Strategy или с помощью свободно распространяемых интерпретаторов, считывающих и интерпретирующих набор правил `if-then`, либо на основе коммерческих интерпретаторов правил.

Для решения этой проблемы используется шаблон Facade.

> *Шаблон Facade*
>
> *Контекст/Проблема*
>
> Как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например с подсистемой, если нежелательно высокое связывание с этой подсистемой или реализация подсистемы может измениться?
>
> *Решение*
>
> Определить одну точку взаимодействия с подсОистемой — фасадный объект, — обеспечивающий общий интерфейс с подсистемой, и возложить на него обязанность по взаимодействию с ее компонентами.

Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы.[4] Реализация других компонентов подсистемы закрыта и не видна внешним компонентам. Фасадный объект обеспечивает реализацию шаблона Protected Variations с точки зрения защиты от изменений в реализации подсистемы.

Например, можно определить подсистему "генератора правил", конкретная реализация которой неизвестна. Она может обрабатывать набор правил и сообщать об отмене операций.

Фасадный объект для такой подсистемы можно назвать `POSRuleEngineFacade`. Разработчик может обращаться к этому объекту в начале метода, определяющего точку подключения правил, как в следующем примере.

```
public class Sale
{
public void makeLineItem( ProductSpecification spec, int quantity)
{
    SalesLineItem sli = new SalesLineItem(spec, quantity);
```

---

[4] В данном случае термин "подсистема" используется в неформальном смысле для обозначения отдельной группы взаимосвязанных компонентов, а не в традиционном смысле UML.

```
        // обращение к фасадному объекту
    if (POSRuleEngineFacade.getInstance().isInvalid(sli, this))
        return;

lineItems.add(sli);
}

//...

}//конец описания класса
```

Обратите внимание на использование шаблона Singleton. Доступ к фасадным объектам зачастую обеспечивается посредством этого шаблона.

В этом примере детали реализации и ее сложность скрыты в подсистеме "генератора правил", доступ к которой осуществляется через объект `POSRuleEngineFacade`. Заметим, что подсистема, скрываемая за "фасадом", может содержать десятки или сотни объектов, либо даже вообще не содержать объектно-ориентированного решения. Внешний пользователь будет видеть только открытую часть этой подсистемы.

При этом различные аспекты реализации обеспечиваются разными частями подсистемы.

## Резюме

Шаблон Facade достаточно прост, поэтому он широко используется. Он призван скрывать подсистему за "фасадом" одного объекта.

*Упражнения*

Упражнение 1.

Предложите способ обработки правил на основе шаблона Strategy, при котором имена классов динамически считываются из внешнего источника.

Упражнение 2.

При реализации на Java воспользуйтесь интерпретатором правил Jess, бесплатно распространяемым в академических целях, который можно найти по адресу `http://herzberg.ca.sandia.gov/jess/`.

**Связанные шаблоны.** К фасадным объектам доступ зачастую обеспечивается посредством шаблона Singleton. Шаблон Facade обеспечивает реализацию шаблона Protected Variations для реализации подсистем за счет добавления объекта, удовлетворяющего шаблону Indirection с целью выполнения шаблона Low Coupling. Внешние объекты "общаются" с подсистемой через фасадный объект.

Как было сказано ранее, для доступа к внешним подсистемам с разными подсистемами можно использовать шаблон Adapter. Это тоже разновидность фасадного объекта, обеспечивающего взаимодействие с разными интерфейсами.

В UML существует обозначение для группы некоторых объектов — так называемого *пакета* (package). Оно напоминает пиктограмму папки с корешком. Пакеты используются для отображения логической группировки объектов и могут соответствовать пакетам в Java, пространствам имен в C++ либо другим логически обоснованным компонентам или подсистемам. На рис. 23.19 открытой частью пакета является только объект `POSRuleEngineFacade`.

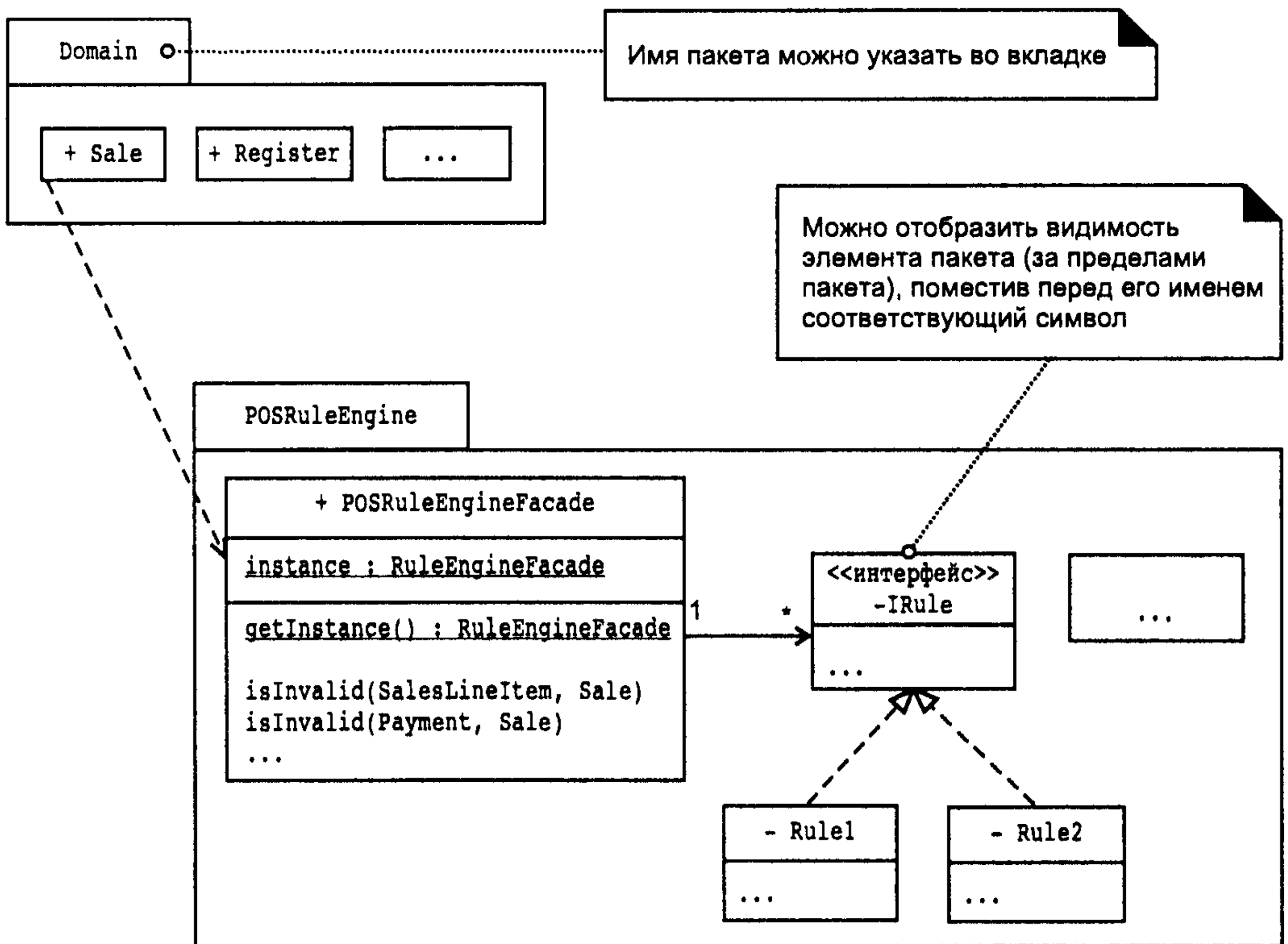



*Рис. 23.19. Обозначение пакета в языке UML*

Для изображения подсистем существуют и другие обозначения, однако мы пока ограничимся только показанными на рис. 23.19. Использование пакетов более подробно будет описано при рассмотрении задач следующей итерации.

## 23.9. Шаблон Observer/Publish-Subscribe/ Delegation Event Model (GoF)

Еще одним требованием для данной итерации является обеспечение возможности обновления окна интерфейса пользователя при изменении общей стоимости покупки (рис. 23.20). Решив эту проблему, на последующих итерациях можно воспользоваться полученным проектным решением для обновления окна интерфейса пользователя при изменении каких-либо других данных.

Почему не воспользоваться следующим решением? При изменении общей стоимости покупки объект `Sale` передает сообщение объекту окна с требованием обновить его содержимое.

Такое решение не согласуется с принципом шаблона Model-View Separation. Согласно этому принципу, объекты уровня предметной области (в частности, `Sale`) не должны знать о представлении объектов в окнах. Это обеспечивает слабое связывание между уровнями интерфейса и объектов предметной области.

Преимуществом такого подхода является возможность замены интерфейса пользователя, не затрагивая объекты уровня предметной области. Если объекты

модели "не знают" об объектах Java Swing, то можно отключить интерфейс Swing и заменить его другими объектами.

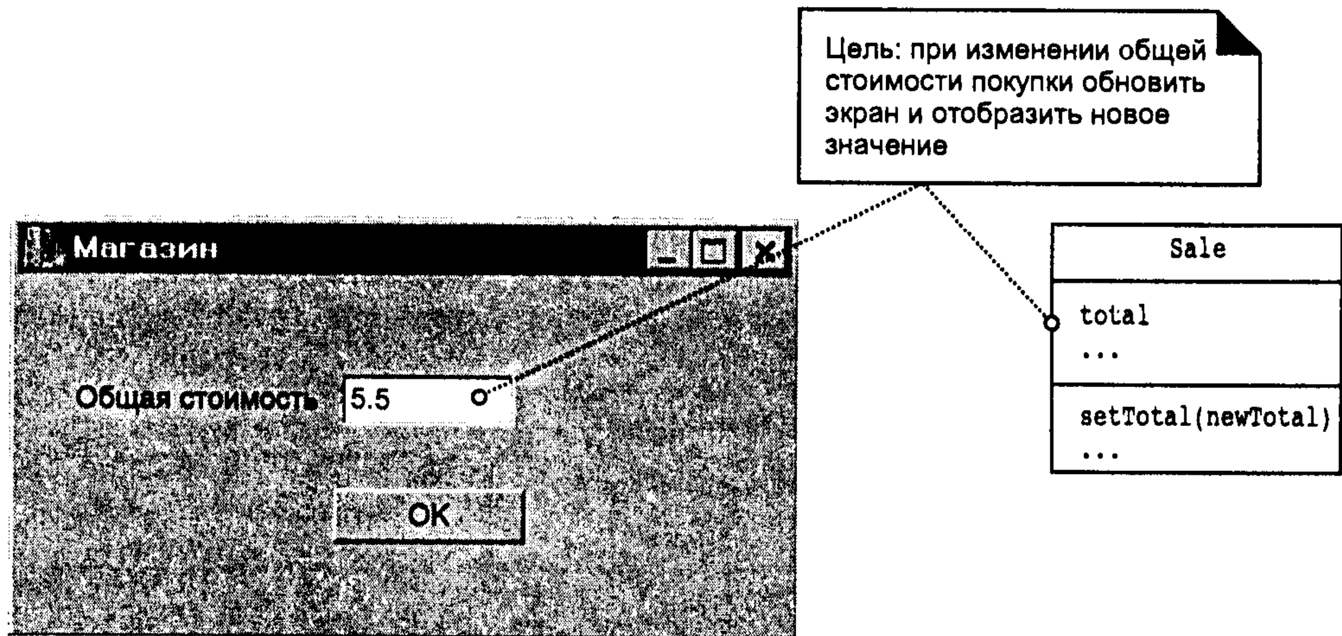



*Рис. 23.20. Обновление интерфейса при изменении общей стоимости покупки*

Таким образом, шаблон Model-View Separation поддерживает требования шаблона Protected Variations при изменении интерфейса пользователя.

Для решения этой проблемы проектирования используется шаблон Observer (Наблюдатель) или Publish-Subscribe (Опубликовать-подписаться).

> *Шаблон Observer (Publish-Subscribe)*
>
> *Контекст/Проблема*
>
> Различные классы объектов-подписчиков должны знать об изменении состояния или некоторых событиях другого объекта и соответствующим образом реагировать на эти изменения. Более того, необходимо поддерживать низкий уровень связывания с объектами-подписчиками. Что делать?
>
> *Решение*
>
> Определить интерфейс "подписки" или "прослушивания". Объекты-подписчики реализуют этот интерфейс и динамически регистрируются для получения информации о некотором событии. Затем при реализации этого события оповещаются все объекты-подписчики.

Пример такого решения показан на рис. 23.21.

Основная идея этого решения сводится к следующему.

1. Определяется интерфейс. В данном случае это интерфейс `PropertyListener` с операцией `onPropertyEvent`.
2. Определяется окно для реализации интерфейса.
   - Окно `SaleFrame1` будет реализовать метод `onPropertyEvent`.
3. При инициализации окна `SaleFrame1` ему передается экземпляр `Sale` для отображения общей стоимости покупки.

4. Объект окна `SaleFrame1` регистрирует (или подписывается на) экземпляр `Sale` для уведомления о данном событии посредством сообщения `addPropertyListener`. Тогда при изменении свойства (например, общей стоимости) объект окна будет получать уведомление.
5. Заметим, что объект Sale "не знает" о существовании объекта `SaleFrame1`. Он лишь "знает" о существовании объектов, реализующих интерфейс `PropertyListener`. Это снижает уровень связывания объекта `Sale` с окном. Объект `Sale` связан только с интерфейсом, а не с классом GUI.
6. Экземпляр `Sale` публикует информацию о событиях. При изменении общей стоимости покупки эта информация передается всем подписчикам `PropertyListener`.

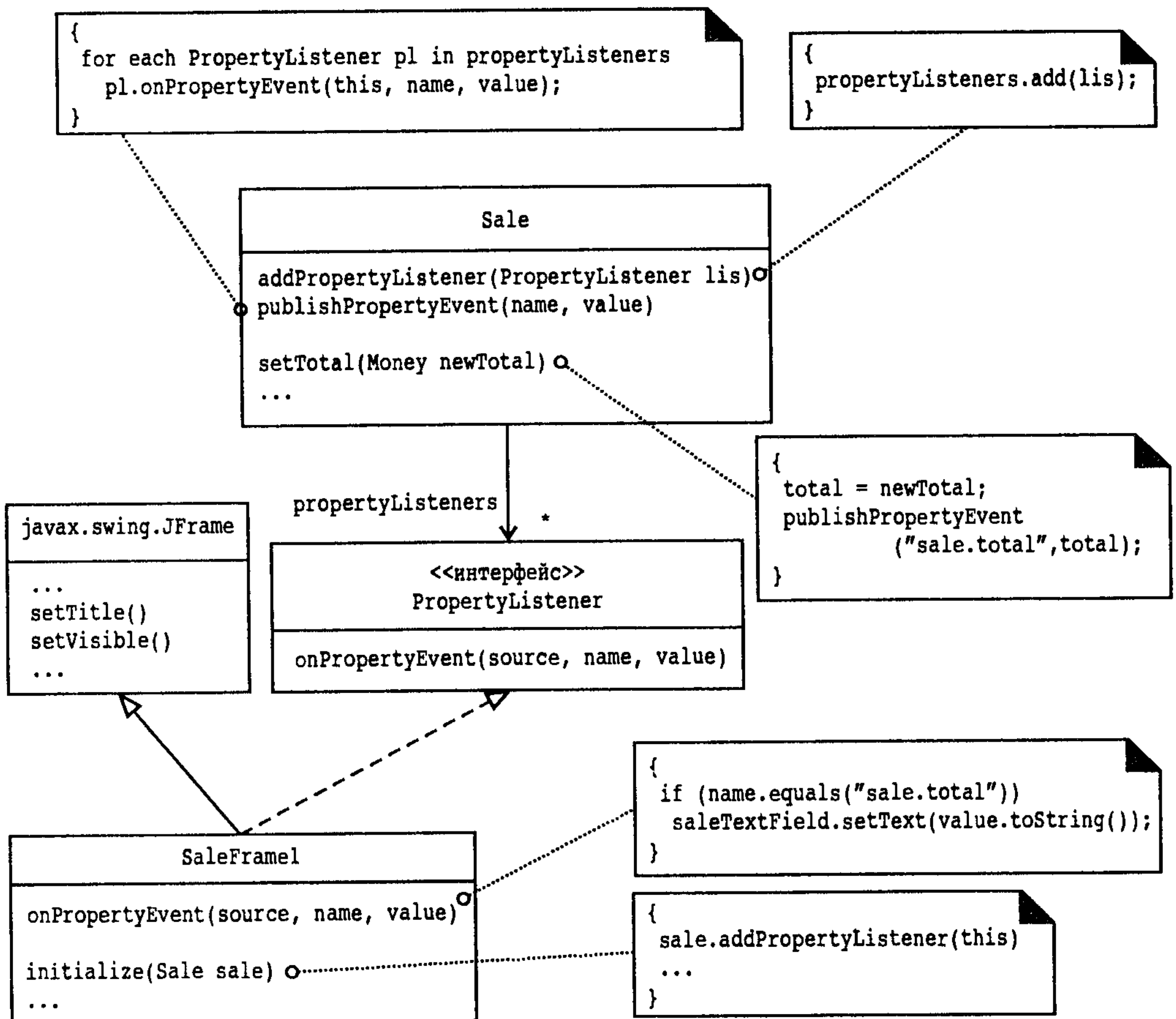



*Рис. 23.21. Шаблон Observer*

Обратите внимание на отображение реализации наиболее важных методов на рис. 23.21. Это добавляет информацию о динамическом поведении объектов на статическую диаграмму. В некоторых случаях диаграммы классов с подобными пояснениями могут полностью заменить собой диаграммы взаимодействия.

Это не значит, что следует вообще отказаться от диаграмм взаимодействия. Просто существуют и другие подходы.

Объект SaleFrame1 в данном случае выступает наблюдателем или подписчиком. Из рис. 23.22 видно, что он "подписывается" на получение информации о событиях объекта Sale. Объект Sale добавляет SaleFrame1 в свой список подписчиков PropertyListener. При этом он "не знает" о существовании самого объекта SaleFrame1, а "знает" лишь о существовании нового объекта PropertyListener. Это снижает уровень связывания между уровнями приложения.

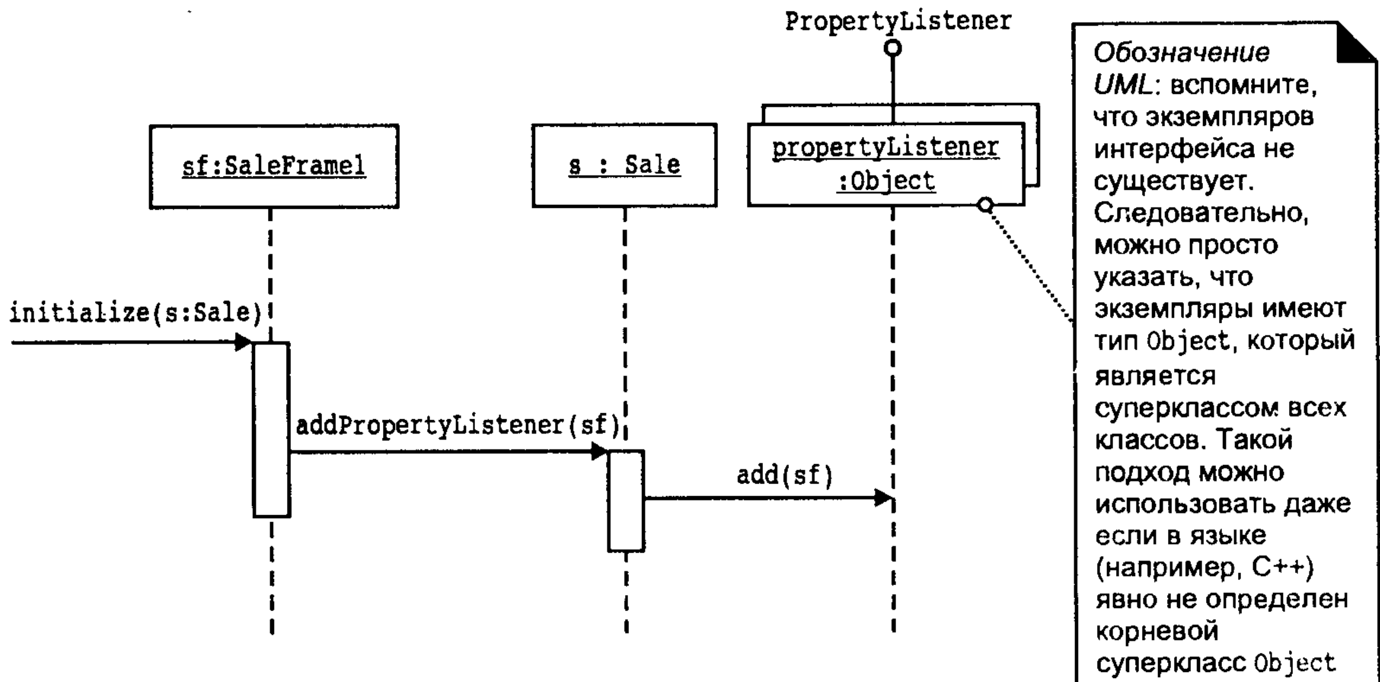



*Рис 23.22. Наблюдатель SaleFrame1 подписывается на получение информации об изменении свойств объекта Sale*

Как видно из рис. 23.23, при изменении общей стоимости покупки объекта Sale эта информация передается всем подписчикам этого события путем передачи сообщения onPropertyEvent.

Обратите внимание на отображение полиморфных сообщений на диаграмме взаимодействия (см. рис. 23.23).

Экземпляр SaleFrame1, реализующий интерфейс PropertyListener, реализует в том числе и метод onPropertyEvent. При получении сообщения он отправляет сообщение объекту JTextField объекта GUI об обновлении значения общей стоимости покупки (рис. 23.24).

При реализации этого шаблона объекты модели (Sale) все же связаны с объектами представления (SaleFrame1). Однако уровень графического интерфейса отделен от уровня предметной области интерфейсом PropertyListener. Такое проектное решение не требует реальной регистрации объектов-подписчиков, т.е. список подписчиков объекта Sale может быть пустым. При этом связывание с общим интерфейсом объектов, которые могут динамически добавляться к списку подписчиков (и удаляться из него), поддерживает принципы шаблона Low Coupling. Следовательно, выполняется шаблон Protected Variations в смысле независимости от интерфейса пользователя.

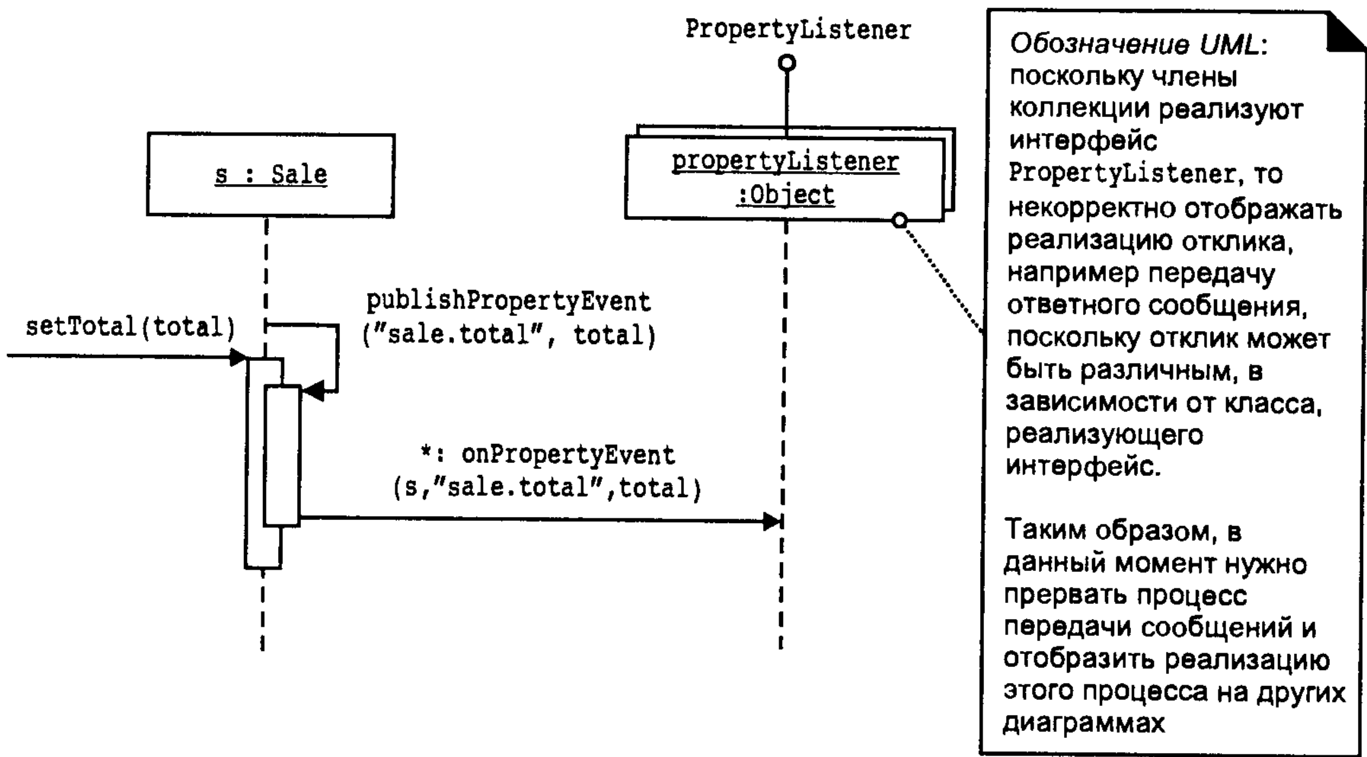



*Рис 23.23. Объект* `Sale` *распространяет информацию об изменении свойств своим подписчикам*

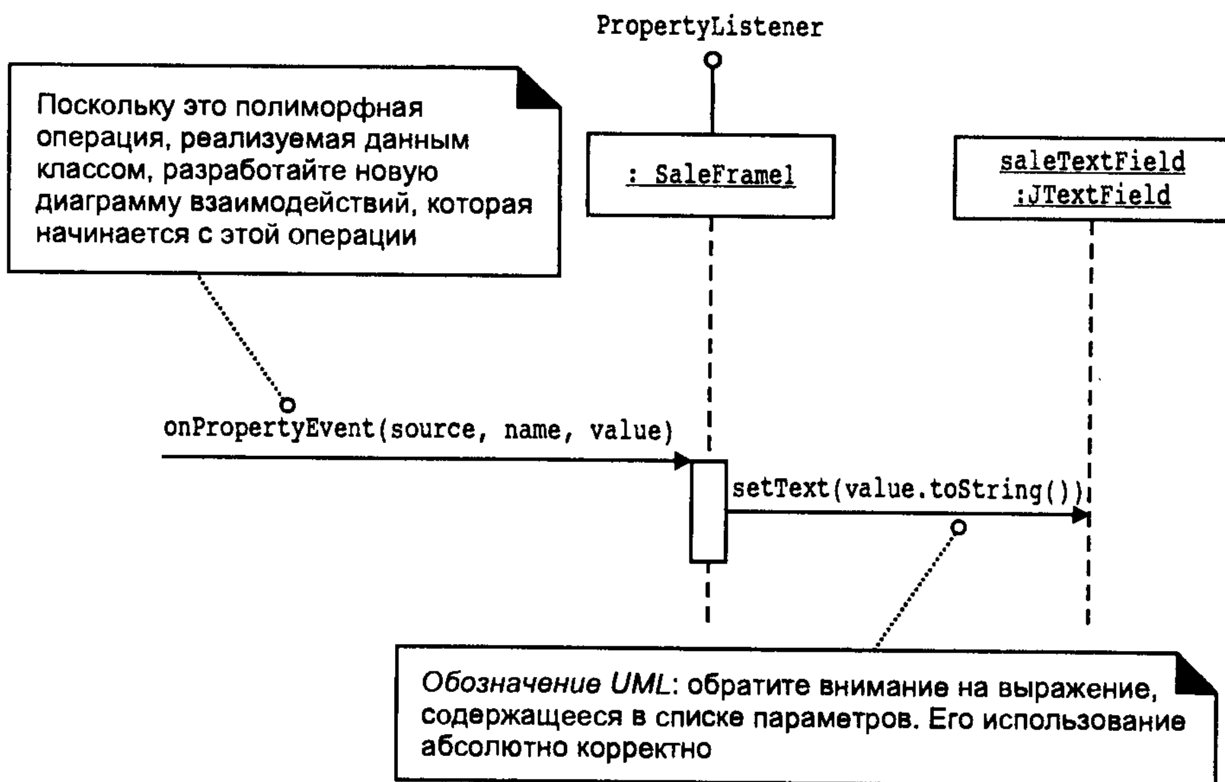



*Рис. 23.24. Подписчик* `SaleFrame1` *получает уведомление о событии*

## Почему этот шаблон называется Observer, Publish-Subscribe или Delegation Event Model

Изначально этот шаблон назывался Publish-Subscribe (Опубликовать-подписаться). Он и до сих пор широко распространен под этим именем. Один объект "публикует" информацию о событии (`Sale`). Эти сведения могут никого

не интересовать, в этом случае у объекта Sale не будет зарегистрированных подписчиков. Однако заинтересованные объекты могут "подписаться" на получение информации, тогда при реализации некоторого события они будут уведомлены. Это делается с помощью сообщения Sale--addPropertyListener.

Этот шаблон также называют Observer (Наблюдатель), поскольку слушатели или подписчики "наблюдают" событие. Этот термин был очень распространен в Smalltalk в начале 1980-х годов.

При реализации на Java этот шаблон называют также Delegation Event Model (Модель делегирования событий), поскольку обработка событий "делегируется" слушателям (или подписчикам) (рис. 23.25).

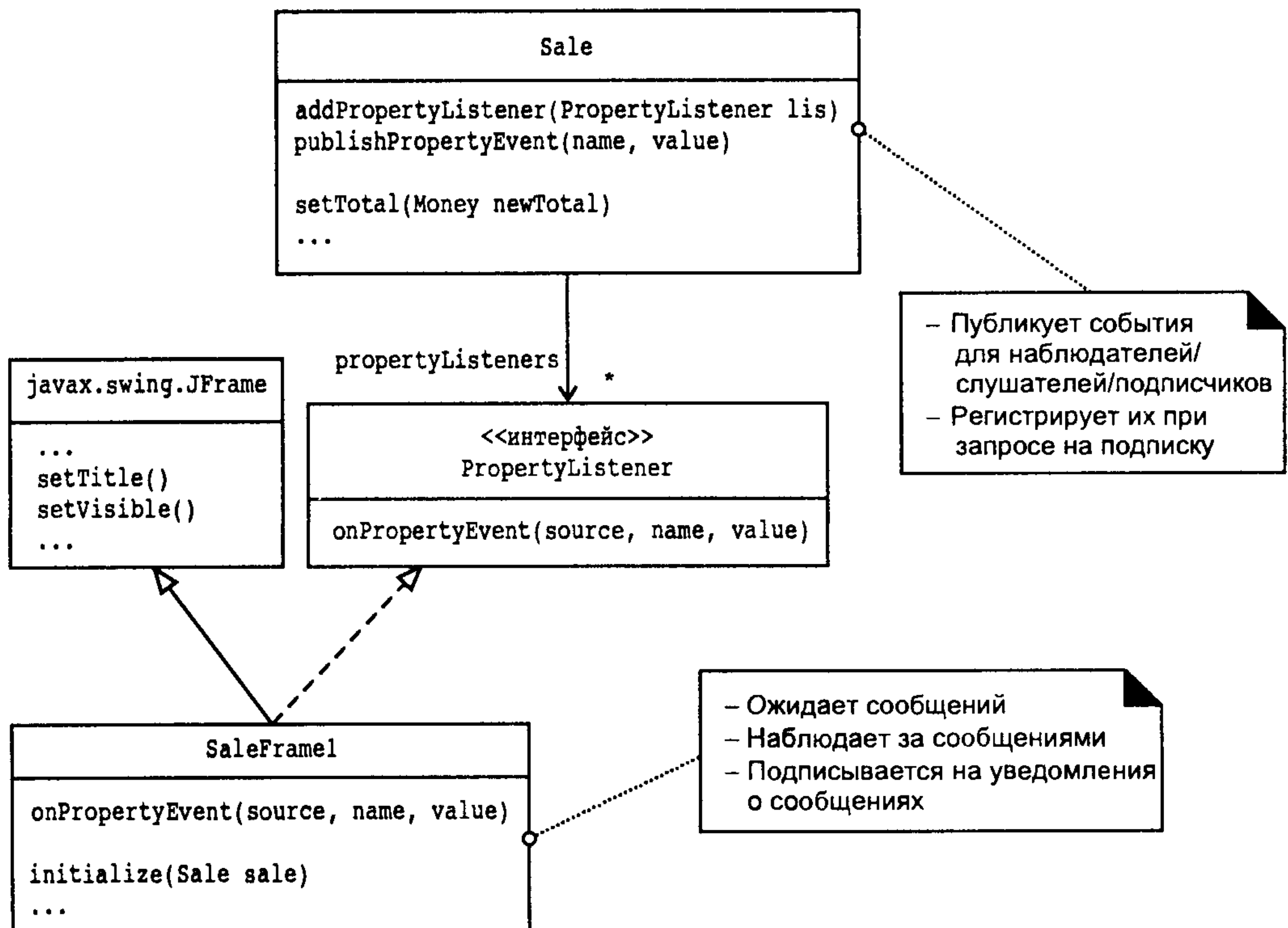



*Рис. 23.25. Кто является наблюдателем, слушателем или подписчиком?*

## Шаблон Observer не только связывает интерфейс пользователя с объектами модели

Предыдущий пример иллюстрирует применение шаблона Observer для связывания объектов GUI с объектами предметной области. Однако это не единственный пример использования шаблона.

Чаще всего этот шаблон применяется для обработки событий объектов GUI при использовании как технологий Java (AWT и Swing), так и Microsoft (.NET). Каждый элемент интерфейса публикует связанные с ним события, а другие объекты могут подписаться на получение уведомлений о них. Например, объект Swing JButton оповещает своих подписчиков о щелчке на кнопке. Объекты-

подписчики при получении такого уведомления могут выполнять некоторые характерные действия.

В качестве другого примера на рис. 23.26 показан объект будильника AlarmClock, уведомляющий о событиях своих подписчиков. Из этого примера видно, что подписчиками могут быть одновременно несколько объектов, и каждый из них может реагировать на уведомление по-своему.

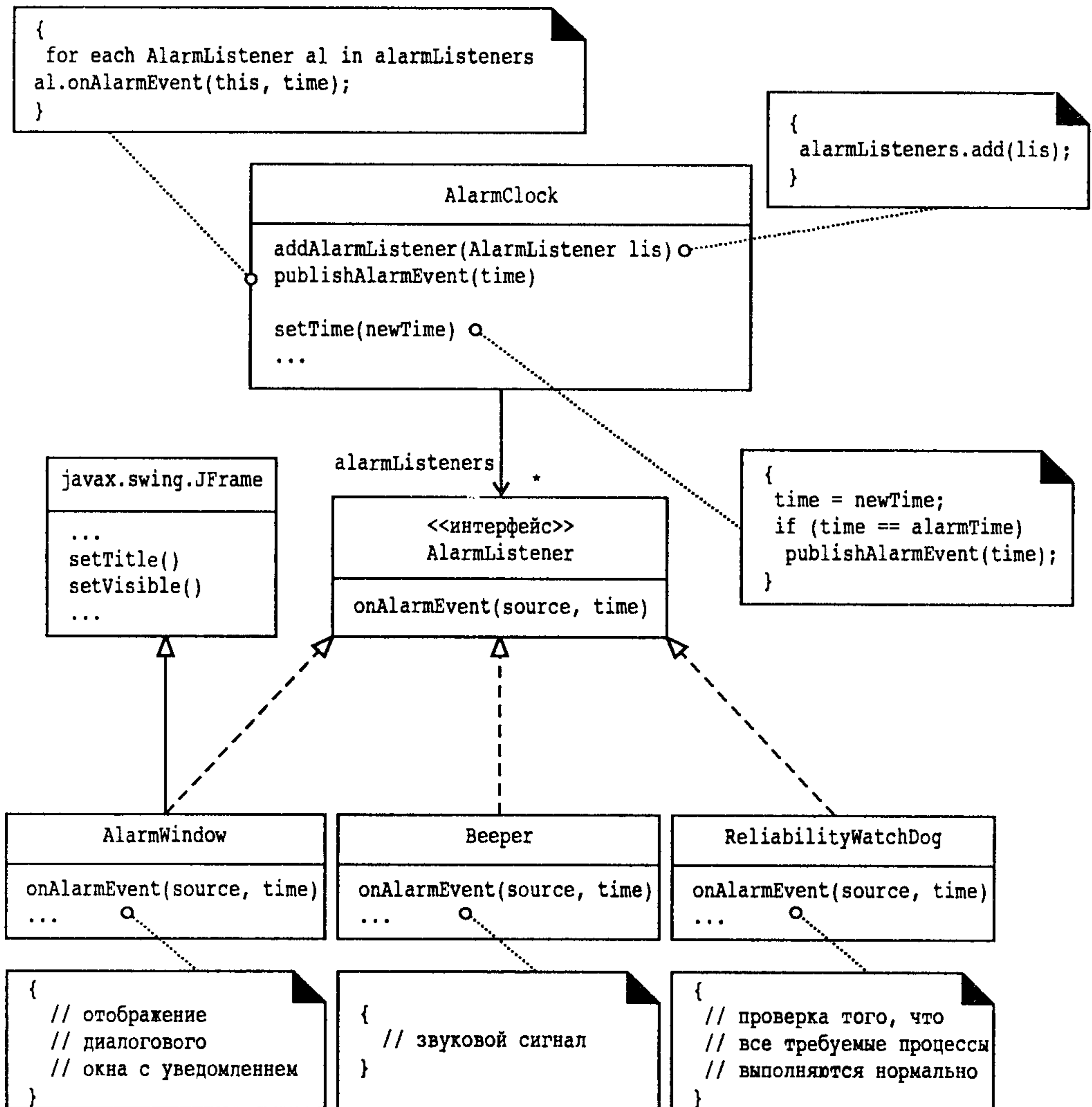



*Рис. 23.26. Шаблон Observer при наличии нескольких подписчиков*

## На одно событие могут подписаться несколько пользователей

Как видно из рис. 23.26, на одно событие могут подписаться несколько пользователей. Например, один экземпляр AlarmClock может иметь три зарегистрированных объекта AlarmWindow, четыре объекта Beeper и один ReliabilityWatchDog. При реализации события все восемь подписчиков AlarmListener будут уведомлены посредством сообщения onAlarmEvent.

## *Реализация*

### События

В обеих реализациях шаблона Observer (на Java или C# .NET) уведомление о событии выполняется с помощью обычного сообщения, например `onPropertyEvent`. Более того, в обоих случаях событие формально определяется как класс с соответствующими данными о событии. Такое событие передается в качестве параметра сообщения.

Рассмотрим следующий пример.

```
class PropertyEvent extends Event
{
    private Object sourceOfEvent;
    private String propertyName;
    private Object oldValue;
    private Object newValue;
    //...
}
//...

class Sale
{
    private void publishPropertyEvent(
        String name, Object old, Object new )
    {
        PropertyEvent evt =
            new PropertyEvent (this, "sale.total", old, new);

            for each AlarmListener al in alarmListeners
                al.onPropertyEvent( evt )
    }

    //...

}
```

### Java

В первой версии набора JDK 1.0, увидевшей свет в январе 1996 года, содержались некоторые средства реализации этого шаблона в виде класса и интерфейса `Observable` и `Observer` соответственно. Эти средства были, по существу, скопированы без всякого улучшения реализации механизма подписки Smalltalk, реализованного в начале 1980-х годов.

В конце 1996 года в версии JDK 1.1 проектное решение `Observable-Observer` было эффективно заменено более робастной версией модели делегирования сообщений DEM (Delegation Event Model) Java, однако прежнее решение было сохранено для обеспечения совместимости с более ранней версией (но в целом его не рекомендовалось использовать).

Решения, описанные в этой главе, ориентированы на модель DEM, но несколько упрощены с целью более ясного изложения идей.

### *Резюме*

Шаблон Observer обеспечивает способ снижения связывания объектов при взаимодействии. О подписчиках известен только их интерфейс. При этом подписчики могут динамически регистрироваться (или отключаться) на получение информации о событиях.

**Связанные шаблоны.** Шаблон Observer основан на принципах полиморфизма и обеспечивает реализацию шаблона Protected Variations в смысле знания конкретных типов подписчиков и их количества.

## 23.10. Заключение

Главный вывод, который можно сделать на основе вышеизложенного материала, заключается в следующем. Распределять обязанности в процессе объектно-ориентированного проектирования можно на основе применения шаблонов. Шаблоны представляют собой набор идиом, на базе которых можно разработать удачную объектно-ориентированную систему.

## 23.11. Дополнительная литература

Всем разработчикам в рамках объектного подхода можно порекомендовать книгу Гамма, Хелма, Джонсона и Влиссидеса [52]. В ней можно найти конструктивную информацию о шаблонах проектирования.

Ежегодно проходит конференция PLOP (Pattern Languages of Programs), по результатам которой публикуется сборник шаблонов в серии *Pattern Languages of Program Design* (том 1, 2 и т.д.). Рекомендую прочитать всю серию.

В [15] (том 1 и 2) продолжается обсуждение шаблонов для больших приложений. В первом томе приводятся принципы систематизации шаблонов.

На сегодняшний день описаны сотни шаблонов, большая часть из которых рассмотрена в [92].

# Часть V

# Третья итерация фазы развития

Глава

# Третья итерация и ее требования

## 24.1. Требования третьей итерации

- Обеспечить возможность использования локальных служб, если удаленные службы оказались недоступными. Например, если не удается получить доступ к удаленной базе данных товаров, то нужно воспользоваться ее локальной версией с кэшированными данными.
- Обеспечить поддержку устройств POS-системы, таких как выдвижной ящик для хранения наличных и аппарат для приема монет.
- Обеспечить авторизацию кредитных карточек.
- Обеспечить возможность использования объектов, подлежащих постоянному хранению.

## 24.2. Наиболее важные вопросы на стадии третьей итерации

В течение начальной фазы и первой итерации были исследованы фундаментальные аспекты процесса анализа требований и ООА/П. На второй итерации основное внимание уделялось вопросам проектирования объектов. На третьей итерации мы снова попытаемся "более широко" взглянуть на вещи, обратившись к анализу и проектированию в следующих областях.

- Связывание прецедентов
- Обобщение и специализация
- Моделирование состояний
- Многоуровневые архитектуры
- Проектирование пакетов
- Архитектурный анализ
- Дополнительные шаблоны проектирования GoF
- Проектирование контуров, в особенности контуров взаимодействия с базой данных

Глава 25

# Взаимосвязь прецедентов

**Основная задача**

- **Связать прецеденты с помощью ассоциаций** включает **и** расширяет.

## Введение

Прецеденты могут быть связаны друг с другом. Например, подчиненный прецедент одного прецедента, такого как Обработка платежа по кредитной карточке (Handle Credit Payment), может быть частью других прецедентов, например, Оформление продажи (Process Sale) и Обработка арендной платы (Process Rental). Определение взаимосвязей прецедентов не влияет на поведение системы или требования к ней. Скорее, такая возможность предоставляет (в идеале) механизм повышения качества взаимосвязей и ясности прецедентов, снижения вероятности дублирования текста и улучшения управляемости документами с описаниями.

## Предостережение

В некоторых организациях в процессе работы над прецедентами много времени тратится на обсуждение того, как связать прецеденты на диаграмме прецедентов, а не на непосредственное написание текста описаний. Несмотря на то, что в данной главе обсуждается взаимосвязь прецедентов, этот вопрос нужно рассматривать более широко. Вне всякого сомнения, взаимосвязь прецедентов очень важна, однако не менее важной задачей является написание и самого текста описаний. Определение требований завершается после разработки соответствующих описаний, а не после организации прецедентов, что является не более чем дополнительным шагом на пути повышения их четкости и снижения дублирования. Если в течение нескольких часов (или, еще хуже, дней) группа разработчиков моделирует прецеденты и обсуждает диаграммы прецедентов с их взаимосвязями (какую ассоциацию лучше использовать, включает или расширяет?; нужно ли прецедент еще больше специализировать?), а не сосредотачивается на написании наиболее важной части текста описания, то вполне возможно, что полученный эффект будет гораздо меньше, чем ожидалось.

Более того, определение взаимосвязей прецедентов можно итеративно выполнять на фазе развития, а не пытаться в самом начале проекта за один раз усовершенствовать и полностью определить диаграмму прецедентов.

## 25.1. Отношение включает

Такая взаимосвязь является наиболее стандартной и важной.

Вполне логично описать фрагмент, который будет встречаться в различных прецедентах. Например, описание оплаты по кредитной карточке может входить в различные прецеденты, в том числе Оформление продажи (Process Sale), Обработка арендной платы (Process Rental), Выполнение плана (Contribute to Lay-away Plan) и т.д. Вместо того чтобы дублировать этот текст, лучше поместить его в отдельный раздел описания прецедента и ссылаться на него при необходимости. Такой подход является простым механизмом управления и связывания текста, который позволяет избежать дублирования.[1]

Рассмотрим следующий пример.

**Прецедент 1:** Оформление продажи

...

**Основной успешный сценарий**

1. Покупатель подходит с выбранными товарами к терминалу POS-системы, чтобы оплатить покупку.

...

7. Покупатель оплачивает покупку, и система обрабатывает этот платеж.

...

**Расширения**

7б. Оплата по кредитной карточке: включает прецедент Обработка платежа по кредитной карточке.

7в. Оплата чеком: включает прецедент Обработка платежа чеком.

...

**Прецедент 7:** Обработка арендной платы

...

**Расширения**

6б. Оплата по кредитной карточке: включает прецедент Обработка платежа по кредитной карточке.

...

**Прецедент 12:** Обработка платежа по кредитной карточке

...

**Уровень:** сценарий прецедента

**Основной успешный сценарий**

1. Покупатель вводит данные о своей кредитной карточке.
2. Система отсылает запрос на авторизацию платежа внешней системе авторизации платежей, чтобы получить подтверждение о возможности оплаты.
3. Система получает подтверждение о возможности оплаты и сообщает об этом кассиру.
4. ...

**Расширения**

2а. При взаимодействии с внешней системой POS-система получает отказ.
   1. Система сообщает о возникновении ошибки кассиру.
   2. Кассир предлагает покупателю другой вид оплаты.

...

В приведенных выше примерах прецеденты связаны отношением включает.

[1] Очень хорошо, если такие связи реализуются с использованием гиперссылок.

Более сжатая (и, следовательно, более предпочтительная) форма включения прецедента заключается в подчеркивании его имени или его выделении каким-либо другим способом, как показано ниже.

...
**Расширения**
7б. Оплата по кредитной карточке: Обработка платежа по кредитной карточке.
7в. Оплата чеком: Обработка платежа чеком.
...

Обратите внимание, что подчиненный прецедент Обработка платежа по кредитной карточке ранее содержался в разделе "Расширения прецедента Оформление продажи", однако был вынесен в отдельный раздел, чтобы избежать дублирования. Кроме того, при описании прецедента разделы "Основной успешный сценарий" и "Расширения" используются так же, как и в прецеденте Оформление продажи.

Практические рекомендации по использованию отношения включает (include) были предложены Фовлером [49]. Приведем цитату.

> "Используйте взаимосвязь включает в том случае, если вам приходится повторяться при описании двух или более прецедентов, а нужно избежать дублирования".

Кроме того, взаимосвязь включает позволяет без проблем осуществить декомпозицию чрезмерно длинных прецедентов на несколько более коротких. Это значительно повысит их четкость.

### *Отношение включает и обработка асинхронных событий*

Отношение включает можно использовать также для описания обработки асинхронного события. Такое событие может наступить, например, в том случае, когда пользователь в произвольный момент времени активизировал какой-либо режим, перешел в определенное диалоговое окно, вызвал какую-нибудь функцию, перешел на Web-страницу или выполнил определенную последовательность действий.

Представление такого асинхронного события в описании прецедента было рассмотрено ранее в главе 6, "Описание требований в контексте модели прецедентов". Однако там рассматривались лишь ключевые вопросы, связанные с обработкой асинхронных сообщений, и не затрагивалась взаимосвязь с подчиненными прецедентами.

Стандартным обозначением асинхронных событий являются метки а*, б*, располагаемые в разделе "Расширения". Вспомните, что такие метки свидетельствуют о расширении или возможности наступления события в любой момент времени. Можно также указать диапазон меток, такой как 3–9, задающий определенные шаги (но не все) в описании прецедента, когда может произойти асинхронное событие.

**Прецедент 1:** Обработка служебной информации

...
**Основной успешный сценарий**
1. ...
**Расширения**
а*. В любой момент покупатель может отредактировать информацию о себе:
Редактирование личной информации.

6*. В любой момент покупатель может перейти к печати справочной информации: Печать текущей справочной информации.

2-11. Покупатель отменил предыдущее действие: Отмена обработки транзакции.

...

### Заключительные замечания

Взаимосвязь включает можно применять для разрешения большинства проблем, возникающих при организации прецедентов. В качестве заключительных рекомендаций можно сформулировать следующее.

Сценарии прецедента следует выделять в отдельные прецеденты, связанные отношением включает, при выполнении следующих условий.

- Эти сценарии дублируются в других прецедентах.
- Прецедент является очень сложным и длинным, поэтому выделение сценариев в отдельный прецедент позволит значительно его упростить.

Как вы узнаете чуть позже, существуют и другие отношения: расширяет (extend) и обобщения (generalization). Однако Кокбурн (Cockburn), эксперт в области моделирования прецедентов, предпочитает использовать именно взаимосвязь включает. Приведем цитату.

> "В качестве первого и основного правила руководствуйтесь следующим: между прецедентами всегда используйте взаимосвязь включает. Те, кто следуют этому правилу, сообщают о том, что у них самих и тех, кто читает их описания, возникает гораздо меньше проблем, чем у тех, кто наряду с отношением включает применяет также отношения расширяет и обобщает [33]".

## 25.2. Новые термины: конкретный, абстрактный, основной и дополнительный прецеденты

**Конкретный прецедент** (concrete use case) инициируется исполнителем и описывает общее поведение системы [97]. В таких прецедентах описываются элементарные бизнес-процессы. Например, прецедент Оформление продажи является конкретным. В отличие от этого, **абстрактный прецедент** (abstract use case) никогда не инстанцируется, а является сценарием, т.е. частью другого прецедента. Прецедент Обработка платежа по кредитной карточке является абстрактным. Он сам никогда не инициируется, однако всегда входит в состав другого прецедента, такого как Оформление продажи.

Прецедент, включающий другой прецедент, или расширяемый/обобщаемый другим прецедентом, называется **основным** (base use case). Прецедент Оформление продажи является основным, поскольку он включает прецедент Обработка платежа по кредитной карточке. Однако прецедент, являющийся включением, расширением или специализацией, называется **дополнительным** (addition use case). Прецедент Обработка платежа по кредитной карточке является дополнительным, поскольку с помощью отношения включает связан с прецедентом Оформление продажи. Обычно дополнительные прецеденты являются абстрактными, а основные — конкретными.

## 25.3. Отношение расширяет

Предположим, что по некоторым причинам текст прецедента не должен изменяться (как минимум, существенно). Это может понадобиться в том случае, когда внесение огромного количества изменений сопряжено с множеством проблем, или прецедент рассматривается как уже установившийся артефакт. Как же добавить к этому прецеденту новый фрагмент, не изменяя исходного текста?

Ответ на этот вопрос предоставляет взаимосвязь расширяет (extend). Основная идея заключается в создании расширяющего или дополнительного прецедента, в котором описывается, где и при каких условиях он расширяет некоторый основной прецедент. Рассмотрим пример.

**Прецедент 1:** Оформление продажи **(основной прецедент)**

...

**Точки расширения:** *особый покупатель*, шаг 1. *Платеж*, шаг 7.

**Основной успешный сценарий**

1. Покупатель подходит с выбранными товарами к терминалу POS-системы, чтобы оплатить покупку.

...

7. Покупатель оплачивает покупку, и система обрабатывает этот платеж.

...

**Прецедент 15:** Обработка платежа с помощью призового сертификата **(расширяющий прецедент)**

...

**Переключатель:** покупатель хочет оплатить покупку с использованием призового сертификата.

**Точки расширения:** платеж при оформлении продажи.

**Уровень:** подчиненный прецедент

**Основной успешный сценарий**

1. Покупатель передает кассиру призовой сертификат.
2. Кассир вводит идентификатор призового сертификата.

...

В этих описаниях содержится пример использования отношения расширяет. Обратите внимание на использование раздела "Точки расширения", а также на то, что расширяющий прецедент инициируется при выполнении некоторого условия. Точки расширения представляют собой метки в описании основного прецедента, в которых содержится ссылка на расширяющий прецедент. Поэтому нумерацию шагов основного прецедента можно без проблем модифицировать, никак при этом не влияя на расширяющий прецедент.

Иногда точка расширения указывается в следующей простой форме: *в любой момент прецедента X*. Это особенно характерно для систем с множеством асинхронных событий, таких как текстовый процессор ("сейчас будет выполнена проверка орфографии", "сейчас будет выполнен поиск в тезаурусе") или системы управления ядерным реактором. Однако обратите внимание на то, что для описания процесса обработки асинхронного события можно использовать также и отношение включает. Взаимосвязь расширяет предоставляет дополнительную возможность в том случае, когда основной прецедент нельзя модифицировать.

Следует отметить, что отличительной особенностью взаимосвязи расширяет является то, что в основном прецеденте (Оформление продажи) не содержится ссылки на расширяющий прецедент (Обработка платежа с использованием

призового сертификата). Следовательно, этот прецедент не определяет и не управляет условиями, при которых инициируется расширяющий прецедент. Прецедент Оформление продажи может так и завершиться, ничего не узнав о расширяющем прецеденте.

Обратите внимание, что на дополнительный прецедент Обработка платежа с использованием призового сертификата можно ссылаться из прецедента Оформление продажи и с помощью отношения включает. Зачастую это оказывается достаточно полезно. Однако рассмотренный пример обусловлен тем, что прецедент Оформление продажи нельзя модифицировать. Именно в этом случае удобнее воспользоваться отношением расширяет, а не включает.

Кроме того, сценарий обработки платежа по призовому сертификату можно также добавить в качестве расширения в раздел "Расширения" прецедента Оформление продажи. Этот подход позволяет избежать необходимости использования обеих взаимосвязей, а также создания отдельного подчиненного прецедента.

> Простое обновление раздела "Расширения" обычно оказывается гораздо лучшим решением, чем создание сложных отношений между прецедентами.

В некоторых руководствах по написанию прецедентов рекомендуется применять расширяющие прецеденты и отношение расширяет для моделирования в рамках основного прецедента условных или необязательных процессов. Однако при этом ничего не говорится о том, что описание таких процессов можно в виде текста просто поместить в раздел "Расширения" основного прецедента.

Использование взаимосвязи расширяет на практике обусловлено невозможностью модификации основного прецедента по каким-либо причинам.

## 25.4. Отношение обобщает

Обсуждение отношения обобщает (generalize) выходит за рамки этой главы. Однако следует упомянуть, что эксперты в области моделирования прецедентов прекрасно обходятся без этого дополнительного отношения, повышающего уровень сложности прецедентов. Кроме того, в настоящий момент не существует хорошего практического руководства по его использованию. Большинство специалистов единодушны в своем мнении: добавление большого количества взаимосвязей прецедентов приводит к непродуктивному использованию времени и проблемам в достижении требуемого результата.

## 25.5. Диаграммы прецедентов

На рис. 25.1 показан пример обозначений языка UML, используемых для отношения включает. На этом рисунке показано лишь одно отношение, что, по мнению экспертов, позволяет обойтись без лишних сложностей.

POS-система NextGen

Кассир

Покупатель

Оформление продажи

<<включает>>

<<включает>>

<<включает>>

Оформление оплаты наличными

<<исполнитель>>
Служба авторизации кредитных карточек

<<включает>>

Оформление оплаты чеком

Оформление оплаты по кредитной карточке

<<включает>>

<<включает>>

<<исполнитель>>
Бухгалтерская система

Оформление кредита

Оформление возврата товара

Управление пользователями

...

*Обозначения UML*:

основной прецедент указывает на включаемый прецедент

*Рис. 25.1. Использование отношения* включает *на диаграмме прецедентов*

Обозначения, используемые для взаимосвязи расширяет, представлены на рис. 25.2.

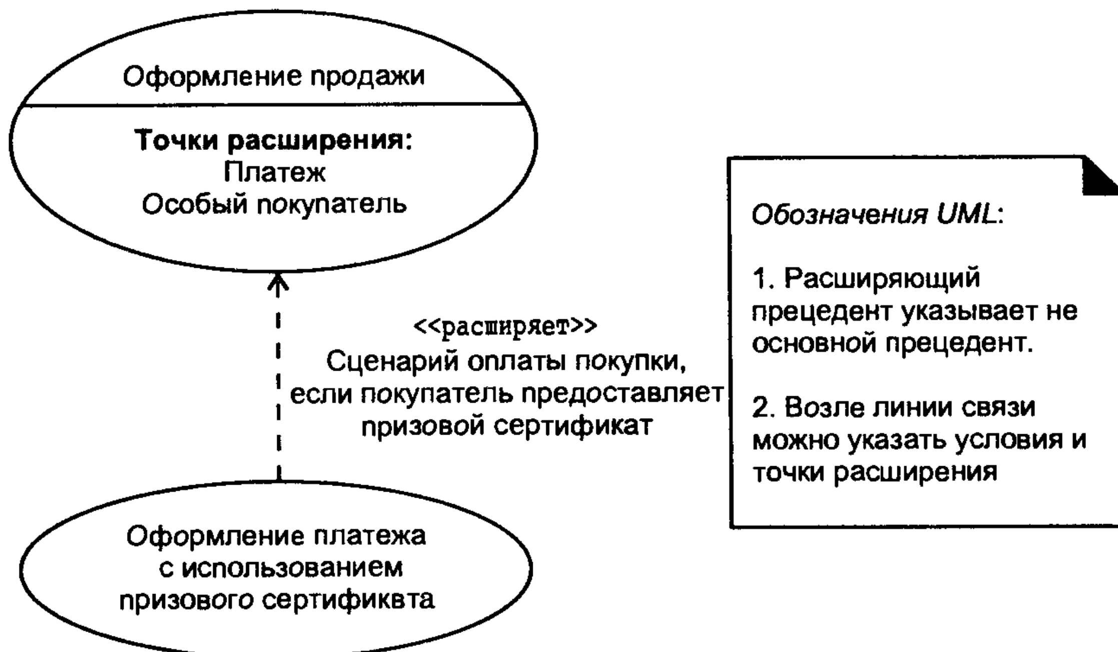



*Рис. 25.2. Взаимосвязь* расширяет

Глава 26

# Обобщение модели

*Непродуманная классификация и некорректное обобщение — это проклятие организованной жизни.*

*Обобщение Х. Дж. Велса (H.G. Wells)*

**Основные задачи**

- **Построить иерархии "обобщение – специализация".**
- **Определить условия для отображения подклассов.**
- **Применить тесты "100%" и "Is-a" для проверки корректности выбора подклассов.**

## Введение

Обобщение и специализация — это базовые концепции процесса моделирования предметной области, позволяющие сэкономить усилия при разработке такой модели. Более того, иерархия концептуальных классов зачастую становится основой иерархии наследования программных классов, позволяющей снизить уровень дублирования кода.

## 26.1. Модель предметной области: новые понятия

Как и на первой итерации, модель предметной области можно последовательно дорабатывать с учетом требований для данной итерации. Понятия можно идентифицировать на основе списка категорий, применяя способ выделения существительных из описания прецедентов. Для разработки робастной и разносторонней модели предметной области очень полезно ознакомиться с работами других авторов по этому вопросу, например [46]. Дело в том, что за пределами рассматриваемых тем этой книги осталось множество важных особенностей процесса моделирования.

### *Список категорий понятий*

В табл. 26.1 приводятся некоторые важные понятия, рассматриваемые на данной итерации.

**Таблица 26.1. Список категорий понятий**

| Категория | Примеры |
|---|---|
| Физические или материальные объекты | CreditCard (Кредитная карточка), Check (Чек) |
| Спецификации, элементы дизайна или описания объектов | |
| Места | |
| Транзакции | CashPayment (Оплата наличными), CreditPayment (Оплата по кредитной карточке), CheckPayment (Оплата чеком) |
| Элементы транзакций | |
| Роли людей | |
| Контейнеры других объектов | |
| Содержимое контейнеров | |
| Другие компьютеры или электромеханические системы, внешние по отношению к данной системе | CreditAuthorizationService (Служба авторизации кредитных карточек) CheckAuthorizationService (Служба авторизации чеков) |
| Абстрактные понятия | |
| Организации | CreditAuthorizationService (Служба авторизации кредитных карточек) CheckAuthorizationService (Служба авторизации чеков) |
| События | |
| Правила и политики | |
| Каталоги | |
| Записи финансовой, трудовой, юридической и другой деятельности | AccountsReceivable (Система оплаты кредитов) |
| Финансовые инструменты и службы | |
| Руководства, книги | |

## *Определение понятий из текстовых описаний*

Как уже отмечалось, процесс выделения понятий, включаемых в концептуальную модель, нельзя назвать механическим. При выборе понятий необходимо использовать определенные абстракции, поскольку слова естественного языка могут иметь по несколько значений и не всегда точно отражают смысл понятий. Однако выделение существительных из текстовых описаний прецедентов — важный практический прием, применяемый при построении модели предметной области.

На данной итерации реализуются сценарии прецедента Оформление продажи, предусматривающие оплату чеком и по кредитной карточке. Поэтому существительные будем выбирать из следующих расширений.

**Прецедент:** Оформление продажи

...

**Расширения**

7б. Оплата по кредитной карточке.

1. Покупатель вводит **информацию о своей кредитной карточке**.
2. Система отправляет **запрос на авторизацию платежа** внешней системе **службы авторизации платежей** и запрашивает **подтверждение платежа**.

2а. Система определяет сбой при взаимодействии с внешней системой.
  1. Система сигнализирует об ошибке кассиру.
  2. Кассир просит покупателя изменить тип платежа.
3. Система получает информацию о **подтверждении платежа** и сообщает об этом кассиру.
  3а. Система получает информацию об **отказе в выполнении платежа**.
    1. Система сообщает об отказе кассиру.
    2. Кассир просит покупателя изменить тип платежа.
4. Система регистрирует **платеж по кредитной карточке**, включая информацию о подтверждении платежа.
5. Система предоставляет механизм ввода подписи для платежа по кредитной карточке.
6. Кассир просит покупателя предоставить подпись на оплату по кредитной карточке. Покупатель вводит подпись.

7в. Оплата чеком.
1. Покупатель выписывает **чек** и передает его и свое **удостоверение личности** кассиру.
2. Кассир записывает **номер удостоверения личности** на чеке, вводит его и отправляет запрос на **авторизацию оплаты чеком**.
3. Система генерирует **запрос на оплату чеком** и отправляет его внешней **службе авторизации чеков**.
4. Система получает подтверждение на оплату чеком и сообщает об этом кассиру.
5. Система регистрирует **оплату чеком** и выдает сообщение об успешной авторизации.

...

### *Транзакции со службами авторизации*

В процессе выделения понятий из описаний прецедентов были идентифицированы следующие понятия: `CreditPaymentRequest` (Запрос на оплату по кредитной карточке) и `CreditApprovalReply` (Подтверждение на оплату по кредитной карточке). Их можно рассматривать как некий вид транзакций с внешними службами. Вообще, такие транзакции полезно идентифицировать, поскольку связанные с ними процессы и виды деятельности могут изменяться по ходу проектирования системы.

Эти транзакции не должны представлять собой записи или последовательности битов. Они лишь представляют абстрактное понятие транзакции, не зависящее от способа ее реализации. Например, запрос на авторизацию платежа по кредитной карточке может выполняться в процессе телефонных переговоров между людьми или в процессе модемного соединения компьютеров.

## 26.2. Что такое обобщение

Понятия `CashPayment`, `CreditPayment` и `CheckPayment` очень похожи. В этом случае можно (и весьма полезно[1]) организовать их в *иерархию обобщения-специализации классов* (generalization-specialization class hierarchy) (рис. 26.1). В этой иерархии *суперкласс* (superclass) `Payment` представляет более общее понятие, а его *подклассы* (subclasses) — более специализированное.

Заметим, что термин "класс" в этой главе относится к концептуальным, а не программным классам.

*Обобщение* (generalization) — это процесс, связанный с идентификацией общности между понятиями и определением суперкласса (основного понятия) и связанных с ним подклассов (специализированных понятий). Обобщение предоставляет метод систематизированной классификации понятий и их представления в виде иерархии классов.

---

[1] Далее в этой главе вы узнаете, для чего необходимо определять иерархии классов.

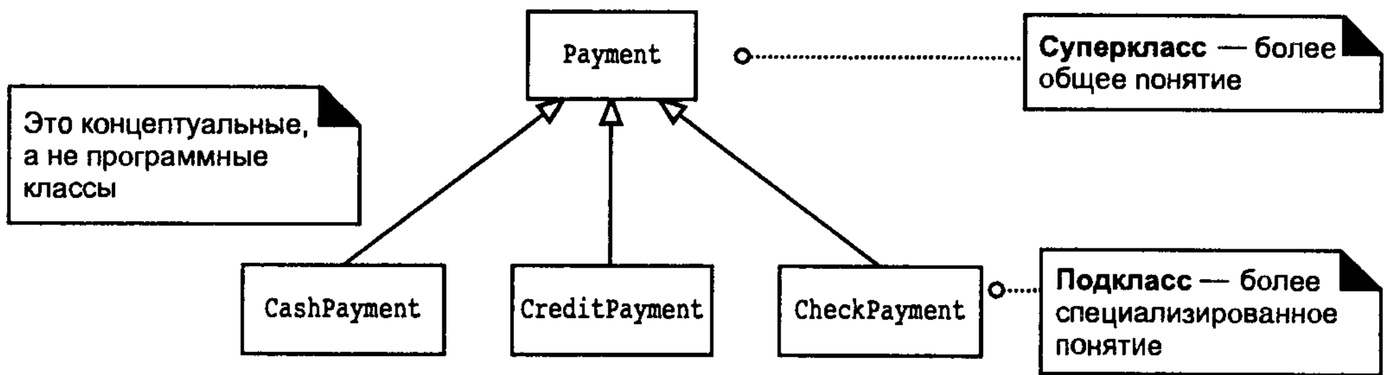



*Рис. 26.1. Иерархия "обобщения–специализации"*

Идентификация супер- и подклассов осуществляется с использованием модели предметной области, поскольку она позволяет проанализировать понятия в более обобщенных, переопределенных и абстрактных терминах. В результате выражения становятся более емкими, улучшается понимание и уменьшается объем повторяемой информации. И хотя в данный момент наше внимание сосредоточено на модели предметной области, а не на модели проектирования, последующая реализация суперклассов и подклассов в форме дерева наследования классов позволит значительно улучшить качество программного кода.

Таким образом, нужно придерживаться следующих рекомендаций.

| Идентифицируйте суперклассы и подклассы предметной области, актуальные на данном этапе исследования, и иллюстрируйте их в модели предметной области. |
|---|

**Обозначения языка UML.** В языке UML обобщающая взаимосвязь между элементами отображается в виде большой треугольной стрелки, направленной от более специализированного элемента к более общему. Для каждого такого элемента можно либо использовать отдельную стрелку, либо указать общую стрелку для всех специализированных элементов (рис. 26.2).

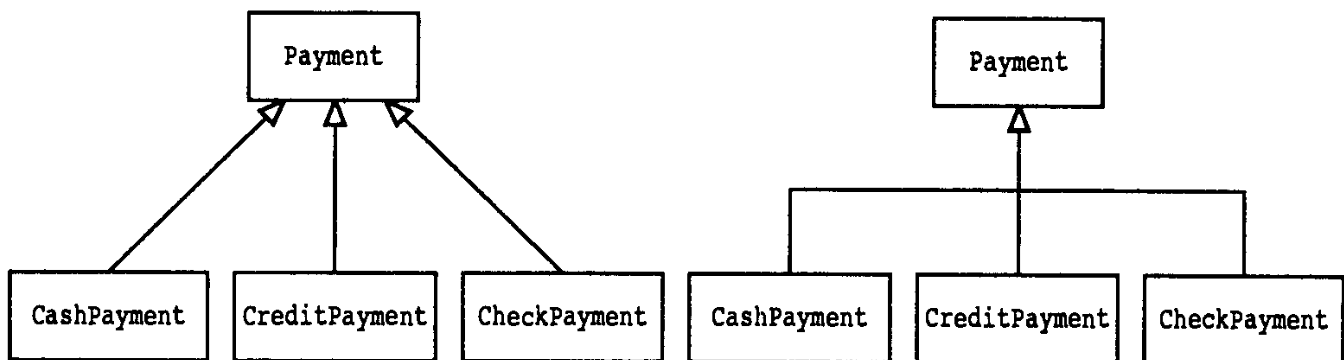



*Рис. 26.2. Иерархия классов с отдельными стрелками для каждого подкласса и общей стрелкой для всех подклассов*

## 26.3. Определение концептуальных суперклассов и подклассов

Поскольку очень важно идентифицировать концептуальные супер- и подклассы, нужно четко понимать, что такое "обобщение", "суперклассы" и "подклассы" в

терминах определения классов и их множеств[2]. Все эти вопросы рассматриваются в последующих разделах.

### Обобщение и определение классов

Что же собой представляет связь суперкласса с подклассом?

| Определение суперкласса является более общим, чем определение подкласса. |
|---|

Например, рассмотрим суперкласс Payment и его подклассы (CashPayment CreditPayment и т.д.). Предположим, что тип Payment определяет транзакцию, связанную с передачей денег (необязательно наличными) для приобретения товаров, и что со всеми платежами связано количество передаваемых денег. Модель, соответствующая этому описанию, представлена на рис. 26.3.

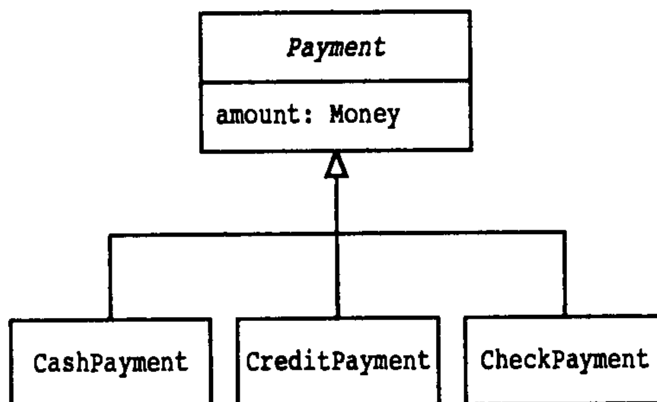



*Рис. 26.3. Иерархия класса* Payment

Класс CreditPayment определяет передачу денег с использованием кредитной карточки, которая должна быть авторизована. Определение класса Payment является более общим, чем определение класса CreditPayment.

### Обобщение и множества классов

Подклассы и суперклассы связаны в терминах элементов множества.

| Все элементы множества подклассов являются элементами множества их суперкласса. |
|---|

Например, в терминах теории множеств все элементы множества CreditPayment являются также элементами множества Payment. На диаграмме Венна (Venn) это можно представить так, как показано на рис. 26.4.

### Совместимость определений подклассов

После создания иерархии классов определитесь с суперклассами, с которыми будут связаны созданные подклассы. Например, на рис. 26.5 показано, что у всех подклассов Payment имеется атрибут amount и все они связаны с объектом Sale.

---

[2] Расширение и сужение классов. Эти вопросы описаны в [81].

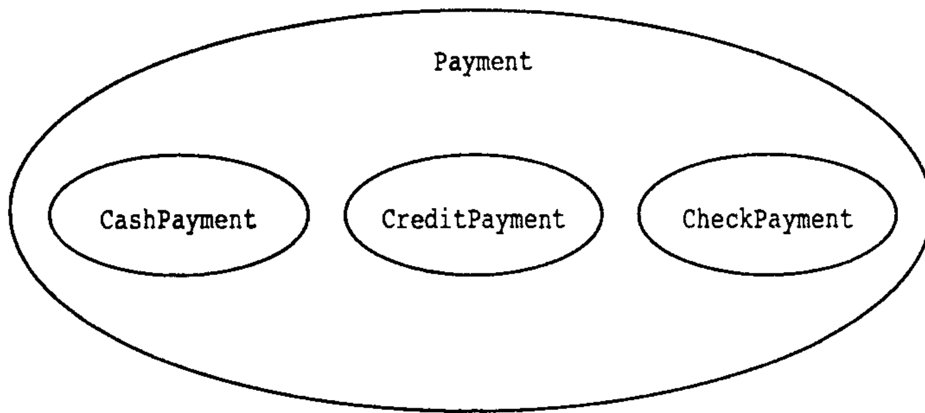



*Рис. 26.4. Диаграмма Венна (отношения множеств)*

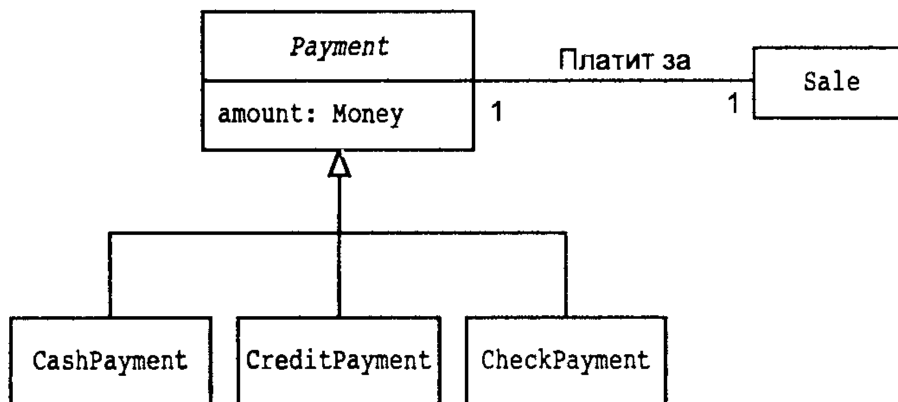



*Рис. 26.5. Совместимость подклассов*

Все подклассы должны содержать атрибут amount и быть связаны с объектом Sale. Это правило совместимости с определением суперкласса называется *правилом 100%* (100% Rule).

*Правило 100%*

100% определения суперкласса должно быть применено к подклассу. Подкласс должен на 100% соответствовать своему суперклассу по следующим параметрам.

- Атрибуты
- Ассоциации

## Совместимость множества подкласса

Концептуальный подкласс должен быть элементом множества суперкласса. Так, подкласс CreditPayment должен быть элементом множества Payment.

Неформально это фраза означает, что подкласс является *разновидностью* (kind) суперкласса. Подкласс CreditPayment является *разновидностью* класса Payment. Более кратко, с учетом английского языка, связь "является" (*is-a-kind-of*) названа *is-a* (это).

Этот тип совместимости называется *правилом Is-a* (Is-a Rule).

*Правило Is-a*

Все элементы множества подкласса должны быть элементами множества его суперкласса.

В обычном языке это правило можно проверить с помощью следующего выражения.

*Подкласс* это (is-a) *Суперкласс*

Например, фраза *Класс* CreditPayment *это класс* Payment имеет смысл и выражает совместимость элементов множеств.

### Что такое корректный подкласс

Как следует из предыдущих разделов, при построении модели предметной области для определения корректного подкласса применяйте следующие тесты[3].

Потенциальный подкласс должен удовлетворять следующим правилам.

- Правило 100% (совместимость определения)
- Правило Is-a (совместимость элементов множеств)

## 26.4. Когда нужно определять концептуальный подкласс

Выше были рассмотрены правила, с помощью которых можно проверить, что подкласс является корректным (правила 100% и Is-a). Однако *когда* нужно приступать к определению подкласса? Введем сначала следующее определение. **Разделение концептуального класса** (conceptual class partition) — это деление класса на непересекающиеся подклассы или, в терминологии Оделла (Odell), типы [81].

Это определение можно сформулировать также в виде следующего вопроса: "Когда полезно отображать разделение класса?".

Например, в предметной области POS-системы понятие Customer можно корректно разделить на подклассы MaleCustomer и FemaleCustomer. Однако следует ли иллюстрировать это разделение в разрабатываемой модели (рис. 26.6)?

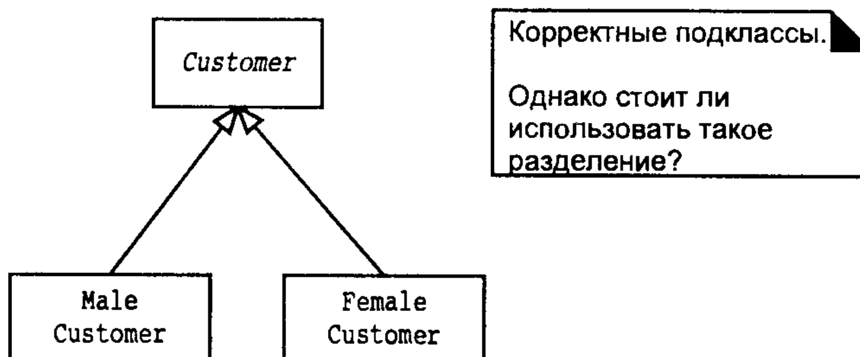



*Рис. 26.6. Допустимое разделение концептуального класса. Однако насколько оно полезно в рассматриваемой предметной области?*

[3] Такие имена правил были выбраны, скорее, из-за их мнемоники, а не для обеспечения точности.

Такое разделение является абсолютно бесполезным. Почему это именно так, вы узнаете из следующего раздела.

### В каких случаях класс нужно разделять на подклассы

Вот случаи, когда класс нужно разделять на подклассы.

Создавайте подкласс суперкласса в следующих случаях.

1. Подкласс имеет дополнительные атрибуты, интересующие разработчика.
2. Подкласс имеет дополнительные ассоциации, интересующие разработчика.
3. Подклассу соответствует понятие, управляемое, обрабатываемое, реагирующее или используемое способом, отличным от способа, определенного суперклассом или другими подклассами.
4. Подкласс представляет понятие "живой" сущности (например, животное, робот), поведение которой отлично от поведения, определяемого суперклассом или другими подклассами.

С учетом приведенных выше критериев, можно прийти к выводу, что класс `Customer` разделять на подклассы `MaleCustomer` и `FemaleCustomer` не стоит, поскольку у них отсутствуют дополнительные атрибуты или ассоциации, они не обрабатываются различными способами и их поведение в контексте рассматриваемой системы ничем не отличается[4].

В табл. 26.2 приводится несколько примеров разделения классов из предметной области платежей и других областей на основе приведенных выше критериев.

**Таблица 26.2. Примеры разделения суперкласса**

| Критерий разделения на подклассы | Примеры |
|---|---|
| Подкласс имеет дополнительные атрибуты, интересующие разработчика | К предметной области платежей неприменим.<br>Библиотека — класс `Book` (Книга), являясь подклассом класса `LoanableResource` (Предоставляемый ресурс), имеет атрибут `ISBN` (Код) |
| Подкласс имеет дополнительные ассоциации, интересующие разработчика | Платежи — класс `CreditPayment`, являясь подклассом класса `Payment`, ассоциирован с объектом `CreditCard`.<br>Библиотека — класс `Video`, являясь подклассом класса `LoanableResource`, ассоциирован с объектом `Director` |
| Подклассу соответствует понятие, управляемое, обрабатываемое, реагирующее или используемое способом, отличным от способа, определенного супертипом или другими подтипами | Платежи — класс `CreditCard`, являясь подклассом класса `Payment`, обрабатывается иначе, чем другие формы платежей в рамках собственного алгоритма авторизации.<br>Библиотека — класс `Software`, являясь подклассом класса `LoanableResource`, перед получением требует внесения задатка |
| Подкласс представляет понятие "живой" сущности (например, животное, робот), поведение которой отлично от поведения, определяемого суперклассом или другими подклассами | К предметной области платежей неприменим.<br>К предметной области библиотеки неприменим.<br>Исследование рынка — класс `MaleHuman` (Мужчина) является подклассом класса `Human` (Человек). В магазине во время приобретения покупок его поведение отлично от поведения класса `FemaleHuman` (Женщина) |

[4] Мужчины и женщины ведут себя в магазине несколько по-разному. Однако эти отличия в ракурсе требований для текущего прецедента являются несущественными, а именно этим критерием определяются рамки исследований.

## 26.5. Когда нужно определять концептуальный суперкласс

Обобщение в рамках суперкласса обычно осуществляется в том случае, если выявлена общность между потенциальными подклассами. Для обобщения и создания суперкласса применяйте следующие рекомендации.

> Создавайте суперкласс в следующих случаях.
>
> - Потенциальные подклассы представляют собой вариации похожего понятия.
> - Подклассы будут удовлетворять правилам 100% и Is-a.
> - Все подклассы имеют один и тот же атрибут, который можно отнести к суперклассу.
> - Все подклассы имеют одну и ту же ассоциацию, которую можно выделить и связать с суперклассом.

Эти критерии рассматриваются в последующих разделах.

## 26.6. Иерархия классов POS-системы NextGen

### *Классы `Payment`*

С помощью рассмотренных выше критериев полезно разделить класс `Payment`, создав иерархию классов, представляющих различные виды платежей. Обоснование создания такого суперкласса и подклассов представлено на рис. 26.7.

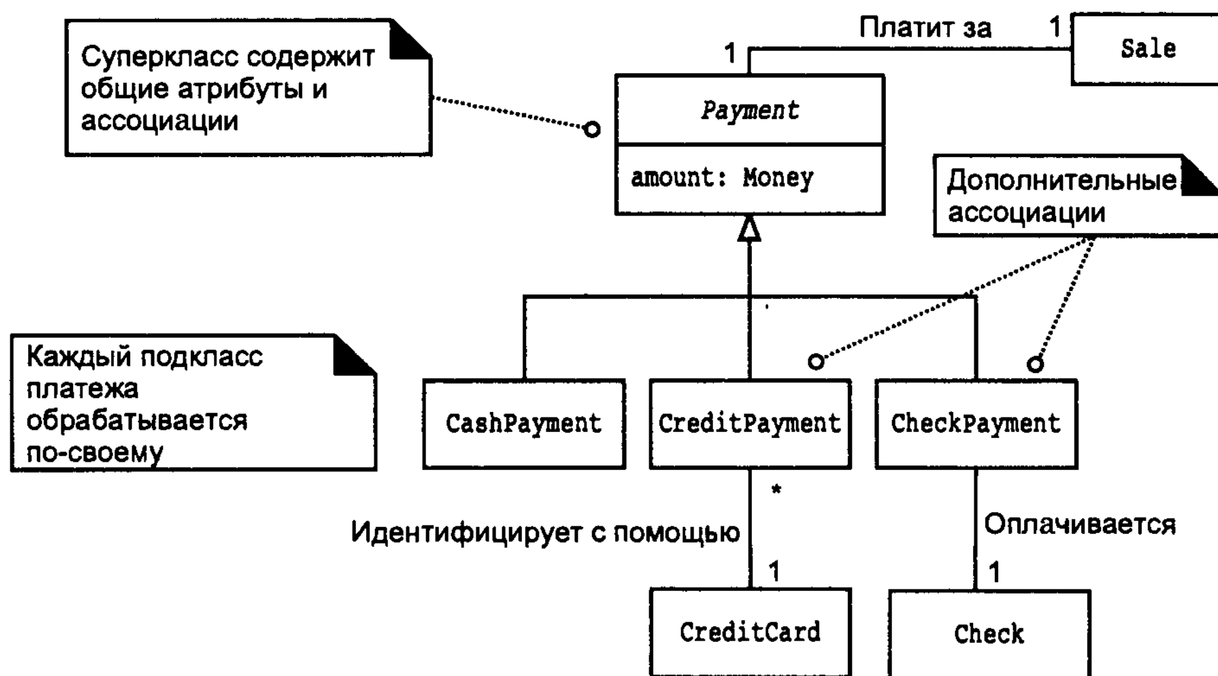



*Рис. 26.7. Обоснование выбора подклассов* `Payment`

### *Классы служб авторизации*

Службы авторизации кредитных карточек и чеков являются вариантами одного и того же понятия и имеют общие атрибуты. Соответствующие им классы можно представить в виде иерархии, показанной на рис. 26.8.

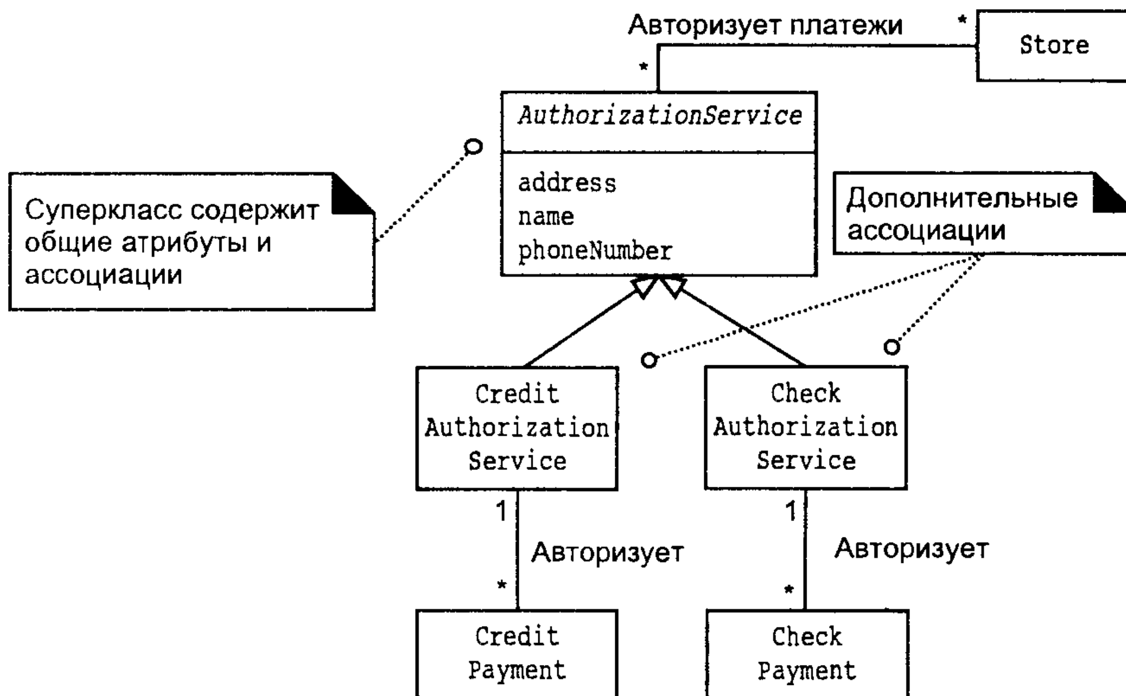



*Рис. 26.8. Обоснование иерархии классов* `AuthorizationService`

## Классы транзакций авторизации

Моделирование различных видов транзакций служб авторизации (запросов и откликов) составляет очень интересную задачу. Вообще говоря, транзакции с внешними службами весьма полезно отображать в модели предметной области, поскольку вокруг них сконцентрированы все виды деятельности и процессы. Такие транзакции очень важны.

Нужно ли иллюстрировать *каждый* вид транзакции внешней службы? Ответ на этот вопрос зависит от конкретной ситуации. Как уже упоминалось, модели предметной области нельзя рассматривать как корректные или ошибочные; скорее, они полезны в большей или меньшей степени. Отображение всех видов транзакций, как правило, оказывается совсем не лишним, поскольку каждый из типов связан с различными понятиями, процессами и бизнес-правилами[5].

Еще одним интересным вопросом является выбор степени обобщения, которую нужно использовать в модели. С учетом приведенных рассуждений, предположим, что с каждой транзакцией связаны дата и время. Эти стандартные атрибуты, а также желание выполнить максимально допустимое обобщение для этого семейства понятий, приведут к созданию класса `PaymentAuthorizationTransaction`.

Однако насколько удачным будет обобщение типов откликов в рамках классов `CreditPaymentAuthorizationReply` и `CheckPaymentAuthorizationReply` (рис. 26.9) или достаточно более низкой степени обобщения, как показано на рис. 26.10?

Иерархия классов, показанная на рис. 26.10, оказывается достаточно полезной в смысле обобщения, поскольку дополнительные классы не приводят к повышению ясности модели. Иерархия, представленная на рис. 26.9, приводит к излишнему дроблению классов, которое не улучшает трактовку понятий и бизнес-правил, а только усложняет модель. Повышение сложности нежелательно, если оно не дает каких-либо других преимуществ.

---

[5] В моделях предметной области коммуникационных систем также полезно идентифицировать все типы сообщений.

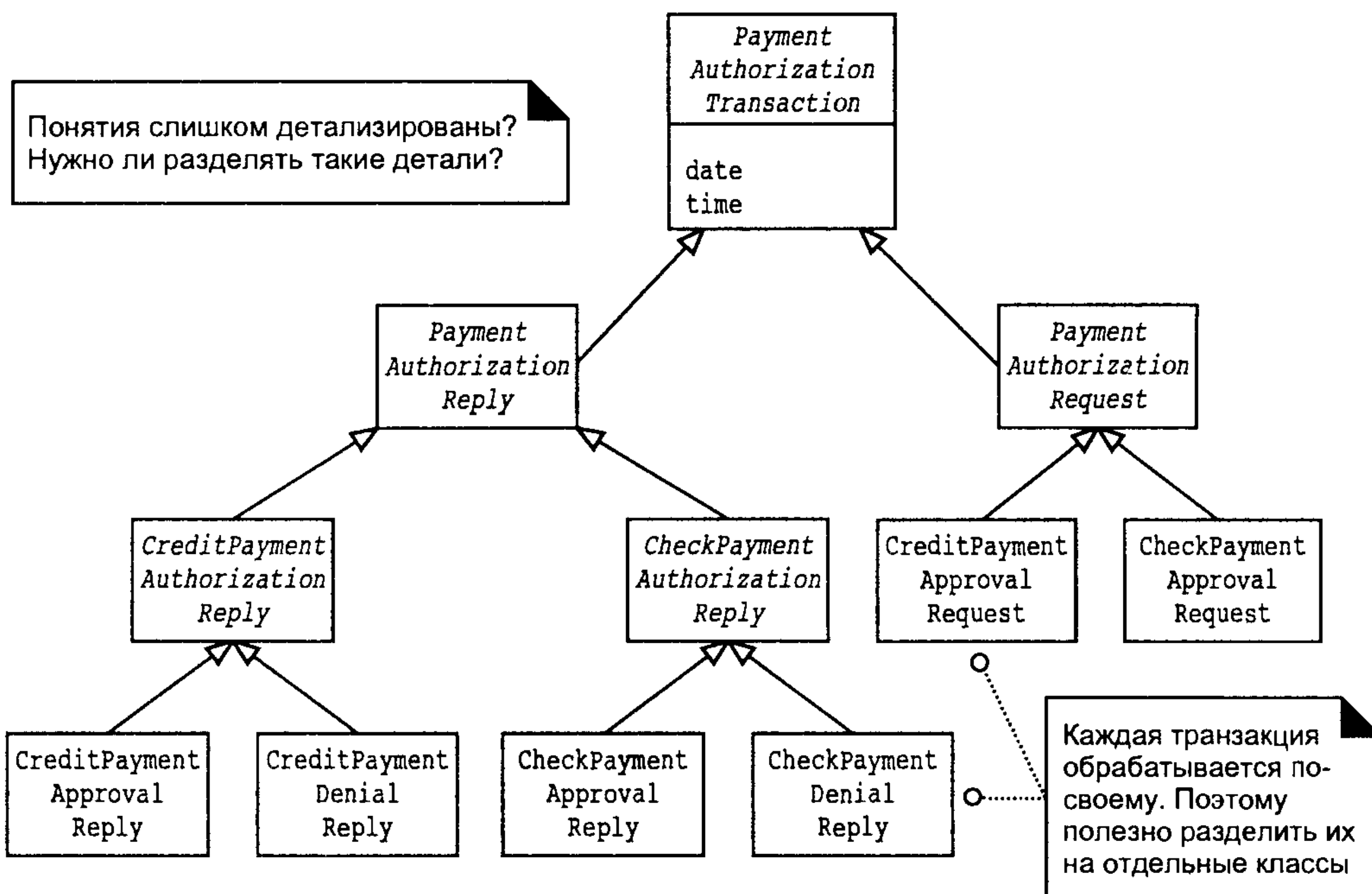



*Рис. 26.9. Одна из возможных иерархий классов транзакций с внешними службами*

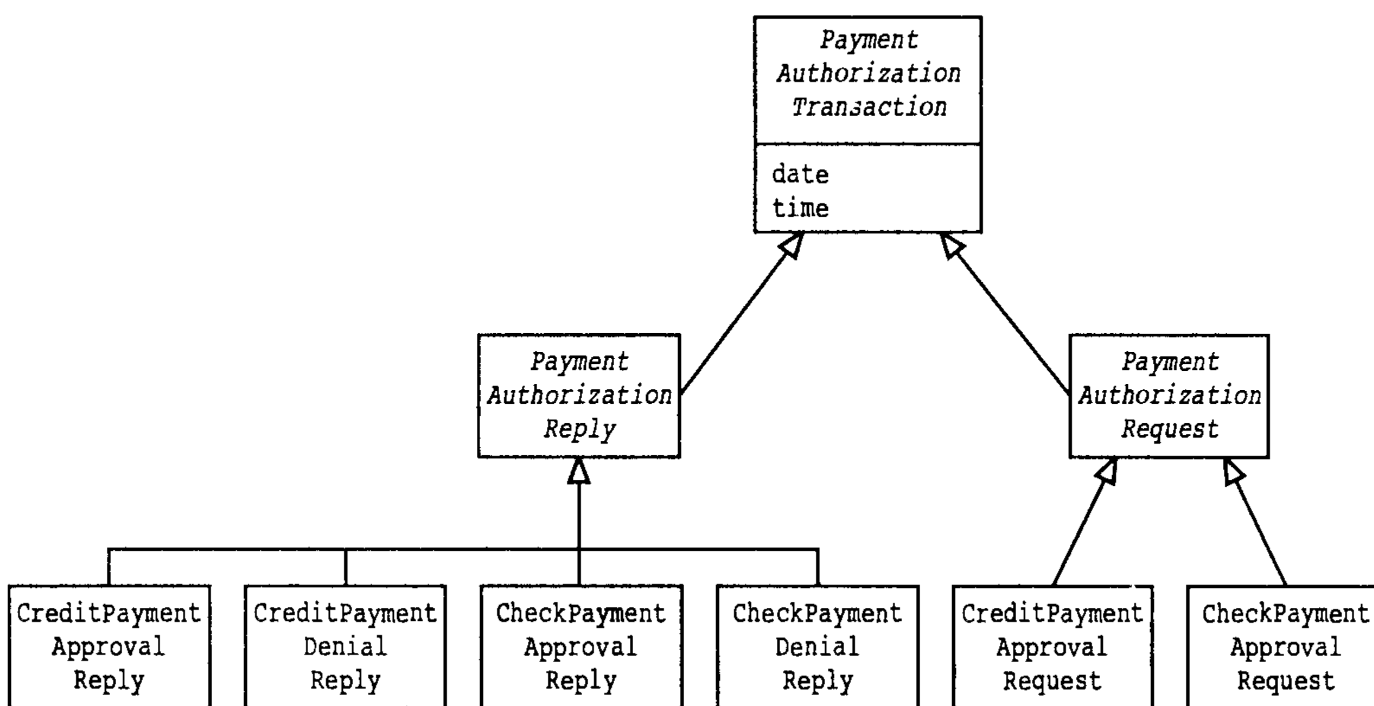



*Рис. 26.10. Альтернативная иерархия классов транзакций*

## 26.7. Абстрактные классы

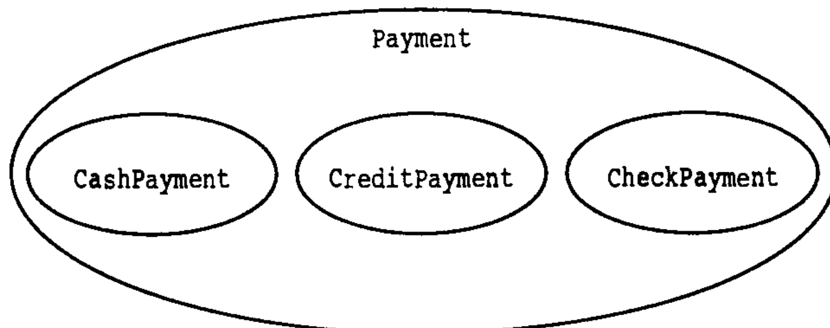
В модели предметной области полезно выделить абстрактные классы, поскольку с их помощью можно уточнить правила предметной области и определить классы, которые могут иметь конкретные экземпляры.

Если каждый член класса C должен быть и членом его подкласса, то класс C называется **абстрактным концептуальным классом** (abstract conceptual class).

Предположим, что каждый экземпляр Payment должен быть также экземпляром подкласса CreditPayment, CashPayment или ChackPayment. Если каждый член класса Payment является также членом своего подкласса, то, по определению, класс Payment является абстрактным.

И наоборот, если могут существовать экземпляры Payment, не являющиеся экземплярами подкласса, то класс Payment не является абстрактным (рис. 26.11, *а*).

В предметной области системы розничной торговли каждый экземпляр Payment является экземпляром своего подкласса. На рис. 26.11, *б* классы представлены корректно, и, таким образом, класс Payment является абстрактным.



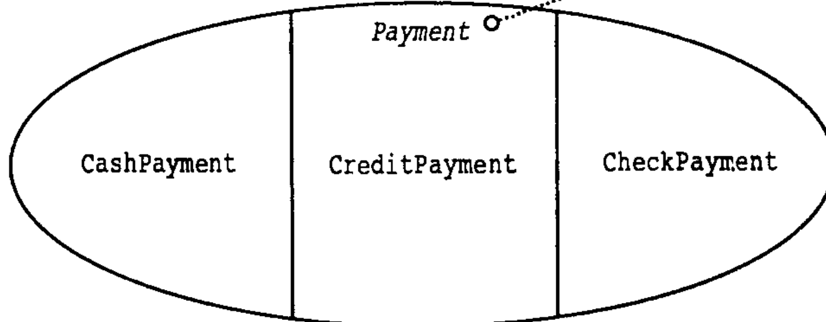




*Рис. 26.11. Абстрактные концептуальные классы*

### *Обозначение абстрактных классов в UML*

В языке UML для отображения абстрактных классов существует собственное обозначение — их имена отображаются курсивом (рис. 26.12).

Идентифицируйте абстрактные классы и отображайте их в модели предметной области, выделяя их имена курсивом.

## 26.8. Моделирование изменения состояний

Предположим, что платеж может быть либо авторизованным, либо неавторизованным и что состояние платежа нужно отобразить в модели предметной области (на самом деле это может быть совсем не так, однако для обсуждения сделаем такое предположение). Как показано на рис. 26.13, одним из возможных спосо-

бов решения этой задачи является определение подклассов класса Payment: UnauthorizedPayment и AuthorizedPayment. Однако заметим, что платеж не остается в одном из этих состояний, а обычно переходит из неавторизованного состояния к авторизованному. На основании приведенных рассуждений можно сформулировать следующие рекомендации.

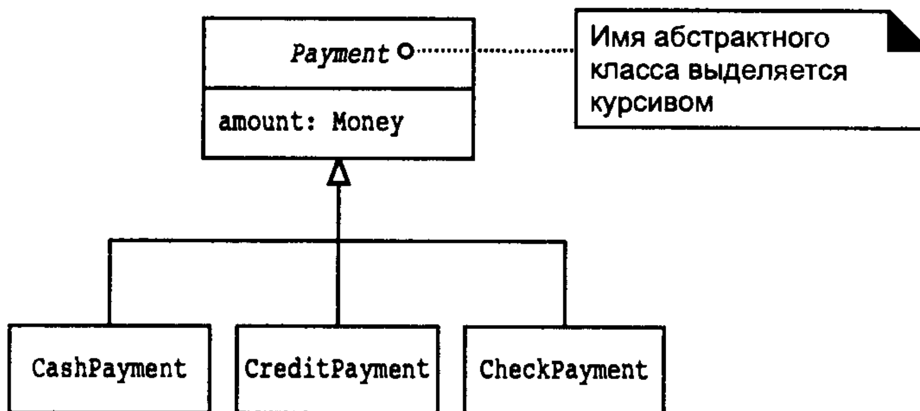



*Рис. 26.12. Обозначение абстрактного класса*

> Не моделируйте состояния понятия X как его подклассы. Лучше выполните одно из следующих действий.
>
> - Определите иерархию состояний и ассоциируйте их с понятием X.
> - Не отображайте состояния понятия в модели предметной области. Лучше проиллюстрируйте их на диаграмме состояний.

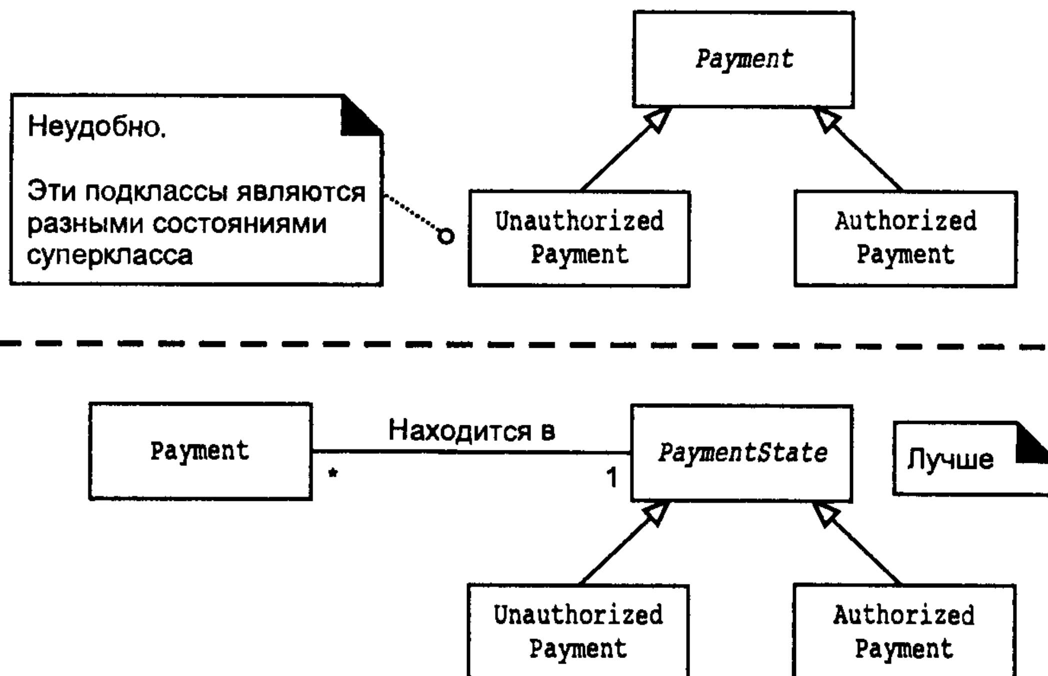



*Рис. 26.13. Моделирование изменения состояний*

## 26.9. Иерархия классов и наследование

Приведенные рассуждения об иерархии концептуальных классов не имеют никакого отношения к *наследованию* (inheritance), поскольку они относятся к модели предметной области и сущностям окружающего мира, а не к артефактам

программного обеспечения. В объектно-ориентированном языке программирования подкласс **наследует** (inherits) атрибуты и определения операций своих суперклассов, создавая тем самым **иерархию программных классов** (software class hierarchy). **Наследование** (inheritance) — это программный механизм реализации совместимости подклассов с определениями суперклассов. Таким образом, наследование находится за пределами обсуждения модели предметной области, хотя при переходе к стадии проектирования и реализации его значение возрастает.

Иерархии концептуальных классов, созданные в данной главе, могут и не отображаться в модели проектирования. Например, иерархия классов транзакций служб авторизации может быть сужена или расширена в альтернативной иерархии программных классов, в зависимости от используемого языка программирования и других факторов. Например, для уменьшения количества классов можно использовать параметризованные классы языка C++.

Глава

# Усовершенствование модели предметной области

*Настоящее — это часть вечности, отделяющая предметную область разочарования от царства надежды.*

*Амброс Бирс (Ambrose Bierce)*

Основные задачи

- Добавить классы ассоциаций в модель предметной области.
- Добавить отношения агрегации.
- Промоделировать временные интервалы.
- Выбрать способ моделирования ролей.
- Организовать элементы модели предметной области в пакеты.

## Введение

В этой главе рассматриваются некоторые полезные идеи и обозначения, которые можно использовать при концептуальном моделировании для всестороннего переосмысления модели предметной области POS-системы NextGen.

## 27.1. Классы ассоциаций

Необходимость задания классов ассоциаций определяется следующими требованиями предметной области.

- Каждому магазину служба авторизации присваивает торговый идентификатор (merchant ID), используемый для его идентификации в процессе взаимодействия.
- В запрос на авторизацию платежа, передаваемый из магазина службе авторизации, требуется включить торговый идентификатор магазина.
- При работе с каждой службой магазин использует отдельный торговый идентификатор.

Где же в модели предметной области должен располагаться атрибут торгового идентификатора?

Помещать торговый идентификатор merchantID в объект Store некорректно, поскольку магазин может иметь несколько таких идентификаторов. Такая же ситуация возникает при добавлении атрибута merchantID к объекту AuthorizationService (рис. 27.1).

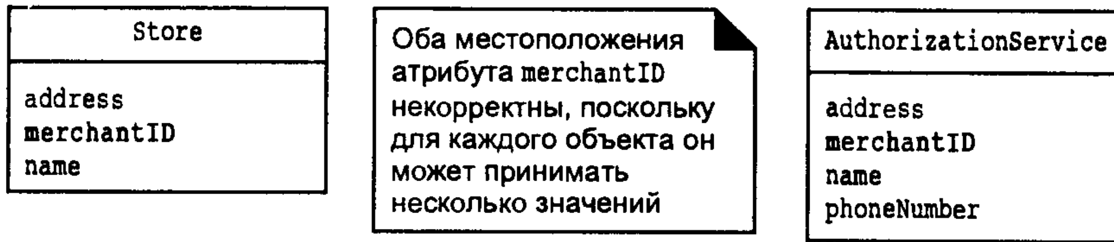



*Рис. 27.1. Неправильное использование атрибута*

Приведенные выше рассуждения приводят к следующему принципу моделирования.

> Если в модели предметной области экземпляр класса C может одновременно иметь несколько значений для одного и того же атрибута A, то атрибут A не следует помещать в класс C. Этот атрибут A следует отнести к другому классу, связанному с классом C, например следующим образом.
>
> - Объект Person может иметь несколько номеров телефона. Разместите номер телефона в другом классе, таком как PhoneNumber или ContactInformation, а затем свяжите несколько экземпляров этого класса с объектом Person.

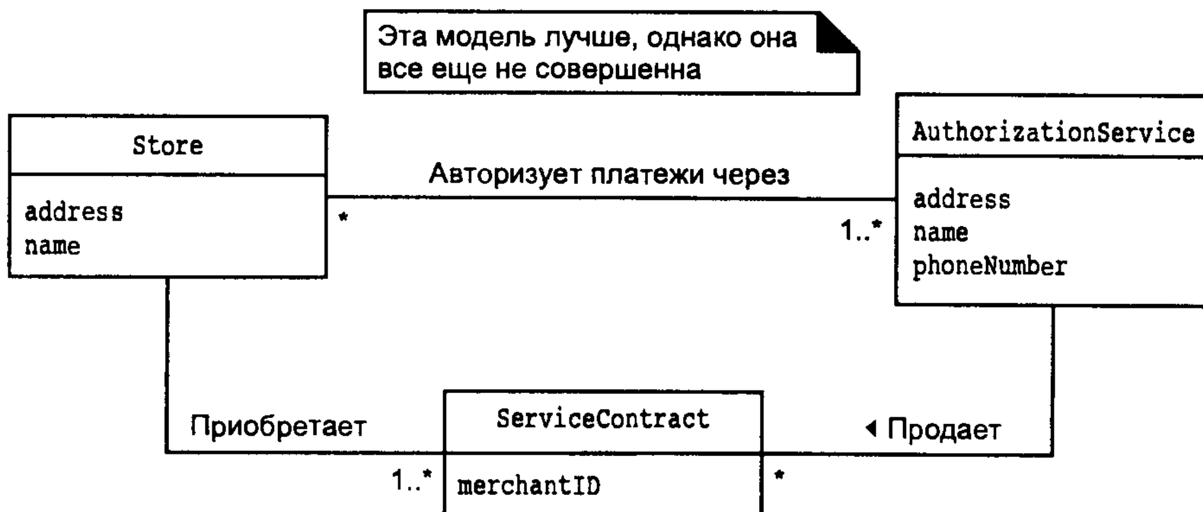



*Рис. 27.2. Первая попытка моделирования проблемы использования торгового идентификатора*

Приведенная на рис. 27.2 модель соответствует сформулированному выше принципу. Действительно, какое понятие делового мира формально отвечает за информацию, связанную с предоставлением услуг клиентам? Такими понятиями могут быть Contract (Договор) и Account (Счет).

Тот факт, что оба объекта, Store и AuthorizationService, связаны с объектом ServiceContract, наводит на мысль о существовании связи между этими двумя

объектами. Идентификатор merchantID можно рассматривать как атрибут, связанный с ассоциацией, соединяющей объекты Store и AuthorizationService.

Это приводит к необходимости создания *класса ассоциации* (association class), в который можно добавить описание самой ассоциации. Класс ServiceContract можно представить как класс ассоциации, связанный с ассоциацией между объектами Store и AuthorizationService.

В языке UML это иллюстрируется с помощью пунктирной линии, направленной от ассоциации к классу ассоциации. Из рис. 27.3 видно, что класс ServiceContract и его атрибуты связаны с ассоциацией между объектами Store и AuthorizationService, а время его жизни зависит от этой связи.

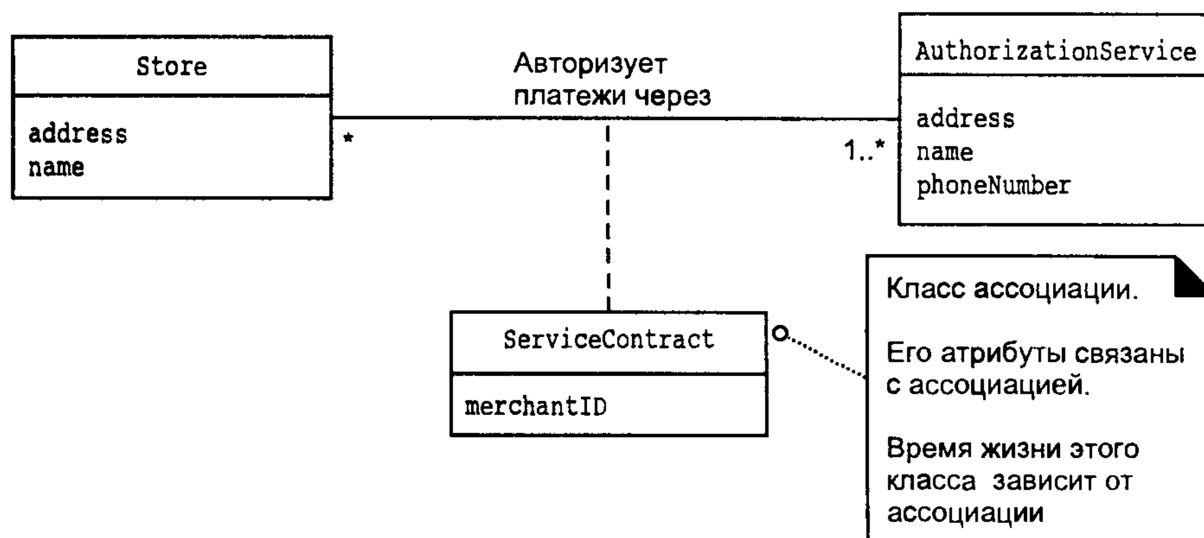



*Рис. 27.3. Класс ассоциации*

## Рекомендации

При добавлении классов ассоциаций руководствуйтесь следующими рекомендациями.

Ниже приведены критерии включения класса ассоциации в модель предметной области.

- Атрибут связан с ассоциацией.
- Время жизни экземпляров класса ассоциации зависит от ассоциации.
- Между двумя понятиями существует ассоциация "многие ко многим", и имеется информация, связанная с самой ассоциацией.

Существование ассоциации "многие ко многим" является стандартным признаком того, что где-то в глубине подсознания зарождается полезный класс ассоциации. Если вы обнаружите такую ассоциацию, то рассматривайте ее как класс ассоциации.

На рис. 27.4 показаны несколько примеров классов ассоциаций.

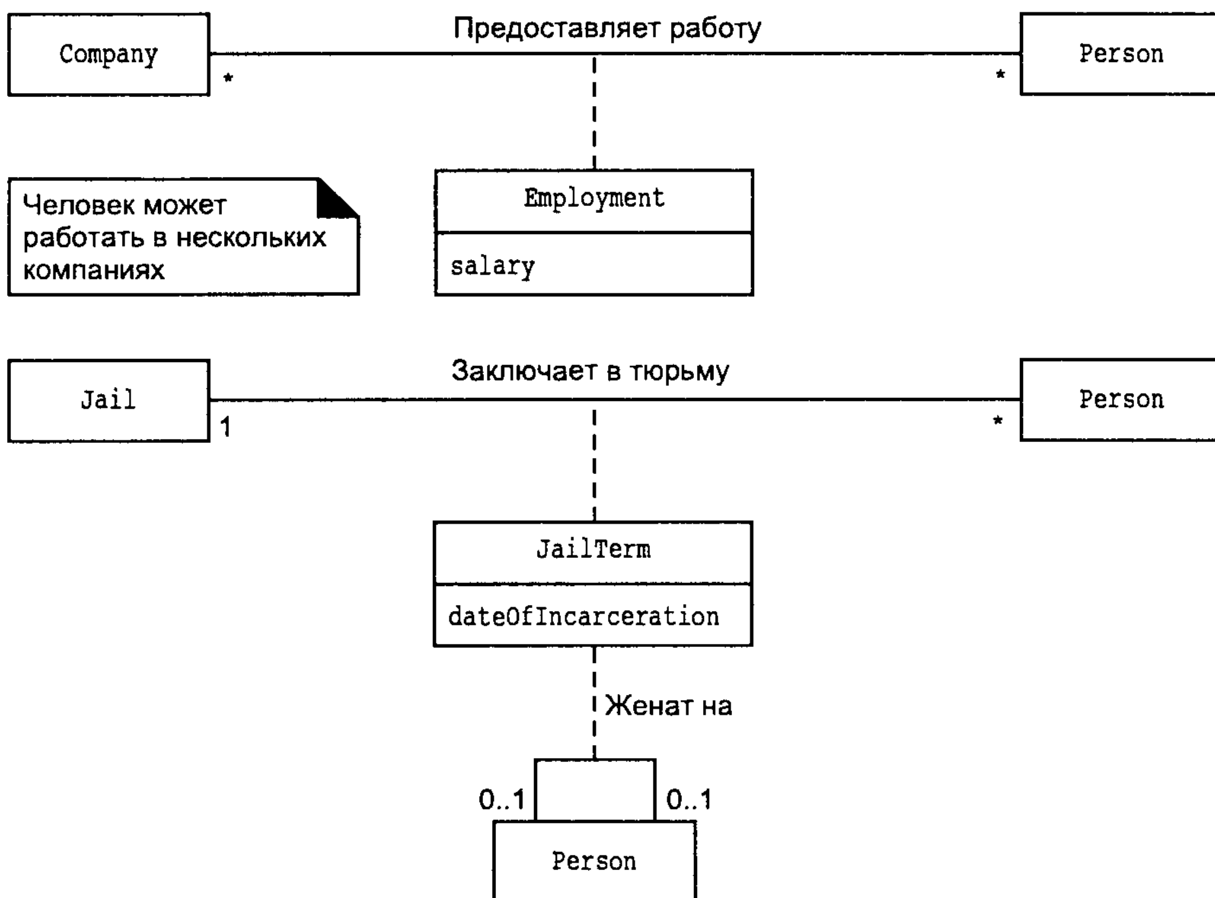



*Рис. 27.4. Примеры классов ассоциаций*

## 27.2. Агрегация и объединение

*Агрегация* (aggregation) — это вид ассоциации, полезный для моделирования взаимосвязей "целое–часть" между сущностями. Целое обычно называется *композитным объектом* (composite).

Например, физические сущности организуются с использованием отношения агрегации, такого как `Hand` (Рука) агрегирует `Finger` (Палец).

### Агрегация в языке UML

В языке UML агрегация отображается с помощью полого или затененного ромбовидного символа, помещенного со стороны композитного объекта ассоциации "целое–часть" (рис. 27.5).

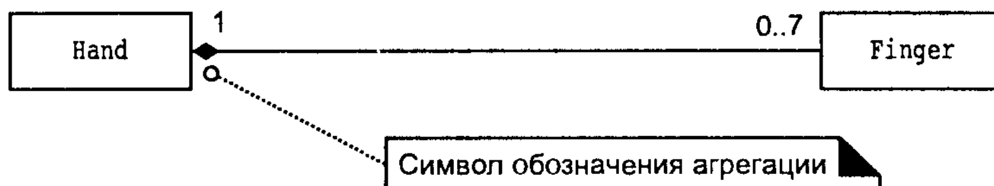



*Рис. 27.5. Обозначение отношения агрегации*

Агрегация является свойством роли ассоциации[1].

[1] Напомним, что каждому концу линии ассоциации соответствует роль и что роль в UML имеет различные свойства, такие как кратность, имя, направление перемещения и свойство `isAggregate`.

Имя ассоциации зачастую не указывается в отношениях агрегации, поскольку в его качестве можно использовать имя `Содержит часть`. Однако для иллюстрации более глубоких семантических связей имя может оказаться полезным.

## Композитная агрегация: затененный ромб

*Композитная агрегация* (composite aggregation) или *объединение* (composition) означает, что части или компоненты связаны лишь с одним композитным объектом. При этом между частями композитного объекта существует некоторая зависимость. В качестве примера композитной агрегации можно привести руку и пальцы.

В модели проектирования наличие композитного объекта и существование зависимости означает, что композитный объект создает (или инициирует создание) своих компонентов (например, объект `Sale` создает экземпляры `SalesLineItem`).

Однако в модели предметной области, на диаграмме которой не показаны программные объекты, ассоциация создания целым объектом своих частей возникает редко (реальная продажа не создает свои элементы). Тем не менее, аналогия налицо. Например, в модели предметной области, относящейся к анатомии человека, рука включает пальцы. Поэтому фраза "рука существует" означает также, что существуют и пальцы этой руки.

Композитная агрегация иллюстрируется с помощью затененного ромба. Считается, что составной объект является "владельцем" своих частей и что эти части можно расположить, согласно иерархии древовидной структуры. Такая форма агрегации при построении моделей стандартна.

Например, палец является частью только одной руки, и для иллюстрации такого отношения используется затененное условное обозначение (рис. 27.6).

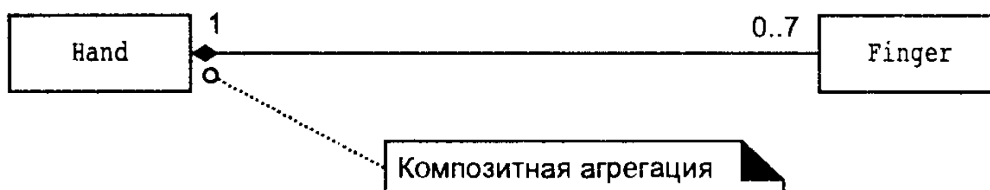



*Рис. 27.6. Композитная агрегация*

Если кратность такой ассоциации со стороны композитного объекта равна 1, значит, его части не могут существовать независимо от самого композита. Например, если удалить палец одной руки, то его нужно немедленно присоединить к другому композитному объекту (другой руке). Именно это иллюстрирует диаграмма безотносительно к медицинским аспектам этой проблемы.

Если кратность ассоциации со стороны композитного объекта составляет `0..1`, значит, часть можно удалить из композита, и она продолжит свое существование даже без взаимосвязи с другим композитным объектом. Поэтому, если вы хотите, чтобы пальцы существовали сами по себе, используйте кратность `0..1`.

## Совместная агрегация: полый ромб

*Совместная агрегация* (shared aggregation) означает, что со стороны составного объекта кратность может быть больше единицы. При этом в условном обозначении ромб остается незатененным. Подразумевается, что с одной частью может быть связано несколько экземпляров составного объекта. В физических

агрегатах совместная агрегация встречается достаточно редко (если встречается вообще); в основном, она связана с нефизическими понятиями.

Например, пакет языка UML можно рассматривать как агрегат его элементов. Однако на элемент можно ссылаться в нескольких пакетах (элемент содержится в одном пакете, а остальные на него ссылаются). Пример совместной агрегации приведен на рис. 27.7.

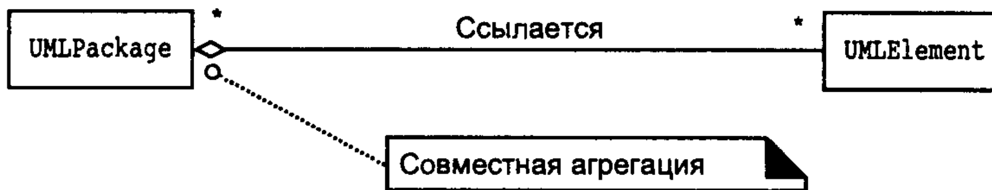



*Рис. 27.7. Совместная агрегация*

## Как идентифицировать отношение агрегации

В некоторых случаях наличие отношения агрегации очевидно. Обычно это имеет место при рассмотрении физических устройств. Однако иногда выявить подобные взаимоотношения гораздо сложнее.

> Если вы не уверены в том, что между понятиями существует отношение агрегации, не включайте его в модель.

Для принятия решения о необходимости иллюстрации отношения агрегации руководствуйтесь следующим рекомендациями.

> Отображайте отношения агрегации в следующих случаях.
>
> - Время жизни компонента ограниченно временем жизни составного объекта, т.е. между частью и целым существует зависимость создания/удаления.
> - В физическом или логическом агрегате очевидно наличие отношений “целое–часть”.
> - Некоторые свойства составного объекта распространяются и на его компоненты, например место их расположения.
> - Операции, применяемые к составному объекту, осуществляются и над его частями, например разрушение, перемещение и запись.

Наличие отношений агрегации между составными частями устройства или агрегата абсолютно очевидно. Еще одним не менее полезным критерием наличия таких отношений является наличие зависимости создания/удаления между целым и его частями.

## Преимущества отображения отношений агрегации

Идентификация и иллюстрация отношений агрегации не очень важны. Их вполне можно исключить из модели предметной области. Большинство (если не все) опытных специалистов по моделированию считают отображение таких ассоциаций пустой тратой времени.

Все же стоит изучить и проиллюстрировать отношения агрегации, поскольку это может обеспечить ряд преимуществ, большинство из которых связано со

стадией генерации программного кода, а не со стадией построения модели предметной области. Именно поэтому отношения агрегации можно не опасаясь, пропустить при моделировании предметной области. Вот эти преимущества.

- Проясняются ограничения предметной области, связанные с возможностью существования частей независимо от целого. При композитной агрегации отдельный компонент не может существовать вне времени жизни целого.
  - На стадии проектирования это будет оказывать влияние на зависимости создания/удаления между целым и частями программных классов или элементов базы данных (в терминах целостности ссылок и каскадного удаления).
- Отображение таких ассоциаций помогает идентифицировать создателя (составной объект) в соответствии с шаблоном Creator.
- Операции, такие как копирование и удаление, применяемые к целому, зачастую должны распространяться также на его части.

### Отношение агрегации в модели предметной области POS-системы

В предметной области POS-системы объекты `SalesLineItem` можно рассматривать как части составного объекта `Sale`. В целом, транзакции с отдельными товарами можно рассматривать как части объединенной транзакции (рис. 27.8). Кроме того, имеется зависимость создания/удаления экземпляров `SalesLineItem` объекта `Sale`, поскольку время их жизни ограничено рамками времени жизни объекта `Sale`.

Можно провести аналогичные рассуждения и сказать, что объект `ProductCatalog` агрегирует экземпляры `ProductSpecification`.

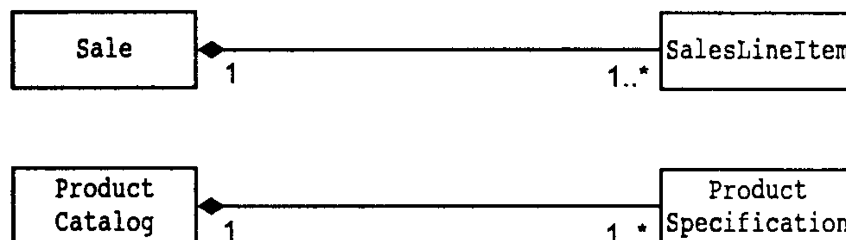



*Рис. 27.8. Отношение агрегации в приложении розничной торговли*

Других отношений, удовлетворяющих семантике “целое–часть” и зависимости создания/удаления, нет, так что на помощь приходит принцип “ не уверен, не используй”.

## 27.3. Временные интервалы и цены товаров: устранение ошибки первой итерации

На первой итерации разработки объекты `SalesLineItem` были связаны с объектами `ProductSpecification`, содержащими информацию о цене товара. Для начальных итераций разработки это было обусловленное упрощение, однако теперь пришло время его детализировать. При этом возникает интересный (и актуальный) вопрос о *временных интервалах* (time interval), связанных с корректностью информации, действием договоров и т.п.

Если объект SalesLineItem всегда получает цену товара от объекта ProductSpecification, значит, при изменении цены товара объекты старых продаж будут связаны с новой ценой, что некорректно. Значит, необходимо отличать цену товара на момент его покупки и текущую цену.

В зависимости от конкретных требований существует как минимум два способа решения этой проблемы. Один состоит в простом копировании цены товара в элемент SalesLineItem и поддержке текущей цены с помощью объекта ProductSpecification.

Второй, более робастный подход, состоит в связывании коллекции объектов ProductPrice с объектом ProductSpecification. При этом каждый экземпляр ProductPrice обеспечивает цену товара, действующую на определенном временном интервале. При таком подходе в организации будут храниться все когда-либо действовавшие цены (для отслеживания изменения цен на распродажах) и текущие цены товаров (рис. 27.9). Более подробная информация о временных интервалах содержится в [27].

Зачастую наряду с самими значениями приходится хранить коллекцию объектов, представляющих временные интервалы. Это относится к данным физических, медицинских и научных измерений, а также ко многим артефактам из области бухгалтерского учета и юриспруденции.

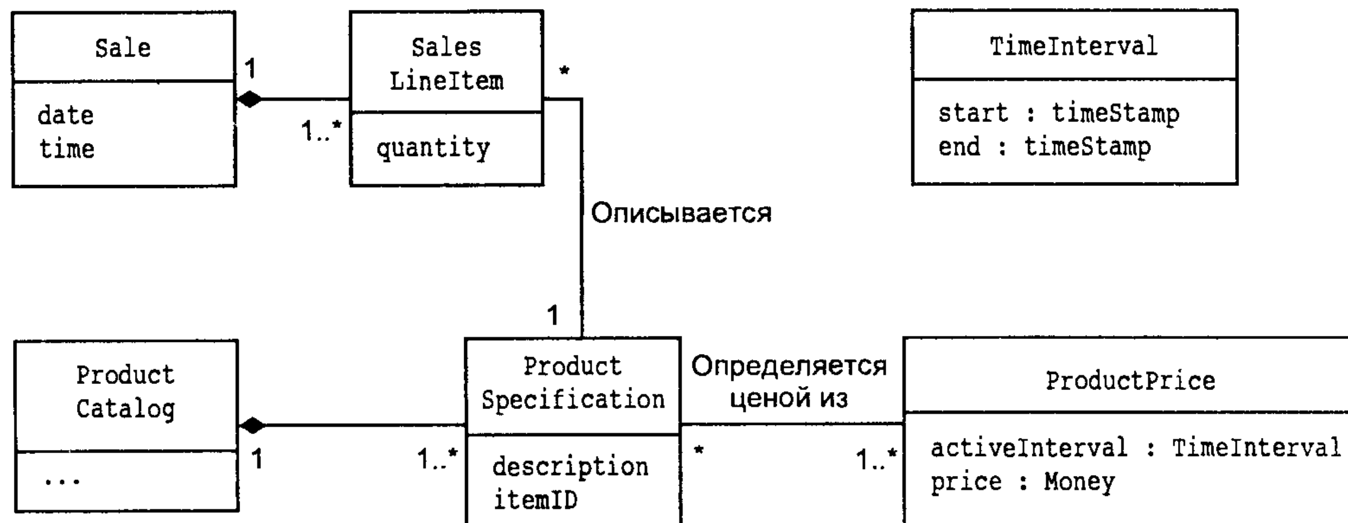



*Рис. 27.9. Временные интервалы и цены товаров*

## 27.4. Имена ролей ассоциаций

Каждый конец линии ассоциации — это роль, с которой связаны различные свойства, а именно:

- имя;
- кратность.

Имя роли идентифицирует одну из сторон ассоциации и, в идеале, описывает роль, которую играет данный объект. На рис. 27.10 представлены примеры имен ролей.

Явное имя роли обычно не требуется. Оно бывает полезным, когда роль объекта является неочевидной. Обычно имя роли начинается со строчного символа. Если имя присутствует неявно, то предполагается, что используется имя

роли по умолчанию, соответствующее имени класса, первый символ которого заменен строчным эквивалентом.

Как уже упоминалось в предыдущих главах, роли, используемые на диаграммах классов на стадии проектирования, можно рассматривать как основу имени атрибута на стадии генерации программного кода.

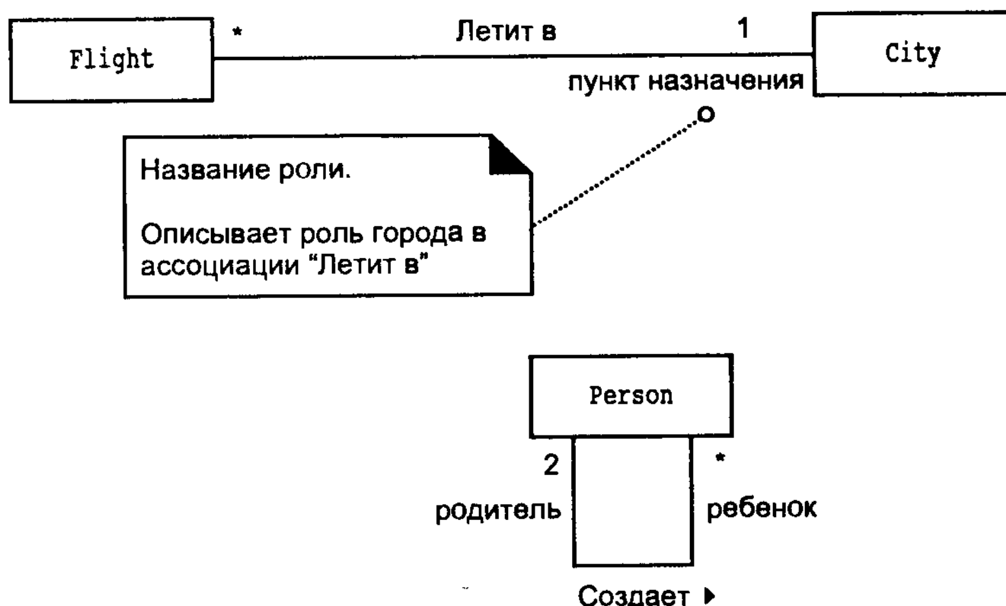



*Рис. 27.10. Имена ролей*

## 27.5. Роли в форме понятий и роли, представленные в ассоциации

В модели предметной области роль из реального мира, в особенности роль, выполняемая человеком, может быть смоделирована несколькими способами, например отдельным понятием или ролью в ассоциации[2].

Например, роли кассира и управляющего могут быть представлены как минимум двумя способами, проиллюстрированными на рис. 27.11.

Первый подход можно назвать представлением ролей в ассоциациях, а второй — представлением ролей в виде понятий. Оба подхода имеют свои преимущества.

Представление ролей в ассоциациях является достаточно привлекательным, поскольку в этом случае можно точно указать, что один и тот же экземпляр может выполнять (в динамике) несколько ролей в различных ассоциациях. Автор как субъект одновременно или последовательно может выполнять роль писателя, разработчика программных систем, родителя и т.д.

Однако роли, представленные в виде понятий, обеспечивают простоту и гибкость при добавлении уникальных атрибутов, ассоциаций и дополнительной семантики. Более того, реализовать роли в форме отдельных классов гораздо проще, поскольку это соответствует ограничениям современных популярных объектно-ориентированных языков программирования. Эти ограничения заключаются в том, что экземпляр одного класса не может динамически видоизменяться и становиться экземпляром другого класса или динамически менять свое поведение и атрибуты в зависимости от изменения своей роли.

---

[2] Для простоты другие возможные способы моделирования (например, описанные в [46]) не рассматриваются.

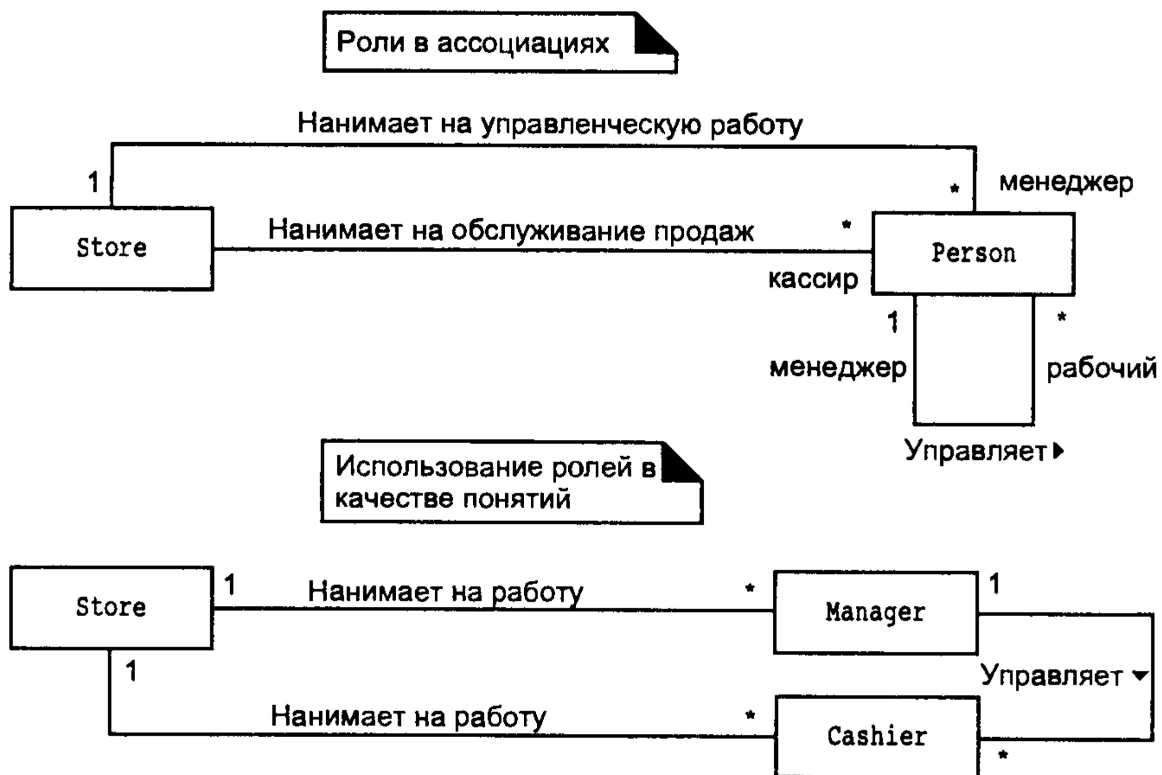



*Рис. 27.11. Два способа моделирования ролей, которые играют люди*

## 27.6. Производные элементы

Производный элемент может быть определен из других объектов. Чаще всего производными элементами являются атрибуты и ассоциации. Когда следует отображать производные элементы?

> Старайтесь не отображать производные элементы на диаграммах, поскольку после их добавления модель усложняется, а новой информации не поступает. Однако если отсутствие производных элементов затрудняет понимание, их следует отображать.

Например, атрибут `total` объекта `Sale` можно создать на основе информации, содержащейся в объектах `SalesLineItem` и `ProductSpecification` (рис. 27.12). В языке UML этот факт отображается с помощью символа "/", расположенного перед именем элемента.

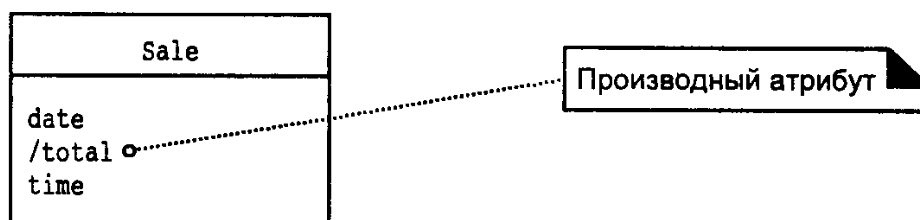



*Рис. 27.12. Производный атрибут*

В качестве еще одного примера можно привести атрибут `quantity` объекта `SalesLineItem`, который является производным от нескольких экземпляров `Item`, связанных с этим объектом (рис. 27.13).

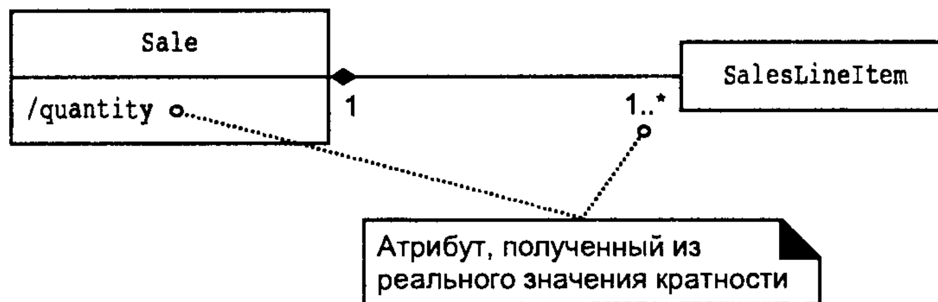



*Рис. 27.13. Производный атрибут, связанный со значением кратности*

## 27.7. Составные ассоциации

В ассоциации можно использовать *спецификатор* (qualifier). С помощью значения спецификатора различаются объекты множества, помещенного со стороны удаленного конца линии ассоциации. Ассоциация со спецификатором называется *составной ассоциацией* (qualified association).

Например, в объекте ProductCatalog объекты ProductSpecification могут различаться идентификатором itemID, как показано на рис. 27.14, *а*. При использовании спецификатора уменьшается кратность на втором конце линии ассоциации, которая становится равной единице (рис. 27.14, *б*). В модели предметной области спецификатор иллюстрирует, чем сущности предметной области одного класса отличаются друг от друга. В модели предметной области спецификаторы нельзя использовать для отражения проектных решений, связанных с ключами поиска, хотя позднее они могут оказаться полезными на диаграммах проектирования.

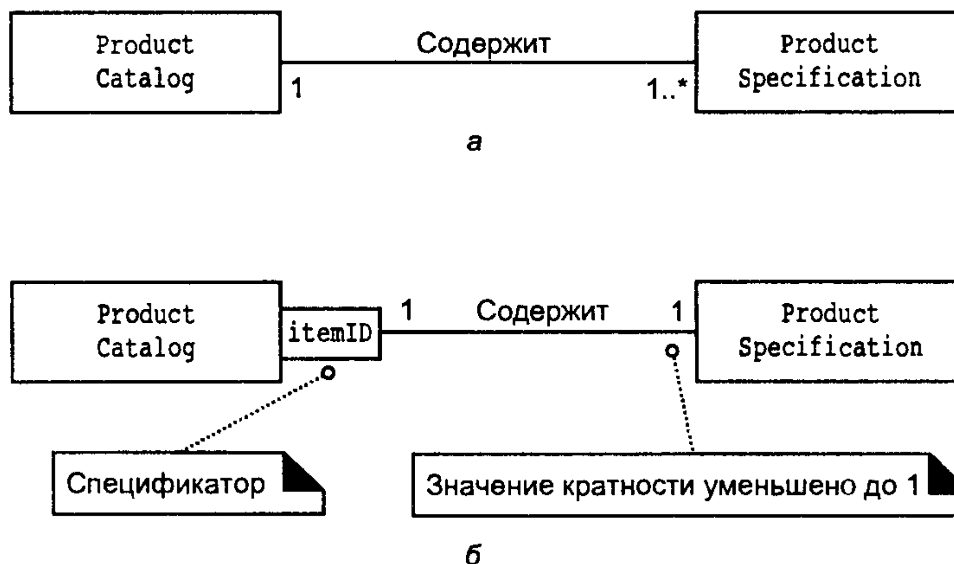



*Рис. 27.14. Составная ассоциация*

Обычно спецификаторы не добавляют новой информации. Более того, при их отображении можно слишком увлечься продумыванием проектного решения. Однако при правильном использовании они могут способствовать улучшению понимания предметной области и происходящих в ней процессов. Составная ассоциация между экземплярами ProductCatalog и ProductSpecification является хорошим примером использования спецификатора, который способствует лучшему пониманию изучаемых процессов.

## 27.8. Рефлексивные ассоциации

Понятие может быть ассоциировано с самим собой. Такая связь называется *рефлексивной ассоциацией* (reflexive association)[3] (рис. 27.15).

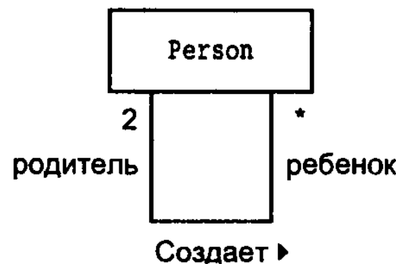



*Рис. 27.15. Рефлексивная ассоциация*

## 27.9. Упорядоченные элементы

Если связанные друг с другом объекты упорядочены, то этот факт можно отобразить на диаграмме (рис. 27.16). Например, выбранные товары, представленные объектами `SalesLineItem`, должны располагаться в порядке ввода.

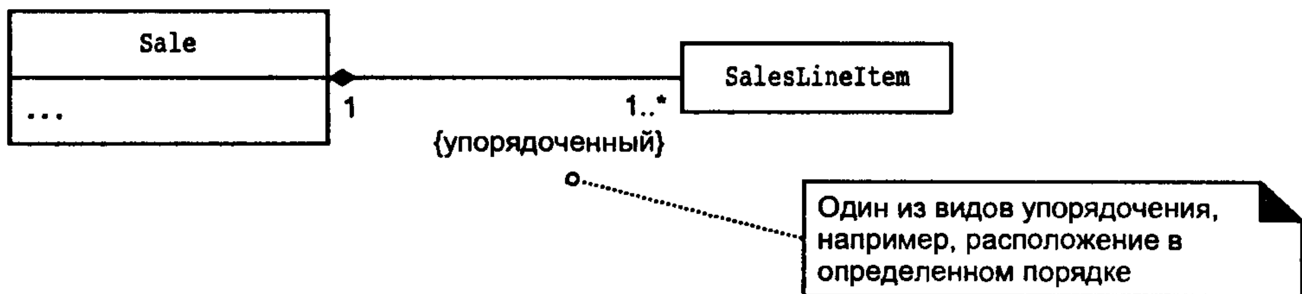



*Рис. 27.16. Упорядоченные элементы*

## 27.10. Использование пакетов для организации элементов модели предметной области

Модель предметной области быстро разрастается, поэтому связанные друг с другом понятия желательно объединять в пакеты, обеспечивая тем самым возможность параллельного анализа отдельных составляющих системы разными специалистами. В следующих разделах будет проиллюстрирована структура пакетов в модели предметной области UP.

### *Обозначение пакетов в языке UML*

Графически пакет представляется в виде папки с корешком (или вкладкой) (рис. 27.17). В нем могут быть размещены подчиненные пакеты. Если в пакете имеются элементы, то его имя указывается во вкладке, в противном случае — в центре самой папки.

#### Принадлежность и ссылки

Элемент может *принадлежать* (owned) пакету, в котором он определен, или на него можно *ссылаться* (referenced) из других пакетов. В этом случае имя

---

[3] В [81] приведено более строгое определение рефлексивной ассоциации.

элемента представляется в формате *ИмяПакета::ИмяЭлемента* (рис. 27.18). Класс, используемый в других пакетах, может быть модифицирован с помощью новых ассоциаций, однако все остальное должно остаться без изменений.

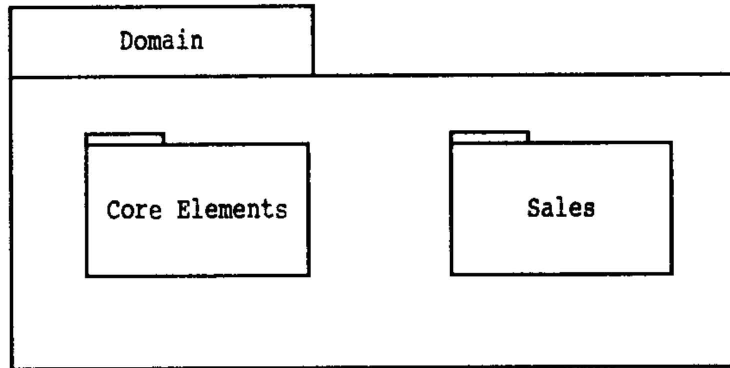



*Рис. 27.17. Пакет UML*

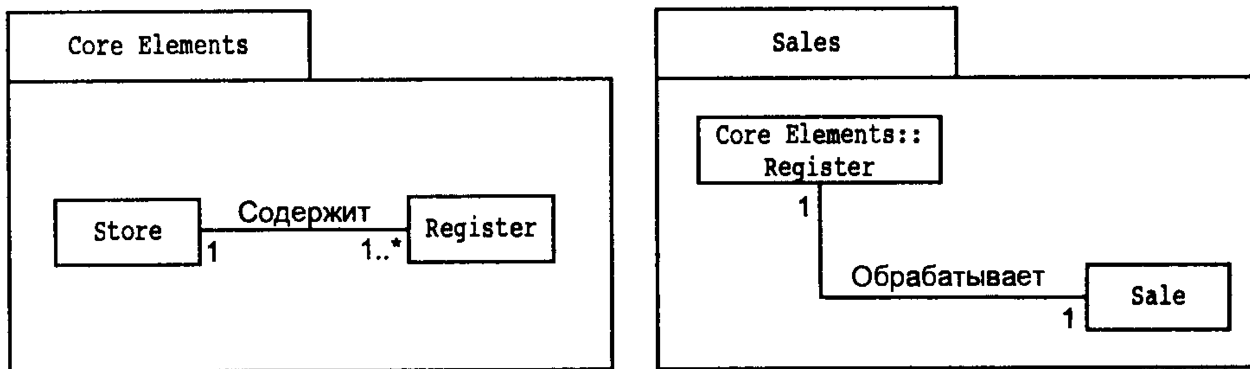



*Рис. 27.18. Ссылка на класс в пакете*

### Зависимости пакетов

Если один элемент модели некоторым образом зависит от другого элемента, то эту зависимость можно отобразить с помощью специальной связи, имеющей вид линии со стрелкой. Зависимость между пакетами определяет, что элементы зависимого пакета каким-либо образом "знают" об элементах целевого (target) пакета или связаны с ними.

Например, если один пакет ссылается на элемент, принадлежащий другому пакету, то между ними существует зависимость. Таким образом, пакет `Sales` зависит от пакета `Core Elements` (рис. 27.19).

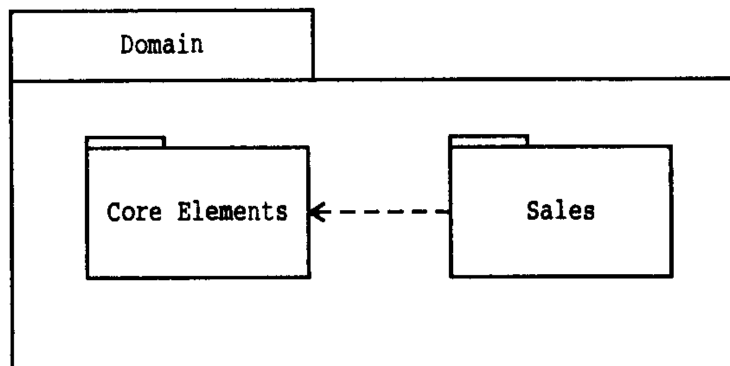



*Рис. 27.19. Зависимость пакетов*

#### Отображение пакетов без использования диаграммы пакетов

Иногда неудобно рисовать диаграмму пакетов, однако в то же время необходимо отобразить пакет, которому принадлежат требуемые элементы.

В этом случае поместите на диаграмму примечание, как показано на рис. 27.20.

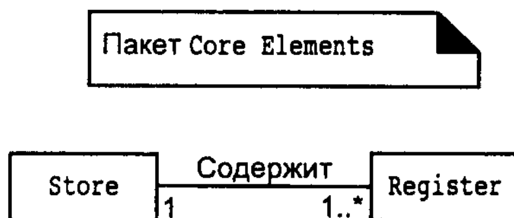



*Рис. 27.20. Отображение принадлежности пакету с помощью примечания*

### Как разделить модель предметной области

Каким образом классы, изображенные в модели предметной области, можно организовать в пакеты? Используйте следующие рекомендации.

При разделении модели предметной области в один пакет помещайте элементы, которые:

- располагаются в одной той же части предметной области, т.е. тесно взаимосвязаны с каким-либо понятием или целью;
- находятся рядом в иерархии классов;
- участвуют в реализации одного и того же прецедента;
- явно связаны друг с другом.

Зачастую бывает полезно все элементы, связанные с моделью предметной области, поместить в пакет `Domain` (Предметная область), а совместно используемые, стандартные, базовые понятия определить в пакете `Core Elements` (Элементы ядра) или `Common Concepts` (Базовые понятия).

## Пакеты модели предметной области POS-системы

С учетом приведенных выше критериев пакеты модели предметной области POS-системы можно организовать следующим образом (рис. 27.21).

### Пакет `Core/Misc`

В пакет `Core/Misc` (рис. 27.22) желательно поместить широко используемые понятия, не относящиеся к другим пакетам. Далее этот пакет будем сокращенно называть `Core`.

На данной итерации с этим пакетом не связаны никакие новые понятия или ассоциации.

### Пакет `Payments`

Как и на первой итерации, новые ассоциации определяются в соответствии с критерием знания. Например, необходимо зафиксировать связь между объектами `CreditPayment` и `CreditCard`. Некоторые ассоциации добавляются просто

для ясности, например, как ассоциация Идентифицирует покупателей для объекта DriverLicense (рис. 27.23).

Заметим, что отклик службы авторизации изображен в виде класса ассоциации PaymentAuthorizationReply. Этот класс обязан своим существованием ассоциации между платежом и службой его авторизации.

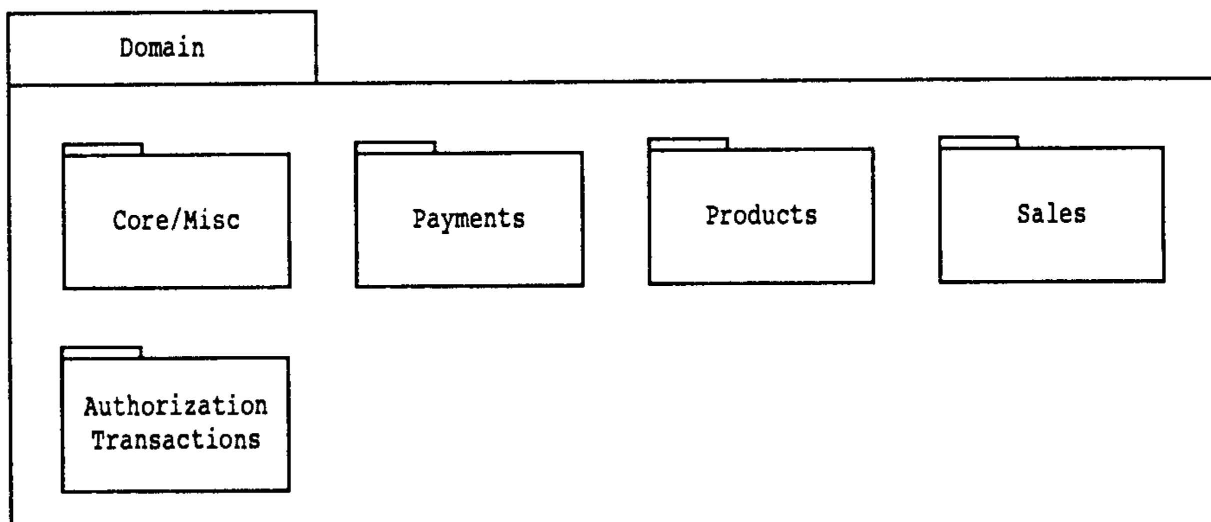



*Рис. 27.21. Пакеты понятий предметной области*

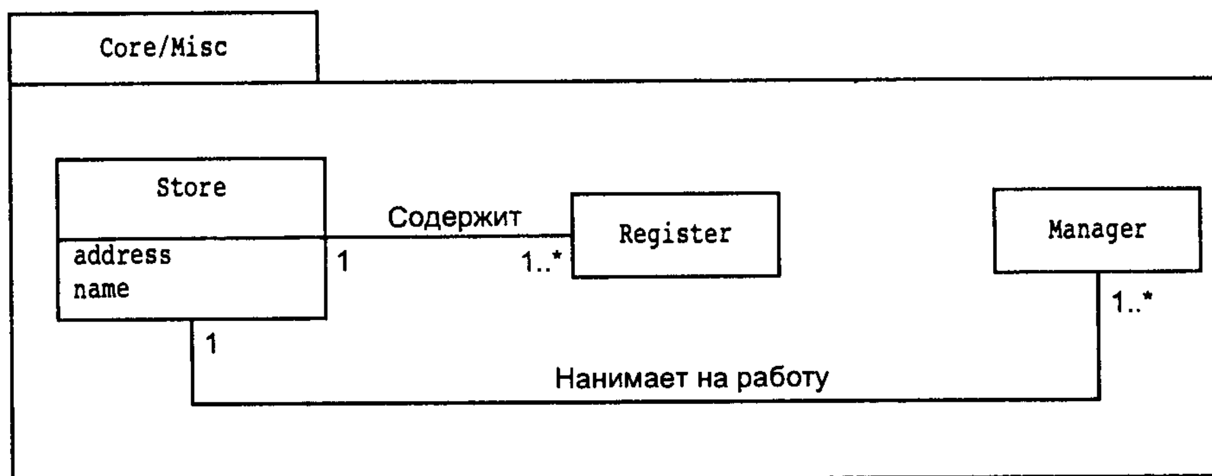



*Рис. 27.22. Пакет* Core

## Пакет Products

На данной итерации не появилось новых понятий и ассоциаций, за исключением композитной агрегации (рис. 27.24).

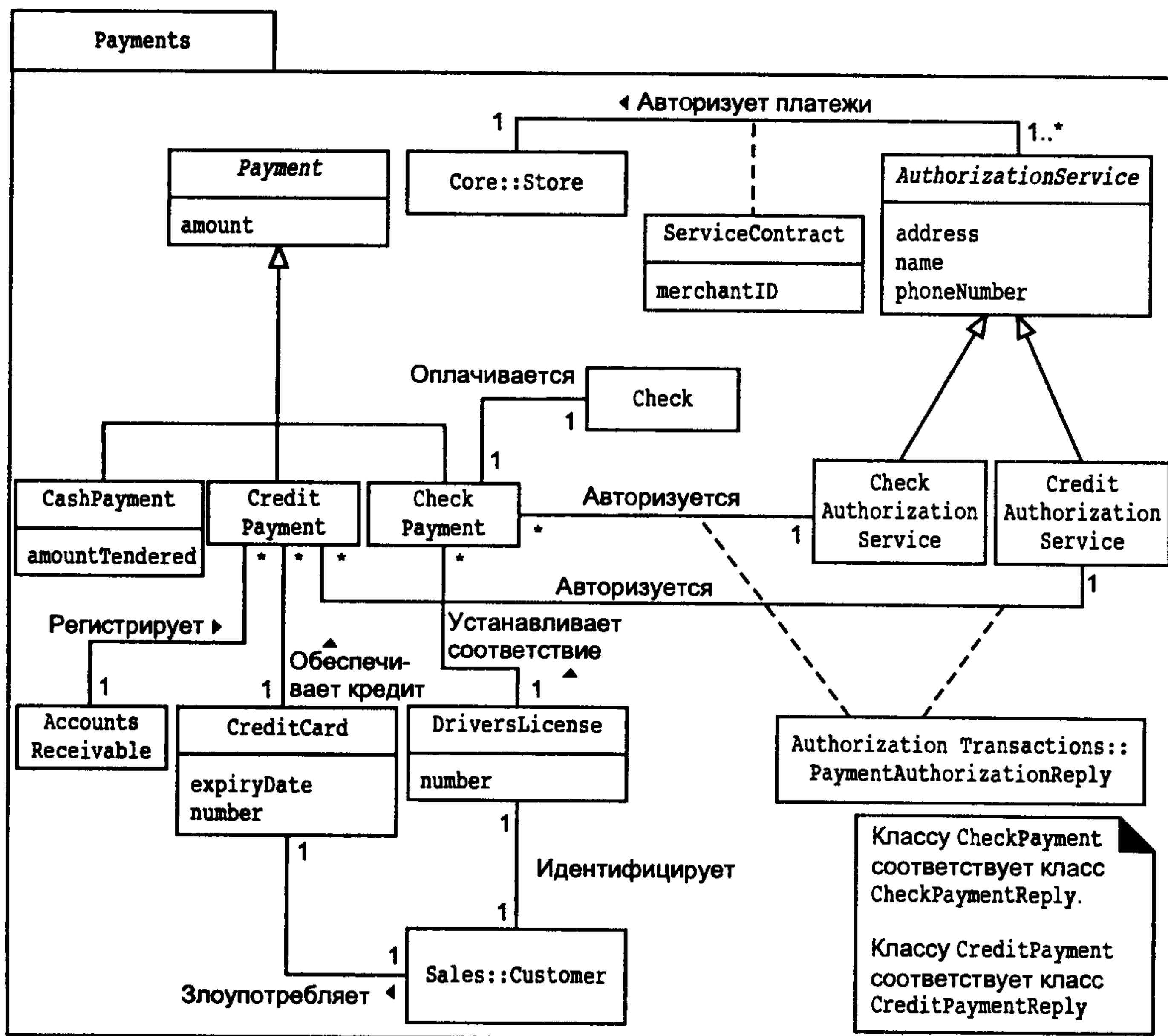


*Рис. 27.23. Пакет* `Payments`

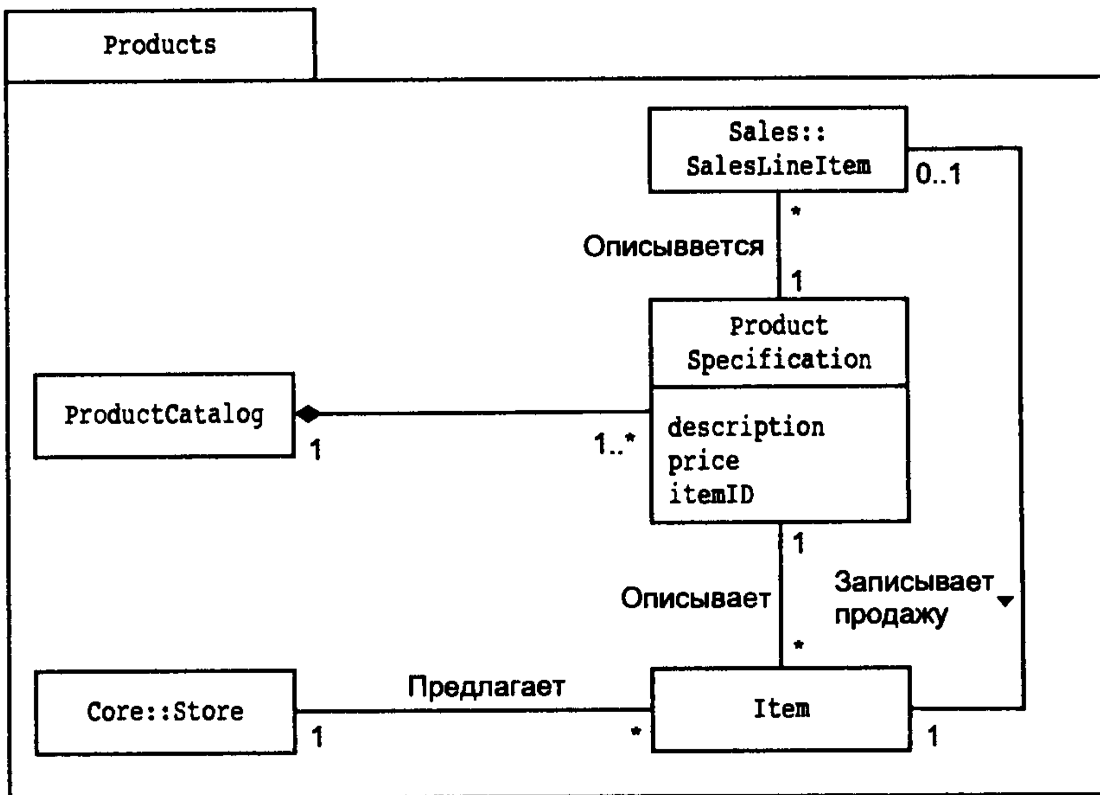



*Рис. 27.24. Пакет* `Products`

### Пакет `Sales`

На данной итерации не появилось новых понятий и ассоциаций, за исключением композитной агрегации и производных атрибутов (рис. 27.25).

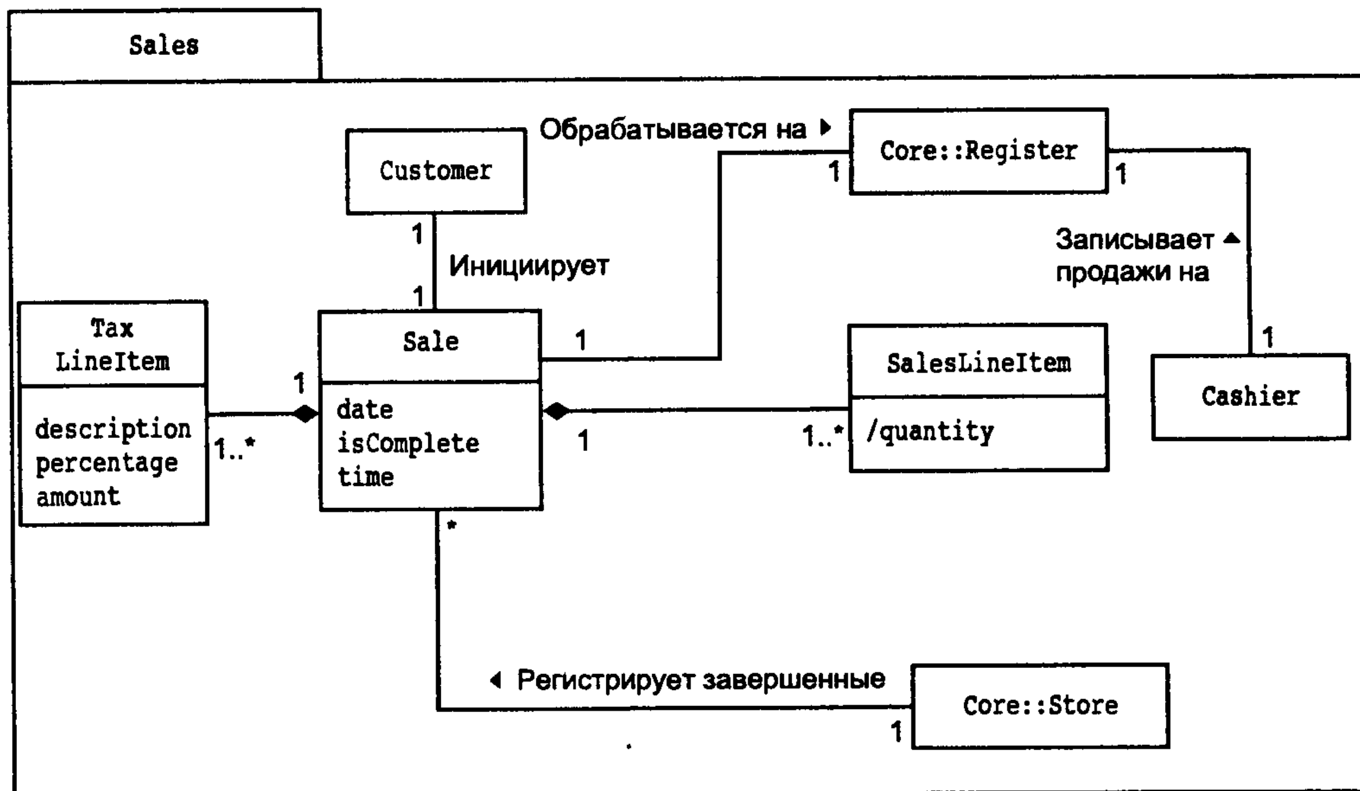



*Рис. 27.25. Пакет* `Sales`

## Пакет Authorization Transactions

Несмотря на общую рекомендацию об использовании осмысленных имен ассоциаций, в некоторых случаях это условие может не выполняться, особенно если тип ассоциации очевиден читателям. Это относится к ассоциациям между объектом платежа и связанными с ним транзакциями. Имена таких ассоциаций можно опустить, поскольку читатели диаграммы классов, представленной на рис. 27.26, легко определят тип этой ассоциации. Добавление имен только усложнит диаграмму.

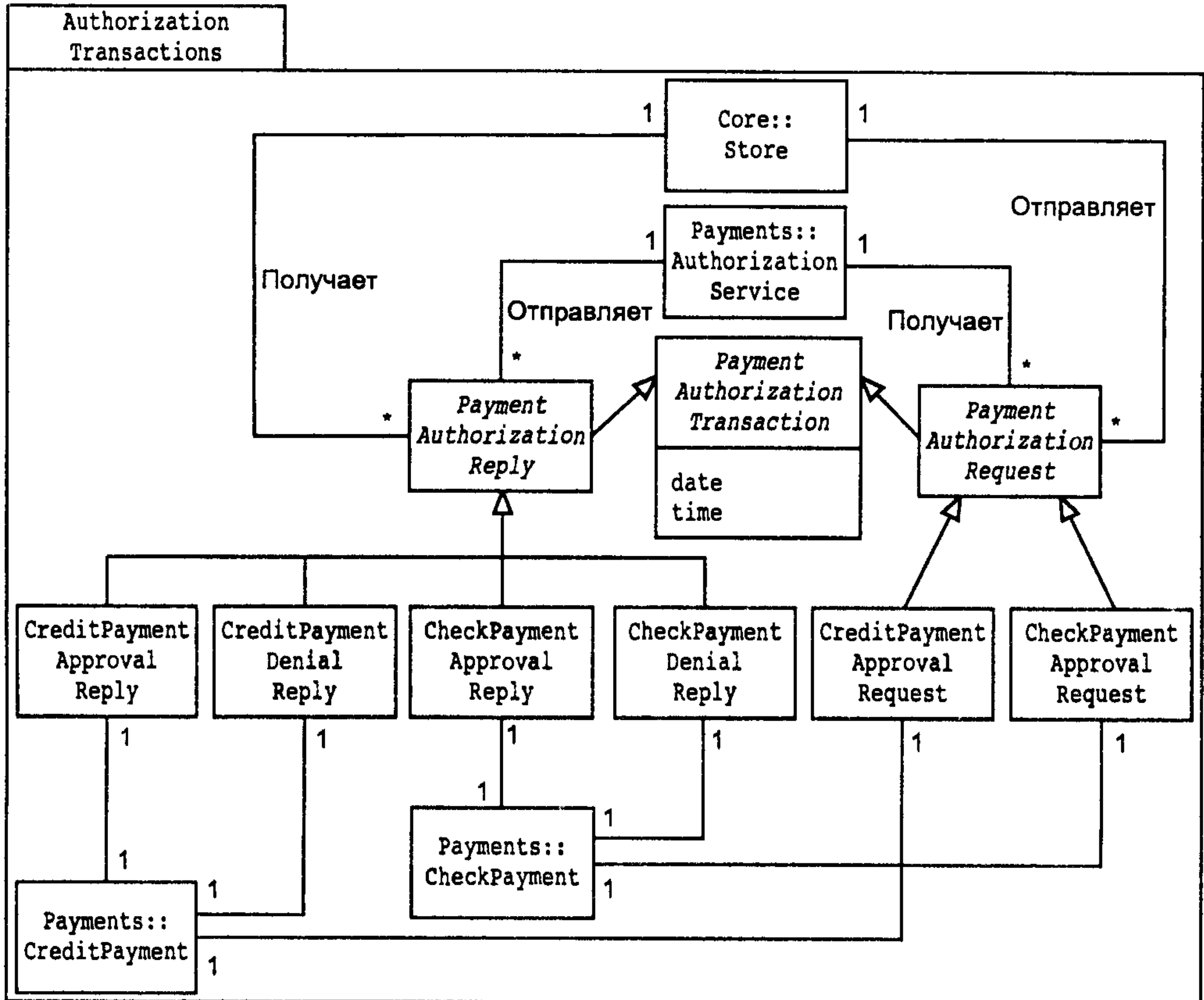



*Рис. 27.26. Пакет* Authorization Transactions

Возможно, эта диаграмма слишком детализирована? Как сказать. Реальным критерием является полезность диаграммы. Диаграмма, безусловно, корректна. Но повышает ли она уровень понимания процессов предметной области? Ответ на этот вопрос определяет степень детализации модели предметной области.

Глава

# Поведение системы

*Добродетель — не лучшее искушение.*

*Джордж Бернард Шоу (George Bernard Shaw)*

---

**Основная задача**

- **Определить диаграммы последовательностей и описания системных операций для текущей итерации.**

---

## 28.1. Новые диаграммы последовательностей

На текущей итерации для реализации требований по обработке различных видов платежей необходимо обеспечить взаимодействие с внешними системами. Следовательно, на диаграммах последовательностей (SSD — System Sequence Diagram) нужно проиллюстрировать межсистемные взаимодействия, рассматривая каждую систему как черный ящик. На диаграммах SSD полезно изобразить новые системные события, проясняющие следующие аспекты функционирования.

- Новые системные операции, которые должна поддерживать POS-система NextGen.
- Обращения к другим системам и их отклики на запросы.

### *Стандартное начало сценария прецедента Оформление продажи*

Диаграмма последовательностей для начала базового сценария включает системные события `enterItem` и `endSale`. Эти события всегда одинаковы, независимо от типа оплаты (рис. 28.1).

### *Оплата с использованием кредитной карточки*

На этой диаграмме последовательностей представлен процесс, происходящий после стандартного начала прецедента Оформление продажи, когда для оплаты используется кредитная карточка (рис. 28.2).

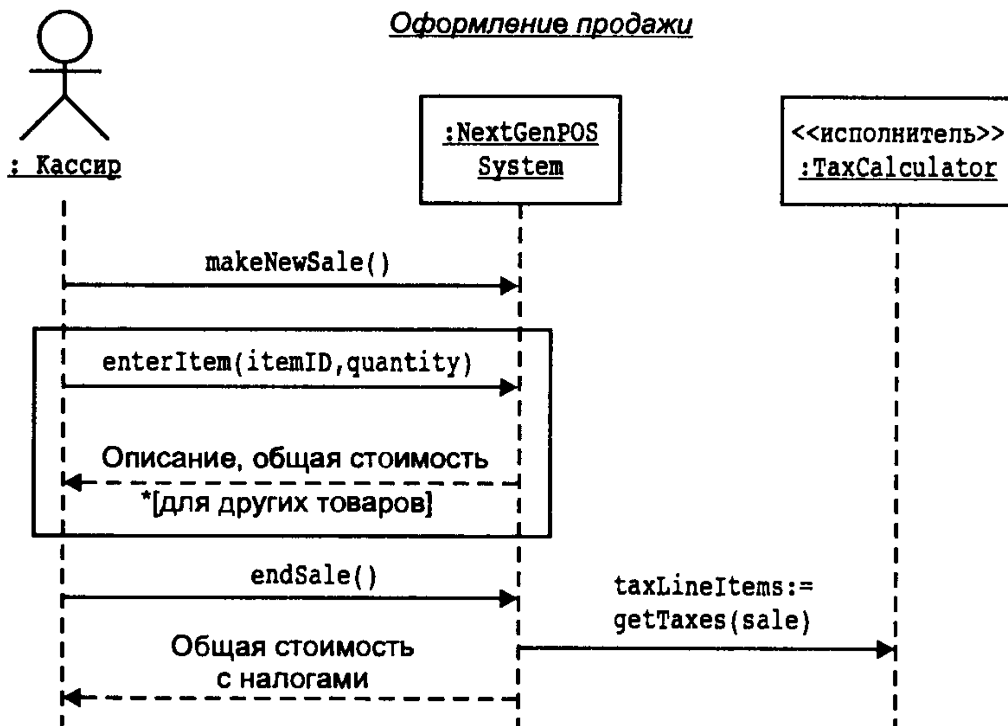



*Рис. 28.1. Диаграмма SSD со стандартным началом сценария*

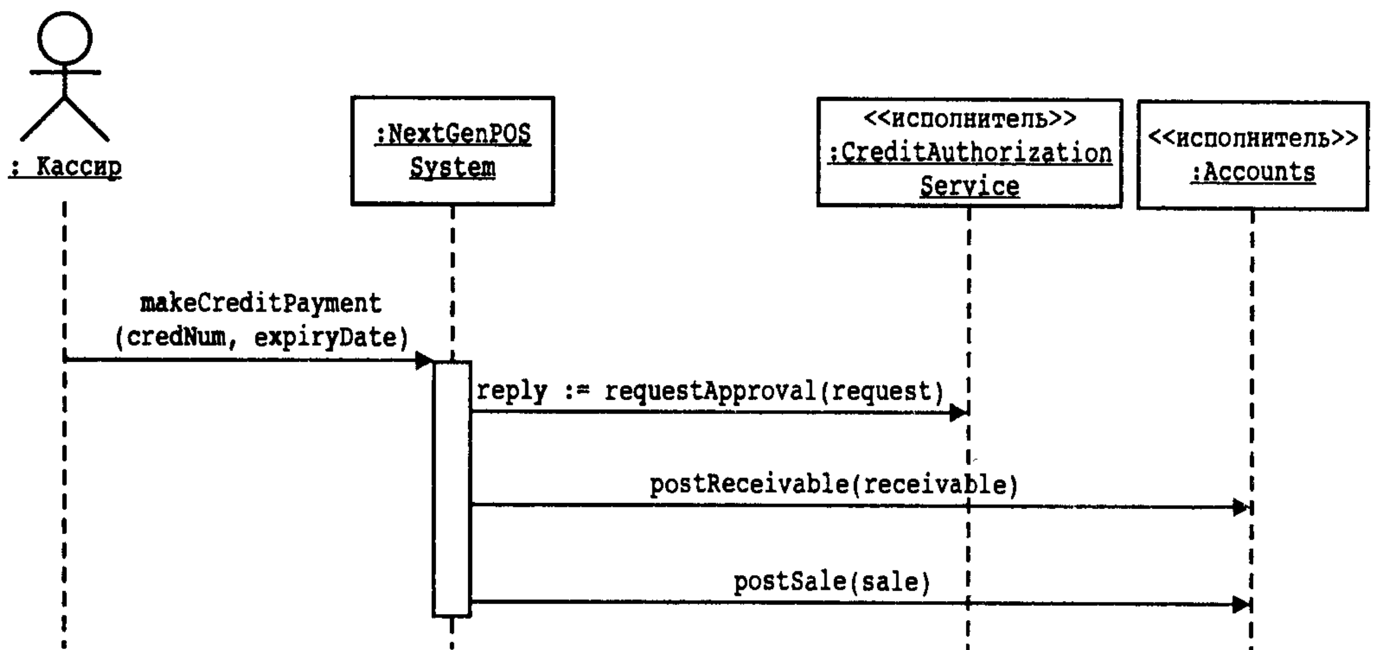



*Рис. 28.2. Диаграмма последовательностей для оплаты по кредитной карточке*

В обоих случаях (при оплате покупки по кредитной карточке и чеком) на данной итерации принимаются некоторые упрощения. В частности, предполагается, что сумма платежа в точности соответствует стоимости покупки, следовательно, параметр "предлагаемая сумма" не требуется.

Обратите внимание, что обращение к внешней службе авторизации `CreditAuthorizationService` моделируется как обычное синхронное сообщение с возвращаемым значением. Это некая абстракция, которую можно будет реализовать с помощью запроса SOAP на основе защищенного протокола HTTPS или любого другого механизма удаленного взаимодействия. Детали использования конкретного протокола скрыты с помощью введенного на предыдущей итерации адаптера ресурса.

Для системной операции makeCreditPayment (и данного прецедента) предполагается, что информация о платежеспособности покупателя поступает из кредитной карточки, поэтому в систему вводится номер кредитного счета и срок его действия (с помощью устройства считывания информации с карточки). Хотя в будущем могут появиться другие механизмы получения информации о платежеспособности покупателя, кредитные карточки с высокой степенью вероятности тоже будут поддерживаться.

Напомним, что в случае подтверждения платежа от службы авторизации платежей по кредитной карточке, этой службе необходимо знать идентификатор магазина, поэтому его нужно указать.

### *Оплата чеком*

На этой диаграмме последовательностей представлен процесс оплаты покупки чеком (рис. 28.3).

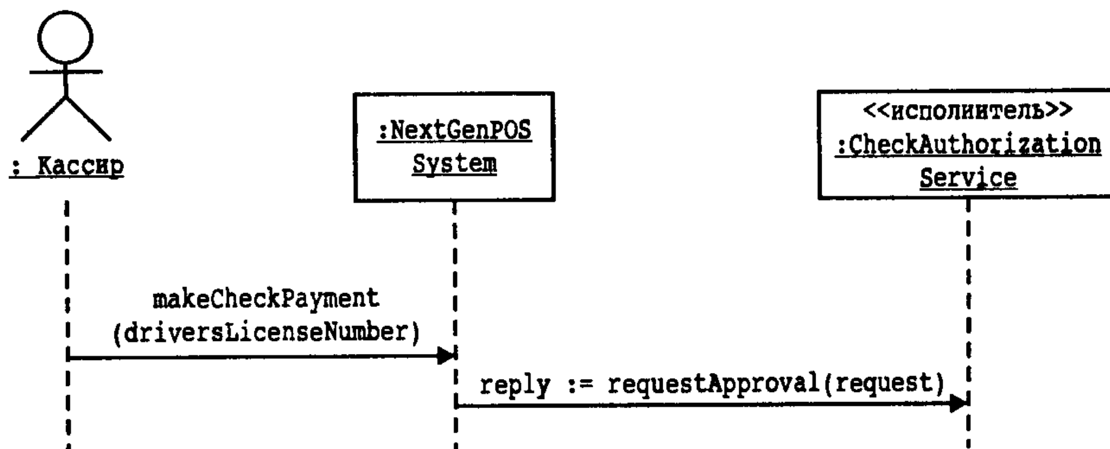



*Рис. 28.3. Диаграмма последовательностей для оплаты чеком*

## 28.2. Новые системные операции

На данной итерации в систему необходимо добавить следующие новые системные операции.

- makeCreditPayment
- makeCheckPayment

На первой итерации была реализована только операция для оплаты наличными под общим названием makePayment. Теперь, после появления различных типов платежей, ее нужно переименовать в makeCashPayment.

## 28.3. Описания новых системных операций

Напомним, что описания системных операций — это необязательные артефакты (относящиеся к модели прецедентов). Иногда для получения информации о системных операциях достаточно описания прецедентов. Однако описания системных операций позволяют точно определить, что происходит в системе в терминах изменения состояния объектов из модели предметной области.

Приведем описания новых системных операций.

### Описание ОП5: makeCreditPayment

| | |
|---|---|
| **Операция** | makeCreditPayment (creditAccountNumber, expiryDate) |
| **Ссылки** | Прецедент Оформление продажи |
| **Предусловия** | Текущая продажа существует, и все товары выбраны |
| **Постусловия** | - Создан экземпляр pmt класса CreditPayment<br>- Объект pmt связан с текущим объектом sale класса Sale<br>- Создан экземпляр cc класса CreditCard; при этом cc.number = creditAccountNumber, cc.expiryDate = expiryDate<br>- Объект cc связан с объектом pmt<br>- Создан экземпляр cpr класса CreditPaymentRequest<br>- Объект pmt связан с объектом cpr<br>- Создан объект re класса ReceivableEntry<br>- Объект re связан с внешним объектом AccountReceivable<br>- Объект sale для завершения продажи связан с объектом Store |

Обратите внимание на постусловие, определяющее создание нового экземпляра счета ReceivableEntry среди бухгалтерских счетов. Хотя эта обязанность выходит за рамки системы NextGen, это постусловие было введено для корректности реализации операции.

Например, на этапе тестирования на основе этого постусловия нужно протестировать бухгалтерскую систему и проверить наличие нового экземпляра счета.

### Описание ОП6: makeCheckPayment

| | |
|---|---|
| **Операция** | makeCheckPayment (driversLicenceNumber) |
| **Ссылки** | Прецедент Оформление продажи |
| **Предусловия** | Текущая продажа существует, и все товары выбраны |
| **Постусловия** | - Создан экземпляр pmt класса CheckPayment<br>- Объект pmt связан с текущим объектом Sale<br>- Создан экземпляр dl класса DriverLicense; при этом dl.number = driversLicenseNumber<br>- Объект dl связан с объектом pmt<br>- Создан экземпляр cpr класса CheckPaymentRequest<br>- Объект pmt связан с объектом cpr<br>- Объект sale для завершения продажи связан с объектом Store |

Глава

# Моделирование поведения на диаграммах состояний

*Удобство использования подобно кислороду — замечают только его отсутствие.*

*Автор неизвестен*

**Основная задача**

- Создать диаграммы состояний для классов и прецедентов.

## Введение

В состав языка UML входит система обозначений диаграмм состояний. С их помощью иллюстрируются события и состояния объектов — транзакций, прецедентов, людей и т.д. В этой главе обсуждаются наиболее важные аспекты использования диаграмм состояний, однако, кроме них, существуют и многие другие особенности.

Применение диаграмм состояний, в основном, ориентировано на отображение системных событий в рамках описания прецедентов, но они могут быть использованы и в ряде других случаев.

## 29.1. События, состояния и переходы

*Событие* (event) — это значимое или заслуживающее внимания происшествие. Например:

- снятие телефонной трубки с рычага.

*Состояние* (state) — это условие, характеризующее объект в некоторый момент между двумя событиями. Например:

- телефон находится в состоянии “ожидания” с момента опускания трубки на рычаг и до момента снятия трубки.

*Переход* (transition) — это такое отношение между двумя состояниями, которое указывает на переход объекта из одного состояния в другое при выполнении некоторого события. Например:

- в случае реализации события "снятие трубки" телефон переходит из состояния ожидания в состояние активности.

## 29.2. Диаграммы состояний

На диаграммах состояний языка UML (рис. 29.1) иллюстрируются интересные события и состояния объекта, а также поведение объекта в ответ на реализацию событий. Переходы между состояниями обозначаются стрелками с указанием соответствующих событий. Сами состояния изображаются в виде прямоугольников со скругленными углами. Зачастую используется некоторое начальное псевдосостояние, которое автоматически переходит в другое состояние при создании экземпляра класса.

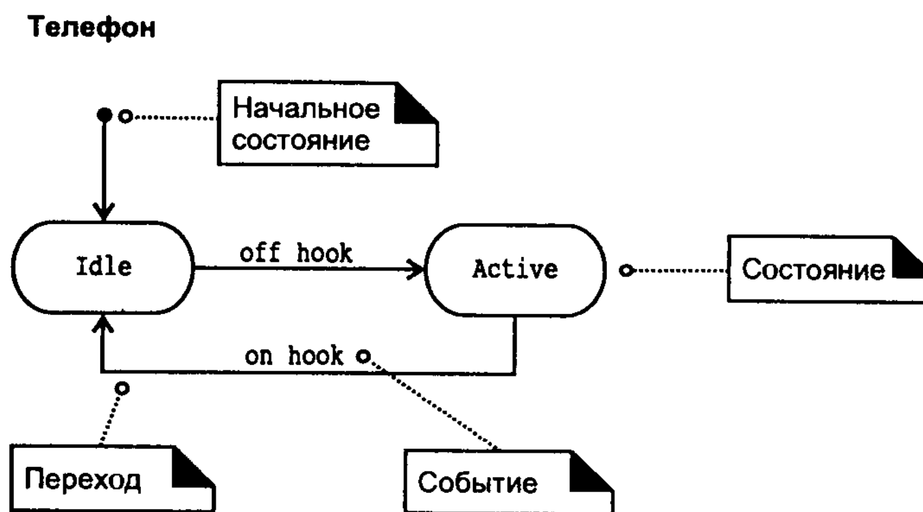



*Рис. 29.1. Диаграмма состояний для телефона*

Диаграмма состояний отражает жизненный цикл объекта. На ней отображаются события, в которых участвует данный объект, переходы и состояния объекта между событиями. Нет необходимости иллюстрировать все возможные события: в случае реализации события, не отображенного на диаграмме, событие игнорируется, согласно диаграмме состояний. Следовательно, можно создавать диаграммы состояний, описывающие жизненный цикл объекта с произвольной степенью детализации или упрощения, в зависимости от текущих потребностей.

### Предмет диаграммы состояний

Диаграммы состояний можно строить для различных элементов языка UML, включая:

- классы (концептуальные или программные);
- прецеденты.

Поскольку всю систему можно представить в виде класса, для нее тоже можно построить диаграмму состояний.

## 29.3. Диаграммы состояний и UP

В контексте UP не определена отдельная модель состояний. Однако для каждого элемента любой модели (проектирования, предметной области и т.д.) можно построить диаграмму состояний для более глубокого понимания динамического поведения этого элемента в ответ на события. Например, можно построить диаграмму состояний для класса Sale в рамках модели проектирования.

## 29.4. Диаграммы состояний для прецедентов

Диаграммы состояний зачастую используются для описания допустимой последовательности внешних системных событий, распознаваемых и обрабатываемых системой в контексте одного прецедента, как, например, показано ниже.

- В процессе реализации прецедента Оформление продажи в POS-приложении NextGen операцию makeCreditPayment нельзя выполнять до события endSale.
- В процессе реализации прецедента Обработка документа в текстовом процессоре нельзя выполнять операцию сохранения файла fileSave до наступления события создания или открытия файла fileNew или fileOpen.

Диаграмма состояний, отражающая события всей системы и их последовательность в рамках одного прецедента, называется *диаграммой состояний прецедента* (use case state diagram). На диаграмме состояний прецедента, представленной на рис. 29.2, показана упрощенная версия системных событий для прецедента Оформление продажи приложения розничной торговли. Согласно этой диаграмме, событие makePayment не может произойти до реализации события endSale, переводящего систему в состояние ожидания платежа WaitingForPayment.

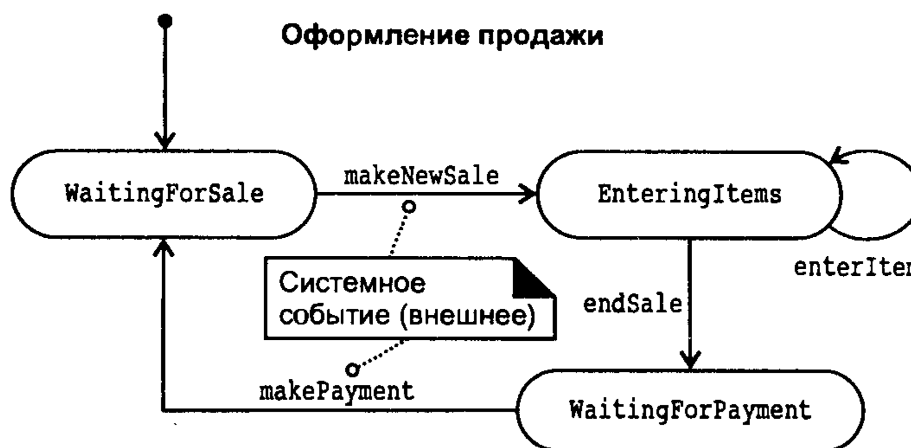



*Рис. 29.2. Диаграмма состояний для прецедента Оформление продажи*

### *Преимущества диаграмм состояний прецедентов*

Большинство системных событий и их допустимый порядок для прецедента Оформление продажи являются относительно тривиальными, поэтому, на первый взгляд, может показаться, что диаграмма состояний для этого прецедента просто не нужна. Однако для сложных прецедентов с большим количеством системных событий (например, при реализации текстового процессора) диаграмма состояний, иллюстрирующая допустимый порядок внешних событий, является очень полезной.

Дело вот в чем. На этапе проектирования и реализации необходимо разработать и реализовать такое проектное решение, которое не допускает выполнения событий в запрещенной последовательности, поскольку это может привести к возникновению ошибки. Например, в POS-системе нельзя обрабатывать платежи до завершения текущей продажи. С учетом этого должен быть написан и программный код приложения.

При наличии диаграмм состояний прецедентов разработчик может методологически грамотно спроектировать систему с учетом корректности порядка выполнения событий. Для этого можно использовать следующие проектные решения.

- Жестко запрограммированная проверка условий для внештатных событий.
- Использование шаблона State (Состояние) (описывается в следующей главе).
- Запрещение использования управляющих элементов, которые могут привести к возникновению внештатных событий, в активных окнах интерфейса (наиболее желательный подход).
- Интерпретатор состояний, запускающий таблицу состояний, соответствующую диаграмме состояний прецедента.

В системах с большим количеством системных событий краткость и завершенность диаграмм состояний прецедентов помогают разработчику охватить все возможные события.

## 29.5. Диаграммы состояний прецедентов для POS-приложения

### *Прецедент Оформление продажи*

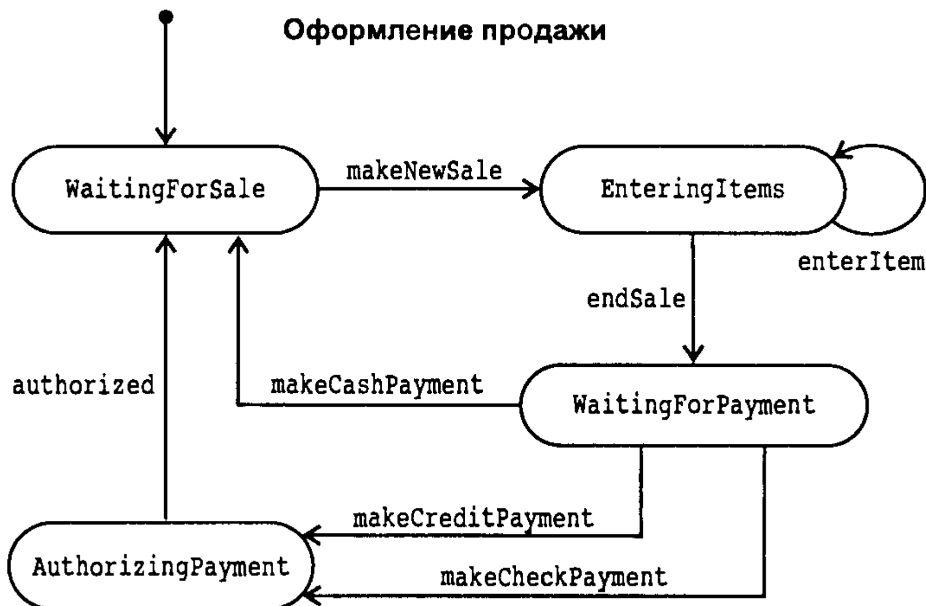



*Рис. 29.3. Пример диаграммы состояний*

## 29.6. Для каких классов необходимы диаграммы состояний

Помимо диаграмм состояний для прецедентов и всей системы в целом, можно создать свою диаграмму состояний практически для любого типа или класса.

### *Объекты, независимые и зависимые от состояний*

Если объект всегда одинаково реагирует на событие, то он считается *независимым от состояния* (state-independent object) (или безрежимным) по отношению к этому событию. Например, если в ответ на получение сообщения всегда вызывается один и тот же метод объекта, в теле которого не содержится операторов ветвления, этот объект считается независимым от состояния по отношению к данному событию. Если на все интересующие нас события некоторый объект реагирует одинаково, то он называется *независимым от состояния объектом* (state-independent object). В отличие от него, *зависимые от состояния объекты* (state-dependent object) по-разному реагируют на одно и то же событие в зависимости от своего состояния.

> Диаграммы состояний создаются для зависимых от состояний типов со сложным поведением.

В целом, в экономических информационных системах содержится очень мало интересных классов, зависимых от состояния, а системы управления процессами и системы телекоммуникаций зачастую содержат множество зависимых от состояния объектов.

### *Зависимые от состояния стандартные классы*

Перечислим стандартные объекты, которые обычно являются зависимыми от состояния и для которых полезно создавать диаграммы состояний.

- **Прецеденты (процессы).**
  - Прецедент `Оформление продажи` как класс по-разному реагирует на событие `endSale` в зависимости от состояния продажи.
- **Сеансы** (существуют программные объекты, относящиеся к серверной части приложения и представляющие сеанс взаимодействия с клиентом).
  - Еще одним типичным примером является обработка на сервере данных, поступающих от клиента Web-приложения, и реализация логики этого приложения. Например, сервлет Java может выступать в роли контроллера, фиксирующего состояние сеанса связи с Web-клиентом и управляющего передачей новых Web-страниц или модификацией текущей страницы на основе состояния сеанса.
  - Сеанс обычно рассматривается как программный класс, представляющий прецедент. Напомним, что, согласно шаблону GRASP Controller, существуют контроллеры прецедентов, представляющие собой объекты сеанса.
- **Системы** (класс, представляющий все приложение или систему).
  - "POS-система".
- **Окна.**
  - Операция вставки допустима только в том случае, если в буфере обмена содержится некоторый фрагмент.
- **Контроллеры** (согласно шаблону GRASP Controller).
  - Класс `Register`, обрабатывающий системные события `enterItem` и `endSale`.

- **Транзакции** (способ реакции транзакции на системные события зачастую зависит от ее текущего состояния в рамках всего жизненного цикла).
  - Если объект `Sale` получает сообщение `makeLineItem` после реализации события `endSale`, то это сообщение либо игнорируется, либо вызывает сообщение об ошибке.
- **Устройства.**
  - Телевизор и микроволновая печь по-разному реагируют на конкретное событие в зависимости от их текущего состояния.
- **Изменяемые классы** (это классы, изменяющие свою роль).
  - Штатский человек, ставший военным.

## 29.7. Изображение внешних и внутренних событий

### *Типы событий*

Обычно события делятся на следующие типы.

- **Внешние события**. Их также называют системными событиями. Такие события инициируются некоторыми внешними по отношению к системе условиями (например, исполнителем). Внешние события отображаются на диаграммах последовательностей системы. Для особо важных внешних событий приводятся инициируемые ими системные операции.
  - Когда кассир щелкает на кнопке ввода товара на терминале POS-системы, инициируется внешнее событие.
- **Внутренние события**. Такие события инициируются некоторыми внутренними по отношению к системе условиями. В терминах программного обеспечения внутренние события возникают при выполнении операции после отправки сообщения или сигнала от другого внутреннего объекта. Внутренние события отображаются в виде сообщений на диаграммах кооперации.
  - Когда объект `Sale` получает сообщение `makeLineItem`, инициируется внутреннее событие.
- **Временные события**. Такие события возникают в определенный день или в определенное время. В терминах программного обеспечения временные события связаны с часами реального или модельного времени.
  - Например, после выполнения операции `endSale` в течение пяти минут должна выполняться операция `makePayment`, в противном случае текущая продажа автоматически аннулируется.

### *Диаграммы состояний для внутренних событий*

На диаграмме состояний могут отображаться внутренние события, которые обычно представляют сообщения, передаваемые от других объектов. Поскольку сообщения и реакция на них (в терминах других сообщений) также отображаются на диаграммах взаимодействия, возникает вопрос: зачем использовать диаграммы состояний для иллюстрации внутренних событий?. Основной принцип объектно-ориентированного проектирования состоит в использовании объектов,

взаимодействующих между собой посредством передачи сообщений для выполнения поставленных задач. Диаграммы взаимодействия языка UML иллюстрируют этот принцип. Поэтому использование диаграмм состояний для отображения передачи сообщений между объектами выглядит несколько нелогичным[1].

Поэтому автор воздерживается от рекомендаций по использованию диаграмм состояний, отражающих внутренние события, для реализации задач объектно-ориентированного проектирования[2]. Однако они могут оказаться полезными при подведении итогов проектирования после его завершения.

В то же время, как уже упоминалось при описании диаграмм состояния прецедентов, диаграммы состояния для внешних событий могут оказаться полезным средством.

> Старайтесь использовать диаграммы состояний для иллюстрации внешних и временных событий, а также реакции на них, а не для проектирования поведения объектов на основе внутренних событий.

## 29.8. Дополнительные обозначения для диаграмм состояний

Язык UML содержит множество обозначений для изображения элементов диаграмм состояний, которые не были описаны в нашем начальном экскурсе. Назовем лишь три основные.

- Действие, связанное с переходом
- Условия перехода
- Вложенные состояния

### *Действия и условия, связанные с переходами*

Переход из одного состояния в другое может инициировать некоторое действие. В программной реализации это может быть вызов метода класса, для которого строится диаграмма состояний.

Переход может осуществляться с учетом некоторых условий (например, логических) (рис. 29.4).

### *Вложенные состояния*

Состояние объекта может включать в себя некоторые производные состояния (подсостояния), которые наследуют переходы состояний более высокого

---

[1] Читатели, знакомые с литературой по объектно-ориентированному анализу и проектированию, наверняка встречали примеры комплексных диаграмм состояний, посвященных внутренним событиям и реакции объектов на них. По существу, создатели таких диаграмм подменяют парадигму взаимодействия объектов посредством передачи сообщений парадигмой представления объектов как конечных автоматов и используют для проектирования поведения объектов не диаграммы кооперации, а диаграммы состояний. Теоретически обе точки зрения являются эквивалентными.

[2] Одним из обоснованных примеров использования диаграмм состояний для отображения структуры объекта на основе внутренних событий являются системы с автоматической генерацией кода на основе диаграмм состояний или системы, для запуска которых используется интерпретатор поведения конечных автоматов.

уровня абстракции (суперсостояний). Подсостояния графически изображаются в виде прямоугольников со скругленными углами, вложенных в блок основного состояния (рис. 29.5).

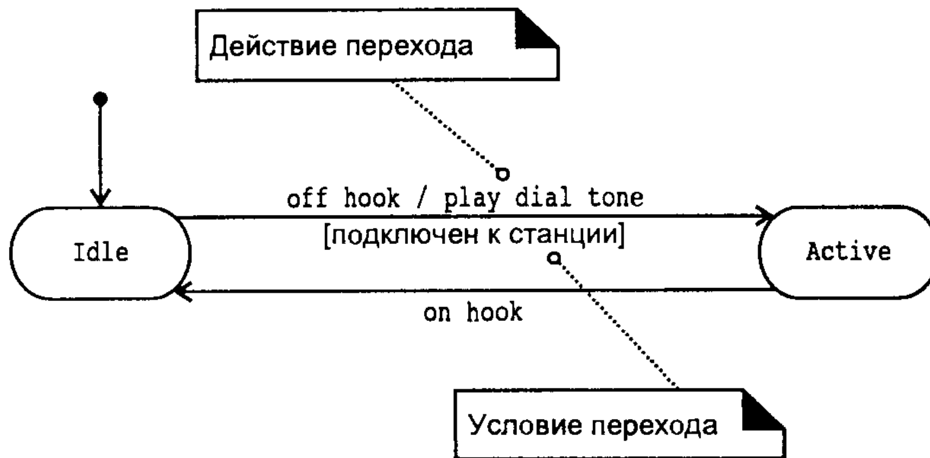



*Рис. 29.4. Обозначения для действий и условий, связанных с переходами*

Например, при переходе в состояние активности Active на самом деле осуществляется переход в подсостояние гудка PlayingDialTone. Независимо от того, в каком из подсостояний находится объект, при реализации события опускания трубки на рычаг on hook, связанного с состоянием активности, выполняется переход в состояние ожидания Idle.

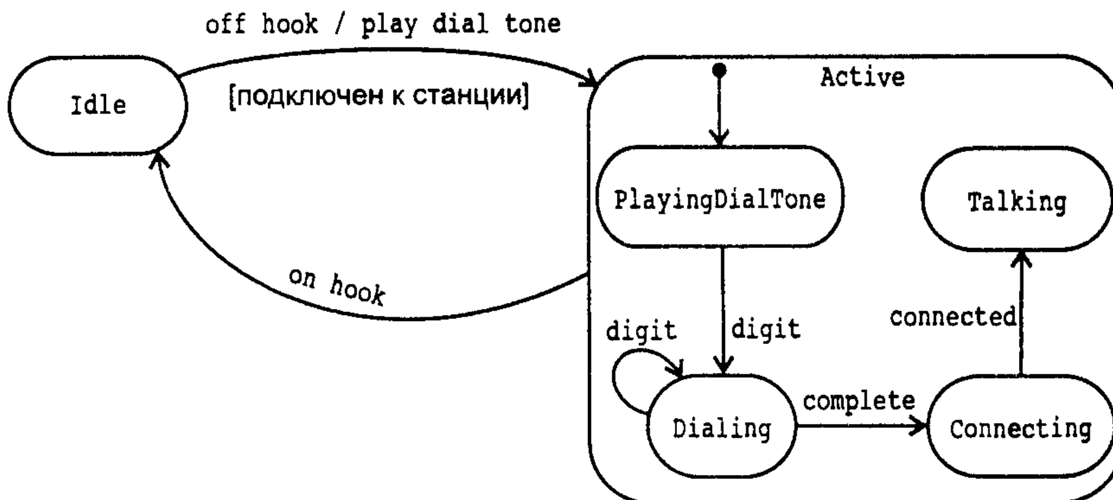



*Рис. 29.5. Вложенные состояния*

## 29.9. Дополнительная литература

Применение моделей состояний в процессе ООА/П хорошо описано в книге Кука и Дэниелса [26]. В одной из работ Дугласа (Douglass) тоже очень хорошо описывается моделирование на основе состояний. Основное внимание в книге уделяется системам реального времени, однако описанные подходы применимы для широкого класса приложений.

Глава

# Проектирование систем на основе шаблонов

**Основные задачи**

- **Разработать логическую архитектуру системы в терминах уровней и разделов на основе шаблона Layers.**
- **Проиллюстрировать разработанную архитектуру на языке UML с использованием диаграмм пакетов.**
- **Применить шаблоны Facade, Observer и Controller.**

## Введение

Для ясности сразу же оговоримся, что в этой главе речь пойдет о *введении* в логическую архитектуру, поскольку эта область очень широка.

В рассмотренных ранее итерациях основное внимание уделялось объектам предметной области (таким, например, как `Sale` и `Payment`), поскольку они определяют основу поведения системы. Интерфейсу пользователя и доступу к ресурсам (в частности, к базе данных) внимание до сих пор не уделялось. Это было сделано для простоты, чтобы иметь возможность сконцентрироваться на основных принципах объектного проектирования.

Однако система состоит из множества логических пакетов, таких как пакет интерфейса пользователя, пакет доступа к базе данных и т.д. Объекты каждого пакета выполняют однотипные обязанности (например, обеспечивают доступ к базе данных). На этом строится модульный принцип проектирования систем.

В этой главе кратко рассматривается архитектура системы, а также затрагиваются вопросы взаимодействия пакетов.

## 30.1. Архитектура программной системы

Приведем одно из определений архитектуры программной системы.

> "Архитектура — это набор важных решений, касающихся организации программной системы, выбора структурных элементов и их интерфей-

сов, затрагивающих поведение и взаимодействие этих элементов, их группировку в более крупные подсистемы и архитектурный стиль приложения." [21]

Независимо от определения (а их существует множество), архитектура системы включает крупномасштабные элементы — основные идеи организации, шаблоны, способ распределения обязанностей, взаимодействие и мотивацию поведения системы и ее основных подсистем.

Архитектура включает организацию и структуру основных элементов системы. Помимо этого статического определения, архитектура подразумевает поведение, особенно в терминах распределения обязанностей системы и подсистем, а также взаимодействие элементов системы. Архитектура включает мотивацию именно этого проектного решения системы.

Разработка архитектуры — это часть исследования и проектирования системы. При этом обычно говорят об исследовании или проектировании архитектуры.

*Исследование архитектуры* (architectural investigation) подразумевает определение функциональных и (особенно) нефункциональных требований, оказывающих большое влияние на проектное решение системы, ее производительность, стоимость, удобство сопровождения и точки эволюции. В широком смысле это анализ требований, на основе которых вырабатывается проектное решение.

*Проектирование архитектуры* (architectural design) — это выражение этих требований в проектном решении, аппаратных средствах, операциях, политиках и сетевой конфигурации.

В рамках UP исследование и проектирование архитектуры называется архитектурным анализом (architectural analysis), который более подробно описан в главе 32.

## Архитектурные представления в унифицированном процессе

Архитектура системы — понятие многомерное. Рассмотрим следующие примеры.

- Логическая архитектура описывает систему в терминах ее принципиальной организации в виде уровней, пакетов, основных контуров, классов, интерфейсов и подсистем.
- Архитектура развертывания описывает систему в терминах выполнения задач системы различными обрабатывающими элементами и различной сетевой конфигурации.

В UP выделяют шесть представлений архитектуры (логическое, представление развертывания и т.д.), которые будут описаны в главе 32.

> В этой главе основное внимание уделяется логическому представлению архитектуры.

## Архитектурные шаблоны и категории шаблонов

Хорошие принципы проектирования архитектуры, особенно логической, нашли свое отражение в архитектурных шаблонах. К ним относится, например шаблон Layers (Уровни). Архитектурные шаблоны впервые были описаны в книге [15].

В этой книге приводится простая и полезная классификация шаблонов.

1. **Архитектурные шаблоны** связаны с крупномасштабными проектными решениями. Обычно они применяются на ранних итерациях разработки (на стадии развития), когда формируются основные структурные элементы системы и связи между ними.
   - Примером является шаблон Layers, предполагающий разделение системы на несколько основных уровней.
2. **Шаблоны проектирования** связаны с небольшими проектными решениями на уровне объектов и контуров. Они применяются для разработки решений по связыванию крупномасштабных элементов, определенных в рамках архитектурных шаблонов, а также для детализации проектного решения. Их называют также микро-архитектурными шаблонами.
   - Примером служит шаблон Facade, который можно использовать для создания интерфейса между соседними уровнями.
   - Шаблон Strategy, обеспечивающий возможность подключения алгоритмов.
3. **Идиомы** — низкоуровневые проектные решения, ориентированные на язык программирования или реализацию.
   - Шаблон Singleton, обеспечивающий глобальный доступ к единственному экземпляру класса.

> В этой главе основное внимание уделяется архитектурным шаблонам, а также применению шаблонов проектирования для обеспечения взаимосвязи между крупномасштабными структурными элементами.

Существуют и другие категории шаблонов. Приведенная классификация оказывается полезной во многих случаях, но она не охватывает все существующие типы шаблонов. Рискнем сказать, что шаблон — это набор рекомендаций, который можно многократно применять в определенной области деятельности. К числу других категорий шаблонов относятся следующие:

- шаблоны планирования и организации процесса разработки;
- шаблоны интерфейса пользователя;
- шаблоны тестирования.

## 30.2. Архитектурный шаблон Layers

**Решение.** Основные принципы шаблона Layers [15] сводятся к следующему.

- Организовать крупномасштабные структурные элементы системы в отдельные уровни со взаимосвязанными обязанностями таким образом, чтобы на нижнем уровне располагались низкоуровневые службы и службы общего назначения, а на более высоких уровнях — объекты уровня логики приложения.
- Взаимодействие и связывание уровней происходит сверху вниз. Нужно избегать связывания объектов снизу вверх.

Уровень — это крупномасштабный элемент, зачастую состоящий из нескольких пакетов или подсистем.

Шаблон Layers связан с логической архитектурой. Он описывает базовую организацию элементов проектирования в виде групп, независимых от их физического размещения.

Шаблон Layers определяет общую N-уровневую модель логической архитектуры. На его основе строится *многоуровневая архитектура*. Его принципы настолько часто формулировались в виде шаблонов, что в книге [92] насчитывается порядка сотни его вариаций.

**Возможные проблемы**

- Изменение исходного кода влечет за собой переделку всех элементов системы, поскольку все части системы тесно связаны друг с другом.
- Логика приложения переплетается с интерфейсом пользователя, поэтому в приложении нельзя изменить интерфейс или принципы реализации логики приложения.
- Общие технические службы тесно связаны с бизнес-логикой приложения, поэтому их нельзя использовать повторно, распространить на другие системы или изменить их реализацию.
- Различные аспекты системы тесно связаны друг с другом, поэтому работу сложно разделить между разработчиками.
- Из-за высокой степени связывания сложно модифицировать функции приложения, масштабировать систему или переходить на новые технологии.

**Пример.** Количество и функции уровней зависят от типа приложения и его предметной области (информационная система, операционная система и т.д.). Типичные уровни информационной системы показаны на рис. 30.1.

На основе приведенной здесь информации на рис. 30.2 показана многоуровневая логическая архитектура приложения NextGen.

Диаграммы пакетов предназначены для иллюстрации разбиения системы на уровни. В UML каждый уровень можно представить в виде пакета.

Обратите внимание на отсутствие уровня приложения на данной итерации разработки. Как будет видно дальше, этот уровень нужен не всегда.

Поскольку процесс разработки является итеративным, то на первых итерациях многоуровневая архитектура может быть очень простой, а затем постепенно усложняться на этапе развития. Основной задачей стадии развития является формирование базовой архитектуры (разработанной и реализованной), однако это не означает выполнения фундаментальной работы по проектированию архитектуры перед началом программирования системы. Логическая архитектура разрабатывается на начальных итерациях, а затем совершенствуется на стадии развития.

Обратите внимание на наличие лишь нескольких типов данных на диаграмме пакетов. Это объясняется не только соображениями экономии места, но и спецификой архитектурного представления. В нем отображаются только важные элементы, отражающие основные идеи и архитектурно значимые аспекты. Документ, описывающий архитектурное представление, призван "сказать" читателю: *этот небольшой набор элементов был выбран для реализации великих идей.*

*Рис. 30.1. Типичные уровни логической архитектуры информационной системы*[1]

[1] Ширина пакета пропорциональна степени его применения, однако это не является стандартным обозначением UML. В круглых скобках приводятся другие названия уровня.

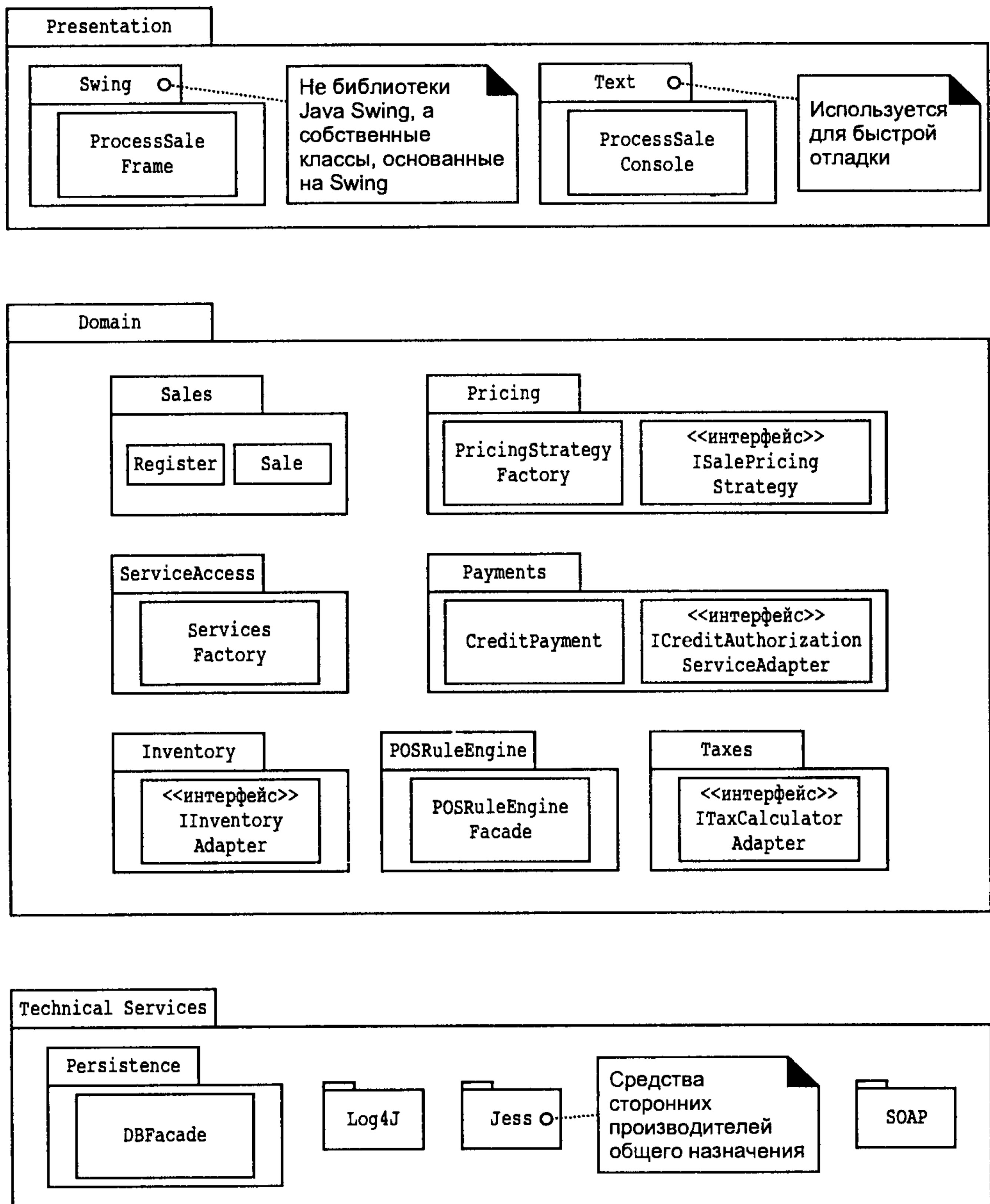



*Рис. 30.2. Фрагмент логического представления уровней приложения NextGen*

К этой диаграмме можно добавить следующие комментарии.

- В пакетах есть и другие типы данных. Здесь показаны лишь наиболее важные из них с точки зрения архитектуры системы.
- Уровень основания не отображен в этом представлении. Архитектор решил, что он не несет интересной информации, хотя разработчикам придется мо-

дифицировать некоторые классы этого уровня, в частности разработать более совершенные утилиты работы со строками.

- Пока не используется отдельный уровень приложения. Обязанности по управлению сеансами выполняет объект Register. Архитектор добавит этот уровень на более поздних итерациях, когда поведение системы усложнится за счет добавления альтернативных интерфейсов пользователя (таких как Web-броузер).

### Связывание уровней и пакетов

В логическое представление можно также включить диаграмму, иллюстрирующую связывание уровней и пакетов. Фрагмент такой диаграммы показан на рис. 30.3.

Расскажем о некоторых обозначениях UML.

- Линии зависимостей можно использовать для обозначения связывания пакетов и типов данных в этих пакетах. Обычные линии зависимостей используются для отображения взаимосвязей общего вида, без указания типа связи (наследования, видимости посредством атрибутов и т.д.).
- Линии связи могут соединять пакеты с другими пакетами или отдельными классами, например пакет Sales с классом POSRuleEngineFacade или пакет Domain с пакетом Log4J. Последнее обозначение используется в том случае, если конкретный зависимый тип данных не играет роли или пакет связан сразу с несколькими типами данных из другого пакета.

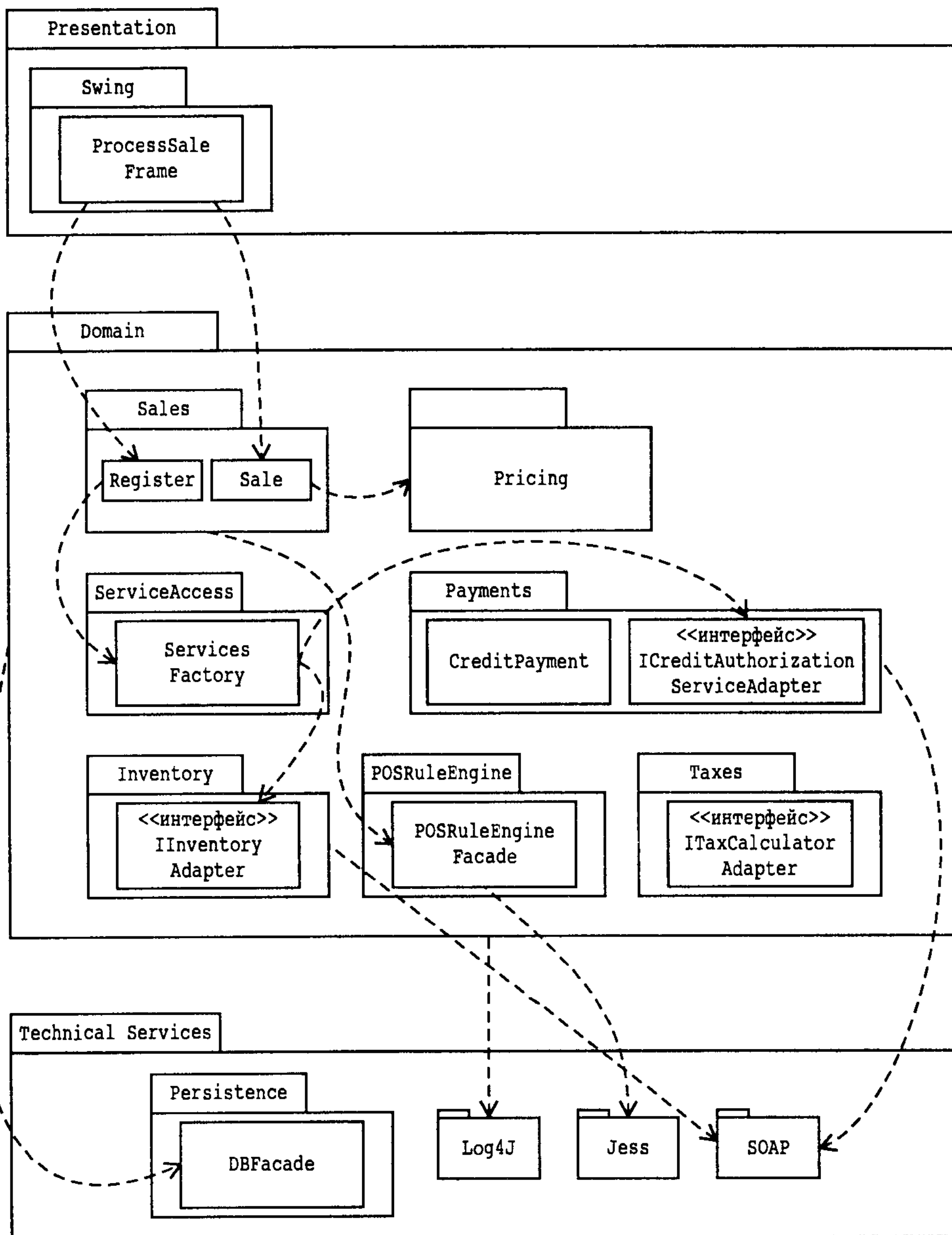



*Рис. 30.3. Фрагмент диаграммы связывания пакетов*

Диаграммы пакетов зачастую используются для сокрытия конкретных типов и концентрации внимания на взаимосвязи более крупных элементов системы (рис. 30.4).

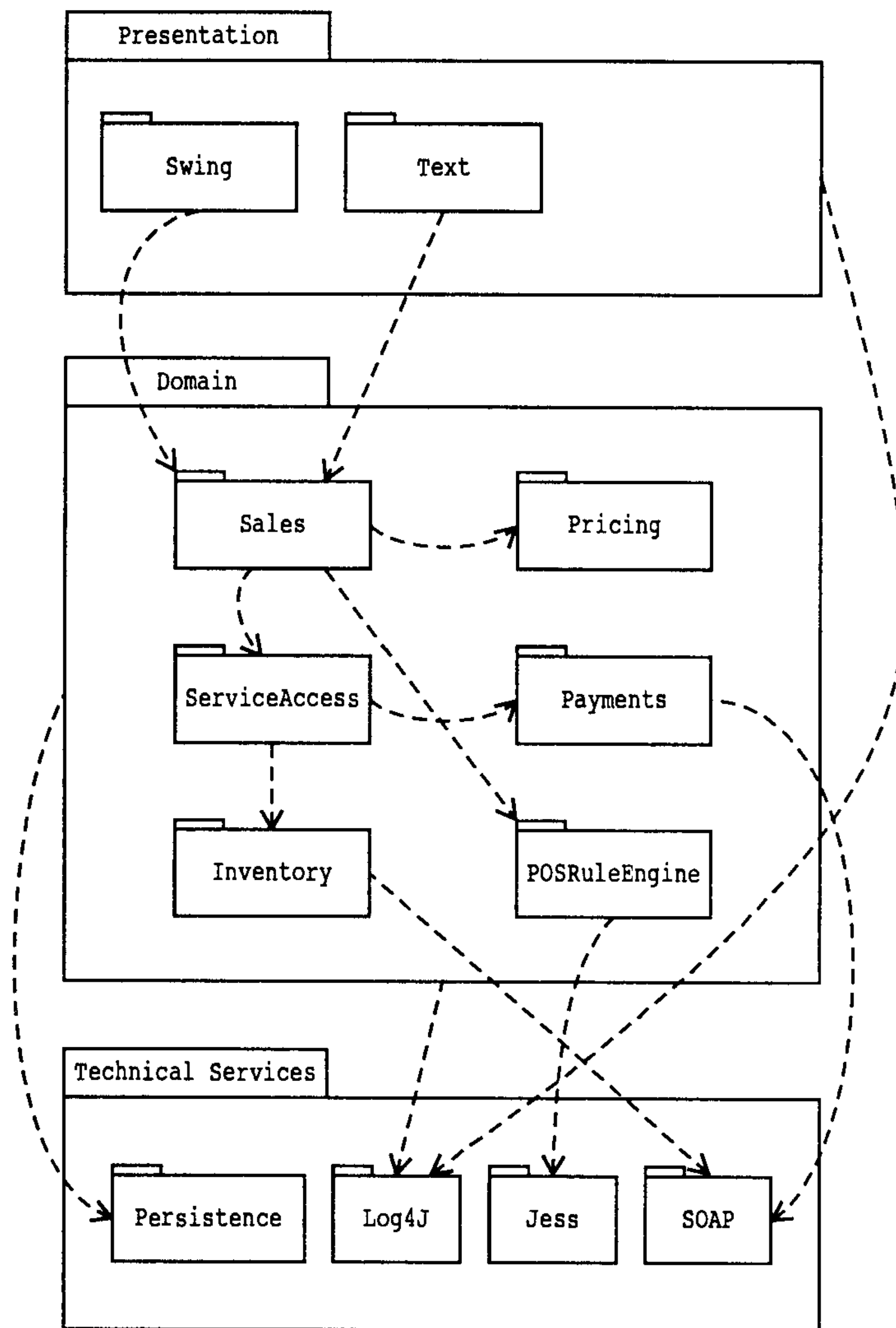



*Рис. 30.4. Фрагмент диаграммы связывания пакетов*

На самом деле на рис 30.4 показан самый типичный вид диаграммы логической архитектуры в UML. На диаграмме пакетов обычно отображают от 5 до 20 пакетов и взаимосвязи между ними.

### Сценарии взаимодействия уровней и пакетов

На диаграммах пакетов отображается статическая информация. Чтобы понять динамику взаимодействия объектов между уровнями и пакетами, используются диаграммы взаимодействия. В стиле "архитектурного представления", скрывающего безынтересные детали, на диаграммах взаимодействия между уровнями и пакетами изображаются *архитектурно значимые сценарии* (architecturally significant scenarios) (в смысле иллюстрации основных крупномасштабных идей проектного решения).

Например, на рис. 30.5 показан фрагмент сценария Оформление продажи, где основное внимание уделяется взаимодействию между уровнями и пакетами.

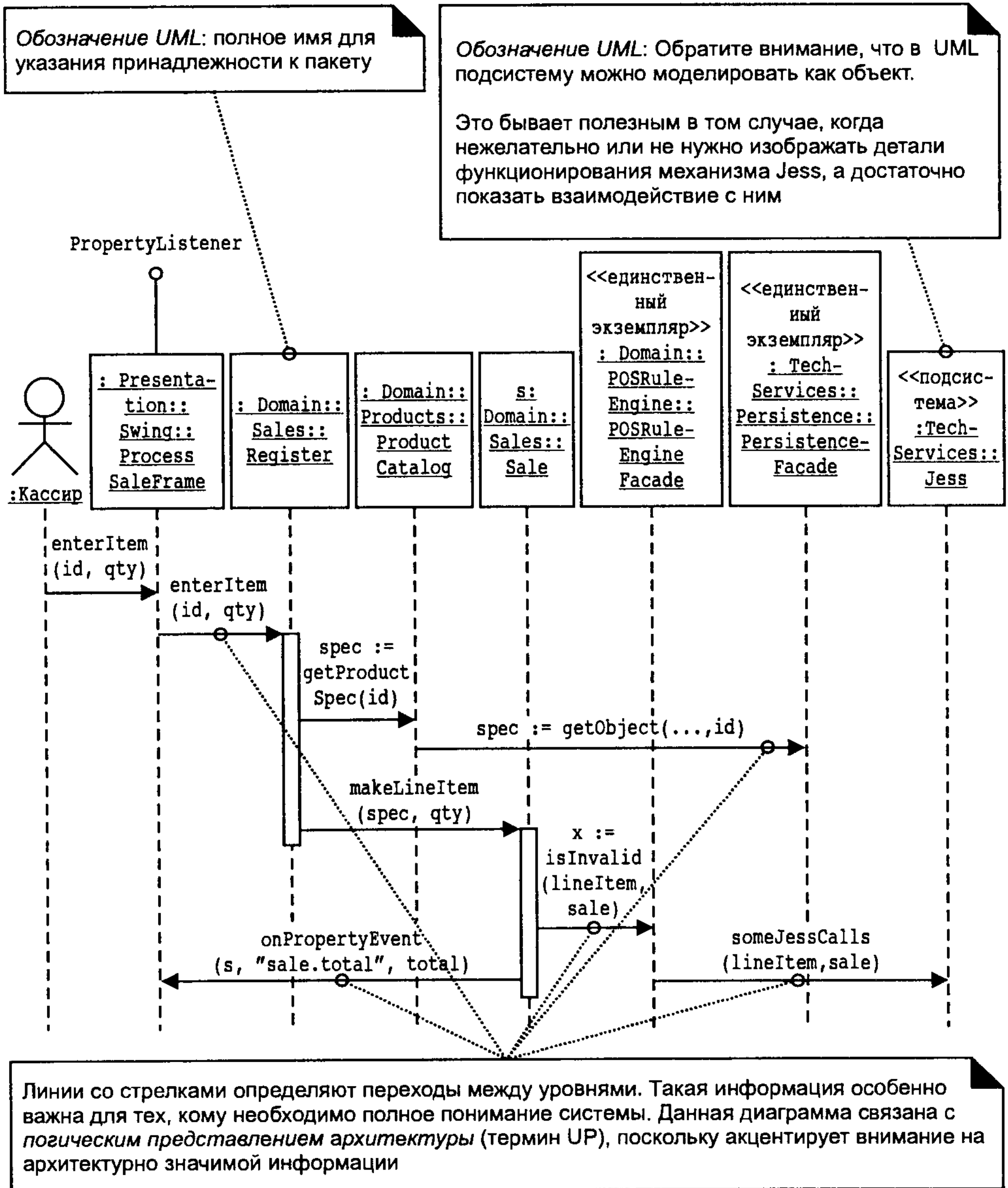



*Рис. 30.5. Архитектурно важная диаграмма взаимодействия*

Здесь использованы следующие обозначения UML.

- Тип в рамках пакета можно определить с помощью выражения *<ИмяПакета>*::*<ИмяТипа>*, например Domain::Sales::Register. Такое обозначение можно использовать для привлечения внимания читателей к взаимосвязи между уровнями в рамках диаграммы взаимодействия.

- Обратите внимание на использование стереотипа <<подсистема>>. В UML подсистемой называют отдельную сущность, обладающую собственным поведением и интерфейсом. Подсистему можно рассматривать как специальный вид пакета или как объект, если нужно отразить взаимодействие между подсистемами (как в данном случае). В UML вся система тоже считается подсистемой (корневой), поэтому ее можно изображать в виде объекта на диаграмме взаимодействия (например, на диаграмме последовательностей).

Заметим, что на этой диаграмме опущены некоторые сообщения, в частности, относящиеся ко взаимодействию с объектом `Sale`. Это сделано для того, чтобы подчеркнуть архитектурно важные взаимодействия.

**Взаимодействия**. При разработке архитектуры нужно принять два важных решения.

1. Какие большие части можно выделить в системе?
2. Как они взаимосвязаны?

Если архитектурный шаблон Layers позволяет выделить большие части системы, то шаблоны проектирования более низкого уровня, такие как Facade, Controller и Observer, призваны определить взаимосвязи между пакетами. В следующем разделе рассматриваются шаблоны, определяющие взаимодействие между уровнями и пакетами.

### Простые пакеты или подсистемы

Некоторые пакеты или уровни представляют собой не просто условные группы объектов, а реальные подсистемы с собственным поведением и интерфейсом. Это видно на следующем примере.

- Пакет `Pricing` не является подсистемой. Это просто набор объектов-фабрик и стратегий, используемых при ценообразовании. Это же можно сказать и о базовых пакетах типа `java.util`.
- Пакеты `Persistence`, `POSRuleEngine` и `Jess` являются подсистемами. Это отдельные сущности с четко выделенными обязанностями.

В UML подсистему можно изобразить с помощью стереотипа, как на рис. 30.6.

### Шаблон Facade

Для доступа к пакетам, представляющим подсистему, чаще всего используется шаблон Facade из группы шаблонов GoF. То есть для доступа к подсистеме определяется открытый фасадный объект, с которым взаимодействуют клиентские объекты. Такими фасадными объектами для доступа к подсистеме правил и базе данных являются объекты `POSRuleEngineFacade` и `PersistenceFacade`.

За "фасадом" обычно скрыто большое количество низкоуровневых операций. Открытыми являются лишь несколько высокоуровневых операций. Если низкоуровневые операции будут открыты, то фасадный объект не будет обладать высоким зацеплением. Более того, если фасадный объект будет распределенным или удаленным (как компонент сеанса EJB или серверный объект RMI), то при удаленном взаимодействии с его службами возникнут проблемы с производительностью, поскольку большое количество удаленных обращений снижает производительность распределенной системы.

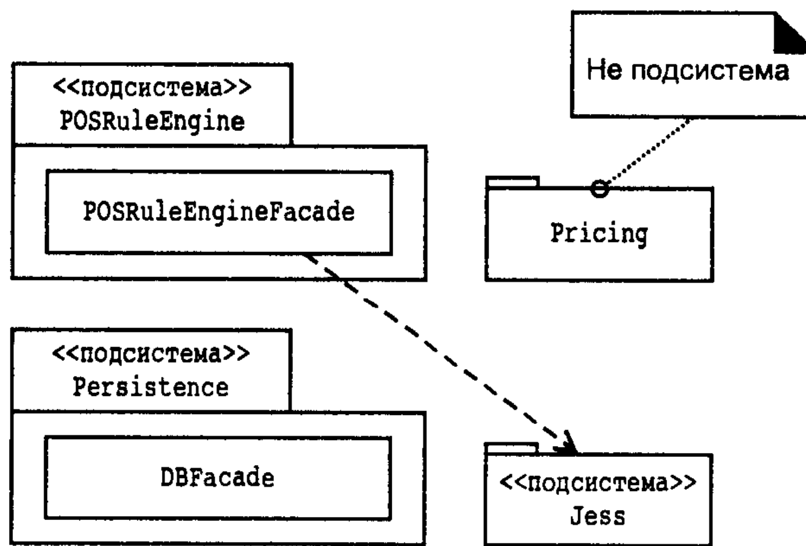



*Рис. 30.6. Стереотипы для подсистемы*

Фасадный объект обычно не выполняет отдельных функций. Он лишь консолидирует и распределяет запросы к конкретным объектам подсистем, которые и обеспечивают нужную функциональность.

Например, объект `POSRuleEngineFacade` обеспечивает оболочку и общую точку входа для подсистемы правил POS-приложения. Реализация этих подсистем скрыта от других пакетов за "фасадным объектом". Допустим (это лишь одна из множества реализаций), подсистема правил реализована на основе механизма Jess. Jess — это подсистема, реализующая множество мелких операций (это довольно типично для общих внешних подсистем). Однако интерфейс объекта `POSRuleEngineFacade` не отражает низкоуровневые операции Jess. Он обеспечивает выполнение лишь нескольких высокоуровневых операций, таких как `isInvalid(lineItem, sale)`.

Если в приложении выполняется лишь небольшое число системных операций, то с верхним уровнем обычно взаимодействует лишь один объект уровня логики приложения. В то же время на уровне технических служб, включающем несколько подсистем, обычно содержится хотя бы один фасадный объект для связи каждой подсистемы с более высокими уровнями (или несколько открытых объектов, если не используется шаблон Facade) (рис. 30.7).

### Фасадные объекты сеансов и уровень приложения

Если приложение включает множество системных операций и поддерживает большое число прецедентов, в нем обычно существует несколько объектов, обеспечивающих связь уровней интерфейса и объектов предметной области (в отличие от ситуации, представленной на рис. 30.7).

В текущей версии системы NextGen предусмотрен единственный объект `Register`, выполняющий функции фасадного объекта для уровня предметной области (согласно шаблону GRASP Controller).

Однако с ростом системы, реализацией новых прецедентов и системных операций зачастую вводится новый уровень объектов (называемый уровнем приложения), поддерживающих состояние сеанса для операций прецедентов, в котором каждый экземпляр сеанса представляет сеанс работы с одним клиентом. Такие объекты называются *фасадными объектами сеансов*. Они строятся в соответствии с шаблоном Controller. На рис. 30.8 показан пример изменения архи-

тектуры приложения NextGen в результате создания уровня приложения и фасадных объектов сеанса.

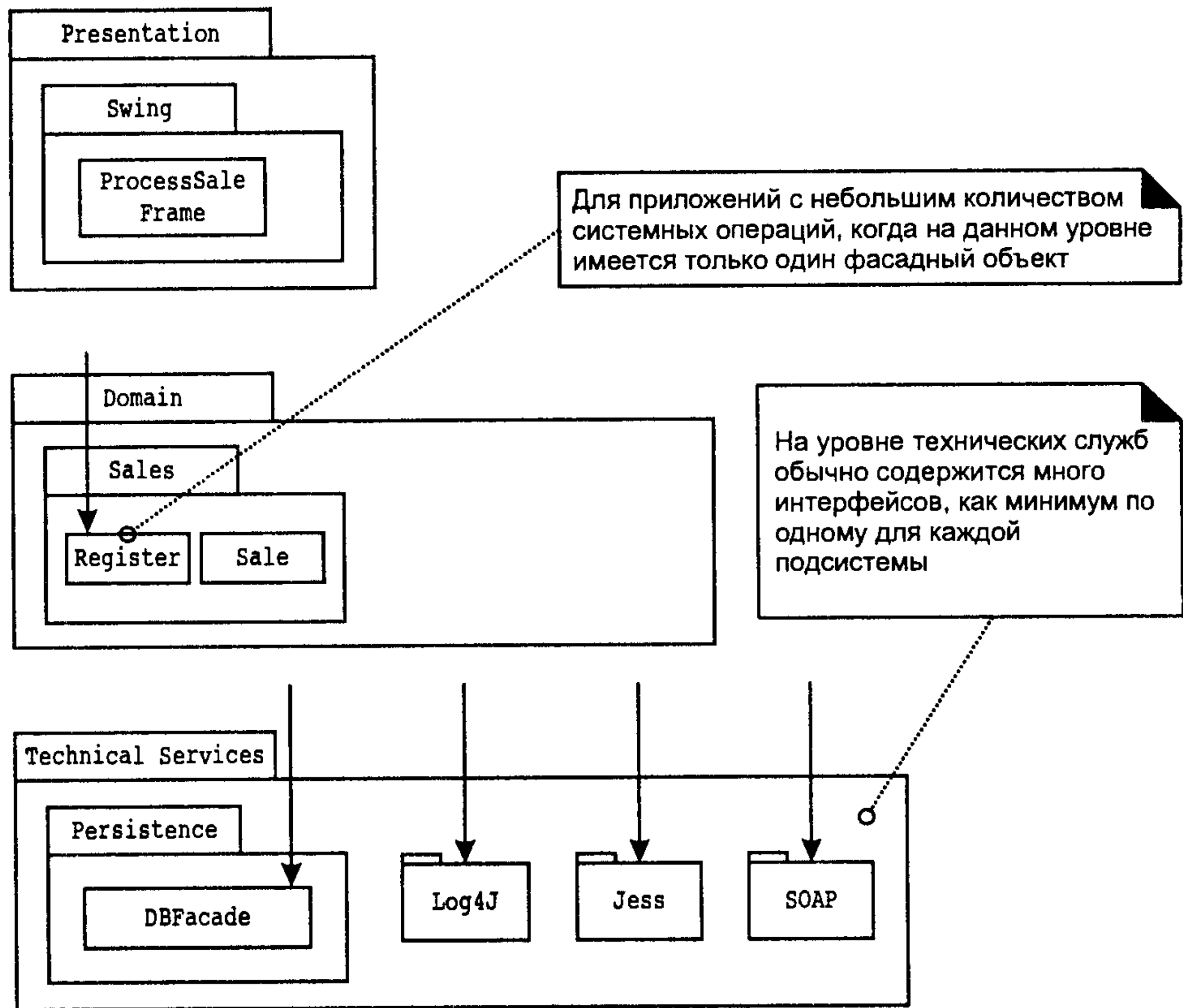



*Рис. 30.7. Интерфейсы связи с более высокими уровнями*

## Шаблон Controller

Шаблон Controller описывает типичное проектное решение (объекты-контроллеры) для обработки в клиентской части приложения запросов системных операций, поступающих от уровня интерфейса пользователя. Реализация шаблона Controller представлена на рис 30.9.

Presentation

Swing

...

ProcessSale Frame

ProcessRental Frame

Application

ProcessSale SessionFacade

ProcessRental SessionFacade

HandleReturns SessionFacade

Domain

Sales

Register

Sale

Rentals

Rental

...

Фасадные объекты сеансов приложения, которые поддерживают состояние сеанса и управляют потоком событий, связанных с некоторыми видами деятельности в рамках прецедента

*Рис. 30.8. Фасадные объекты сеансов и уровень приложения*

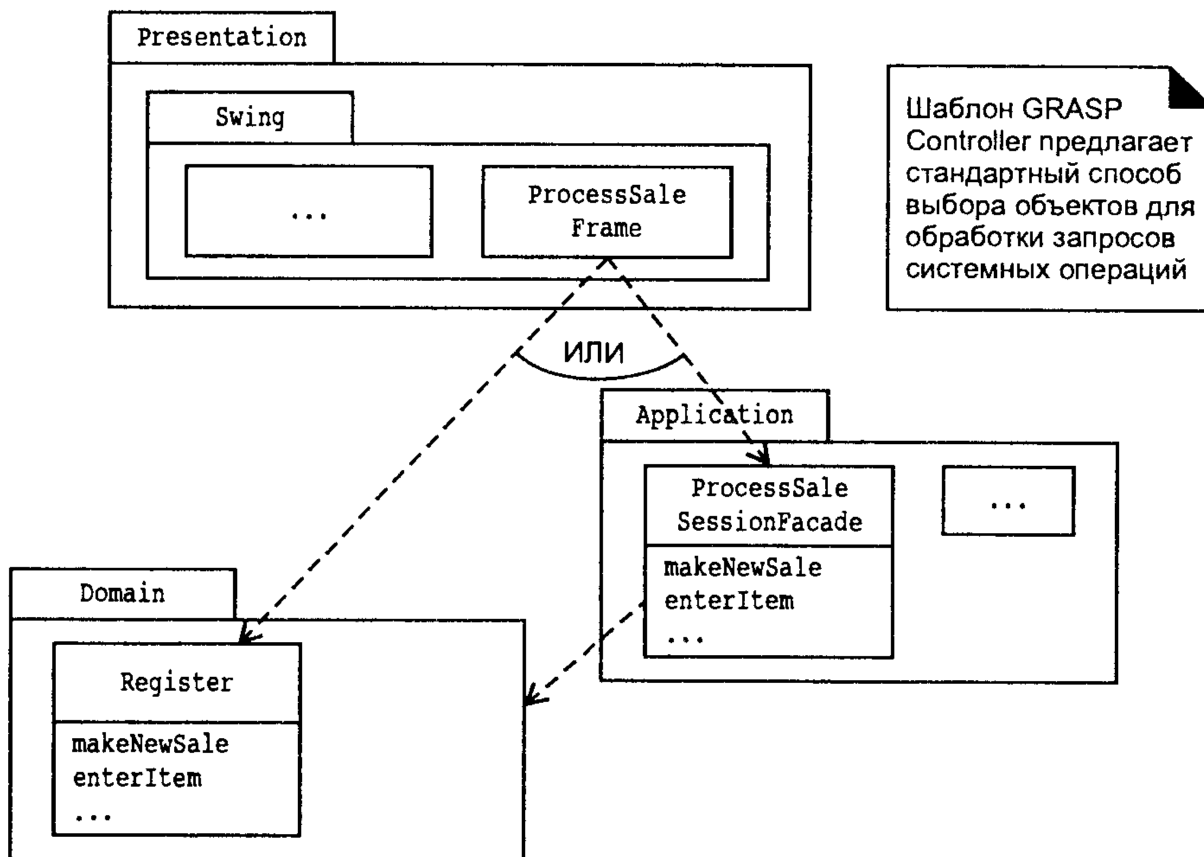



*Рис. 30.9. Шаблон Controller*

### Системные операции и уровни

На диаграммах последовательностей изображают системные операции, а не представления объектов. На рис. 30.10 показаны системные операции, представляющие собой запросы исполнителей, передаваемые с уровня интерфейса пользователя на уровень приложения или предметной области.

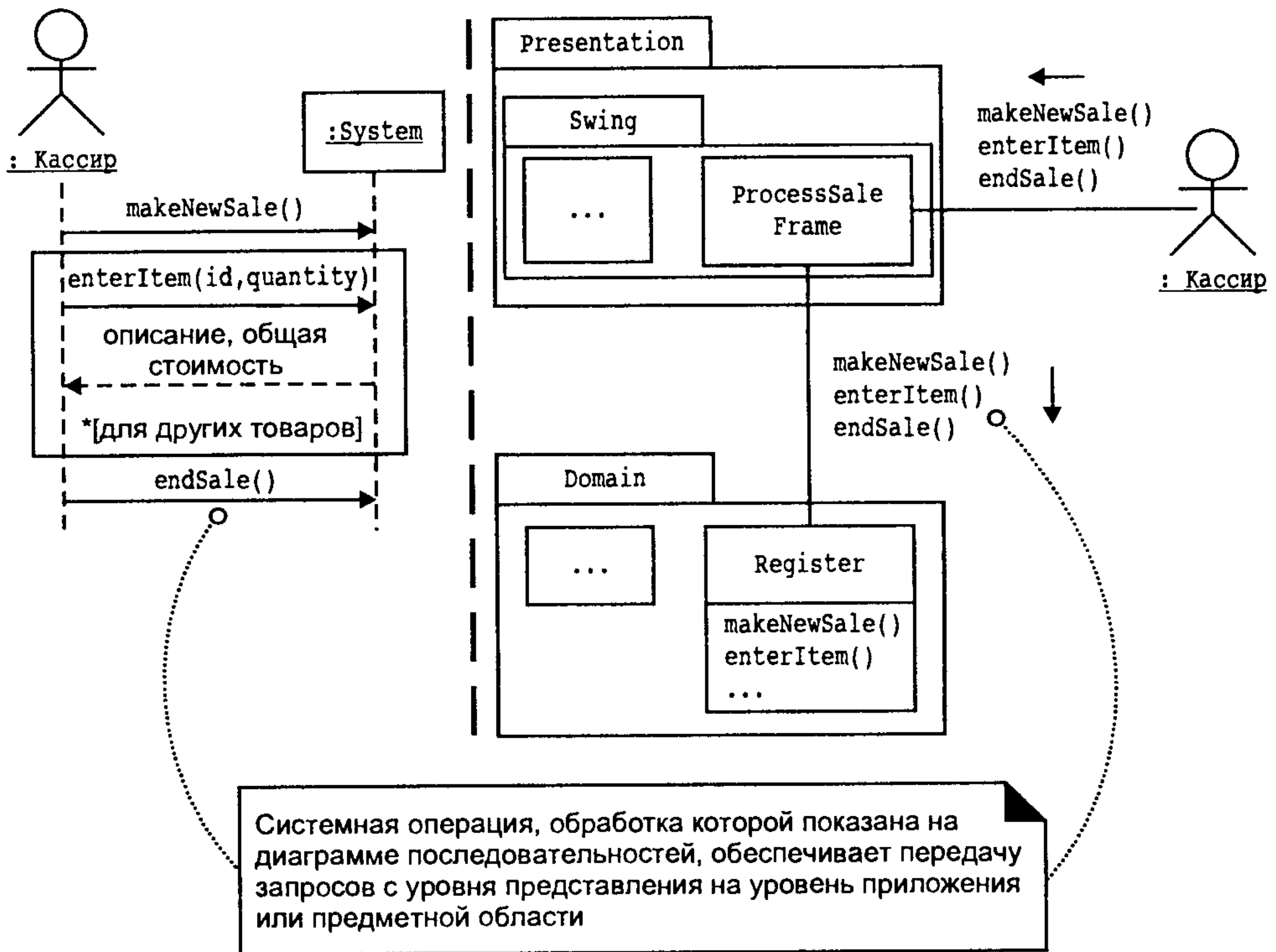



*Рис. 30.10. Системные операции на диаграмме последовательностей в терминах уровней системы*

### Взаимодействие с верхними уровнями на основе шаблона Observer

Шаблон Facade зачастую используется для взаимосвязи вышестоящих уровней с нижними уровнями или для доступа к службам и другим подсистемам того же уровня. Если же нижестоящим уровням приложения или предметной области необходимо взаимодействовать с вышестоящим уровнем интерфейса пользователя, то обычно используют шаблон Observer. При этом объекты уровня представления реализуют интерфейс, наподобие `PropertyListener` или `AlarmListener`. Эти объекты являются подписчиками или слушателями для событий, инициируемых объектами нижних уровней. Объекты нижних уровней напрямую передают сообщения объектам более высокого уровня интерфейса пользователя, но связываются при этом только с объектами, реализующими интерфейс (например, `PropertyListener`), а не с конкретными окнами GUI.

Эта ситуация рассматривалась при описании шаблона Observer. Идея таких взаимоотношений показана на рис. 30.11.

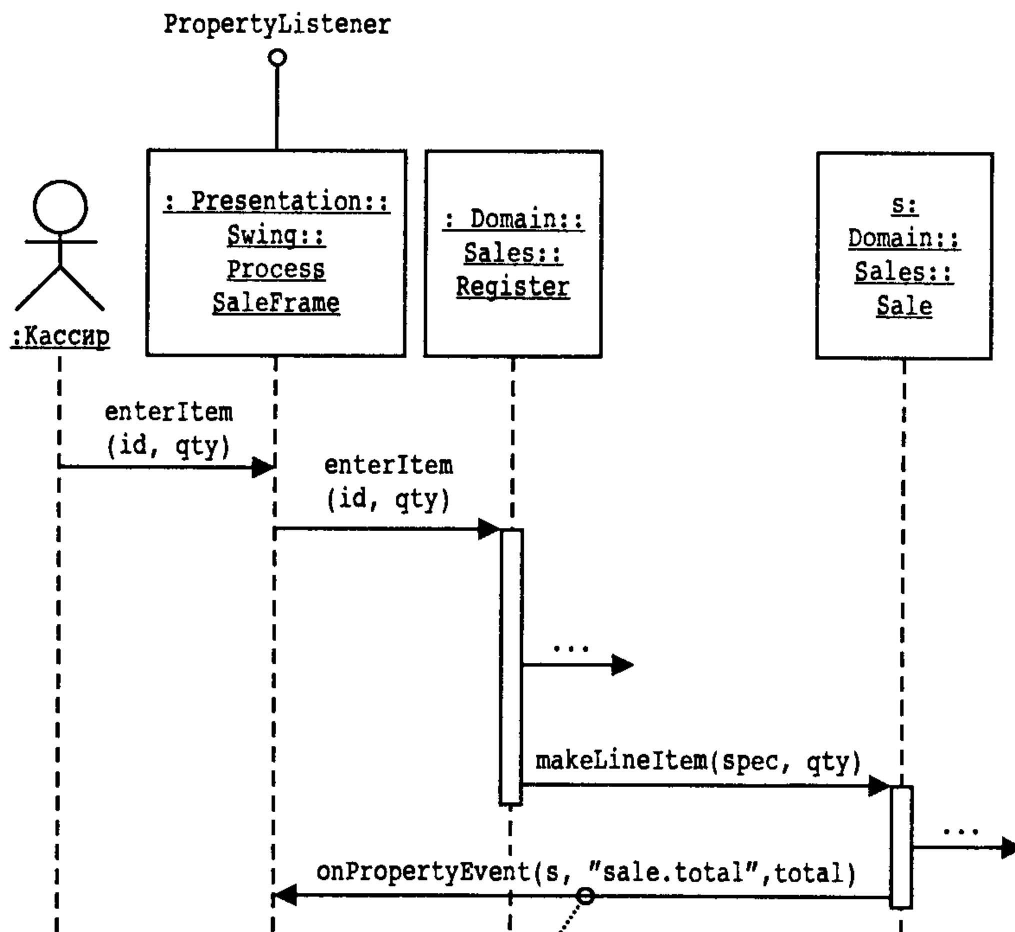



*Рис. 30.11. Использование шаблона Observer для взаимодействия с уровнем интерфейса пользователя*

### Связь уровней

В большинстве многоуровневых приложений уровни связаны между собой не столь жестко, как в семиуровневой модели сетевых протоколов OSI. В модели протоколов существует строгое ограничение, предполагающее связь элементов уровня N лишь со службами нижестоящего уровня N–1.

Такой принцип очень редко используется в архитектуре информационных систем. Здесь чаще применяется "прозрачная многоуровневая" архитектура [15], в которой элементы каждого уровня связаны с несколькими другими уровнями.

Рассмотрим связь уровней более подробно.

- Все вышестоящие уровни зависят от технических служб и базового уровня.
  - Например, в Java все уровни зависят от элементов пакета `java.util`.
- Объекты уровня предметной области зависят от уровня экономической инфраструктуры.

- Объекты уровня представления обращаются к уровню приложения, который, в свою очередь, обращается к уровню предметной области. При наличии уровня приложения объекты уровня интерфейса не взаимодействуют напрямую с уровнем предметной области.
- В однопоточных "настольных" приложениях программные объекты уровня предметной области "видны" или передаются уровням представления, приложения и, несколько реже, объектам технических служб.
  - Допустим, к этому типу относится приложение NextGen. Тогда объекты `Sale` и `Payment` могут быть "видимы" для объектов уровня GUI и передаваться в подсистему взаимодействия с базой данных уровня технических служб.
- В распределенных системах объекты уровня предметной области обычно реплицируются (как *объекты значений*) и передаются на уровень представления. В этом случае уровень предметной области размещается на сервере, а клиентские узлы получают копии данных с сервера.

***Не опасно ли связывание с базовым уровнем и уровнем технических служб?***

Как указывалось при описании шаблонов GRASP Protected Variations и Low Coupling, проблемой является не само связывание как таковое, а излишнее связывание с неустойчивыми объектами в точках вариации и эволюции. Вряд ли стоит тратить время и деньги на попытки абстрагироваться от неизменяемых элементов, или элементов, изменение которых не повлечет за собой никаких последствий. Например, при создании Java-приложения не имеет смысла скрывать доступ к библиотекам Java. Высокая степень связывания с библиотеками Java не составляет проблемы, поскольку эти библиотеки достаточно устойчивы.

**Обсуждение.** Помимо описанных выше структурных проблем и вопросов взаимодействия, для этого шаблона следует учесть еще некоторые моменты.

### Внешние ресурсы или уровень внешней базы данных

Большинство систем опираются на внешние ресурсы или службы, такие как базы данных Oracle или службу имен и каталогов Novell LDAP. В логическом представлении архитектуры можно использовать не только уровни, но и физические реализации компонентов.

Если ниже базового уровня изобразить внешние ресурсы или конкретную базу данных, то такая диаграмма будет совмещать логическое представление системы с представлением реализации или развертывания.

В терминах логического представления архитектуры доступ к конкретной базе данных отображается как пакет уровня предметной области. При этом для доступа к базе данных можно использовать пакет `Persistence` (Хранилище) раздела технических служб (рис. 30.12).

### Логическое представление архитектуры и представление развертывания

Архитектурные уровни приложения составляют логическое представление архитектуры. Такое представление не отражает обрабатываемые и обрабатывающие элементы. В зависимости от платформы, уровни приложения могут взаимодействовать в рамках одного процесса на одном узле либо распределяться между многими компьютерами и процессами для крупномасштабных Web-приложений.

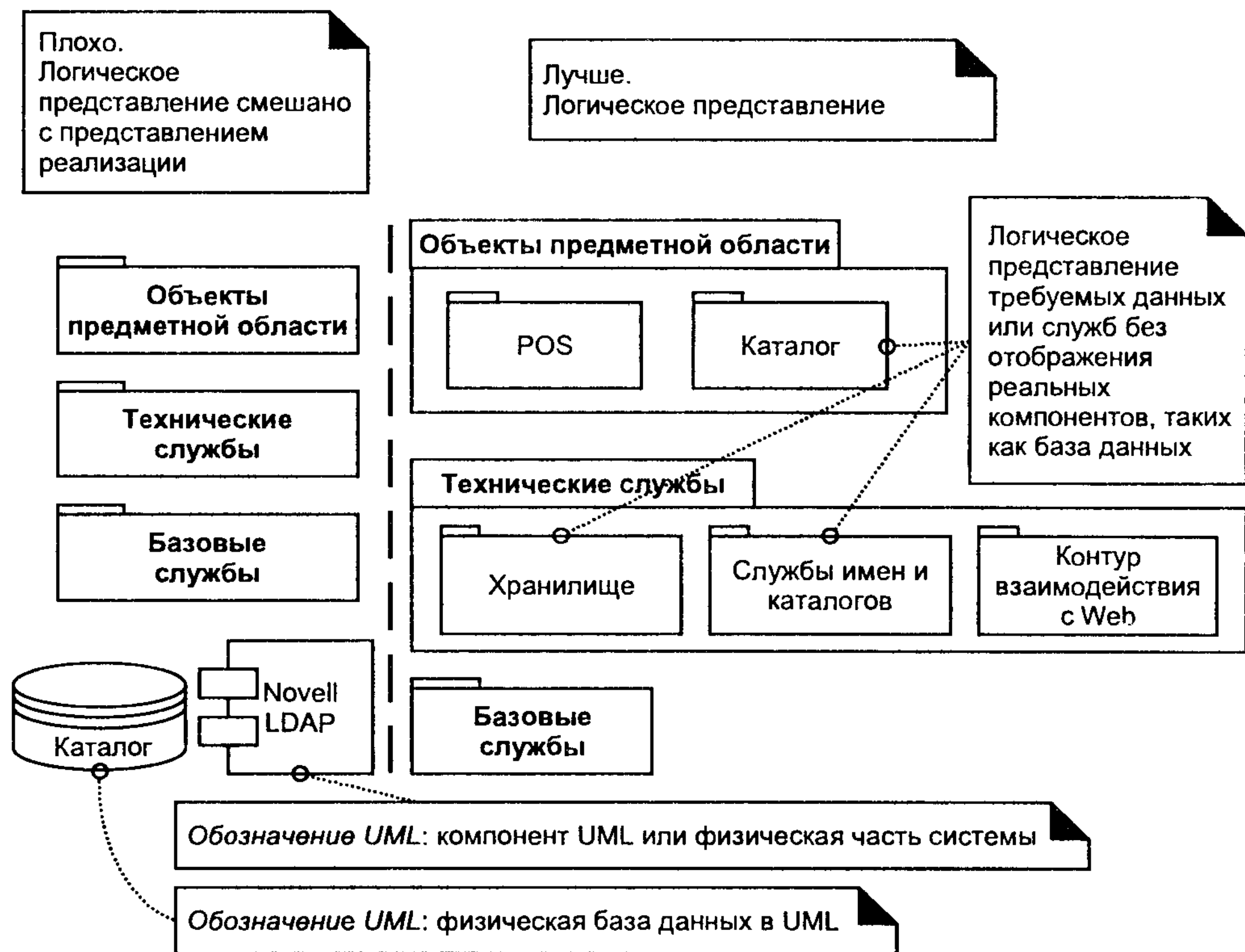



*Рис. 30.12. Комбинация разных представлений архитектуры*

Модель развертывания UP, отражающая логическую архитектуру и ее распределение по разным процессам и узлам, существенно зависит от выбора программной и аппаратной платформы, а также используемых контуров приложений. Например, архитектура развертывания зависит от того, будет использован модуль .NET или J2EE.

Существует множество способов отражения логической архитектуры в модели развертывания. Однако аспекты развертывания выходят за рамки рассматриваемых здесь вопросов, поскольку они являются нетривиальными и зависят от выбора программной платформы, в частности J2EE.

### Дополнительный уровень приложения

Если в системе существует уровень приложения, то он содержит объекты, обеспечивающие управление состоянием сеанса клиентов, взаимодействие с уровнями представления и предметной области, а также управление потоками.

Поток может управлять, например, порядком отображения окон и Web-страниц.

В терминах шаблонов GRASP объекты-контроллеры, например фасадный контроллер прецедента, тоже относятся к этому уровню. В распределенных системах к этому уровню относятся также компоненты, например EJB (и объекты сеансов в целом).

В некоторых приложениях этот уровень не требуется. Он нужен при выполнении одного или нескольких из следующих условий (этот список не является исчерпывающим).

- В системе используется несколько интерфейсов пользователя (например, Web-страницы и интерфейс Swing). Объекты уровня приложения могут выступать в роли адаптеров, коллекционирующих и консолидирующих данные для различных интерфейсов пользователя, а также в роли фасадных объектов, скрывающих доступ к уровню предметной области.
- Уровень приложения необходим в распределенных системах, где объекты уровня предметной области располагаются на одном узле, а интерфейс пользователя — на другом. Он обычно необходим для отслеживания состояния сеанса.
- Объекты уровня предметной области не могут или не должны поддерживать состояние сеанса.
- Существует определенный порядок выполнения приложения, например, заданный порядок отображения окон или Web-страниц.

### Нечеткая принадлежность к различным уровням

Некоторые элементы принадлежат строго к одному уровню. Так, класс `Math` относится к базовому уровню. Однако некоторые элементы сложно отнести к одному конкретному уровню (технических служб или базовому, уровню предметной области или экономической инфраструктуры), поскольку различие между этими уровнями характеризуется качественными показателями, такими как "высокий" — "низкий", "конкретный" — "абстрактный". Эти характеристики относятся к теории нечетких множеств. Это нормальная жизненная ситуация, и строгое разграничение здесь требуется далеко не всегда. Разработчики могут рассматривать уровень технических служб и базовый уровень одновременно, назвав их уровнем инфраструктуры.[2]

Рассмотрим следующие примеры.

- Допустим, Java-приложение разрабатывается на основе открытого контура регистрации Log4J (являющегося частью проекта Jakarta). К какому уровню в этом случае относится подсистема регистрации — к базовому или уровню технических служб? Log4J — это низкоуровневый маленький контур общего назначения. Его можно отнести и к техническим службам, и к базовым объектам.
- Предположим, при разработке Web-приложения используется контур Struts проекта Jakarta. Это относительно высокоуровневый большой специализированный технический контур. Его с большой долей достоверности можно отнести к техническим службам и с меньшей — к базовому уровню.

Однако то, что в одной системе считается высокоуровневой технической службой, в другой может быть базовым элементом.

И наконец, библиотеки для конкретных программных платформ не всегда относятся к низкоуровневым базовым службам. Например, в библиотеках .NET

---

[2] Заметим, что не существует общепринятой системы имен для разных уровней приложения, поэтому имена уровней отражают общепринятые термины.

и J2SE+J2EE содержатся относительно высокоуровневые функции, относящиеся к службам имен и каталогов.

### Терминология: уровни и разделы

Изначально термин "уровень" относился только к логическому представлению архитектуры, а не к физическому узлу. Однако он стал широко использоваться и в значении физического обрабатывающего элемента (узла или кластера узлов), например "уровень клиента" (в смысле клиентского компьютера). В этой книге для ясности автор старался не смешивать эти два понятия, однако в другой литературе по архитектуре приложений можно встретить оба значения этого термина.

Архитектурные *уровни* (layer) представляют деление системы по вертикали, а *разделы* (partition) — по горизонтали на параллельные подсистемы в рамках одного уровня. Например, уровень служб можно разделить на разделы, отвечающие за выполнение требований безопасности и формирование отчетов (рис. 30.13).

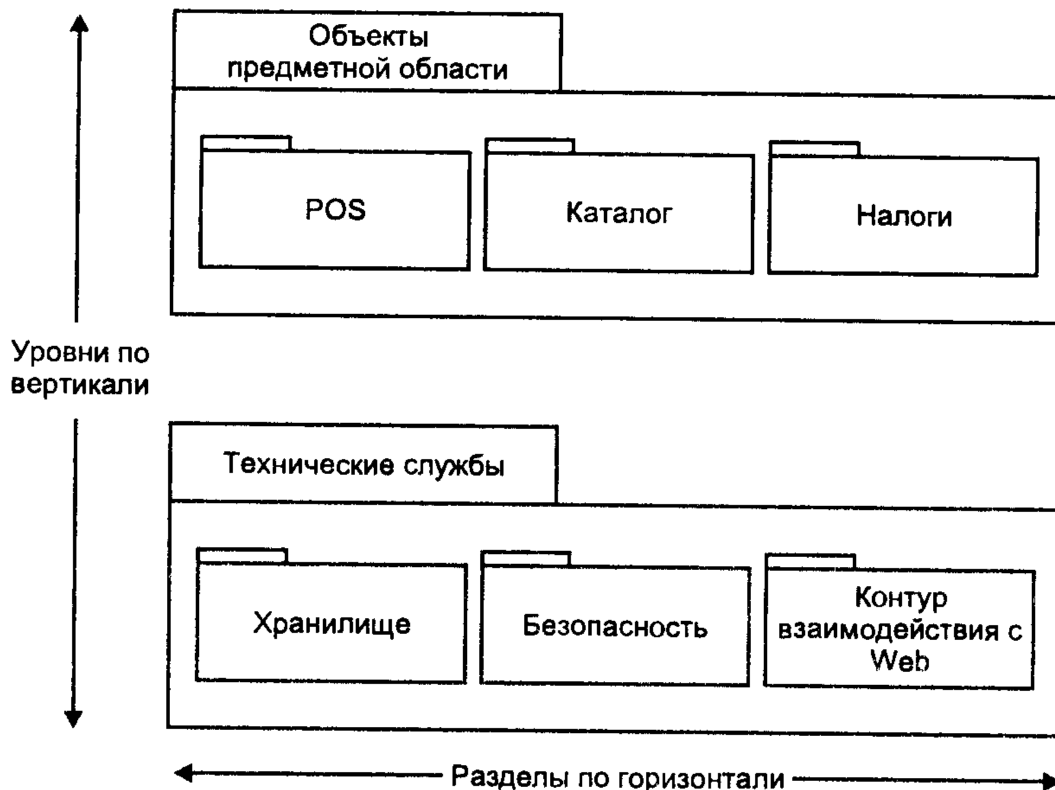



*Рис. 30.13. Уровни и разделы*

#### Ограничения и предупреждения

- В некоторых контекстах добавление новых уровней вызывает проблемы с производительностью. Например, при добавлении уровней для абстракции и перенаправления в ресурсоемкую графическую игру вместо прямого доступа к компонентам графической платы может снизиться производительность приложения.
- Шаблон Layers является одним из нескольких базовых архитектурных шаблонов. Он применим не ко всем приложениям. Например, вместо него можно использовать шаблоны Pipes (Конвейер) и Filters (Фильтры) [15]. Эти шаблоны пригодятся в приложениях, выполняющих последовательные преобразования, например, трансформацию изображений в выбранном порядке. При этом, если на высоком уровне архитектура всего приложения

соответствует шаблону Pipes или Filters, отдельные фильтры или конвейеры могут быть реализованы на базе шаблона Layers.

#### Преимущества

- В целом, этот шаблон обеспечивает отделение разных аспектов, высокоуровневых служб от низкоуровневых, специализированных функций от общих. Это снижает уровень связывания и зависимости в приложении, повышает степень зацепления, увеличивает потенциал повторного использования и вносит дополнительную ясность.
- Сложные элементы подвергаются декомпозиции и подлежат инкапсуляции.
- Некоторые уровни заменяются новыми реализациями. В общем случае это не возможно для низкоуровневых технических служб и базового уровня (в частности, `java.util`), однако вполне реально для уровней интерфейса, приложения и предметной области.
- Нижние уровни содержат повторно используемые функции.
- Некоторые уровни (особенно уровни предметной области и технических служб) могут быть распределенными.
- Логическая сегментация обеспечивает возможность работы над приложением группы разработчиков.

### Реализация шаблона Layers: разработчики и процесс UP

Рекомендуется, чтобы на каждой итерации разработчик отвечал за один уровень или службу.

Это не означает, что в течение одной итерации вся группа разработчиков должна сконцентрироваться на создании одного уровня или службы. Чаще всего реализуется некий вертикальный срез системы. В UP на стадии развития используется следующий подход: на каждой итерации выбирается сценарий и требования, подлежащие реализации. При этом охватывается широкий спектр архитектурно важных пакетов уровней и подсистем. Поэтому на ранних итерациях реализуются и стабилизируются основные архитектурные элементы.

В этой книге при рассмотрении приложения NextGen автор несколько отошел от такого подхода для систематизации изложения. В противном случае ему пришлось бы сразу рассматривать широкий круг вопросов — от программирования GUI до объектно-реляционных преобразований и оптимизации SQL-запросов. В этой книге основное внимание уделяется проектированию объектов уровня предметной области, однако нужно понимать, что в реальной жизни параллельно разрабатываются и другие уровни и подсистемы.

Описанные здесь принципы разработки применимы для проектирования любых уровней.

### Представление реализации: распределение исходного кода по уровням и пакетам

Частью модели реализации UP является организация исходного кода. Для таких языков как Java или C#, обеспечивающих простую поддержку пакетов (пространств имен), логическая архитектура естественным образом отражается в пакетах модели реализации, за исключением использования библиотек сторон-

них производителей.[3] Значительные различия могут наблюдаться лишь на начальных стадиях разработки, когда окончательная структура пакетов еще не сформировалась.

Со временем в результате увеличения объема кода пользователи зачастую отходят от ранней модели, а вместо нее используют результат обратного проектирования — диаграмму пакетов, построенную CASE-средствами на основе исходного кода. Впоследствии автоматически сгенерированная диаграмма пакетов, точно отражающая исходный код, становится основой для логического представления архитектуры.

При использовании Java диаграмма пакетов, показанная на рис. 30.4, преобразуется в следующую модель реализации.

```
//--- УРОВЕНЬ ИНТЕРФЕЙСА
com.foo.nextgen.ui.swing
com.foo.nextgen.ui.text

//--- УРОВЕНЬ ПРЕДМЕТНОЙ ОБЛАСТИ
    // пакеты, специфические для проекта NextGen
com.foo.nextgen.domain.sales
com.foo.nextgen.domain.pricing
com.foo.nextgen.domain.serviceaccess
com.foo.nextgen.domain.posruleengine

    //пакеты, которые можно спроектировать как стандартные
    //универсальные бизнес-службы для многих приложений
com.foo.domain.inventory
com.foo.domain.creditpayment

//--- ТЕХНИЧЕСКИЕ СЛУЖБЫ
    // разработанные нашей группой разработчиков
com.foo.service.persistencelite

    // сторонних производителей
org.apache.log4j
org.apache.soap.rpc
jess

// БАЗОВЫЕ СЛУЖБЫ

    // созданные нашей группой разработчиков
com.foo.util
com.foo.stringutil
```

Обратите внимание, что в именах пакетов без необходимости не упоминается квалификатор конкретного приложения (nextgen). Он участвует только в именах пакетов, связанных с интерфейсом пользователя приложения NextGen.

Для облегчения повторного использования имена элементов должны, по возможности, не зависеть от имени приложения. Например, созданная группой разработчиков библиотека утилит общего назначения для работы со строками размещена

[3] Язык C++ также поддерживает пространства имен, однако использовать десятки и сотни частных пространств имен достаточно неудобно. В Java и C# ситуация совсем другая.

в пакете com.foo.stringutil, а не в com.foo.nextgen.stringutil. Вообще, эту библиотеку можно поместить в общее хранилище исходного кода организации, а не в папки проекта NextGen, чтобы она всегда была на виду.

В качестве другого примера рассмотрим службы доступа к внешним системам авторизации кредитных платежей и хранилищу. Хотя они созданы группой разработчиков в рамках проекта NextGen, это общие службы, доступ к которым может понадобиться из других приложений. Поэтому их поместили в пакет com.foo.domain.creditpayment, а не в com.foo.nextgen.domain.creditpayment.

В то же время, пакет POSRuleEngine тесно связан с проектом NextGen, поэтому ему присвоено имя com.foo.nextgen.domain.posruleengine.

В целом, при возникновении сомнений в полное имя пакета нужно включать имя проекта. Впоследствии его всегда можно изменить.

**Известные применения.** Большинство современных объектно-ориентированных систем (от настольных приложений до распределенных Web-систем) строится на основе шаблона Layers. Трудно найти систему, не основанную на этом шаблоне. Поэтому обратимся к прошлому.

### Виртуальные машины и операционные системы

Начиная с 1960-х годов сформировалась ярко выраженная многоуровневая архитектура операционных систем, в которой нижние уровни инкапсулируют доступ к физическим ресурсам и обработку ввода/вывода, а более высокие уровни обращаются к этим службам. Эти принципы описаны в работах [43] и [44].

До этого, в 50-е годы, была предложена идея виртуальных машин с универсальным машинным языком байт-кодов (например, UNCOL [37]). Предполагалось, что приложения будут разрабатываться на более высоком уровне, чем уровень виртуальной машины (и выполняться без перекомпиляции на различных платформах), который, в свою очередь, является надстройкой над операционной системой и машинными ресурсами. Многоуровневая архитектура виртуальных машин была использована Аланом Кеем (Alan Kay) в его персональной объектно-ориентированной компьютерной системе Flex [70], а позднее (в 1972 году) в виртуальной машине Smalltalk [55], которая стала праобразом более современных виртуальных машин, таких как Java Virtual Machine.

### Информационные системы: классическая трехуровневая архитектура

Типичная архитектура информационных систем, включающая интерфейс пользователя и хранение данных на постоянном носителе, называется *трехуровневой архитектурой* (three-tier architecture) (рис. 30.14). Такая архитектура впервые была описана в 1970-х годах в [102]. Однако этот подход приобрел популярность лишь в середине 90-х годов, возможно, благодаря выходу в свет книги [50], в которой предлагаются решения проблем, связанных с двухуровневой архитектурой.

Поэтому первая публикация, в которой упоминается этот термин, используется довольно редко.

При такой архитектуре приложение делится по вертикали на три уровня.

1. *Уровень интерфейса* (Interface) — окна, отчеты и т.д.

2. *Уровень логики приложения* (Application Logic) — задачи и правила управления процессом.

3. *Уровень данных* (Storage) — механизм постоянного хранения данных.

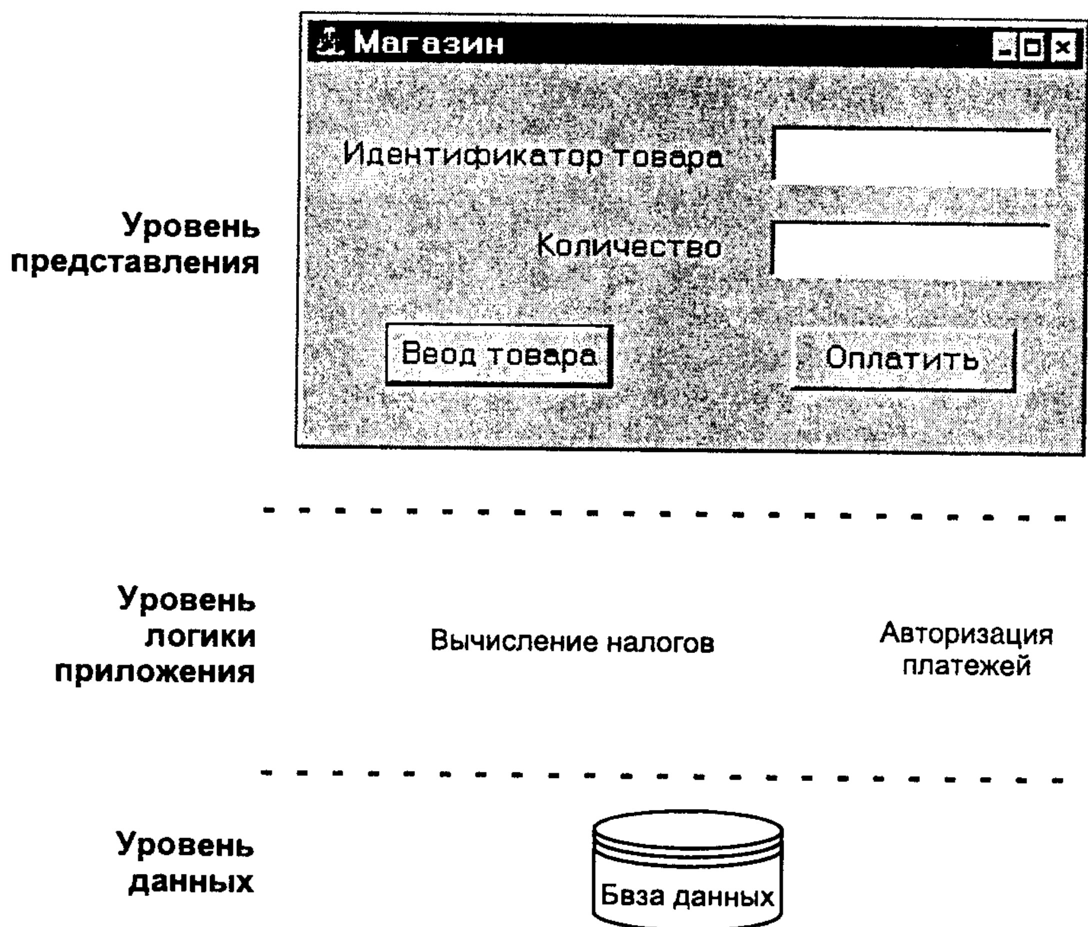



*Рис. 30.14. Классическая схема трехуровневой архитектуры*

Неотъемлемой частью трехуровневой архитектуры является вынесение логики приложения на отдельный уровень. Уровень интерфейса является относительно независимым от выполнения основных задач приложения. Его окна или Web-страницы лишь направляют запросы на средний уровень логики приложения, а средний уровень, в свою очередь, взаимодействует с самым нижним уровнем хранения данных.

Иногда термин "трехуровневая архитектура" неверно трактуется как требование развертывания системы на трех компьютерах. Однако он подразумевает лишь логическое деление приложения. Число компьютеров, на которых размещается система, при этом варьируется от 1 до 3 (рис. 30.15).

Эта архитектура существенно отличается от *двухуровневой архитектуры* (two-tier architecture), при которой логика приложения включается в определения окон графического интерфейса. Объекты графического интерфейса взаимодействуют (считывают и записывают информацию) непосредственно с базой данных, и отдельного среднего уровня логики приложения не существует. Недостатком двухуровневой архитектуры является невозможность представления логики приложения посредством отдельных компонентов, обеспечивающих возможность повторного использования объектов. Двухуровневая архитектура клиент/сервер приобрела особую популярность с появлением таких средств, как Visual Basic и PowerBuilder.

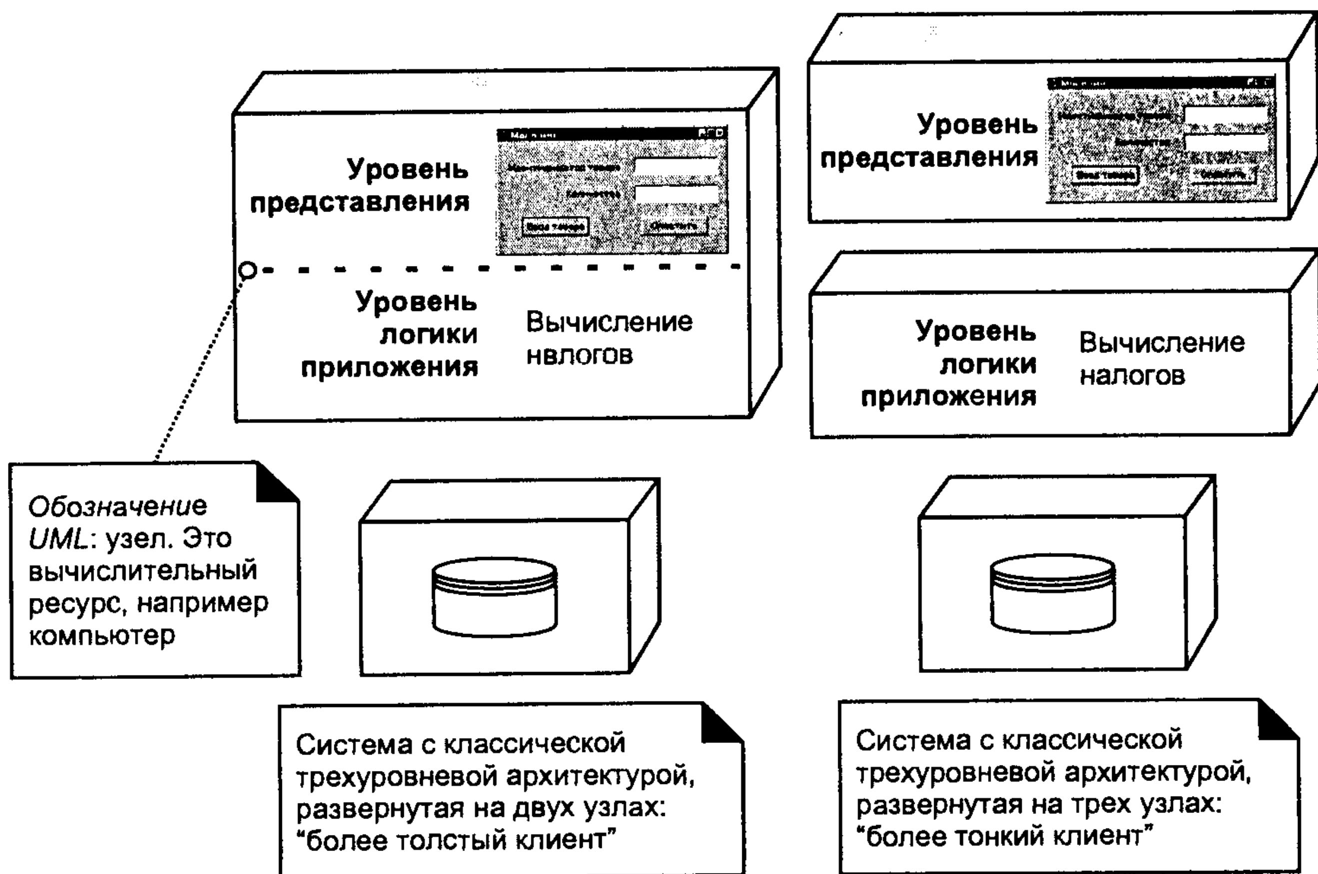



*Рис. 30.15. Размещение трехуровневого приложения в виде двух физических архитектур*

Преимуществом двухуровневой архитектуры (в некоторых случаях) становится быстрота разработки. Однако ее использование вызывает множество проблем. Тем не менее существуют простые приложения для обработки данных, для которых эта архитектура достаточно удобна.

**Связанные шаблоны**

- Indirection (Перенаправление) — к двухуровневой архитектуре добавляется новый уровень перенаправления.
- Protected Variations (Защищенные вариации) — уровни защищены от влияния конкретной реализации.
- Low Coupling (Слабое связывание) и High Cohesion (Высокое зацепление) — уровни обеспечивают реализацию этих шаблонов.
- Применение этого шаблона к проектированию объектно-ориентированных информационных систем описано в [46].

**Также известен как** шаблон Layered Architecture (Многоуровневая архитектура) [51, 99].

## 30.3. Принцип Model-View Separation

Этот принцип уже несколько раз упоминался в книге. Здесь мы подведем итог его обсуждению.

Как можно обеспечить взаимодействие других пакетов с уровнем представления? Как классы, не имеющие отношения к интерфейсу пользователя, должны взаимодействовать с окнами? Желательно, чтобы другие компоненты не взаимодействовали напрямую с объектами окон, поскольку окна связаны с кон-

кретным приложением, а остальные компоненты (в идеале) могут повторно использоваться в новых приложениях или дополняться другим интерфейсом. В этом состоит принцип Model-View Separation (Отделение модели от внешнего представления).

В этом контексте "модель" — это синоним объектов уровня предметной области, а "внешнее представление" — это объекты уровня представления, такие как окна, аплеты и отчеты.

Согласно принципу Model-View Separation[4], объекты модели (уровня предметной области) не должны *напрямую* взаимодействовать с объектами уровня представления. Например, объект `Register` или `Sale` не должен напрямую отправлять сообщения об изменении цвета или о закрытии объекту окна `ProcessSaleFrame`.

Как указывалось ранее, некоторое "смягчение" этого принципа обеспечивает шаблон Observer, согласно которому объекты уровня предметной области отправляют сообщения объектам интерфейса пользователя, рассматриваемым только в качестве связующих объектов между уровнями (`PropertyListener` или `AlarmListener`).

Этот шаблон содержит еще одну рекомендацию: классы уровня предметной области должны инкапсулировать информацию и поведение, связанные с логикой приложения, а классы окон являются относительно "тонкими". Они отвечают лишь за ввод и вывод информации, но не поддерживают обработку данных или функции приложения.

Применение принципа Model-View Separation предоставляет следующие возможности.

- Поддерживает высокий уровень зацепления специальных объектов и позволяет сконцентрировать внимание на процессах предметной области, а не на вопросах интерфейса.
- Позволяет разделить процесс разработки уровней модели и интерфейса пользователя.
- Минимизирует влияние изменений интерфейса на специальные объекты.
- Позволяет легко подключить новый интерфейс, не затрагивая уровень реализации.
- Позволяет одновременно использовать несколько различных представлений для одного и того же специального объекта, например, вывести информацию о продажах в виде таблицы и диаграммы.
- Обеспечивает выполнение задач уровня реализации независимо от интерфейса пользователя, например обработку сообщений в пакетном режиме.
- Упрощает переход к другому контуру интерфейса пользователя.

---

[4] Этот принцип является ключевым в шаблоне Model-View Controller (MVC). Шаблон MVC изначально появился как мелкомасштабный шаблон для языка Smalltalk-80. Согласно этому шаблону, объекты данных (модели) взаимодействовали с объектами GUI (внешним представлением) через обработчики событий мыши и клавиатуры (контроллеры). В настоящее время термин MVC в сообществе разработчиков распределенных систем применяется при обсуждении крупномасштабных архитектурных решений. В качестве модели здесь выступает уровень предметной области, а в качестве внешнего представления — уровень интерфейса. Контроллерами при этом являются объекты уровня приложения.

## Принцип Model-View Separation и взаимодействие “снизу вверх”

Как передавать окнам отображаемую информацию? Как правило, объекты интерфейса отправляют сообщения специальным объектам с запросами на получение информации, которую необходимо отобразить. Такая модель отображения изменений называется моделью с использованием *опросов* (polling) или *опрашиванием сверху* (pull-from-above).

Однако иногда модели опросов оказывается недостаточно. Например, неэффективно опрашивать тысячи объектов каждые несколько секунд в надежде выявить одно или два изменения, которые необходимо отразить в интерфейсе. В такой ситуации предпочтительнее, чтобы изменившиеся объекты предметной области сами взаимодействовали с окнами и инициировали отображение изменений. Типичные ситуации такого вида встречаются в следующих случаях.

- В приложениях мониторинга, например в системах управления телекоммуникационными сетями.
- В приложениях моделирования с визуализацией результатов, например в системах аэродинамического моделирования.

В таких ситуациях требуется модель *предоставления информации снизу* (push-from-below). Из-за ограничений шаблона Model-View Separation это приводит к непрямому взаимодействию объектов с окнами, т.е. непрямому уведомлению об обновлении информации нижнего уровня.

Вот два возможных решения этой проблемы.

1. Использование шаблона Observer, при котором на уровне представления создается объект, реализующий функции интерфейса (например, `PropertyListener`).
2. Использование фасадного объекта для уровня представления. В этом случае к уровню представления добавляется фасадный объект, получающий запросы с нижних уровней приложения. В этом примере используется шаблон Indirection (Перенаправление) для обеспечения принципа Protected Variations (Защищенные вариации) от изменений GUI (рис. 30.16).

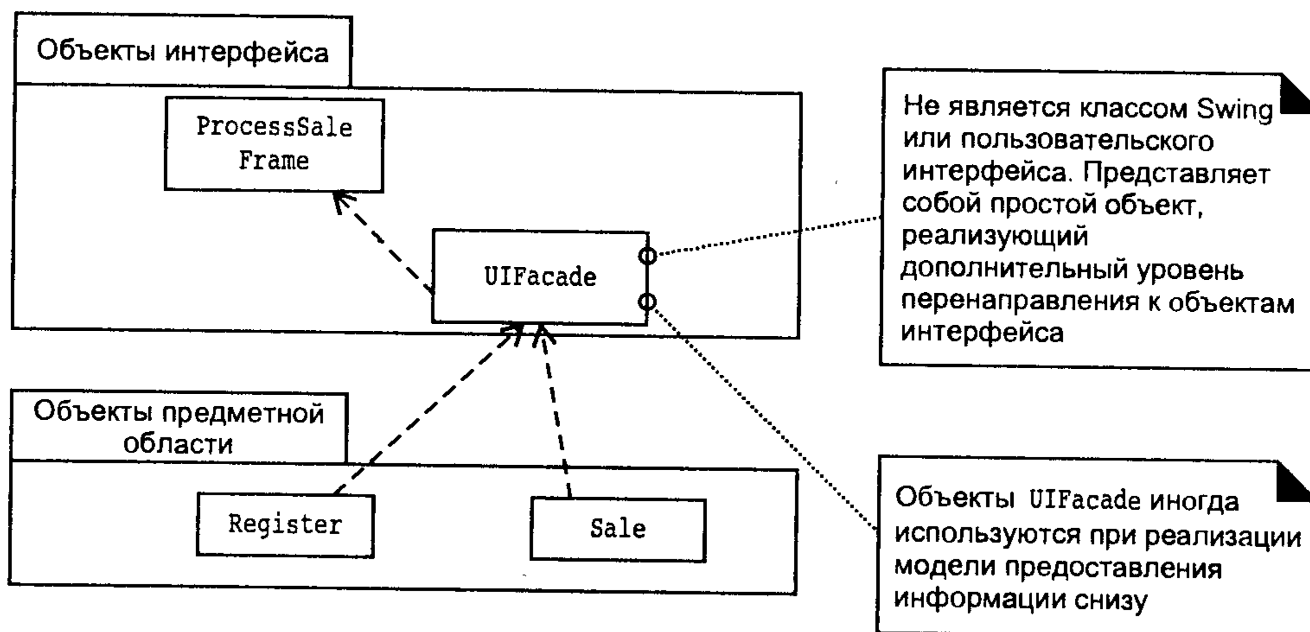



*Рис. 30.16. Использование объекта* `UIFacade` *уровня интерфейса пользователя для опрашивания сверху*

## 30.4. Дополнительная литература

Многоуровневой архитектуре приложений посвящено множество публикаций, как печатных, так и электронных. Шаблоны для проектирования такой архитектуры впервые предложены в книге [40], однако сама архитектура описана во многих работах начиная с 60-х годов. Во втором томе приводятся новые шаблоны, относящиеся к многоуровневой архитектуре. Шаблон Layers хорошо описан в книге [15] (том 1).

Глава

# Создание модели проектирования и реализации на основе пакетов

*Если бы вам пришлось вспахать поле, что бы вы предпочли: две сильных лошади или 1024 цыпленка?*

*Сеймур Край (Seymour Cray)*

**Основные задачи**

- **Реализовать пакетную структуру для снижения степени влияния изменений.**
- **Ознакомиться с альтернативной системой обозначений UML.**

## Введение

Если некоторый пакет X в значительной мере зависит от конкретной группы разработчиков, то его нужно сделать максимально устойчивым (не подлежащим изменению в новых версиях системы), поскольку в противном случае повышается зависимость от этих разработчиков. Именно их придется привлекать для модернизации пакета в случае необходимости.

Это достаточно очевидно, но иногда данному вопросу не уделяется должного внимания, что приводит к неоправданно высоким трудозатратам.

Материал этой главы базируется на рассмотренных в предыдущей главе вопросах организации уровней и пакетов, но здесь предлагаются более "тонкие" эвристические советы по организации пакетов, позволяющие снизить зависимость от возможных изменений внутри пакетов. Основная задача — создать робастную физическую структуру пакетов.

Зависимость от правильной пакетной организации более остро чувствуют разработчики на языке C++, чем на Java, поскольку C++ более чувствителен к реализации компилятора и компоновщика. Изменение одного класса может отразиться на множестве других классов и потребовать перекомпоновки системы.[1] Поэтому предлагаемые здесь рекомендации особенно полезны при разработке систем на C++ и менее важны для реализации проектов на Java, Smalltalk или C#.

Некоторые из приведенных ниже советов навеяны рассуждениями Роберта Мартина [79] о проектировании физической структуры приложения на C++ и его пакетной организации.

### *Исходный код физической архитектуры в модели реализации*

Этот вопрос относится к физическому проектированию системы, которое выполняется в рамках модели реализации путем разбиения исходного кода на пакеты.

В процессе построения диаграмм на бумаге или с помощью CASE-средств разработчик может разместить типы данных и пакеты достаточно произвольно. Однако в процессе проектирования физической архитектуры, подразумевающей организацию типов данных в физические пакеты на C++ или Java, способ разбиения системы на пакеты определяется взаимным влиянием элементов системы.

## 31.1. Рекомендации по организации пакетов

### *Вертикальное и горизонтальное зацепление функциональности пакетов*

Основной интуитивный подход к разбиению системы на модули подразумевает группировку элементов по принципу высокого зацепления — тесной взаимосвязи частей пакета в рамках реализации общих задач, служб, политик и функций. Например, все типы данных пакета `Pricing` системы NextGen связаны с определением стоимости товаров. Уровни и пакеты системы NextGen разбиты на группы по функциональному назначению.

Менее очевидным критерием группировки является высокая степень внутреннего связывания, определяющая принадлежность типов некоторому кластеру. Например, класс `Register` тесно связан с классом `Sale`, который, в свою очередь, связан с объектами `SalesLineItem`.

Степень внутреннего связывания пакетов или его относительное зацепление можно выразить в числовом представлении, хотя эта характеристика используется достаточно редко. Для общего сведения приведем формулу ее вычисления.

```
RC=ЧислоВнутреннихСвязей/КоличествоТипов
```

Здесь `ЧислоВнутреннихСвязей` включает связи между параметрами и атрибутами, отношения наследования, реализацию интерфейсов среди типов данных пакета.

Для пакета с 6 типами данных и 12 связями коэффициент связывания `RC` составляет 2. Для пакета с 6 типами и 3 внутренними связями `RC=0.5`. Более высокие значения коэффициента означают более высокую степень зацепления элементов данного пакета.

---

[1] В C++ пакеты можно реализовать через пространства имен, однако лучше разделить файлы исходного кода на несколько каталогов, по одному на каждый пакет.

Заметим, что эта характеристика не очень информативна для пакетов, содержащих в основном интерфейсы. Она гораздо показательнее для пакетов, включающих реализации классов.

Слишком маленькое значение RC означает выполнение одного из следующих условий.

- Пакет содержит не связанные между собой элементы и плохо нормализован.
- Пакет содержит не связанные между собой элементы, но это не должно беспокоить разработчиков. Это случается с пакетами утилит или разрозненных служб (например, java.util). В этом случае значение RC не играет роли.
- Пакет содержит один или несколько внутренних кластеров с высоким коэффициентом RC, но в целом связность элементов пакета невысока.

### Пакет, представляющий собой семейство интерфейсов

Семейство функционально связанных интерфейсов можно поместить в отдельный пакет, не содержащий реализации классов. Речь идет не о паре связанных друг с другом интерфейсов, а о семействе, включающем не менее трех интерфейсов. Примером такого пакета для Java-технологий является пакет javax.ejb, содержащий порядка 12 интерфейсов, реализации которых содержатся в других пакетах.

### Формирование пакетов на базе групп неустойчивых классов

Основная мысль данного обсуждения сводится к следующему. Пакет — это базовая структурная единица разработки и программного продукта. Отдельные классы разрабатываются гораздо реже и совсем редко становятся приложениями.

Допустим, существует большой пакет P1, содержащий порядка тридцати классов, некоторое подмножество которого (порядка 10 классов: C1...C10) постоянно модифицируется.

В этом случае стоит разделить пакет P1 на два отдельных пакета P1-a и P1-b, включив в P1-b десять неустойчивых классов.

Таким образом пакет делится на устойчивую и неустойчивую части, либо на группы классов, работа над которыми ведется одновременно. Следовательно, приходим к выводу: если работа над классами пакета ведется одновременно, значит, пакет сгруппирован правильно.

В идеале пакеты нужно сформировать таким образом, чтобы от классов пакета P1-b зависела работа меньшего числа разработчиков, чем от пакета P1-a. Тогда очередная версия пакета P1-b скажется на дальнейшей работе ограниченного круга разработчиков.

Заметим, что такая перегруппировка является следствием учета трудоемкости создания пакета. На ранних итерациях очень сложно сформировать хорошую структуру пакетов приложения. Она постепенно изменяется на стадии развития, и по окончании этой стадии разработки должна быть сформирована устойчивая структура большинства пакетов.

Данный совет сводится к следующей основной стратегии: **снижение зависимости от неустойчивых пакетов.**

### *Базовые пакеты должны быть более устойчивыми*

Если базовые пакеты (от которых зависит большинство других) неустойчивы, то повышается вероятность ошибок в приложении. Например, если широко используемый пакет утилит, такой как com.foo.util, часто изменяется, то это может повлечь за собой неработоспособность многих других элементов системы. На рис. 31.1 показана рекомендуемая структура пакетов с точки зрения их взаимозависимости.

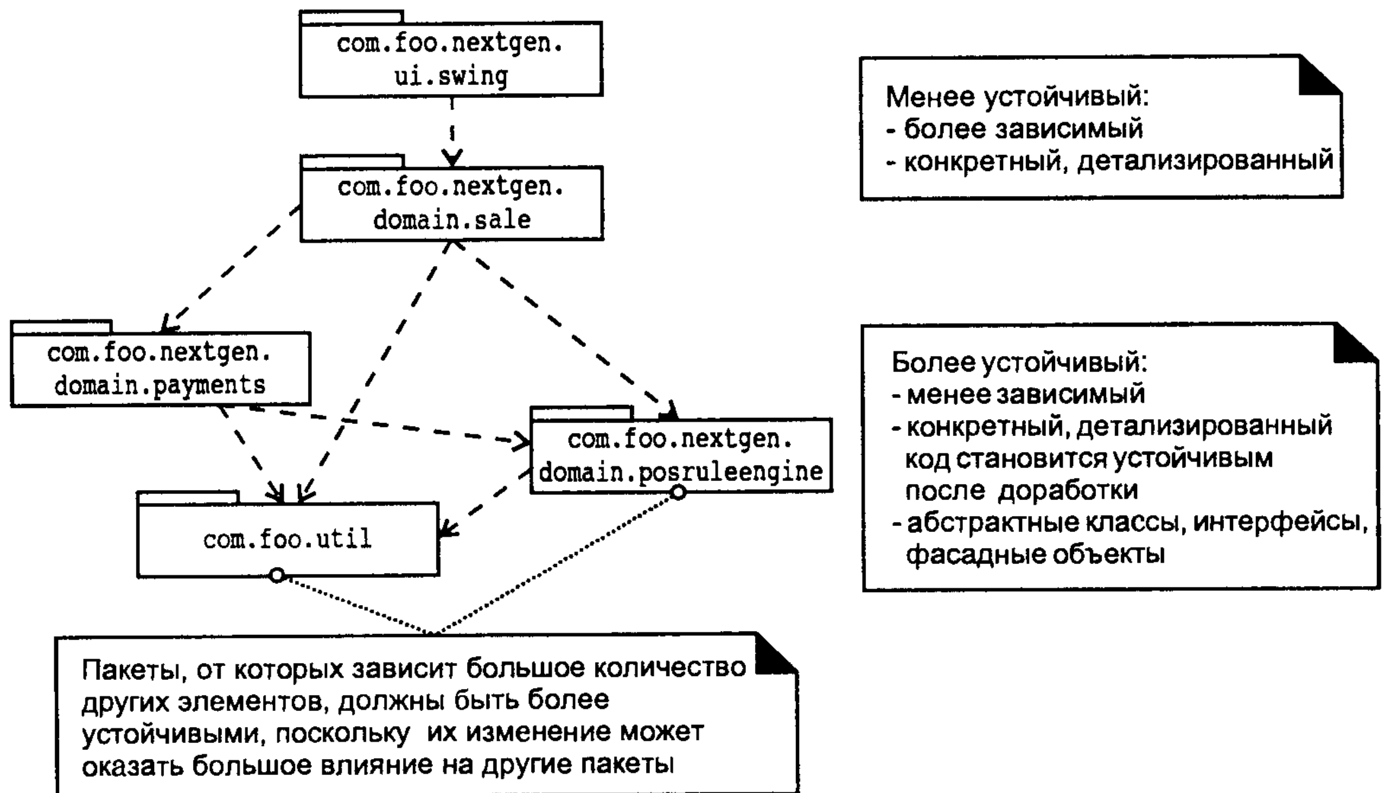



*Рис. 31.1. Базовые пакеты должны быть более устойчивыми*

Чем ниже расположены пакеты на этой диаграмме, тем устойчивее они должны быть. Существует несколько способов повышения устойчивости пакета.

- Включение в пакет преимущественно интерфейсов и абстрактных классов.
  - Например, пакет java.sql содержит восемь интерфейсов и шесть классов, преимущественно простых и устойчивых, как Time и Date.
- Отсутствие зависимости от других пакетов, или зависимость только от очень устойчивых пакетов, либо инкапсуляция зависимости внутри пакета.
  - Например, в пакете com.foo.nextgen.domain.posruleengine можно скрыть реализацию правил за "фасадным" объектом. Тогда даже при изменении реализации правил зависимые пакеты не будут оказывать влияния друг на друга.
- Включение в пакет относительно устойчивого кода, апробированного и испытанного ранее.
  - Например, java.util.
- Утверждение стабильного графика разработки, предполагающего нечастое внесение изменений.
  - Например, основной пакет библиотеки Java java.lang просто не может изменяться довольно часто.

### Факторизация независимых типов

Классы, используемые независимо друг от друга или в различном контексте, следует помещать в разные пакеты. Без внимательного изучения группировка на основе сходной функциональности может не обеспечить нужного уровня детализации пакетов.

Допустим, что подсистема служб взаимодействия с базой данных содержится в одном пакете com.foo.service.persistence. Предположим, в этом пакете имеются два общих класса утилит JDBCUtilities и SQLCommand. Раз эти общие утилиты предназначены для взаимодействия с реляционной базой данных, значит, их можно использовать независимо от остальной части подсистемы для доступа к базе данных через JDBC. Следовательно, эти типы лучше выделить в отдельный пакет, например com.foo.util.jdbc (рис. 31.2).

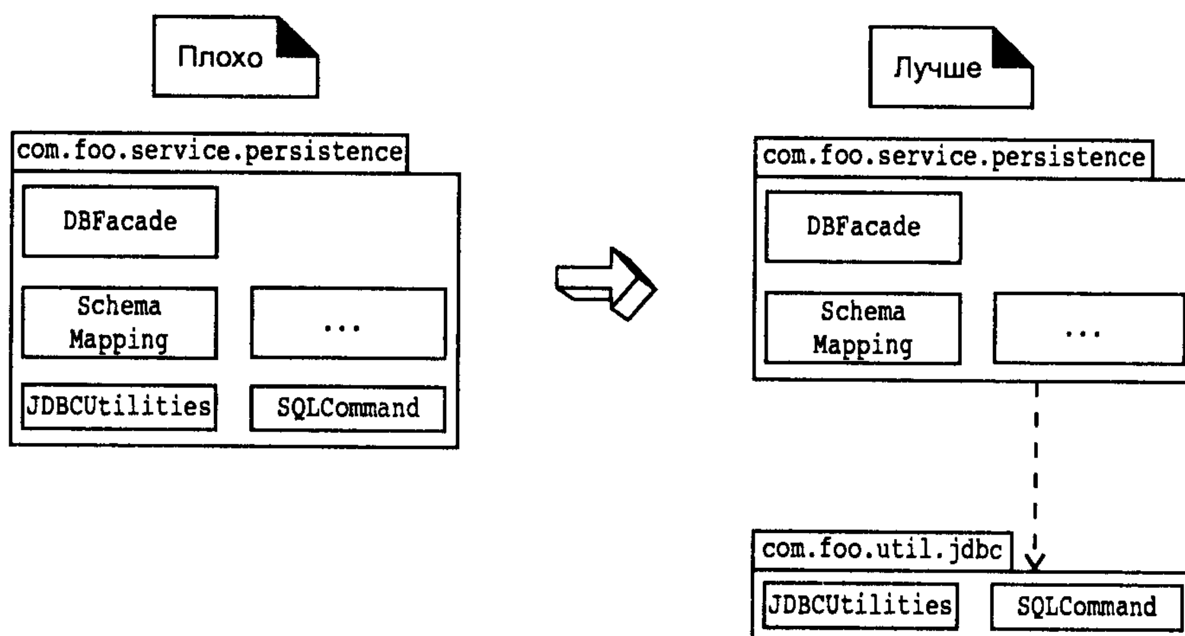



*Рис. 31.2. Факторизация независимых типов*

### Факторизация для снижения зависимости от конкретных пакетов

Одним из способов повышения устойчивости пакетов является снижение зависимости от конкретных классов других пакетов. Проблемная ситуация показана на рис. 31.3.

Предположим, классы Register и PaymentMapper (класс, осуществляющий преобразование объектов платежа в формат реляционной базы данных и обратно) создают экземпляры объектов CreditPayment из пакета Payments. Одним из путей повышения устойчивости пакетов Sales и Persistence является предотвращение создания экземпляров классов, определенных в других пакетах (объектов CreditPayment из пакета Payments).

Такое связывание можно уменьшить за счет использования объекта-фабрики, отвечающего за создание экземпляров. Методы этого объекта-фабрики должны возвращать объекты, объявленные в терминах интерфейсов, а не классов (рис. 31.4).

#### Шаблон Domain Object Factory

Использование объектов-фабрик для создания всех экземпляров классов уровня предметной области — это стандартная идиома проектирования. В лите-

ратуре по проектированию ее можно встретить под именем шаблона Domain Object Factory (Фабрика объектов предметной области), однако формально она не описана в виде шаблона.

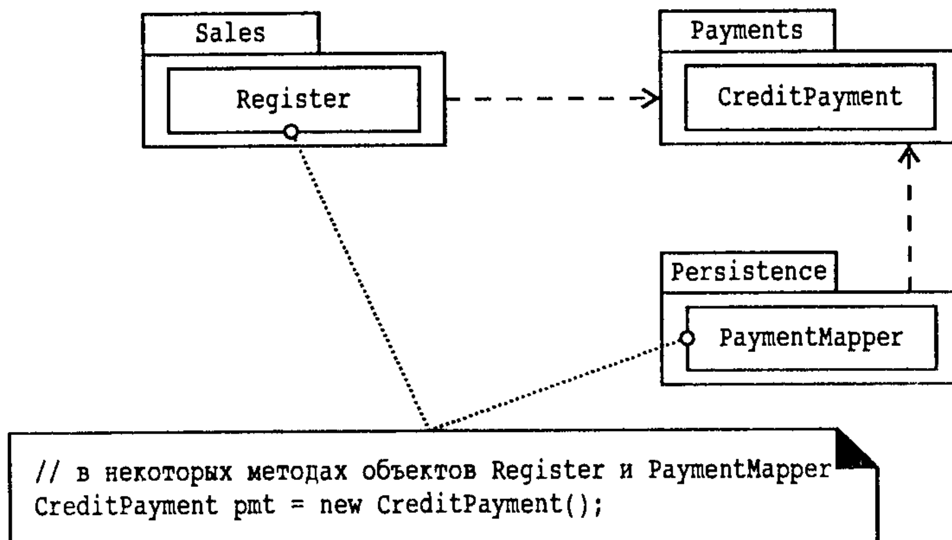



*Рис. 31.3. Непосредственное связывание с конкретным пакетом*

## Не используйте циклы в пакетах

Если группа пакетов связана циклической зависимостью, их можно объединить в один пакет. Такая ситуация нежелательна, поскольку увеличение размера пакета повышает вероятность возникновения ошибок в системе.

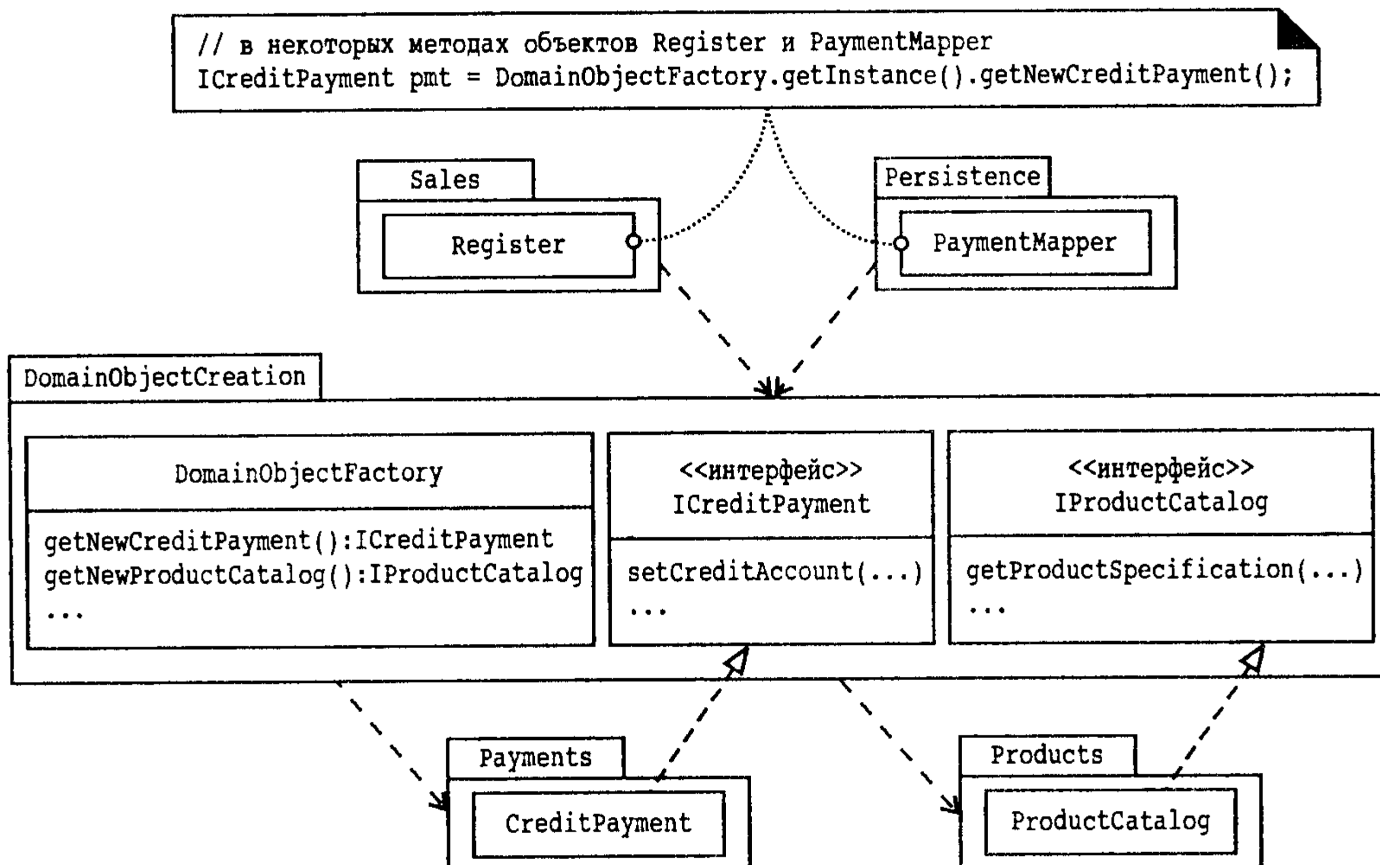



*Рис. 31.4. Снижение связывания с конкретным пакетом за счет использования объекта-фабрики*

Существует два решения подобной проблемы.

1. Выделение типов, связанных циклической зависимостью, в отдельный пакет меньшего размера.
2. Прерывание цикла с помощью интерфейса.

Для этого можно предпринять следующие действия.

1. Переопределить зависимые классы в одном из пакетов таким образом, чтобы они реализовывали новые интерфейсы.
2. Определить новые интерфейсы в новом пакете.
3. Переопределить зависимые типы таким образом, чтобы они зависели от интерфейсов в новом пакете, а не от исходных классов.

Эта стратегия показана на рис. 31.5.

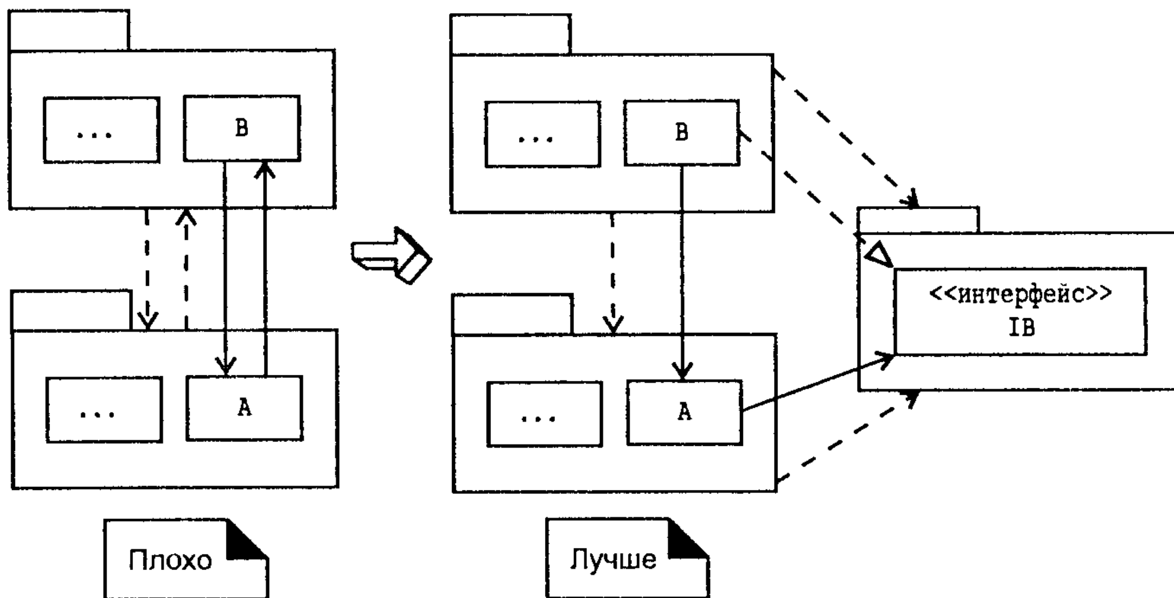



*Рис. 31.5. Предотвращение циклической зависимости*

## 31.2. Дополнительные обозначения UML для пакетов

Для обозначения пакетов в UML существует еще одна группа обозначений, призванная выделить внешние и внутренние пакеты. Иногда нежелательно помещать пакеты внутри внешнего пакета. Альтернативное обозначение показано на рис. 31.6.

## 31.3 Дополнительная литература

Не удивительно, что большинство публикаций по улучшению структуры пакетов и снижению зависимостей принадлежит перу авторов, разрабатывающих приложения на C++. Однако изложенные в этих книгах принципы применимы и к другим языкам программирования. Эти вопросы хорошо проработаны в книгах [74] и [79]. Данной тематике посвящена также книга [56].

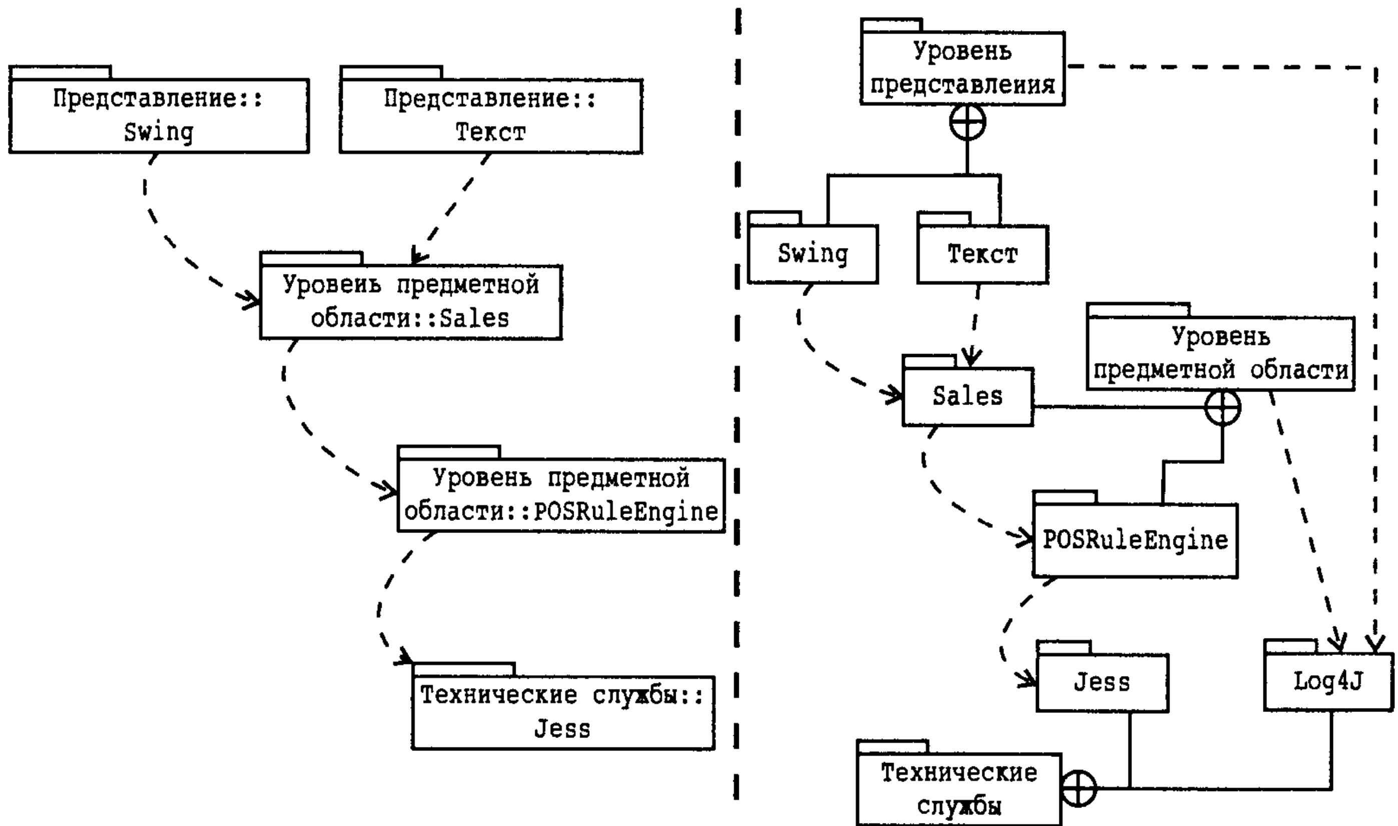



*Рис. 31.6. Альтернативное обозначение для структуры пакетов с использованием путей UML*

Глава

# Введение в архитектурный анализ

*Ошибка, не связанная с клавиатурой, — нажмите <F1> для продолжения.*

*(Сообщение ранних версий PC BIOS)*

**Основные задачи**

- **Создать таблицы архитектурных факторов.**
- **Создать технические описания архитектурных решений.**
- **Ознакомиться с базовыми принципами архитектурного проектирования.**
- **Ознакомиться с ресурсами для изучения архитектурных шаблонов.**

## Введение

Задача архитектурного анализа сводится к выявлению факторов, определяющих архитектуру системы, осознанию их приоритетов и возможности изменения, а также решению связанных с этим проблем. На этом этапе важно осознать стоящие перед разработчиками проблемы, взвесить все "за" и "против" различных решений, оценить степень влияния архитектурно важных факторов, проранжировать их в соответствии с приоритетами, выработать проектные решения и оценить продукты сторонних производителей.

В контексте UP архитектурные факторы описываются в дополнительной спецификации, а соответствующие архитектурные решения — в документе SAD (Software Architecture Document), описывающем программную архитектуру. Документ SAD более подробно описывается ниже в этой главе.

Архитектурный анализ начинается на ранних стадиях разработки проекта, возможно, в процессе начальной фазы, а основные усилия нужно сконцентрировать на стадии развития. Это высокоприоритетный и очень важный вид деятельности в рамках разработки программных систем. В этой книге вопросы архитектурного анализа рассматриваются одними из последних, поскольку сначала были

изложены основные принципы объектно-ориентированного анализа и проектирования. Тем не менее, архитектурный анализ очень важен для решения следующих проблем.

- Снижение риска отсутствия важных элементов в проектном решении системы
- Снижение трудозатрат на реализацию низкоприоритетных моментов
- Обеспечение конкурентоспособности программного продукта

В этой главе рассматриваются основные моменты и идеи архитектурного анализа в контексте UP. То есть здесь излагается метод, а не приводятся готовые советы и архитектурные решения, позволяющие повысить профессиональный уровень архитектора системы. Это не справочник архитектурных решений, здесь рассматриваются лишь вводные вопросы. Тем не менее, на примере рассматриваемого в данной книге приложения NextGen приводятся конкретные образцы проектных решений.

## 32.1. Архитектурный анализ

*Архитектурный анализ* (architectural analysis) связан с определением и удовлетворением нефункциональных (например, качественных) требований к системе в контексте функциональных требований.

В контексте UP этот термин охватывает как исследование архитектуры (определение), так и архитектурное проектирование (удовлетворение требований). Ниже приведены вопросы, решаемые на архитектурном уровне системы.

- Как требования надежности и защиты от сбоев влияют на проектное решение системы?
  - Предположим, для каких удаленных служб POS-системы NextGen (например, системы вычисления налоговых платежей) нужно обеспечить дублирование на локальном компьютере? Почему? Будут ли функции локальной реализации в точности соответствовать функциям удаленного прототипа или предусматриваются какие-то отличия?
- Как стоимость лицензирования или покупки программных компонентов повлияет на рентабельность всей системы?
  - Например, производитель прекрасного сервера базы данных Clueless требует за свои услуги по 2% от каждой продажи, оформленной через POS-систему NextGen, если их продукт входит в состав системы. Использование этого продукта ускорит разработку, поскольку он обладает свойством робастности и поддерживает многие важные службы, однако за отдельную плату. Нужно ли в этом случае использовать менее робастный бесплатный сервер баз данных YourSQL? Какой риск связан с таким решением? Насколько оно повлияет на стоимость всей системы?
- Как распределенность системы повлияет на функциональные и качественные требования?
  - Например, использование удаленной (централизованной) системы вычисления налоговых платежей снизит нагрузку на клиентскую часть системы NextGen, уменьшит стоимость лицензирования этого программного продукта (потребуется только одна его копия) и минимизирует усилия по конфигури-

рованию системы (при каждой установке почти неделя уходит на настройку программного продукта в соответствии с изменением законодательства или экономической политики). Однако время отклика удаленной службы можно значительно сократить, если вычислять налог один раз после выбора всех товаров. Тогда после добавления каждого нового товара общая стоимость покупки с учетом налогов не будет обновляться. Такое решение также приводит к созданию единственной проблемной точки.

- Какое влияние на проектное решение окажут требования адаптируемости и быстрой настройки?
  - Например, большинство торговых организаций хотят видеть в своих POS-приложениях различные вариации бизнес-правил. В чем состоят эти различия? Как их лучше всего разработать? Каковы критерии выбора бизнес-правил, подлежащих реализации в системе? Нужно ли программно настраивать приложение NextGen для каждого пользователя в отдельности или пользователи сами могут сконфигурировать приложение? Нужно ли стремиться заработать максимум денег за минимальное время?

### *Общие методы архитектурного анализа*

Существует несколько методов архитектурного анализа. Тем не менее, большинство из них предполагает выполнение в той или иной форме следующих шагов.

1. Определить и проанализировать нефункциональные требования, оказывающие влияние на архитектуру. Функциональные требования тоже нужно учитывать (особенно в терминах возможности изменений), но основное внимание следует уделить нефункциональным требованиям. В целом, эти требования можно назвать *архитектурными факторами* (или *архитектурными предпосылками*).
   - Этот шаг можно рассматривать как элемент обычного анализа требований, однако поскольку он выполняется в контексте идентификации влияния архитектуры и принятия высокоуровневых решений, он относится к архитектурному анализу.
   - В рамках UP некоторые из этих требований в целом определяются и описываются в дополнительной спецификации или описании прецедентов на начальной стадии проекта. В процессе архитектурного анализа, выполняемого на первых итерациях фазы развития, разработчики исследуют эти требования более глубоко.
2. Для требований, существенно затрагивающих архитектуру системы, проанализировать альтернативы и выработать приемлемые решения. Такие решения называются *архитектурными* (architectural decision).
   - Решения могут варьироваться в широком диапазоне: от “отменить требование” до “прекратить проект”, “привлечь экспертов” или “найти решение проблемы”.

Рассмотрим выполнение этих действий в контексте POS-системы NextGen. Для простоты вопросы архитектурного развертывания, конфигурирования аппаратных средств и операционной системы рассматриваться не будут.

## 32.2. Типы представлений архитектуры

В некоторых описаниях вводятся различные типы архитектуры, в частности, "архитектура приложения" (размещение средств и компонентов) или "системная архитектура" (конфигурация аппаратных средств или операционной системы).

В UP архитектура тоже рассматривается с разных точек зрения, но здесь они названы *архитектурными представлениями* (architecture view). Каждое из этих представлений призвано сконцентрировать внимание на определенной стороне вопроса. Например, *логическое представление* архитектуры, введенное в главе 30, подытоживает организацию и функциональность основных программных элементов (в частности, уровней). Это понятие аналогично термину "архитектура приложения". *Представление развертывания* (deployment view) описывает топологию системы, типы взаимодействия компонентов и распределение выполняемых элементов системы между узлами. Это понятие аналогично термину "системная архитектура".

В UP определены шесть представлений архитектуры, которые будут подробно описаны в конце этой главы. Эти представления включают текстовые описания и диаграммы, помещаемые в документе SAD (если они создаются вообще).

Архитектурный анализ связан с архитектурными представлениями, поскольку именно в них описываются архитектурные решения.

## 32.3. Определение и анализ архитектурных факторов

### Архитектурные факторы

Все требования FURPS+ оказывают существенное влияние на архитектуру системы. К таким требованиям относятся надежность, график реализации проекта, ограничения по стоимости или высокий уровень квалификации. Например, в условиях плотного графика, недостаточного профессионализма и достаточного финансирования лучше приобрести готовые компоненты, чем пытаться разрабатывать все элементы системы кустарным способом.

Факторы, оказывающие наибольшее влияние на архитектуру системы, можно определить в контексте высокоуровневых категорий функциональности, надежности, производительности, простоты сопровождения и реализации, а также интерфейса (эти требования более подробно описаны в главе 5). Интересно, что эти требования являются качественными атрибутами системы (например, надежность и производительность) и придают конкретной архитектуре особый "дух", не связанный с функциональными требованиями. Примерами таких требований для системы NextGen является поддержка внешних компонентов с их специфическими интерфейсами и поддержка подключения различных наборов бизнес-правил.

В UP эти факторы получили название *архитектурно значимых требований* (architecturally significant requirements). Термин "факторы" используется только для краткости изложения.

Многие технические и организационные факторы можно рассматривать как *ограничения*, накладываемые на проектное решение (например, система должна работать под управлением операционной системы Linux, или необходимо приобрести компонент X).

## Сценарии реализации качественных требований

При определении качественных требований в процессе анализа архитектурных факторов рекомендуется разработать специальные сценарии[1] с реальным выходом. Недостаточно задекларировать, что "система будет легко расширяемой", поскольку за этой фразой не стоит никакого конкретного решения.

Зачастую требования обеспечения производительности системы и бесперебойной работы в течение некоторого периода времени облекают в числовую форму. Однако сценарии реализации качественных требований дополняют этот процесс и требуют записи всех (или большинства) факторов в количественной форме.

*Сценарии реализации качественных требований* — это краткие утверждения вида *<предпосылка><измеряемый результат>*. Рассмотрим несколько примеров.

- После завершения продажи отклик от внешней системы вычисления налоговых платежей должен поступать в среднем через 2 секунды.
- После получения информации от специалистов по бета-тестированию ответ в виде телефонного звонка должен быть отправлен в течение одного рабочего дня.

Заметим, что термин "в среднем" требует дополнительного исследования архитектором системы. Сценарии реализации качественных требований не могут считаться корректными до момента их апробации. Заметим также, что временные оценки в первом сценарии зависят от среды разработки и работы самого приложения. Этот сценарий может успешно реализовываться только в среде разработки.

### Определение узких мест

Иногда написание сценариев реализации качественных требований не приносит желаемого результата. Написать детальную спецификацию очень легко, а реализовать ее очень трудно. Можно ли будет проверить выполнение этих требований? Как и какими силами? К написанию подобных сценариев нужно подходить реалистично. Не имеет смысла тратить силы на формулировку сложных задач, если ни одна из них не будет решена или невозможно проверить качество решения.

Какие факторы играют главную роль при написании сценариев? Для системы резервирования авиабилетов действительно критичным является быстрое выполнение транзакций в условиях высокой нагрузки на систему. Это требование обязательно нужно реализовать и протестировать качество его реализации. Система NextGen должна быть устойчивой к сбоям в работе внешних систем. Это достигается за счет локальной репликации важных внешних служб. Выполнение этого свойства необходимо проверить. Следовательно, сценарии реализации качественных требований нужно составлять для критичных элементов системы, причем эти сценарии должны быть проверяемыми.

## Описание архитектурных факторов

Важной задачей архитектурного анализа является осознание влияния различных факторов, расстановка их приоритетов и определение степени гибкости системы. Поэтому в рамках многих методов разработки архитектуры (например,

---

[1] Термин "сценарий реализации качественных требований" используется в различных методах анализа архитектуры, разработанных Институтом программных технологий SEI (Software Engineering Institute), например методе проектирования системы на основе архитектуры.

в [63]) предлагается строить деревья или таблицы, формат которых зависит от конкретного метода. В табл. 32.1 показаны факторы, описываемые в дополнительной спецификации.

**Таблица 32.1. Пример факторной таблицы (в — высокий, с — средний)**

| Фактор | Мера качества и сценарии | Изменяемость (текущая гибкость и возможность эволюции) | Влияние фактора на заинтересованных лиц, архитектуру или другие факторы | Приоритет | Сложность или риск |
|---|---|---|---|---|---|
| **Надежность/восстанавливаемость** | | | | | |
| Защита от сбоев внешних служб | В случае сбоя в работе внешней службы при нормальной работе магазина нужно восстановить соединение в течение 1 минуты | Текущая гибкость — использовать упрощенные службы из клиентской части приложения.<br><br>Эволюция: в течение 2 лет реалии могут измениться, что потребует полной локальной репликации удаленных служб. Вероятность такого изменения очень высока | Сильное влияние на общее проектное решение<br><br>Торгующие организации не любят связываться с ненадежными внешними службами, что может "оттолкнуть" их от системы NextGen | в | с |
| ... | ... | ... | ... | ... | ... |

Обратите внимание на название категории — **Надежность/восстанавливаемость**. Оно позаимствовано из модели FURPS+. Такая категоризация не является единственно возможной, но ее удобно использовать для группировки архитектурных факторов. Например, некоторые категории, такие как надежность и производительность, в значительной мере зависят от планов тестирования, поэтому их стоит сгруппировать в одну категорию.

Шкала измерения приоритетов и рисков (высокий/средний/низкий) является весьма условной. Возможны и другие числовые и качественные описания. Их можно позаимствовать из различных архитектурных методов и стандартов, в частности ISO 9126. Но стоит сделать одно предупреждение: если использование сложной схемы не приведет к ощутимому результату, то от такой схемы следует отказаться.

### *Факторы и артефакты UP*

В UP основные функциональные требования определяются в описании прецедентов, которое наряду с дополнительной спецификацией и описанием в*и*дения системы является базисом для построения таблицы факторов. В описании прецедентов наиболее важны разделы "Специальные требования", "Технологические вариации" и "Открытые вопросы". Архитектурные факторы в явной форме описываются также в дополнительной спецификации.

Некоторые факторы можно описать в процессе описания прецедентов. Однако лучше разместить все архитектурные факторы в одном месте — в таблице факторов дополнительной спецификации.

***Прецедент П1. Оформление продажи***

**Основной успешный сценарий (или основной процесс)**

1. ...

**Специальные требования**

- Отклик службы авторизации в 90% случаев приходит в течение 30 секунд.
- Каким-то образом нужно обеспечить робастное восстановление информации в случае сбоя при доступе к удаленным службам, таким как система складского учета.
- ...

**Список технологий и типов данных**

3a. Идентификатор товара считывается со штрих-кода (при наличии последнего) лазерным сканером или вводится с клавиатуры.

...

**Открытые вопросы**

- Изучить законодательство по налогообложению.
- Исследовать вопрос восстановления удаленных служб.

## 32.4. Пример: фрагмент таблицы архитектурных факторов POS-системы NextGen

В табл. 32.2 приводятся некоторые архитектурные факторы, связанные с рассматриваемым приложением.

**Таблица 32.2. Фрагмент факторной таблицы для архитектурного анализа приложения NextGen**

| Фактор | Мера качества и сценарии | Изменяемость (текущая гибкость и возможность эволюции) | Влияние фактора на заинтересованных лиц, архитектуру или другие факторы | Приоритет | Сложность или риск |
|---|---|---|---|---|---|
| **Надежность/восстанавливаемость** | | | | | |
| Защита от сбоев внешних служб | В случае сбоя в работе внешней службы при нормальной работе магазина нужно восстановить соединение в течение 1 минуты | Текущая гибкость — использовать упрощенные службы из клиентской части приложения.<br><br>Эволюция: в течение 2 лет реалии могут измениться, что потребует полной локальной репликации удаленных служб. Вероятность такого изменения очень высока | Сильное влияние на общее проектное решение<br><br>Торгующие организации не любят связываться с ненадежными внешними службами, что может "оттолкнуть" их от системы NextGen | в | с |
| Защита от сбоев удаленной базы данных | То же | Текущая гибкость — кэшировать информацию о товарах в клиентской части приложения до восстановления соединения. | То же | в | с |

*Продолжение табл. 32.2*

| Фактор | Мера качества и сценарии | Изменяемость (текущая гибкость и возможность эволюции) | Влияние фактора на заинтересованных лиц, архитектуру или другие факторы | Приоритет | Сложность или риск |
|---|---|---|---|---|---|
| | | Эволюция: в течение 3 лет устройства хранения информации и средства репликации станут дешевле и эффективнее, что приведет к полной локальной репликации удаленной базы данных. Вероятность такого изменения очень высока | | | |
| **Поддержка/Адаптация** | | | | | |
| Поддержка многих внешних служб (бухгалтерской системы, системы вычисления налоговых платежей и складского учета). Службы могут отличаться при каждой установке системы | При необходимости интеграция с новой внешней службой должна занять не более 10 человеко-дней | Текущая гибкость — определяется описанием фактора. Эволюция отсутствует | Требуется для обеспечения конкурентоспособности продукта. Незначительное влияние на проектное решение | в | н |
| Поддержка беспроводных терминалов для клиентов POS-системы | Добавление этой возможности не должно потребовать изменения проектного решения уровней архитектуры, не связанных с интерфейсом пользователя | Текущая гибкость — в настоящее время не требуется. Эволюция: в течение трех лет с высокой вероятностью на рынке появятся беспроводные терминалы для клиентов POS-системы | Высокое влияние с учетом возможного изменения многих элементов. Например, может измениться операционная система и интерфейс пользователя | н | в |

*Окончание табл. 32.2*

| Фактор | Мера качества и сценарии | Изменяемость (текущая гибкость и возможность эволюции) | Влияние фактора на заинтересованных лиц, архитектуру или другие факторы | Приоритет | Сложность или риск |
|---|---|---|---|---|---|
| **Другие факторы, в том числе правовые** | | | | | |
| Необходимость учета текущих правил вычисления налоговых платежей | Правила вычисления налоговых платежей должны быть полностью согласованы с аудитором. При изменении налоговых ставок их нужно учесть в системе в указанный правительством срок | Текущая гибкость — правила вычисления налоговых платежей могут изменяться даже еженедельно. Эволюция — отсутствует | Несоблюдение законов влечет уголовную ответственность. Система зависит от служб вычисления налоговых платежей. Собственную службу написать очень трудно, поскольку правила сложны, постоянно изменяются и их нужно своевременно отслеживать. Риск значительно снижается при покупке готового пакета | в | н |

## 32.5. Определение архитектурных факторов

Можно сказать, что *наука* архитектурного анализа сводится к выявлению архитектурных факторов и составлению соответствующей таблицы. *Профессионализм* архитектора состоит в умении определить эти факторы с учетом затрат, взаимозависимостей и приоритетов.

Опытные архитекторы обладают обширными знаниями в разных областях (например, знакомы с архитектурными стилями и шаблонами, технологиями, продуктами и тенденциями), на основе которых строят свои решения.

### Альтернативы, решения и мотивировка

Независимо от принципов принятия архитектурных решений, практически все методы проектирования архитектуры предполагают запись различных вариантов архитектурных решений, факторов влияния и мотивов принятия решений.

Такие записи называют *техническим описанием* (technical memo) [41], *тематическими картами* (issue card) [63] или *описанием архитектурных подходов* (архитектурными предложениями). Глубина проработки материала в различных методах варьируется. В некоторых методах такие документы являются лишь введением к следующим видам деятельности.

В UP технические описания помещаются в документ SAD.

Важным элементом технического описания является мотивировка. Если разработчику или архитектору в будущем придется модифицировать систему,[2] то на осно-

[2] Или если по прошествии 4 недель архитектор системы забыл свой собственный ход мыслей!

ве этого раздела он сможет разобраться, почему было принято то или иное архитектурное решение. Это поможет ему принять решение по изменению системы.

Мотивировка принятия решений и описание возможных альтернативных решений помогут в будущем при модификации программы. Архитектор может воспользоваться этими вариантами или хотя бы ознакомиться с ними.

Рассмотрим пример технического описания для POS-системы NextGen. Точный формат описания значения не имеет. Нужно стремиться к простоте и записывать важную информацию, которая может пригодиться в будущем.

---

**Техническое описание**

**Вопрос. Надежность — восстановление информации при отказе удаленной службы.**

**Идея решения. Обеспечить прозрачный поиск нужной службы, переход от удаленной службы к локальной и осуществить частичную репликацию локальной службы.**

**Факторы**

- Робастное восстановление при выходе из строя удаленной службы (системы складского учета, системы вычисления налоговых платежей).
- Робастное восстановление информации при выходе из строя удаленной базы данных с описаниями товаров.

**Решение**

Реализовать шаблон Protected Variations (Защищенные вариации) для защиты от влияния местоположения службы с помощью интерфейса адаптера и объекта `ServicesFactory`. По возможности обеспечить локальную реализацию удаленных служб с упрощенным и ограниченным поведением. Например, локальная система вычисления налогов может использовать фиксированные ставки налогов. Локальная база данных товаров может представлять собой небольшой буфер с информацией о наиболее популярных товарах. Информацию для обновления системы складского учета можно хранить локально и передавать удаленной системе после восстановления соединения.

Вопросы конкретной реализации этих решений описаны в разделе технической документации "Адаптация — службы сторонних производителей", поскольку реализации удаленных служб могут изменяться.

Для обеспечения качественного восстановления соединения с удаленными службами используйте объекты-посредники, проверяющие дееспособность каждой удаленной службы и перенаправляющие по возможности информацию напрямую удаленным службам.

**Мотивировка**

Владельцы магазинов не желают прекращать торговлю. Следовательно, в системе NextGen нужно предусмотреть высокий уровень надежности и возможность восстановления информации при сбоях. Тогда она станет очень привлекательным продуктом, поскольку конкурирующие системы такой возможности не обеспечивают. Небольшой размер буфера объясняется ограниченностью ресурсов клиентских компьютеров. Системы налоговых платежей не устанавливаются на каждом клиентском компьютере из-за высокой стоимости лицензии и сложности настройки (настройка каждого рабочего места занимает почти неделю). Такое проектное решение обеспечивает точку эволюции системы, и в будущем можно будет поместить службу вычисления налоговых платежей на каждом клиентском компьютере.

**Неразрешенные вопросы**

Отсутствуют.

**Альтернативные решения**

Обеспечение наивысшего уровня качества связи с удаленными службами авторизации платежей по кредитной карточке для повышения надежности. Это возможно, но очень дорого.

---

Заметим, что это иллюстративный пример. В одном техническом описании можно рассматривать целую группу факторов, а не только один.

## Приоритеты

Рассмотрим иерархию задач, определяющих архитектурные решения.

1. Негибкие ограничения, включая защищенность и соответствие законодательству.
   - В POS-системе нужно корректно реализовать текущие политики вычисления налоговых платежей.
2. Экономические задачи.
   - Важнейшие элементы системы нужно реализовать к выставке в Гамбурге, которая состоится через 18 месяцев.
   - Обеспечить свойства, необходимые для магазинов Европы (например, настройку бизнес-правил).
3. Другие задачи.
   - Это тоже экономические задачи, но сформулированные в неявной форме. Например, "легкая расширяемость" подразумевает возможность выпуска системы каждые 6 месяцев.

Многие из этих задач в UP описываются в артефакте "В*и*дение". Их приоритеты указываются в соответствующем столбце таблицы факторов.

Отличительной особенностью принятия решений на этом высоком уровне (по сравнению с низкоуровневым объектным проектированием) является рассмотрение множества важнейших задач в комплексе. Причем в процессе принятия технических решений главную роль играют экономические задачи. Рассмотрим еще один пример.

---

**Техническое описание**

**Вопрос. Соответствие правил вычисления налоговых платежей законодательным нормам.**

**Идея решения. Приобрести отдельный компонент для вычисления налоговых платежей.**

**Факторы**

- Применение действующих правил вычисления налоговых платежей.

**Решение**

Приобрести систему вычисления налоговых платежей с соответствующей лицензией и регулярно получать обновления. Заметим, что на разных клиентских компьютерах можно использовать разные системы вычисления налоговых платежей.

**Мотивировка**

Такое решение корректно, требует минимальных затрат на поддержку и облегчает труд разработчиков. Готовые продукты стоят дорого, однако альтернативные решения неприемлемы.

**Неразрешенные вопросы**

Какие продукты имеются на рынке и какими свойствами они обладают?

**Альтернативные решения**

Можно разработать компонент самостоятельно. Однако это займет много времени, обойдется дорого и потребует дополнительных усилий разработчиков. При этом в "доморощенной" системе возможны ошибки.

---

### Приоритеты и точки эволюции: недоработанные и слишком детализированные проектные решения

Еще одной особенностью принятия архитектурных решений является выявление *точек эволюции* (evolution points) — т.е. элементов, которые в будущем могут измениться. Например, со временем могут появиться беспроводные терминалы для клиентов системы NextGen. Это обстоятельство существенно влияет на архитектуру системы, поскольку с появлением новых аппаратных средств могут измениться операционные системы, интерфейс пользователя и т.д.

На подготовку к будущим изменениям можно затратить очень много денег. Если когда-нибудь окажется, что эти затраты были напрасны, то архитектурные "упражнения" станут слишком дорогим удовольствием. Такая ситуация является результатом слишком детализированной проработки. Заметим, что прогнозирование дает очень ненадежные результаты, поэтому не стоит закладывать в архитектуру системы слишком много элементов "на будущее". Даже если впоследствии прогнозы оправдаются, систему, возможно, все равно придется дорабатывать.

Конечно, в системе нужно предусмотреть решение основных возможных проблем (в свое время такой проблемой была "Проблема 2000"). На ее решение необходимо затратить средства, поскольку недоработка обернется слишком большими затратами в будущем.

> Профессионализм архитектора состоит в правильной расстановке приоритетов и выборе наилучшего способа вложения средств для защиты от последующих изменений.

Чтобы решить, насколько серьезно нужно готовиться к возможным изменениям в будущем, необходимо взвесить возможные сценарии развития событий. Насколько потребуется изменить проектное решение и код? Каких затрат это потребует? Возможно, при ближайшем рассмотрении сложнейшая проблема окажется легко устранимой в течение нескольких человеко-недель.

Здесь уместно снова привести цитату, приписываемую Нильсу Бору: "Прогнозирование — это очень сложная задача, особенно если оно касается будущего".

## Основные принципы проектирования архитектуры

Основные принципы проектирования, рассматриваемые в этой книге, в равной степени применимы к крупномасштабному архитектурному уровню. К ним относятся следующие.

- Низкая степень связывания
- Высокое зацепление
- Защита от вариаций (интерфейсы, перенаправления и т.д.)

Однако архитектурные элементы рассматриваются в более крупном масштабе. Здесь обеспечивается низкая степень связывания приложений, подсистем и процессов, а не отдельных объектов.

Более того, на архитектурном уровне существует больше различных механизмов обеспечения качества разработки. В качестве примера рассмотрим следующее описание.

**Техническое описание**

**Вопрос. Адаптация — службы сторонних производителей.**

**Идея решения. Реализовать шаблон Protected Variations с использованием интерфейсов и адаптеров.**

**Факторы**

- Поддержка различных внешних служб, перечень которых может изменяться (систем авторизации платежей, вычисления налогов и т.д.).

**Решение**

Обеспечить защиту от вариаций следующим образом. Проанализировать различные коммерческие продукты и разработать стандартные интерфейсы. Затем на основе шаблона Adapter обеспечить перенаправление информации. Например, создать объект-адаптер ресурса, реализующий этот интерфейс и обеспечивающий взаимодействие с реальной системой вычисления налоговых платежей.

Вопросы обеспечения прозрачности такого решения рассмотрены в техническом описании "Надежность — восстановление информации при отказе удаленной службы".

**Мотивировка**

Простота. Это решение дешевле и быстрее, чем использование службы сообщений (см. раздел "Альтернативные решения"). Службу сообщений нельзя использовать для прямого взаимодействия со службой авторизации платежей.

**Неразрешенные вопросы**

Не будет ли минимальный общий интерфейс слишком ограниченным?

**Альтернативные решения**

Можно реализовать перенаправление с использованием службы сообщений или подписки (реализации JMS) между клиентским компьютером и системой вычисления налоговых платежей. Однако это решение нельзя применять для авторизации платежей, оно слишком дорого и больше подходит для доставки сообщений, а не для практических нужд приложения NextGen.

Отсюда видно, что на архитектурном уровне расширяются возможные способы защиты от вариаций. Зачастую они основаны на применении компонентов сторонних производителей, таких как JMS (Java Messaging Service) или сервер EJB.

## Разделение функций и снижение влияния

Еще одним принципом архитектурного анализа является *разделение функций* различных объектов. Этот принцип применим и для отдельных объектов, но наиболее ярко проявляется в процессе анализа архитектуры.

Существуют функции, относящиеся к приложению в целом, например, хранение данных или обеспечение безопасности. Можно, конечно, спроектировать приложение NextGen таким образом, чтобы каждый объект сам сохранял себя и взаимодействовал с базой данных. Однако в этом случае уровни логики приложения и базы данных будут слишком связаны между собой. Это же относится и к вопросам безопасности. Такие решения снижают уровень зацепления и повышают связывание.

Разделение функций относится к целым подсистемам. Объекты логики приложения должны реализовывать только логику приложения и ничего более. Подсистема взаимодействия с базой данных должна выполнять только свои функции.

Разделение функций подсистем обеспечивает низкое связывание и высокое зацепление на архитектурном уровне. Эти принципы применимы и к отдельным объектам и их обязанностям. Однако особо важное значение они имеют при про-

ектировании архитектуры приложения, поскольку здесь требуются более глобальные и важные решения.

Для разделения функций крупномасштабных компонентов существует несколько общих приемов.

1. Разделить элемент на отдельные части (например, подсистемы) и использовать их службы.
   - Это самый общий подход. Например, в системе NextGen функции взаимодействия с базой данных можно возложить на отдельную подсистему, доступ к которой от других компонентов осуществляется через фасадный объект. Многоуровневая архитектура приложения тоже иллюстрирует разделение функций.
2. Использовать объекты-декораторы.
   - Это еще один очень распространенный подход. Впервые он был использован компанией Microsoft для службы Microsoft Transaction Service, а затем в серверах EJB. В рамках этого подхода некая функция (например, обеспечение безопасности) реализуется через объект-декоратор, инкапсулирующий другие объекты. В терминологии EJB объект-декоратор называется *контейнером* (container). Например, функцию обеспечения безопасности в системе NextGen можно возложить на контейнер EJB.
3. Применить аспектно-ориентированные технологии и посткомпиляторы.
   - Например, взаимодействие с базой данных таких классов, как `Sale` можно обеспечить с помощью компонентов EJB. Характеристики класса `Sale`, сохраняемые в базе данных, задаются в файле описания свойств. Затем посткомпилятор (под которым понимается еще один компилятор, запускаемый после "обычного" компилятора) добавляет необходимую поддержку взаимодействия с базой данных для класса `Sale` или его подклассов (модифицируется только байт-код). Разработчик при этом продолжает работать с исходным классом как с объектом уровня логики приложения. Можно использовать и аспектно-ориентированную технологию, например AspectJ (`www.aspectj.org`), которая также поддерживает посткомпиляцию для обеспечения общих функций приложения. При таких подходах в процессе разработки поддерживается видимость разделения функций.

## Архитектурные шаблоны

Подробное описание архитектурных шаблонов и их применения к приложению NextGen выходит за рамки задач этой книги. Остановимся лишь на нескольких моментах.

Наиболее общим шаблоном, обеспечивающим реализацию принципов низкого связывания, высокого зацепления и разделения функций, является шаблон Layers, согласно которому функции системы разбиваются по компонентам или уровням.

Известно множество архитектурных шаблонов, и их количество постоянно растет. Список полезной литературы по архитектурным решениям приводится в конце этой главы.

## 32.6. Выводы

Вопросы выбора архитектуры связаны, в основном, с нефункциональными требованиями и включают экономические или рыночные аспекты работы приложения. При этом не нужно упускать из виду и функциональные требования (например, оформление продажи). Именно в их контексте рассматриваются главные функции приложения. Архитектурно значимым является выявление точек вариации.

Архитектурный уровень включает общесистемные крупномасштабные проблемы, устранение которых требует принятия фундаментальных проектных решений. К таким проблемам относится, в частности, выбор сервера приложений.

В процессе архитектурного анализа нужно учитывать взаимозависимость факторов. Например, повышение безопасности связано с производительностью и удобством использования системы, большинство факторов связаны с повышением стоимости.

В процессе архитектурного анализа нужно оценивать все возможные решения. Опытный архитектор может предложить проектные решения, связанные с разработкой новых программных компонентов и покупкой готовых программ или аппаратных средств. Например, восстановление информации на удаленном сервере POS-системы NextGen можно обеспечить за счет репликации и служб защиты от сбоев, обеспечиваемых некоторыми операционными системами и аппаратными средствами. Хороший архитектор должен знать о таких продуктах.

Таким образом, в процессе архитектурного анализа нужно выявить крупномасштабные проблемы и решить их.

> Архитектурный анализ связан с идентификацией и реализацией нефункциональных (качественных) требований к приложению в контексте функциональных требований.

## 32.7. Архитектурный анализ в UP

### *Предупреждение: каскадный архитектурный анализ*

В книгах по архитектурному анализу зачастую рассматривается каскадный метод проектирования архитектуры, выполняемый до начала стадии реализации. В контексте UP и итеративной разработки такой анализ проводится на каждой итерации с учетом обратной связи и адаптации. Наиболее сложные элементы системы, связанные с самым высоким риском, реализуются на ранних итерациях. Затем на основе обратной связи по мере углубления понимания проблемы они настраиваются и модифицируются.

### *Архитектурная информация в артефактах UP*

- Архитектурные факторы описываются в дополнительной спецификации, например в виде таблицы.
- Архитектурные решения описываются в документе SAD. Туда заносятся технические описания и описания архитектурных представлений.

### *Документ SAD и архитектурные представления*

Помимо диаграмм пакетов UML, классов и последовательностей, важным артефактом модели проектирования в рамках UP является документ SAD. В нем

описываются важнейшие архитектурные решения. С его помощью разработчики могут ознакомиться с основными идеями, заложенными в систему.

Если в группу приходит новый разработчик, менеджер проекта может сразу направить его на Web-узел проекта для ознакомления с документом SAD. Из этого документа можно быстро получить общие сведения о системе.

Поэтому при создании этого документа нужно задать себе вопрос: что бы я рассказал новому члену группы разработчиков, чтобы помочь ему поскорее войти в курс дела?

В документе SAD описываются главные технические решения и архитектурные представления UP.

### Архитектурные представления

Разработка архитектуры — это одна сторона вопроса, а ее описание — другая.

В [73] предложены несколько представлений архитектуры. Основная идея *архитектурного представления* сводится к следующему.

> *Архитектурное представление*
>
> Это представление архитектуры системы, в котором основное внимание уделяется структуре, модулям, главным компонентам и потокам управления [97].
>
> Важным аспектом такого представления является *мотивировка*, объясняющая сделанный выбор.
>
> Архитектурное представление — это взгляд на систему с определенных позиций, когда выделяются только основные идеи, а остальные отбрасываются.

Архитектурное представление — это средство общения, изучения системы и ее осмысления. Оно создается в виде текстового описания и диаграмм UML.

В UP существует 6 основных представлений архитектуры (но можно добавить и другие, например, представление с позиций безопасности)[3]. Все их составлять необязательно, но рекомендуется создать логическое представление, представления прецедентов и процессов, а также представление развертывания. Вот эти 6 представлений.

1. **Логическое представление**
   - Отражает структуру программной системы в терминах уровней, подсистем, пакетов, контуров, наиболее важных классов и интерфейсов. В нем обобщается функциональность основных программных элементов, таких как подсистемы.
   - Отражает реализацию важных прецедентов (подобно диаграмме взаимодействия), иллюстрирующих основные аспекты работы системы.
   - Это представление модели проектирования UP, построенное с использованием обозначений пакетов UML, классов и диаграмм взаимодействия.
2. **Представление процессов**
   - Отражает процессы и потоки, их обязанности, взаимодействие и размещение логических элементов (уровней, классов, подсистем и т.д.).

---

[3] В ранних версиях UP описывались “4+1” представлений, однако впоследствии их стало 6 [73].

- Это представление модели проектирования UP, построенное с использованием обозначений пакетов UML, классов, диаграмм взаимодействия, а также процессов и потоков.

3. **Представление развертывания**

- Отражает физическое распределение процессов и компонентов по процессорам и физическую конфигурацию сети.
- Это представление модели развертывания UP, построенное с использованием обозначений диаграмм развертывания UML. В этом представлении обычно отображается вся система, а не ее подмножество. Система обозначений диаграммы развертывания UML описана в главе 38.

4. **Представление данных**

- Отражает схему построения базы данных и преобразования объектного представления данных в реляционное, описывает хранимые процедуры и триггеры базы данных.
- Это представление модели данных UP, построенное с использованием обозначений диаграмм классов UML.

5. **Представление прецедентов**

- Краткое описание наиболее архитектурно важных прецедентов и их нефункциональных требований. Реализация таких прецедентов затрагивает многие архитектурно значимые элементы системы. В качестве примера можно привести прецедент `Оформление продажи`.
- Это представление модели прецедентов UP, построенное с использованием обозначений диаграмм прецедентов UML.

6. **Представление реализации**

- Вспомним сначала определение модели реализации. В отличие от других моделей UP, представляющих собой диаграммы и текстовые описания, эта "модель" включает исходный и исполняемый код системы. Она состоит из двух частей: исполняемого кода и исходного кода с графическими ресурсами. В модель реализации включаются также Web-страницы, библиотеки DLL и другие элементы системы, в том числе пакеты Java и байт-код в виде файлов JAR.
- В представление реализации включают краткое описание организации исходного и исполняемого кода системы.
- Это представление модели реализации UP, состоящее из текстового описания и диаграмм пакетов и компонентов UML.

Например, диаграммы пакетов и взаимодействия для системы NextGen, приведенные в главе 30, отражающие уровни приложения и его логическую архитектуру, иллюстрируют основные идеи, заложенные в архитектуру программной системы. В документе SAD нужно создать раздел "Логическое представление", добавить в него эти диаграммы и дополнить их текстовыми комментариями по каждому уровню и пакету. В этот документ нужно также включить представления процессов и развертывания.

Главное понять, что архитектурные представления (представляющие собой текстовые описания и диаграммы) не должны отражать все элементы системы.

Они иллюстрируют только ключевые идеи основных проектных решений. Это те сведения, которые бы архитектор системы сообщил новому разработчику при первом знакомстве.

Архитектурные представления можно разрабатывать на следующих этапах.

- После создания системы для следующих "поколений" разработчиков.
- После завершения очередного этапа разработки (например, в конце фазы развития) для обучения разработчиков системы и новых членов команды.
- На ранних итерациях, в рамках проектирования системы.

### Примерная структура документа SAD

---

**Описание архитектуры программы**

**Архитектурное представление**

(Краткое описание структуры этого документа. Это полезно для читателей, не знакомых с идеей технических описаний или представлений. Заметим, что все представления описывать не обязательно.)

**Архитектурные факторы и решения**

(Сослаться на таблицу факторов в дополнительной спецификации. Привести технические описания основных решений.)

**Логическое представление**

(Диаграммы UML пакетов, классов и основных элементов. Основные элементы нужно прокомментировать.)

**Представление процессов**

(Диаграммы UML классов и взаимодействия, иллюстрирующие процессы и потоки в системе. Взаимосвязанные потоки и процессы нужно сгруппировать, а их совместную работу прокомментировать (например, с помощью RMI).)

**Представление прецедентов**

(Краткое описание наиболее архитектурно важных прецедентов. Диаграммы взаимодействия UML для реализаций некоторых архитектурно значимых прецедентов или их сценариев с комментариями относительно основных архитектурных элементов.)

**Представление развертывания**

(Диаграммы развертывания UML, отражающие распределение процессов и компонентов по узлам сети с комментариями по организации сети.)

---

## *Фазы разработки*

**Начало.** Если непонятно, насколько технически реально удовлетворить архитектурно значимые требования, можно реализовать *проверку архитектурной концепции* (architectural proof-of-concept). В UP эта процедура называется *архитектурным синтезом* (architectural synthesis). Она отличается от принятых в прежние времена программных экспериментов, связанных с реализацией отдельных технических решений. Архитектурный синтез позволяет проверить выполнимость архитектурно значимых требований в комплексе.

**Развитие.** Основная задача этой фазы — реализация наиболее сложных архитектурных элементов, поэтому основные усилия по архитектурному анализу затрагиваются именно на этой стадии. К ее завершению должны быть составлены таблица факторов, технические описания и документ SAD.

**Передача.** Хотя в идеале все архитектурные решения должны быть реализованы задолго до этой фазы, следует просмотреть документ SAD, сверить описанную в нем архитектуру с реальной и удостовериться в корректности описания системы.

**Последующая эволюция системы.** Перед переходом к разработке новых версий системы следует еще раз оценить архитектурные факторы и решения. Например, решение по использованию в версии 1.0 единственного экземпляра системы вычисления налоговых платежей, вызванное в свое время соображениями экономии, может быть пересмотрено, поскольку со временем такие системы могут подешеветь. Тогда в последующих версиях системы можно использовать локальные службы вычисления налоговых платежей.

## 32.8. Дополнительная литература

Перечень литературы по архитектурным шаблонам и общим принципам создания систем постоянно дополняется. Автор советует ознакомиться с работами [9, 15, 25]. По архитектурным шаблонам много информации представлено в Internet, в частности на Web-узлах компаний Sun и IBM, а также на Web-узле института SEI (Software Engineering Institute) по адресу `www.sei.cmu.edu`.

Глава 33

# Реализация новых прецедентов на основе объектов и шаблонов

*Члены парламента меня дважды спрашивали: "Мистер Бабедж, выдаст ли ваша машина правильные ответы, если ввести в нее неверные данные?" Я не могу представить, какая цепочка мыслей привела их к такому вопросу.*

*Чарльз Бабедж (Charles Babbage)*

**Основная задача**

- Применить для проектирования шаблоны GRASP и GoF.

## Введение

В этой главе рассматриваются проектные решения для текущей итерации, призванные удовлетворить такие требования, как обработка отказа локальных служб, обслуживание аппарата с POS-системой и авторизация платежей.

## 33.1. Отказ локальных служб и обеспечение локальной буферизации

Требования к приложению NextGen предусматривают защиту от отказов удаленных служб, например временной недоступности базы данных. Сначала рассмотрим проектное решение, обеспечивающее обработку отказа и восстановление информации о товарах. Затем исследуем доступ к бухгалтерской системе, для обеспечения работоспособности которой предлагается несколько другая стратегия.

Напомним фрагмент технического описания.

### Техническое описание

**Вопрос. Надежность — восстановление информации при отказе удаленной службы.**

**Идея решения. Обеспечить прозрачный поиск нужной службы, переход от удаленной службы к локальной и осуществить частичную репликацию локальной службы.**

**Факторы**

- Робастное восстановление при выходе из строя удаленной службы (системы складского учета, системы вычисления налоговых платежей).
- Робастное восстановление информации при выходе из строя удаленной базы данных с описаниями товаров.

**Решение**

Реализовать шаблон Protected Variations (Защищенные вариации) для защиты от влияния местоположения службы с помощью интерфейса адаптера и объекта `ServicesFactory`. По возможности обеспечить локальную реализацию удаленных служб с упрощенным и ограниченным поведением. Например, локальная система вычисления налогов может использовать фиксированные ставки налогов. Локальная база данных товаров может представлять собой небольшой буфер с информацией о наиболее популярных товарах. Информацию для обновления системы складского учета можно хранить локально и передавать удаленной системе после восстановления соединения.

Вопросы конкретной реализации этих решений описаны в разделе технической документации "Адаптация — службы сторонних производителей", поскольку реализации удаленных служб могут изменяться.

Для обеспечения качественного восстановления соединения с удаленными службами используйте объекты-посредники, проверяющие дееспособность каждой удаленной службы и перенаправляющие по возможности информацию напрямую удаленным службам.

**Мотивировка**

Владельцы магазинов не желают прекращать торговлю. Следовательно, в системе NextGen нужно предусмотреть высокий уровень надежности и возможность восстановления информации при сбоях. Тогда она станет очень привлекательным продуктом, поскольку конкурирующие системы такой возможности не обеспечивают.

---

Прежде чем приступать к реализации защиты от сбоев, заметим, что архитектор (в данном случае автор) рекомендует использовать локальный буфер (базу данных на локальном жестком диске в виде простого файла) объектов `ProductSpecification`. Это позволяет повысить производительность и надежность системы. Следовательно, прежде чем обращаться к удаленной базе данных, нужно всегда выполнять поиск товара в этом буфере.

Это можно легко реализовать на основе шаблонов Adapter и Factory.

1. Объект-фабрика `ServicesFactory` всегда возвращает адаптер для локальной службы информации о товарах.
2. Локальный "адаптер" для товаров на самом деле не является адаптером для других компонентов, а сам выполняет обязанности локальной службы.
3. Локальная служба инициализируется ссылкой на другой адаптер для реальной удаленной службы.
4. Если данные о товаре содержатся в локальном буфере, то они и возвращаются. В противном случае запрос перенаправляется адаптеру внешней службы.

Обратите внимание на двухуровневый буфер в клиентской части приложения.

1. Расположенный в оперативной памяти объект `ProductCatalog` будет поддерживать коллекцию (например, объект `HashMap` для Java) нескольких (например, 1000) объектов `ProductSpecification`, содержащих информацию о товарах. Размер коллекции можно настраивать в зависимости от объема оперативной памяти компьютера.
2. Локальные службы информации о товарах будут поддерживать более мощный (расположенный на жестком диске) буфер с данными о большем количестве товаров (занимающий от 1 до 100 Мбайт дискового пространства). Размер этого буфера тоже можно настраивать в зависимости от конфигурации локального компьютера. Этот буфер играет важную роль для защиты от сбоев, поскольку не выходит из строя даже при сбоях в самом приложении NextGen.

Такое проектное решение не приводит к необходимости переделки существующего кода — новый локальный объект службы можно добавить, не затрагивая объект `ProductCatalog` (взаимодействующий со службой информации о товарах).

Здесь не применялись новые шаблоны. Решение основано на шаблонах Adapter и Factory.

На рис. 33.1 показано описанное проектное решение, а на рис. 33.2 проиллюстрирован процесс инициализации.

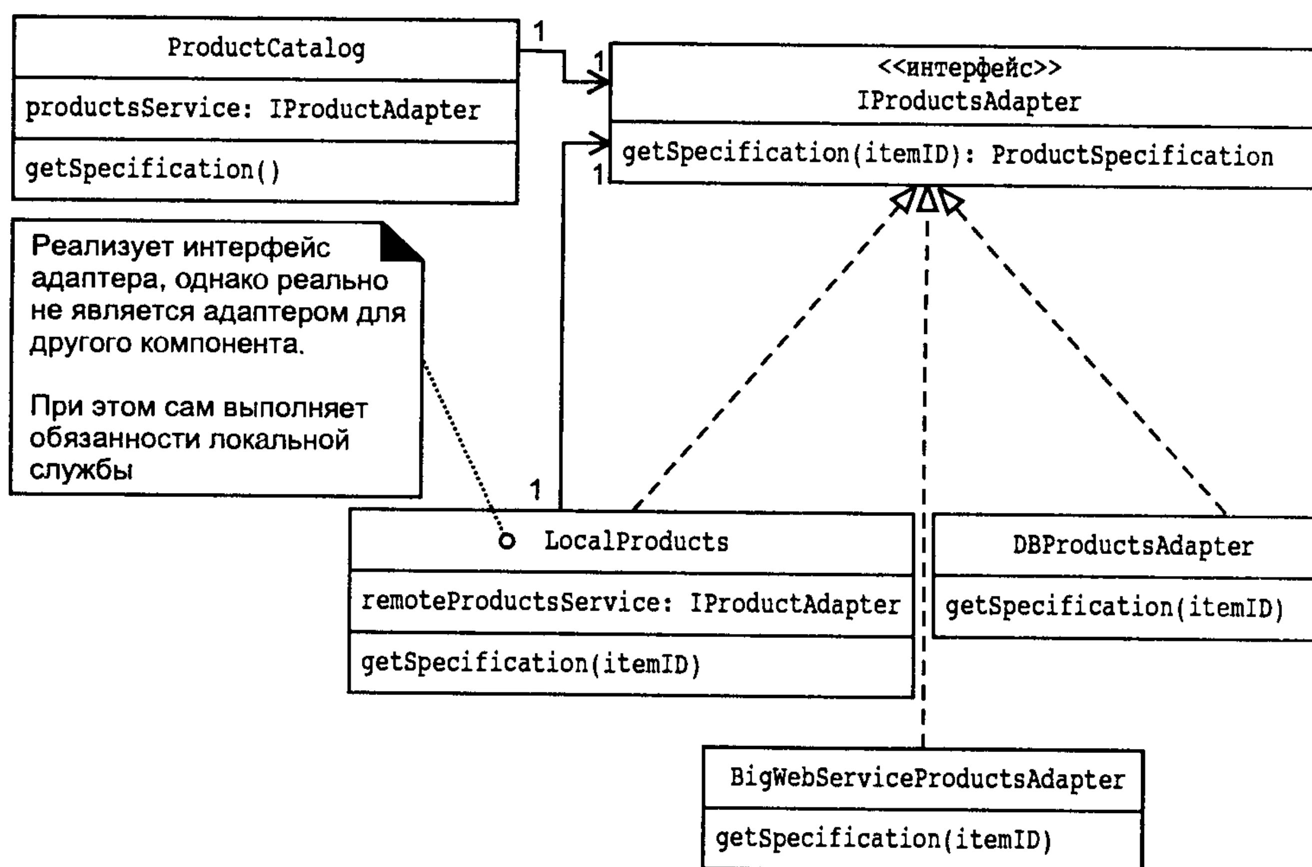



*Рис. 33.1. Адаптеры для информации о товарах*

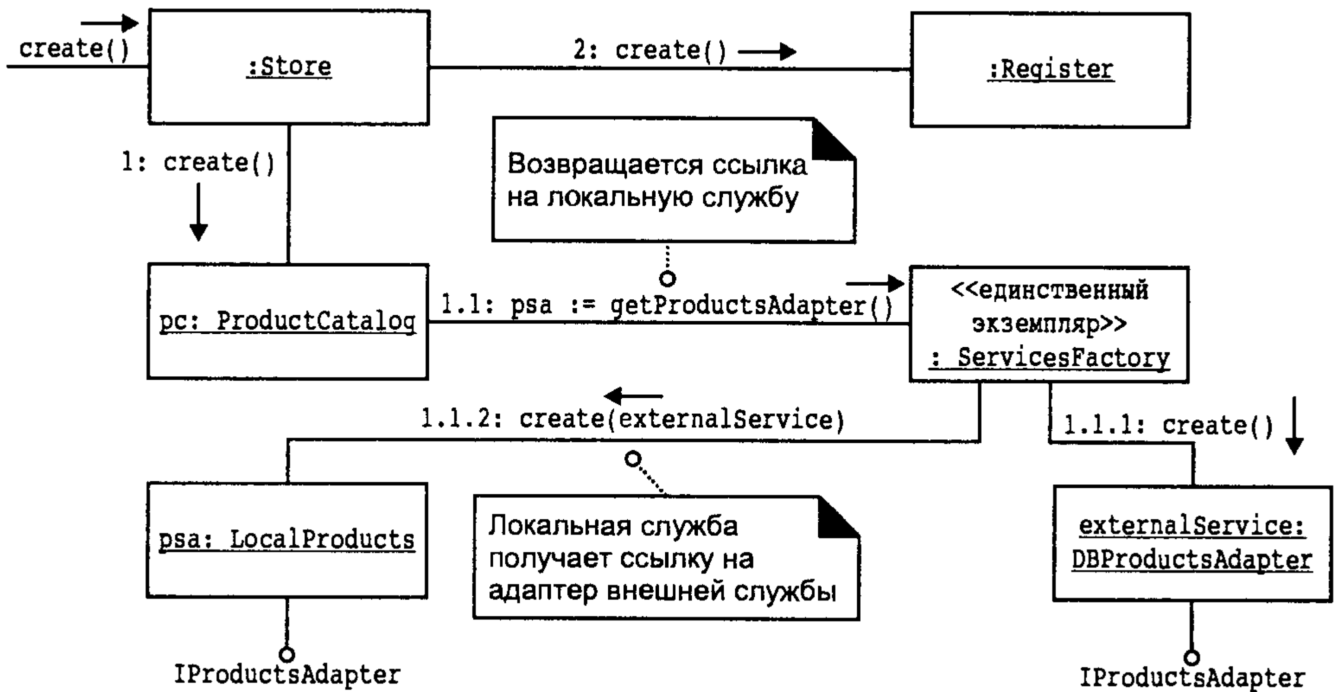



*Рис. 33.2. Инициализация службы информации о товарах*

На рис. 33.3 показано исходное взаимодействие объекта-каталога со службой информации о товарах.

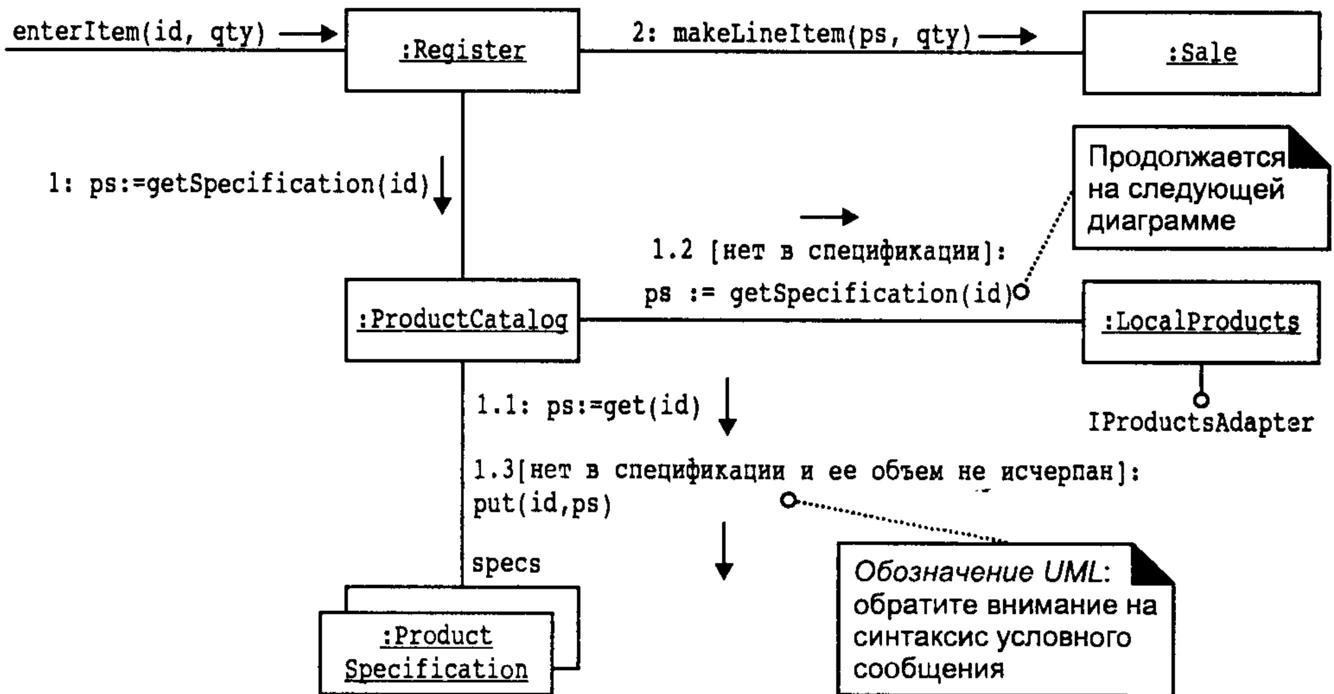



*Рис. 33.3. Начало взаимодействия объекта-каталога со службой информации о товарах*

Если в буфере локальной службы информация о товаре отсутствует, происходит обращение к адаптеру внешней службы (рис. 33.4). Обратите внимание, что локальная служба товаров кэширует объекты `ProductSpecification` как обычную последовательность объектов.

При изменении архитектуры внешней службы (переходе от обычной базы данных к новой Web-службе) потребуется переделать только конфигурацию объекта-фабрики удаленных служб (рис. 33.5).

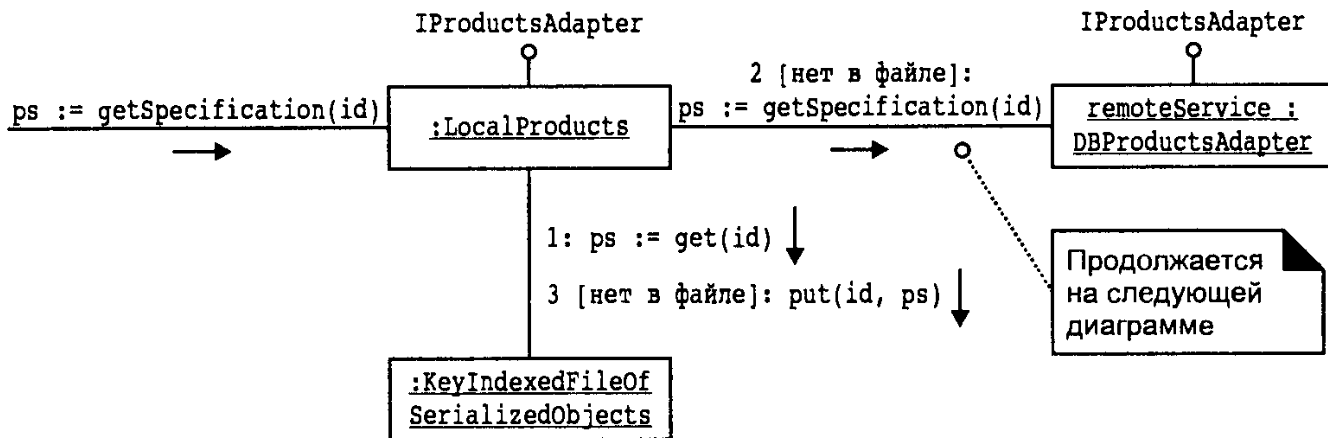



*Рис. 33.4. Продолжение взаимодействия со службой информации о товарах*

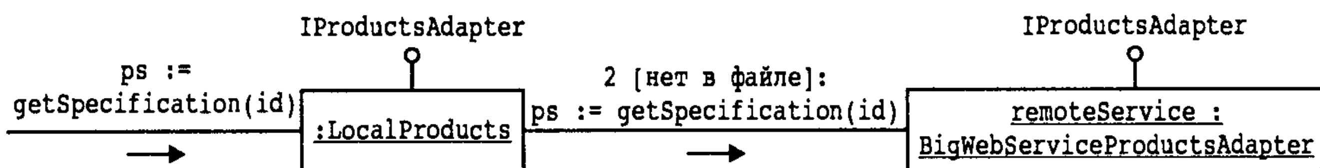



*Рис. 33.5. Новые внешние службы не влияют на существующее проектное решение*

Далее объект DBProductsAdapter будет взаимодействовать с подсистемой объектно-реляционного преобразования (рис. 33.6).

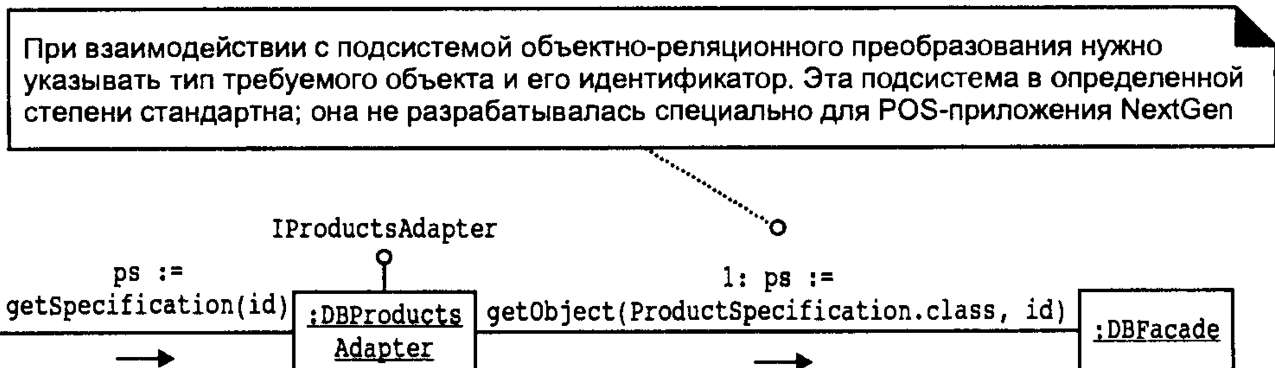



*Рис. 33.6. Взаимодействие с подсистемой базы данных*

### Стратегии кэширования

Рассмотрим возможные варианты загрузки оперативного буфера ProductCatalog и файлового буфера LocalProducts. Для этого можно использовать пассивную инициализацию, при которой буферы заполняются постепенно по мере поступления информации от внешних служб. Другой подход подразумевает активную инициализацию, при которой оба буфера заполняются в рамках прецедента Запуск системы. Если разработчик еще не определился с выбором подхода и хочет поэкспериментировать с обоими, то для решения этой проблемы можно воспользоваться семейством различных объектов CacheStrategy и шаблоном Strategy.

### Устаревший буфер

Поскольку цены на товары изменяются довольно часто, причем иногда просто по прихоти менеджера магазина, значение цены в буфере может устареть. Это общая проблема репликации данных. Решение этой проблемы — добавление

операции удаленной службы, отслеживающей ежедневные изменения. Объект `LocalProducts` каждые n минут может отправлять запрос и обновлять буфер.

### Потоки в UML

Если объект `LocalProducts` должен решать проблему поддерживания актуального состояния данных, то его можно сделать *активным объектом* (active object) — владельцем процесса управления. Этот процесс может засыпать на n минут, просыпаться, обеспечивать получение данных и снова засыпать. Для отображения процессов и асинхронных вызовов в UML имеется специальное обозначение (рис. 33.7).

На диаграмме взаимодействия экземпляр активного объекта может быть снабжен свойством `{активный}`. На диаграмме классов класс активных объектов (*активный класс*) — владелец собственного процесса помечается стереотипом `<<поток>>` (рис. 33.8).

## 33.2. Обработка отказов

В предыдущем разделе описан способ кэширования объектов `ProductSpecification` в отдельном файле в клиентской части приложения. Это повышает производительность системы и обеспечивает частичное решение проблемы защиты от сбоев внешней службы информации о товарах. Поскольку в файле локального компьютера можно хранить данные о 10000 товаров, то такой объем информации может удовлетворить большую часть запросов в случае отказа внешней службы.

Но что делать, если внешняя служба не работает, а в локальном файле нужная информация отсутствует? Допустим, в этом случае необходимо уведомить кассира о необходимости ввести цену товара и его описание вручную либо отменить ввод информации об этом товаре вообще.

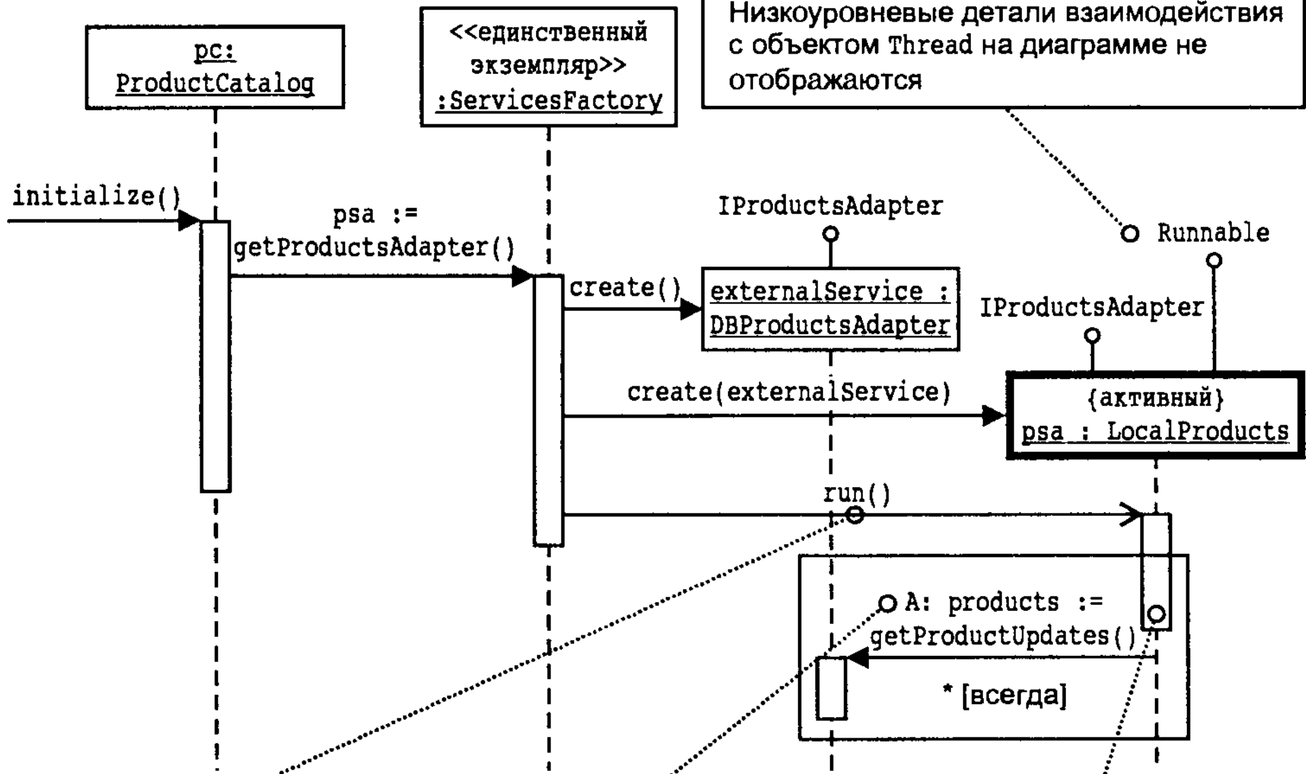
Это пример условия ошибки или сбоя, который можно использовать для описания некоторых общих шаблонов обработки сбоев и исключений. Обработка ошибок или исключений — отдельный вопрос, поэтому здесь мы сконцентрируемся лишь на некоторых шаблонах, относящихся к данной проблеме. Сначала введем несколько терминов.

- **Повреждение** (fault) — первопричина неправильного поведения.
  - Программист перепутал имя базы данных.
- **Ошибка** (error) — выявление сбоя в работающей системе. Ошибки определяются (или нет).
  - При обращении к службе имен для получения ссылки на базу данных (с неправильным именем) она сигнализирует об ошибке.
- **Отказ** (failure) — отказ в обслуживании, вызванный ошибкой.
  - Подсистема информации о товарах не отвечает.

Обратите внимание, что активный объект со своим собственным потоком выделяется жирной линией. Свойство {активный} является необязательным, однако его рекомендуется использовать.

Например, в Java активные объекты реализуют интерфейс Runnable. Кроме того, для каждого интерфейса используется отдельное условное обозначение.

Низкоуровневые детали взаимодействия с объектом Thread на диаграмме не отображаются




В Java вызов метода run класса Thread или интерфейса Runnable рассматривается как асинхронное сообщение.

В UML такие сообщения изображаются с помощью стрелки специального вида. Обратите внимание, что это изменение появилось в UML версии 1.4

Если методы запускаются в другом потоке, соответствующее выражение на диаграмме последовательностей UML может начинаться с имени или символьного обозначения потока. Этот символ применяется для указания потока и является необязательным, но позволяет добиться дополнительного визуального эффекта.

Например, все сообщения обрабатываемые в потоке LocalProducts, будут помечаться символом A

```
//это сообщение обрабатывается
//в своем собственном потоке
{
всегда выполняется циклически:
  - "засыпает" иа n минут
  - обновляет буфер
}
```

*Рис. 33.7. Потоки и асинхронные сообщения в UML*

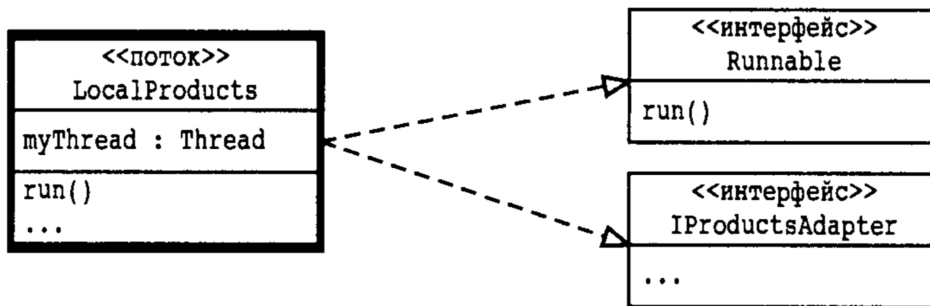



*Рис. 33.8. Обозначение активного класса*

## Генерирование исключений

Самый распространенный способ сигнализировать об отказе — генерирование исключений.

> Исключения чаще всего применяются для обработки отказов ресурсов (диска, памяти, отсутствия доступа к сети, базе данных или другим внешним службам).

Исключение генерируется в подсистеме взаимодействия с базой данных (на самом деле, в реализации JDBC для Java), когда впервые обнаруживается невозможность использования внешней базы данных. При генерации исключения вызывается его обработчик.[1]

Допустим, сгенерировано исключение `java.sql.SQLException` (для Java). Нужно ли передавать это исключение верхним уровням системы, вплоть до уровня представления? Нет, этого не делают. Для обработки исключений используют следующий шаблон.

> *Шаблон Convert Exceptions (Конвертирование исключений)* [23]
>
> В рамках подсистемы избегайте дальнейшей передачи исключений, поступающих от подсистем или служб более низкого уровня. Преобразуйте исключения низкого уровня в форму, соответствующую уровню данной подсистемы. Исключения более высокого уровня обычно включают исключения более низких уровней и добавляют информацию, делающую исключение контекстно-значимым для более высокого уровня.
>
> Это рекомендация, а не жесткое правило.
>
> Термин "исключение" используется здесь в смысле генерируемой информации.
>
> Этот шаблон известен также под названием Exception Abstraction (Абстрагирование исключений) [91].

Например, подсистема взаимодействия с базой данных перехватывает некоторое исключение `SQLException` и (в предположении, что сама его не обрабатывает[2]) генерирует новое исключение `DBUnavailableException`, содержащее

[1] Вопросы проверки исключений здесь не обсуждаются, поскольку она поддерживается не во всех популярных объектно-ориентированных языках, например, в C++, C# и Smalltalk проверка отсутствует.

[2] Желательно разрешать исключения на уровне, близком к уровню его генерации, однако это достаточно трудная задача, поскольку способ обработки исключений обычно определяется спецификой приложения.

SQLException. Заметим, что интерфейс DBProductAdapter выступает в роли фасадного объекта для логической подсистемы обеспечения информации о товарах. Поэтому на более высоком уровне DBProductAdapter перехватывает исключение DBUnavailableException более низкого уровня и (в предположении, что сам его не обрабатывает) генерирует новое исключение ProductInfoUnavailableException, включающее в себя DBUnavailableException.

Обратите внимание на имена исключений. Почему исключение названо DBUnavailableException, а не, скажем, PersistenceSubsystemException? Это объясняется следующим шаблоном.

*Шаблон Name The Problem Not The Thrower (Использование имени проблемы, а не ее генератора) [58]*

Как именовать исключения? Нужно выбирать имена, описывающие саму проблему, а не ее генератор. Тогда программистам легче понять проблему и единообразно отразить ее суть во многих классах исключений (что невозможно при использовании имен генераторов исключений).

### Исключения в UML

Самое время перейти к обозначениям UML для генераторов[3] и обработчиков исключений.

При обсуждении обозначений UML возникает два вопроса.

1. Как отобразить классы, генерирующие и перехватывающие исключения, на диаграмме классов?
2. Как отобразить генерацию исключений на диаграмме взаимодействия?

Обозначения для диаграммы классов показаны на рис. 33.9.

В UML исключение является частным случаем сигналов, описывающих асинхронное взаимодействие между объектами. Это означает, что на диаграмме взаимодействия исключения изображаются как *асинхронные сообщения*.[4]

На рис. 33.10 показано обозначение для исключений на примере преобразования исключения SQLException в DBUnavailableException.

В целом, для отображения исключений в UML существует специальное обозначение, но оно используется очень редко.

Это не значит, что следует забыть о приведенной выше информации об обработке исключений. Совсем наоборот. На архитектурном уровне с самого начала следует предусмотреть основные шаблоны, политики и способы взаимодействия объектов по обработке исключений, поскольку на поздних стадиях разработки сделать это очень сложно. Однако низкоуровневую обработку исключений многие разработчики оставляют на потом, до стадии программирования, а не построения диаграмм UML.

---

[3] Официально в UML исключения "передаются", однако термин "генерируются" является более стандартным.

[4] Обратите внимание, что начиная с версии UML 1.4 изменилось обозначение для асинхронного сообщения.

*Обозначение UML*: в UML для обозначения операций имеется синтаксис по умолчанию. Однако в него официально не включены средства отображения исключений, сгенерированных операцией. Для разрешения этой ситуации существует как минимум три возможности.

1. Язык UML позволяет использовать для операций синтаксис любого другого языка, такого как Java. Кроме того, некоторые CASE-средства с поддержкой UML позволяют явно представлять операции с применением синтаксиса Java. Например,

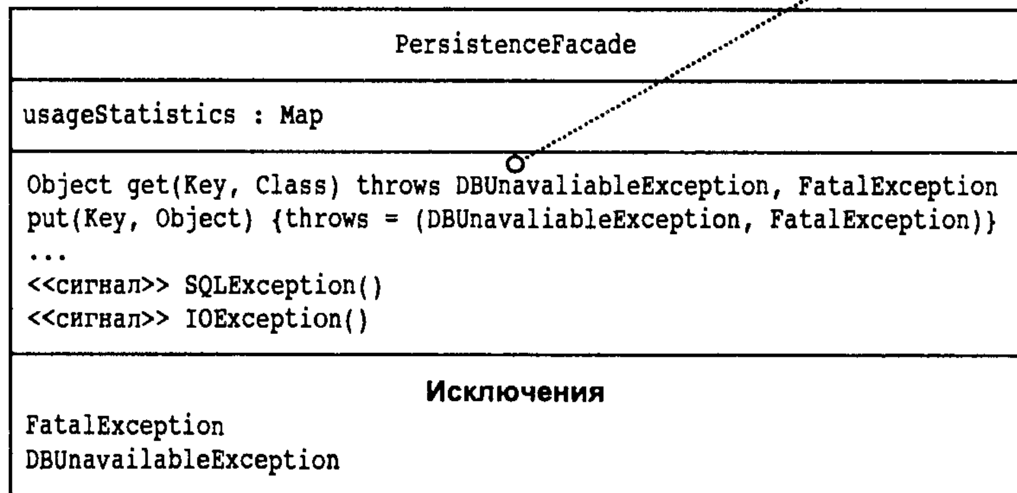
*Object get(Key, Class) throws DBUnavailableException, FatalException*

2. Предлагаемый по умолчанию синтаксис допускает применение "строки свойств" в качестве последнего элемента. Она представляет собой произвольный список пар свойство/значение, например {автор=Крэг, дети = (Анна, Халли)}. Таким образом,

*put(Object, id) {throws = (DBUnavailableException, FatalException)}*

3. Некоторые CASE-средства с поддержкой UML позволяют задавать (в специальном диалоговом окне) исключения, генерируемые операциями

Перехват исключения можно смоделировать как специальную операцию обработки сигналов




Сгенерированные исключения можно перечислить в отдельном разделе с меткой "исключения"

*Рис. 33.9. Перехват и генерация исключений некоторым классом*

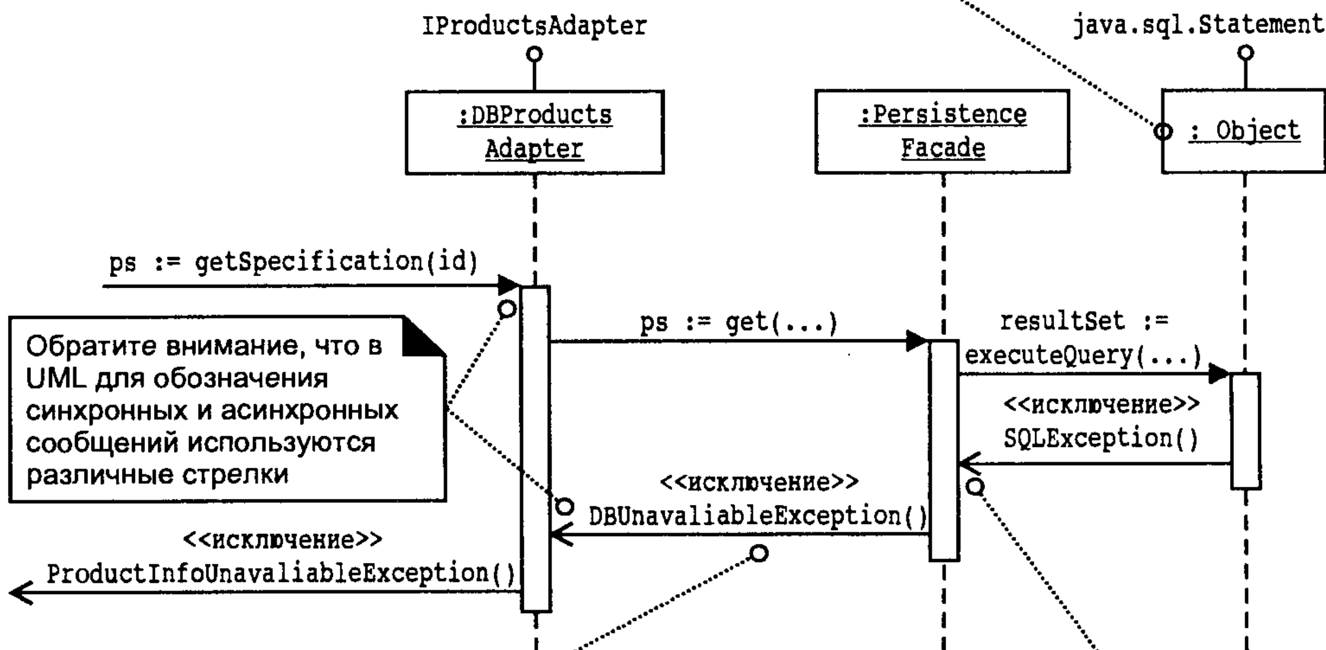



*Рис. 33.10. Исключения на диаграмме взаимодействия*

## Обработка ошибок

До сих пор мы рассматривали одну сторону вопроса — генерацию исключений с точки зрения преобразования исключений, их именования и отображения. Теперь перейдем к задаче обработки исключений.

Для этого чаще всего применяются два шаблона.

> *Шаблон Centralized Error Logging (Централизованная регистрация ошибок) [91]*
>
> Для централизованной регистрации ошибок используется единственный экземпляр, которому сообщается обо всех исключениях в системе. В распределенных системах локальные объекты, соответствующие шаблону Singleton, взаимодействуют с центральным журналом регистрации ошибок. Это дает следующие преимущества.
>
> ■ Преемственность сообщений

- Гибкость в определении выходного потока и формата данных

Этот шаблон известен также под именем Diagnostic Logger (Журнал диагностики) [60].

Это достаточно простой шаблон. Существует еще один.

*Шаблон Error Dialog (Диалоговое окно ошибок) [91]*

Для уведомления пользователя об ошибках используется стандартный независимый от приложения неграфический объект, соответствующий шаблону Singleton. Он включает несколько объектов-окон графического интерфейса (модальное диалоговое окно, текстовый консольный объект, звуковой сигнал или генератор речи) и делегирует обязанность уведомления об ошибках объектам пользовательского интерфейса. Тогда выходное сообщение может передаваться и диалоговому окну, и генератору голосового сообщения. Этот же объект уведомляет об ошибке централизованный журнал регистрации ошибок. Соответствующие объекты интерфейса пользователя может генерировать объект-фабрика на основе системных параметров. Такой подход дает следующие преимущества.

- Реализуется шаблон Protected Variations в смысле защиты от влияния изменений в механизме вывода.
- Поддерживается общий стиль отчета об ошибках. Например, этот объект может вызывать окна графического интерфейса для отображения сообщений об ошибках.
- Обеспечивается централизованное управление общей стратегией уведомления об ошибках.
- Минимальные потери производительности. При использовании "дорогостоящих" ресурсов, в частности диалогового окна, его можно скрыть и хранить в буфере для последующего использования, а не создавать диалоговое окно при возникновении каждой ошибки.

Должны ли объекты интерфейса пользователя (например, `ProcessSaleFrame`) обрабатывать ошибку путем перехвата исключения и уведомления пользователя? Для приложений с небольшим количеством окон и простым способом навигации между ними такой подход применим. Это относится и к приложению NextGen.

Однако следует помнить, что при такой обработке ошибок часть "логики приложения" переносится на уровень пользовательского интерфейса. Об обработке ошибок нужно уведомлять пользователей, это вполне логично, но этот процесс нужно отслеживать. Это не проблема для приложений с простым графическим интерфейсом и низкой вероятностью его изменения, однако это может вызвать проблемы в дальнейшем. Допустим, решено заменить интерфейс Java Swing на контур пользовательского интерфейса IBM Java MicroView. Тогда придется определить, где в интерфейсе Swing задействована логика приложения, и перенести ее в версию MicroView. По сравнению с заменой интерфейса пользователя эта задача может показаться незначительной, однако при усложнении логики приложения она может превратиться в проблему. Вообще, чем больше логики приложения передается на уровень пользовательского интерфейса, тем сложнее поддерживать систему.

Для систем с множеством окон и сложными (возможно даже изменяемыми) переходами между ними существуют другие решения. Например, между уровнем представления и предметной областью можно добавить уровень приложения с одним или несколькими контроллерами.

Более того, можно добавить объект управления представлением [15, 52], содержащий ссылки на все открытые окна и обладающий информацией о переходах между ними при реализации некоторого события (например, ошибки).

Этот объект по существу представляет собой конечный автомат, инкапсулирующий состояния (открытые окна) и переходы между ними на основе событий. Он может считывать модель перехода состояний из внешнего файла, осуществляя навигацию на основе данных (не требующую изменения исходного кода). Он также может закрывать окна приложения, сворачивать или упорядочивать их.

При таком проектном решении контроллер уровня приложения может иметь ссылку на этот объект управления представлением (тогда он будет связан с вышестоящим уровнем представления). Контроллер приложения может перехватывать исключение и взаимодействовать с менеджером представления для уведомления пользователя (на основе шаблона Error Dialog). В этом случае обработка ошибок выносится за пределы графического интерфейса.

Детальное рассмотрение вопросов управления интерфейсом пользователя не входит в задачи данной книги. Общее проектное решение на основе шаблона Error Dialog показано на рис. 33.11.

## 33.3. Взаимодействие с локальными службами на основе шаблона Proxy (GoF)

Взаимодействие с локальной службой информации о товарах было реализовано за счет обращения к этой локальной службе перед обращением к внешней службе. Однако такое решение применимо не всегда. Иногда сначала следует попытаться обратиться к внешней службе, а уже затем к ее локальной версии. Например, рассмотрим вопрос передачи информации о продаже в бухгалтерскую систему. Эти сведения нужно передать как можно скорее, чтобы в реальном времени отслеживать деятельность магазина.

Для решения этой проблемы существует еще один шаблон GoF — Proxy (Объект-посредник). Это простой шаблон, чаще всего используемый в виде Remote Proxy (Удаленный объект-посредник). Например, в пакете RMI и спецификации CORBA для доступа к удаленным объектам служб используется локальный клиентский объект (называемый "опорным"). Этот опорный объект является локальным объектом-посредником или представителем удаленного объекта.

В приложении NextGen используется другой, отличный от Remote Proxy вариант шаблона Proxy — Redirection Proxy (Посредник для перенаправления), известный также под именем Failover Proxy (Посредник для устранения проблем).

Независимо от варианта, идея шаблона остается одинаковой. Отличия проявляются после вызова объекта-посредника.

Объект-посредник — это объект, реализующий тот же интерфейс, что и основной объект, содержащий ссылку на этот объект и применяемый для управления доступом к нему. Общая структура объектов показана на рис. 33.12.

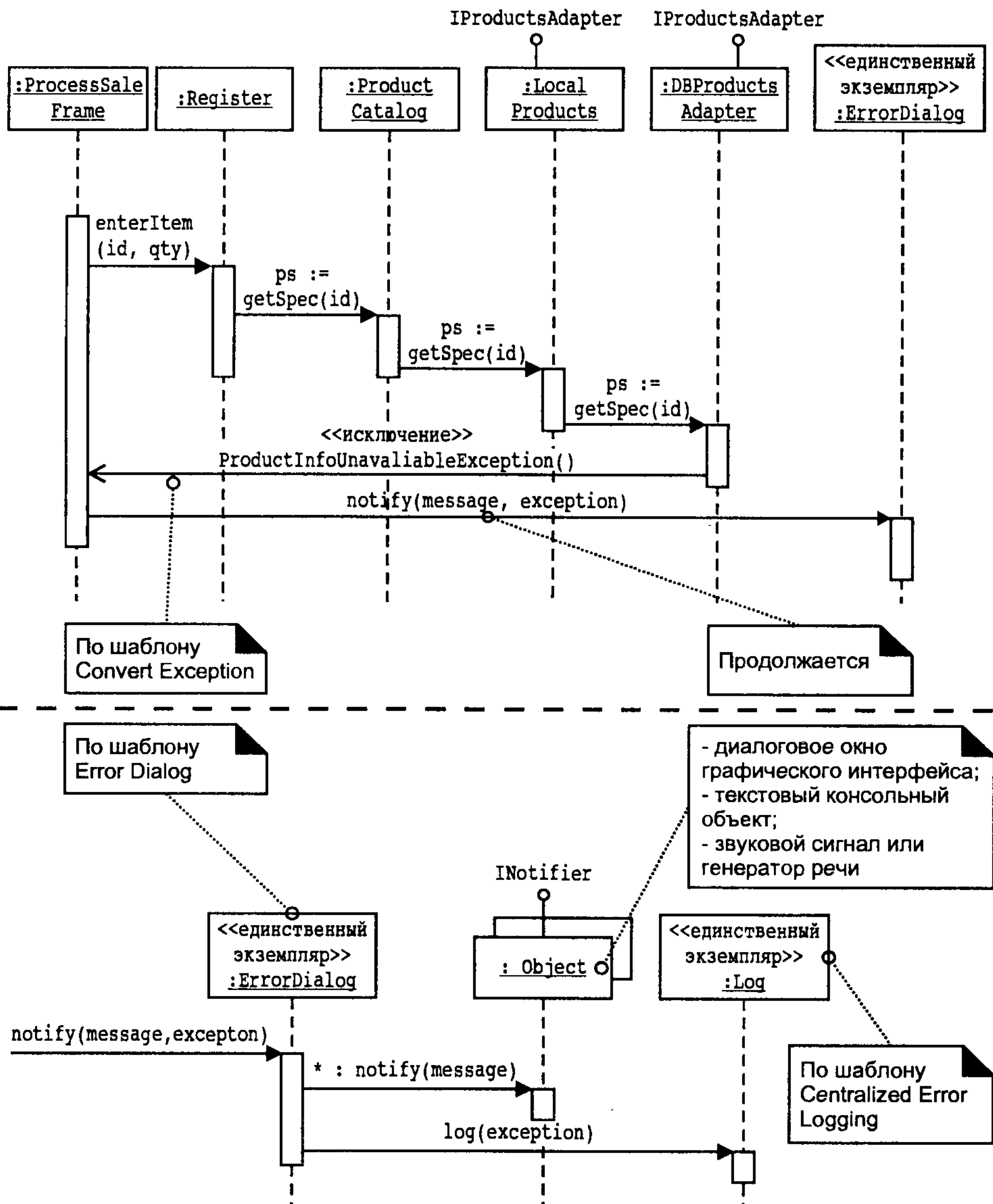



*Рис. 33.11. Обработка исключения*

*Шаблон Proxy*

**Контекст/Проблема**

Прямой доступ к объекту не желателен или не возможен. Что делать?

**Решение**

Добавить новый уровень перенаправления с искусственным объектом-посредником, реализующим тот же интерфейс, что и основной объект, и делегировать ему обязанность по управлению доступом к основному объекту.

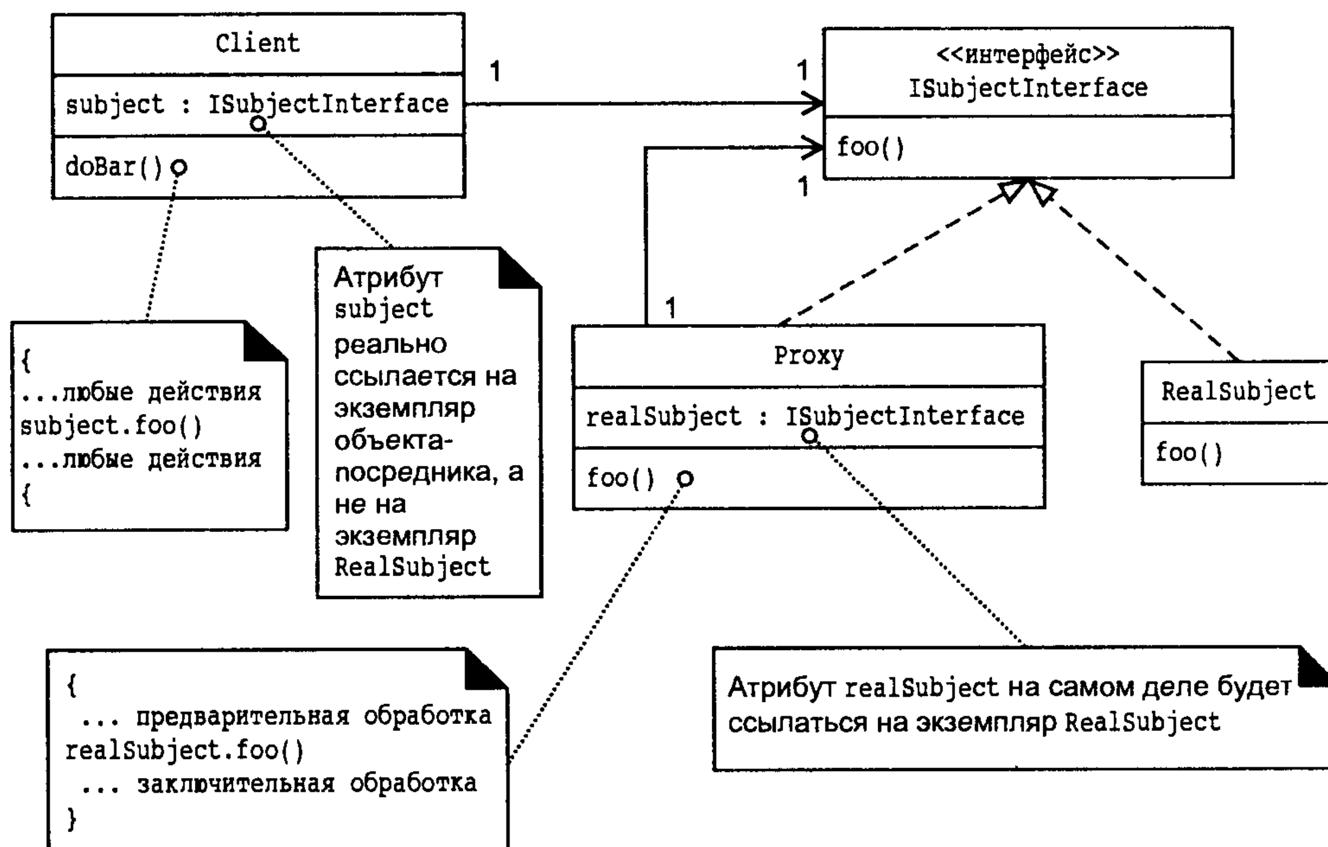



*Рис. 33.12. Общая структура объектов шаблона Proxy*

В соответствии с рассматриваемым примером приложения NextGen для обеспечения доступа к внешней бухгалтерской системе объект-посредник для перенаправления используется следующим образом (рис. 33.13).

1. Сообщение postSale передается объекту-посреднику, обрабатывающему его как реальная внешняя бухгалтерская система.
2. Если объекту-посреднику не удается установить соединение с внешней системой (через адаптер), он перенаправляет сообщение postSale локальной службе, хранящей информацию о продажах для последующей ее передачи внешней системе (после восстановления связи).

Вот некоторые обозначения UML.

- Обратите внимание, как на этой статической диаграмме показана последовательность взаимодействия (что позволяет избежать построения диаграммы взаимодействия для отражения динамического поведения). Обычно лучше все же использовать диаграммы взаимодействия, однако здесь показаны альтернативные обозначения.
- Обратите внимание на маркеры открытой и закрытой областей видимости (+, -) рядом с методами объекта Register. Если эти маркеры не указаны, значит, область видимости не определена (а не задается по умолчанию). Тем не менее по общепринятому соглашению большинство читателей (и генераторы кода CASE-средств) интерпретируют отсутствие маркера области видимости как принадлежность атрибута или метода к закрытой области. На приведенной диаграмме специально показано, что метод makePayment является открытым, а метод completeSaleHandling — закрытым. Лишняя информация перегружает диаграмму, поэтому желательно использовать на диаграммах общепринятые обозначения.

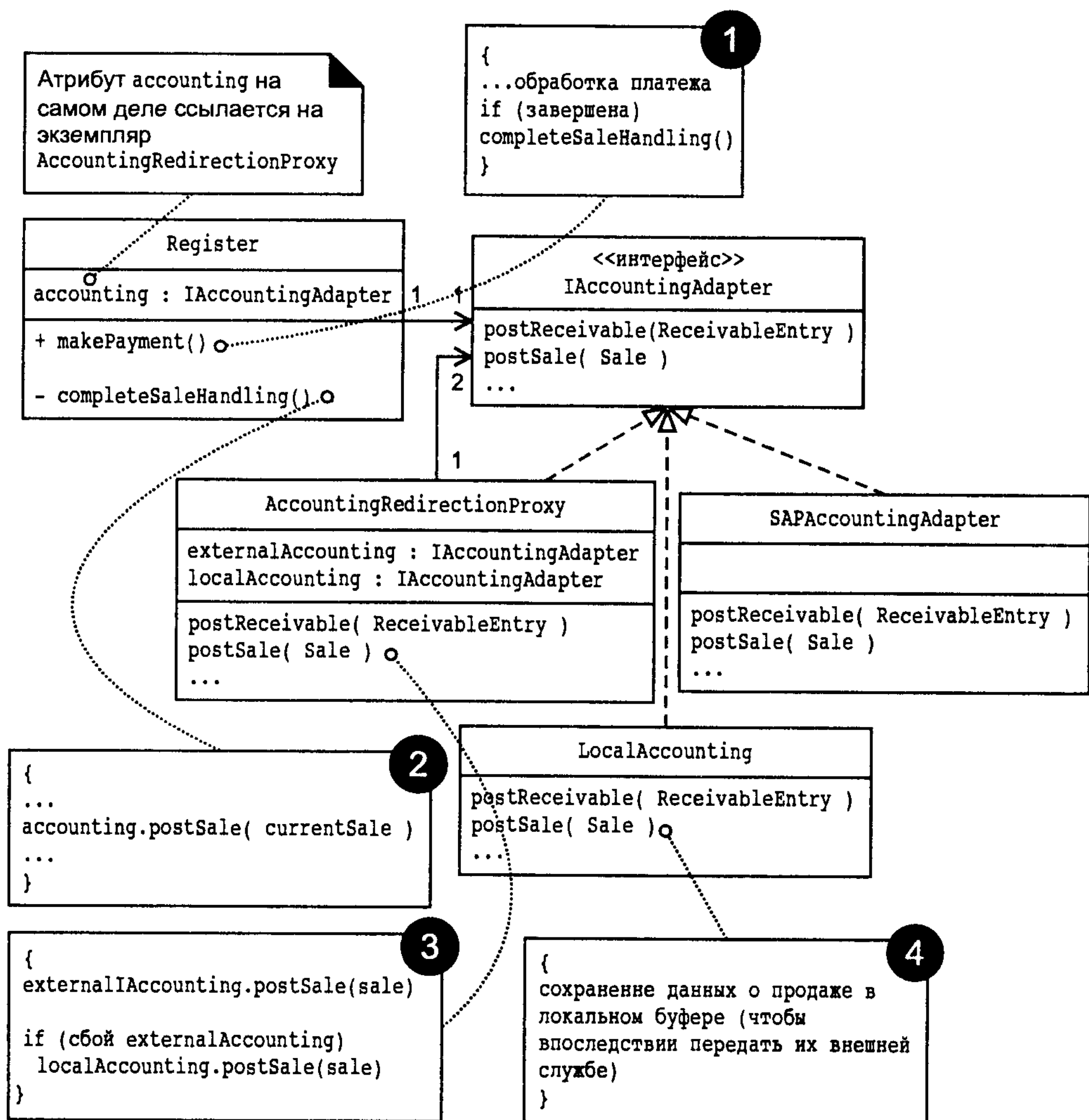



*Рис. 33.13. Использование объекта-посредника для перенаправления информации в системе NextGen*

В заключение отметим, что объект-посредник — это внешний объект, выступающий в качестве оболочки для внутреннего объекта и реализующий тот же интерфейс. Клиентский объект (например, `Register`) не знает, что общается с посредником. Для него взаимодействие ничем не отличается от работы с реальным объектом (к примеру, `SAPAccountingAdapter`). Объект-посредник перехватывает вызовы для улучшения доступа к реальному объекту, в данной ситуации за счет перенаправления обращения к локальной службе (`LocalAccounting`) в случае отказа внешней службы.

## 33.4. Реализация нефункциональных или качественных требований

Прежде чем перейти к следующему разделу, отметим, что рассматриваемые до сих пор в этой главе проектные решения не были связаны с бизнес-логикой, а относились лишь к реализации нефункциональных или качественных требований надежности и восстановления данных.

Интересно, что большинство сложных вопросов, шаблонов и типов программной архитектуры связаны как раз с реализацией нефункциональных качественных требований, а не с реализацией бизнес-логики.

## 33.5. Доступ к внешним физическим устройствам с помощью шаблона Adapter: купить или разработать

Еще одно требование этой итерации связано со взаимодействием с физическими устройствами, сопряженными с терминалом POS-системы, в частности, открытием кассового аппарата, извлечением сдачи или считыванием подписи с помощью цифрового устройства.

POS-система NextGen должна взаимодействовать с различным оборудованием, в том числе разработанным компаниями IBM, Epson, Fujitsu и т.д.

К счастью в настоящее время существует промышленный стандарт UnifiedPOS (`www.nrf-arts.org`), определяющий общепринятый объектно-ориентированный интерфейс (в смысле UML) для всех типичных POS-устройств. Более того, существует реализация этого стандарта для Java — JavaPOS (`www.javapos.com`).

Поэтому в описание программной архитектуры нужно добавить раздел о взаимодействии с другими устройствами.

---

**Техническое описание**

**Вопрос. Управление аппаратными средствами POS-системы.**

**Идея решения. Использовать специальное программное обеспечение для Java от производителей аппаратных средств, удовлетворяющее стандарту интерфейса JavaPOS.**

**Факторы**

- Корректное управление устройствами
- Проще купить, чем разработать и поддерживать

**Решение**

Стандарт UnifiedPOS (`www.nrf-arts.org`) определяет промышленный стандарт для моделирования на UML интерфейсов для POS-устройств. JavaPOS (`www.javapos.com`) — это промышленная версия стандарта UnifiedPOS для Java. Производители оборудования для POS-систем (IBM, NCR) продают реализации этих интерфейсов, управляющие произведенными ими устройствами.

Лучше купить эти программы, чем разрабатывать самим.

Для загрузки набора классов IBM или NCR можно использовать шаблон Factory и возвращать экземпляры, основанные на этих интерфейсах.

**Мотивировка**

По неофициальным данным, эти интерфейсы работают хорошо, производители оборудования регулярно обновляют их версии в соответствии с модификацией аппаратных средств. Написать их самостоятельно гораздо сложнее.

**Альтернативные решения**

Написать интерфейсы самостоятельно — это сложно и рискованно.

---

На рис. 33.14 показаны некоторые интерфейсы, объединенные в новый пакет уровня предметной области модели проектирования.

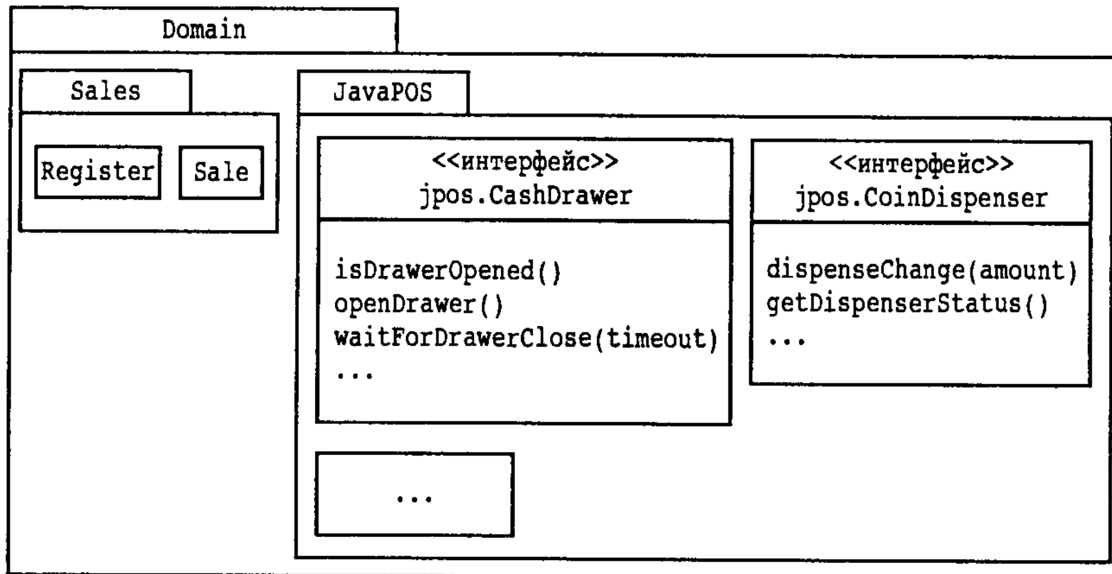



*Рис. 33.14. Стандартные интерфейсы JavaPOS*

Допустим, основные производители оборудования для POS-систем поставляют на рынок также свои реализации стандарта JavaPOS. Например, при покупке терминала с кассовым аппаратом от компании IBM можно также приобрести классы Java, реализующие интерфейсы JavaPOS и управляющие физическими устройствами.

> Следовательно, эта часть архитектуры строится на базе стандартных программных компонентов, а не их самостоятельной разработки. Один из советов UP — приобретение стандартных компонентов.

Как работают эти компоненты? На низком уровне существуют драйверы физических устройств, взаимодействующие с их операционной системой. Класс Java (например, одна из реализаций `jpos.CashDrawer`) использует интерфейс JNI (Java Native Interface) для обращения к драйверам этих устройств.

> Эти классы Java адаптируют низкоуровневые драйверы устройств к интерфейсам JavaPOS, поэтому их можно рассматривать как объекты-адаптеры в смысле шаблона GoF. Их можно также назвать объектами-посредниками, управляющими доступом к физическим устройствам.

## 33.6. Шаблон Abstract Factory (GoF) для семейства взаимосвязанных объектов

Реализации интерфейсов JavaPOS можно приобрести у производителей оборудования. Вот несколько примеров.[5]

```
// Драйверы компании IBM
com.ibm.pos.jpos.CashDrawer (implements jpos.CashDrawer)
com.ibm.pos.jpos.CoinDispenser (implements jpos.CoinDispenser)
```

[5] Это фиктивные имена пакетов.

```
...
// Драйверы компании NCR
com.ncr.posdrivers.CashDrawer (implements jpos.CashDrawer)
com.ncr.posdrivers.CoinDispenser (implements jpos.CoinDispenser)
...
```

Как же спроектировать работу приложения NextGen с драйверами компании IBM или NCR?

Требуется создать семейства классов (`CashDrawer`, `CoinDispenser` и т.д.), каждое из которых реализует свой интерфейс.

Для этого существует шаблон GoF Abstract Factory (Абстрактная фабрика).

*Шаблон Abstract Factory*

**Контекст/проблема**

Как создать семейства взаимосвязанных классов, реализующих общий интерфейс?

**Решение**

Определить интерфейс объекта-фабрики (абстрактную фабрику). Определить конкретный класс-фабрику для каждого создаваемого семейства. Кроме того, определить реальный абстрактный класс, реализующий интерфейс фабрики и обеспечивающий однотипную работу с конкретными объектами-фабриками, расширяющими его.

Основная идея этого шаблона проиллюстрирована на рис. 33.15.

### Абстрактный класс абстрактной фабрики

Типичная реализация шаблона Abstract Factory подразумевает создание абстрактного класса-фабрики, доступ к которому осуществляется по шаблону Singleton (Единственный экземпляр), определение на основе системного свойства конкретных типов создаваемых подклассов-фабрик и получение соответствующего экземпляра подкласса. Этот принцип реализуется, например, в библиотеках Java с помощью класса `java.awt.Toolkit` — абстрактного класса абстрактной фабрики, предназначенной для создания семейства элементов графического пользовательского интерфейса для различных операционных систем.

Преимуществом такого подхода является решение следующей проблемы: какую абстрактную фабрику использовать — `IBMJavaPOSDevicesFactory` или `NCRJavaPOSDevicesFactory`?

Решение этой проблемы показано на рис. 33.16.

При наличии абстрактного класса-фабрики и метода `getInstance` шаблона Singleton объекты могут взаимодействовать с абстрактным подклассом, получая ссылку на один из экземпляров подкласса. Например, рассмотрим следующий оператор.

```
CashDrawer = JavaPOSDevicesfactory.getInstance().getNewCashDrawer();
```

Вызов `JavaPOSDevicesfactory.getInstance()` возвращает экземпляр класса `IBMJavaPOSDevicesFactory` или `NCRJavaPOSDevicesFactory`, в зависимости от считанного значения системного свойства. Заметим, что при изменении внешнего системного свойства `jposfactory.classname` (представляющего имя класса в виде

строки) в файле конфигурации, система NextGen будет использовать другое семейство драйверов JavaPOS. При этом соблюдается принцип шаблона Protected Variations в смысле защиты от влияния изменения типа фабрики за счет считывания данных из файла конфигурации и рефлексивного проектного решения.

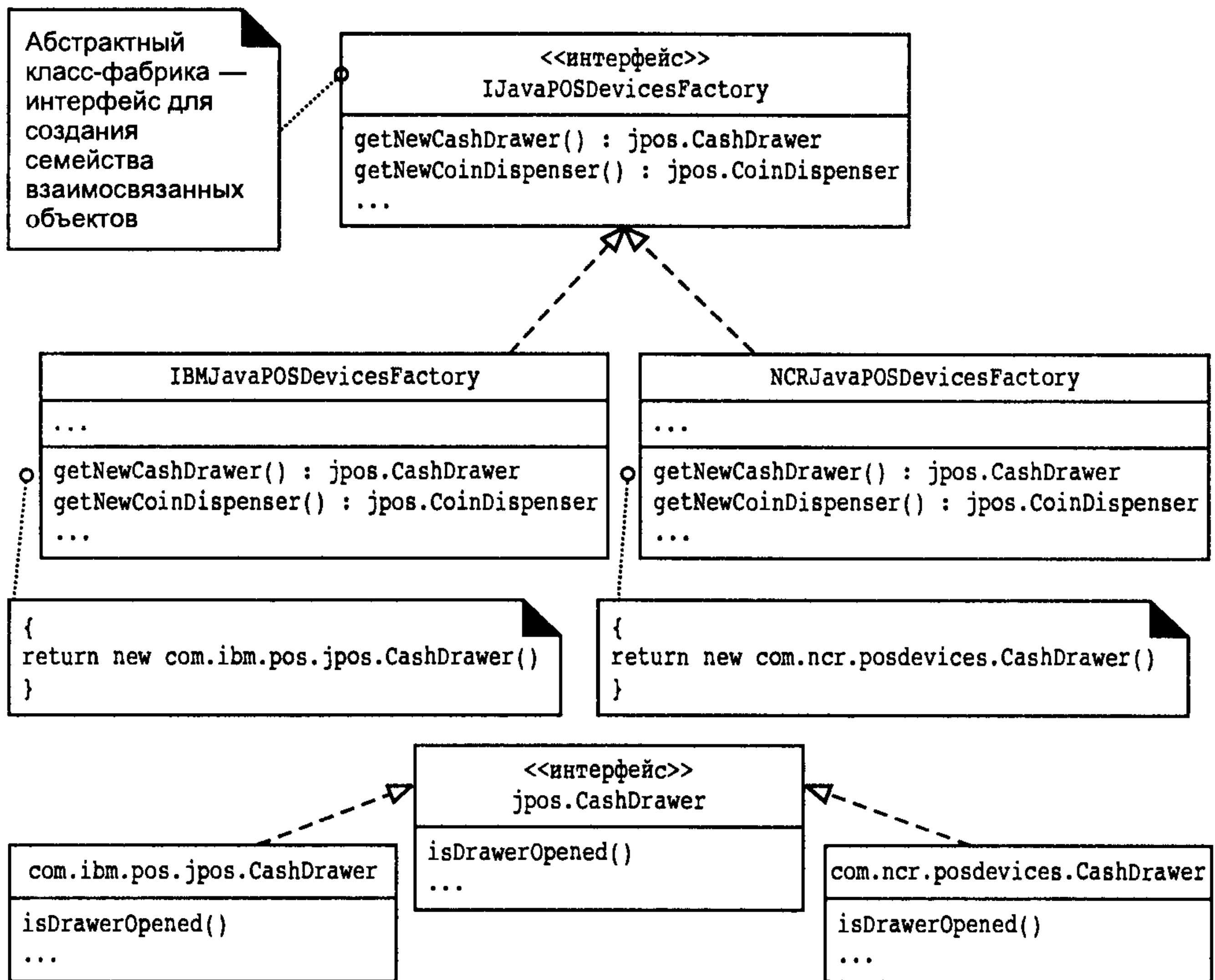



*Рис. 33.15. Базовая абстрактная фабрика*

Взаимодействие с объектом-фабрикой осуществляет объект Register. С целью сокращения разрыва между содержанием и представлением целесообразно поместить в этот объект ссылки на устройства, в частности CashDrawer. Например,

```
class Register
{
private jpos.CashDrawer cashDrawer;
private jpos.CoinDispenser coinDispenser;

public Register()
{
    cashDrawer = JavaPOSDevicesfactory.
                getInstance().getNewCashDrawer();
    //...
}
//...
}
```

```
{
// ЭТОТ МЕТОД РЕАЛИЗУЕТ ПОЛЕЗНЫЙ ПРИЕМ
public static synchronized
IJavaPOSDevicesFactory getInstance()
{
 if (instance == null)
 {
   String factoryClassName =
     System.getProperty("jposfactory.classname");

   Class c = Class.forName(factoryClassName);

   instance = (IJavaPOSDevicesFactory)c.newInstance();
 }
 return instance;
}

}
```

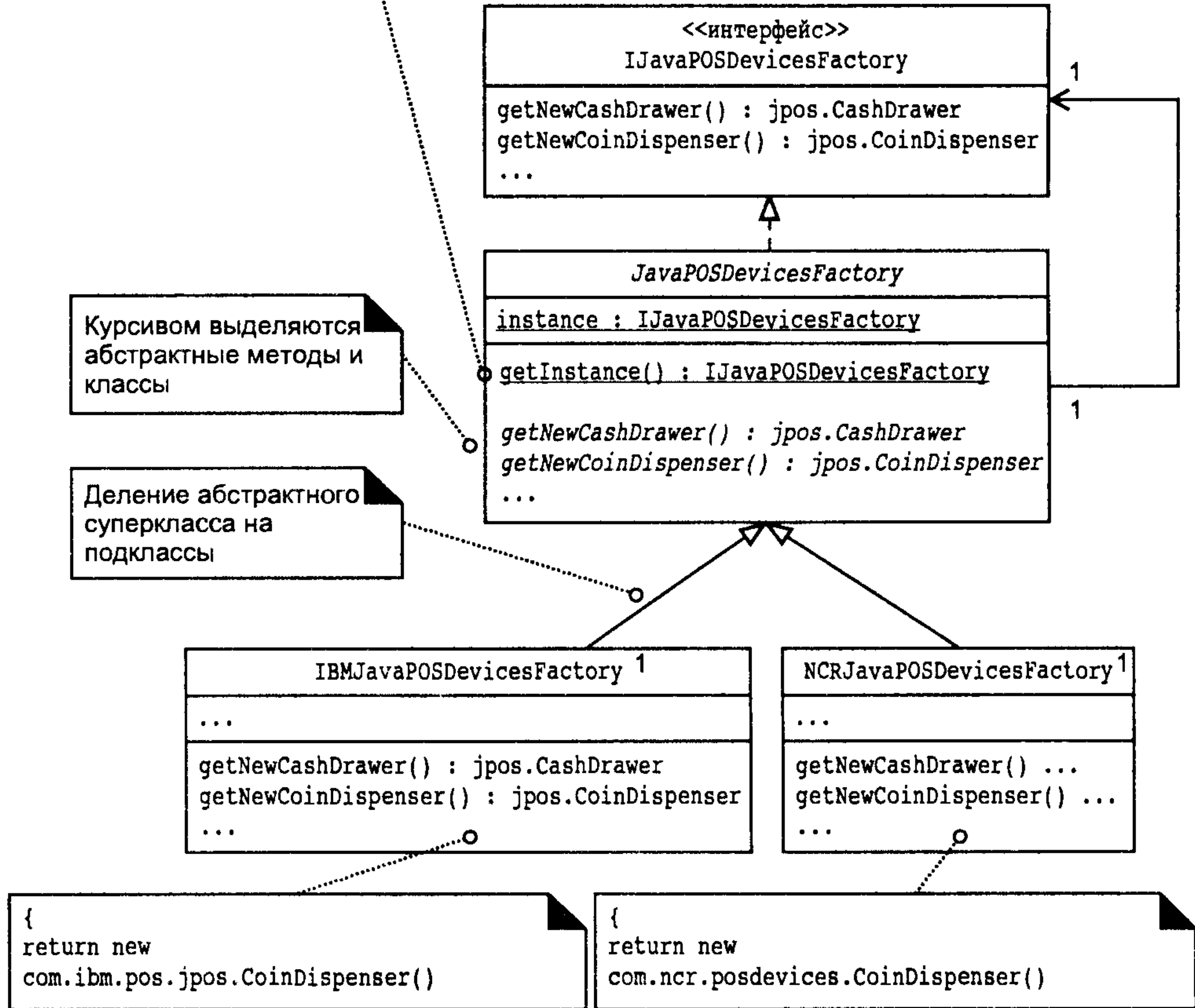



*Рис. 33.16. Абстрактный класс абстрактной фабрики*

## 33.7. Обработка платежей на основе шаблонов Polymorphism и Do It Myself

Одним из распространенных способов применения принципа полиморфизма (и шаблона Information Expert) является стратегия "Сделай сам", оформленная в виде отдельного шаблона [30]. Она сводится к следующему.

> *Шаблон Do It Myself*
>
> "Программный объект выполняет обязанности реальных объектов, абстракцию которых он представляет." [30]

В этом проявляется классический стиль объектно-ориентированного проектирования: объекты `Circle` и `Square` отвечают за отображение самих себя, объект `Text` проверяет орфографию и т.д.

Заметим, что проверка орфографии объектом `Text` — классический пример реализации шаблона Information Expert, согласно которому функциональная обязанность присваивается тому объекту, который обладает для ее выполнения достаточной информацией.

Шаблоны Do It Myself и Information Expert провозглашают схожие принципы.

Отображение объектов `Circle` и `Square` — это в то же время пример реализации шаблона Polymorphism, согласно которому обязанности для типов с различным поведением распределяются на основе полиморфных операций.

Шаблоны Do It Myself и Polymorphism обычно приводят к одинаковому распределению обязанностей.

В соответствии с шаблоном Pure Fabrication, перегрузка класса функциональными обязанностями часто противопоказана, поскольку это приводит к проблемам со связыванием и зацеплением, поэтому разработчики зачастую прибегают к созданию чисто синтетических классов стратегий, фабрик и т.д.

Тем не менее, шаблон Do It Myself очень привлекателен, поскольку позволяет сократить разрыв между реализацией и ее представлением. Поэтому обработку платежей будем выполнять на основе шаблонов Do It Myself и Polymorphism.

Одним из требований для данной итерации является обработка различных типов платежей, требующих авторизации и взаимодействия с внешними системами. Разные типы платежей авторизируются по-разному.

- Платежи по кредитной и дебитной карточке подлежат авторизации через внешнюю службу.
- Платежи наличными авторизируются в некоторых магазинах с помощью специального устройства проверки подлинности банкнот, подключенного к POS-терминалу. В других магазинах авторизация не проводится.
- Платежи чеком авторизируются в некоторых магазинах с помощью компьютеризированной службы авторизации, в других — авторизация отсутствует.

Таким образом, объекты `CreditPayment` подлежат авторизации одним способом, а объекты `CheckPayment` — другим. Это классическая ситуация для применения шаблона Polymorphism.

Как видно из рис. 33.17, каждый подкласс класса `Payment` имеет свой метод `authorize`.

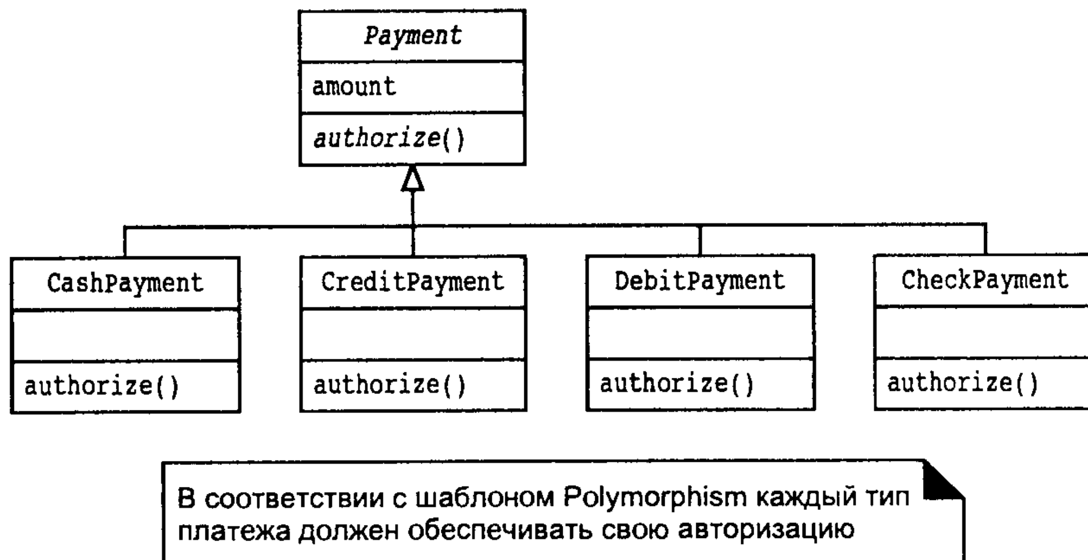



*Рис. 33.17. Классический пример использования принципа полиморфизма для различных методов* `authorize`

Например, как видно из рис. 33.18 и 33.19, объект `Sale` инстанцирует объекты `CreditPayment` и `CheckPayment` и передает им запрос на авторизацию соответствующего типа платежа.

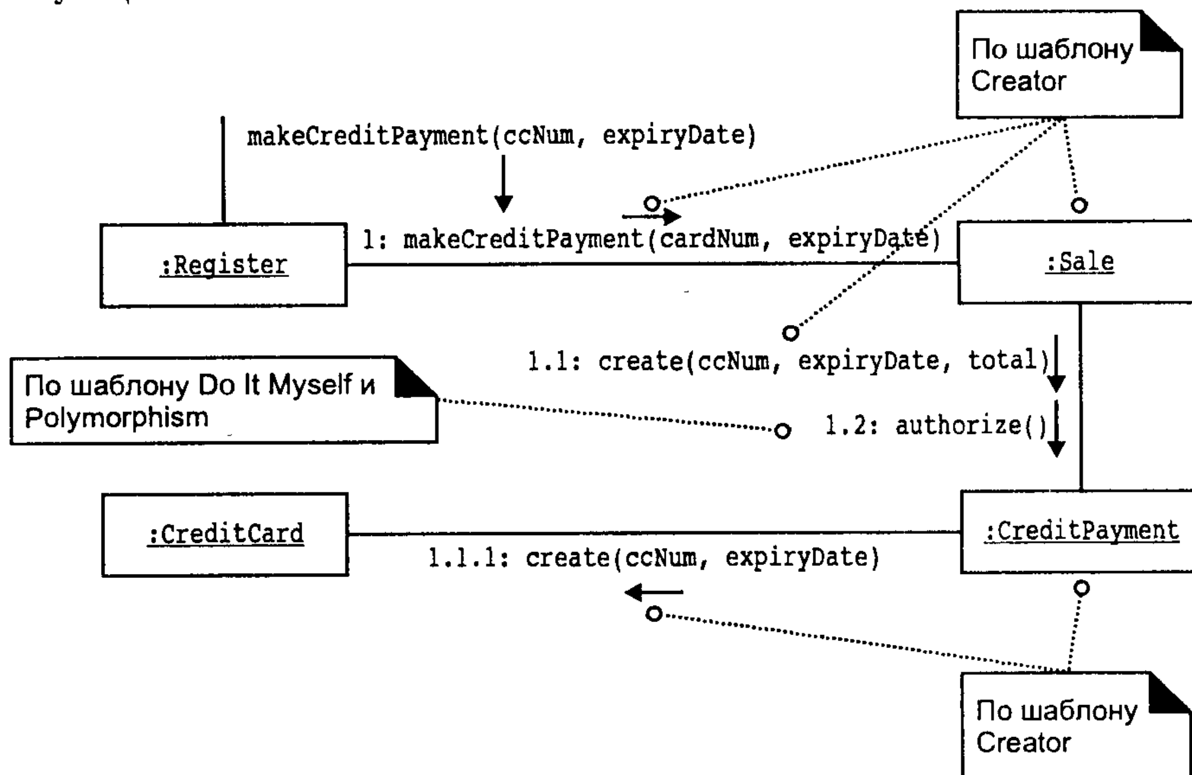



*Рис. 33.18. Создание экземпляра* `CreditPayment`

## "Мелкие" классы

Рассмотрим программные объекты `CreditCard`, `DriversLicense` и `Check`. Сразу же приходит мысль записать данные этих классов в соответствующие классы платежей и не связываться со столь "мелкими классами". Однако зачастую более разумно эти классы оставить, поскольку они полезны и их можно будет использовать повторно. Например, класс `CreditCard` является естественным экспертом для определения типа кредитной карточки (Visa, MasterCard и т.п.). А это очень нужная для приложения информация.

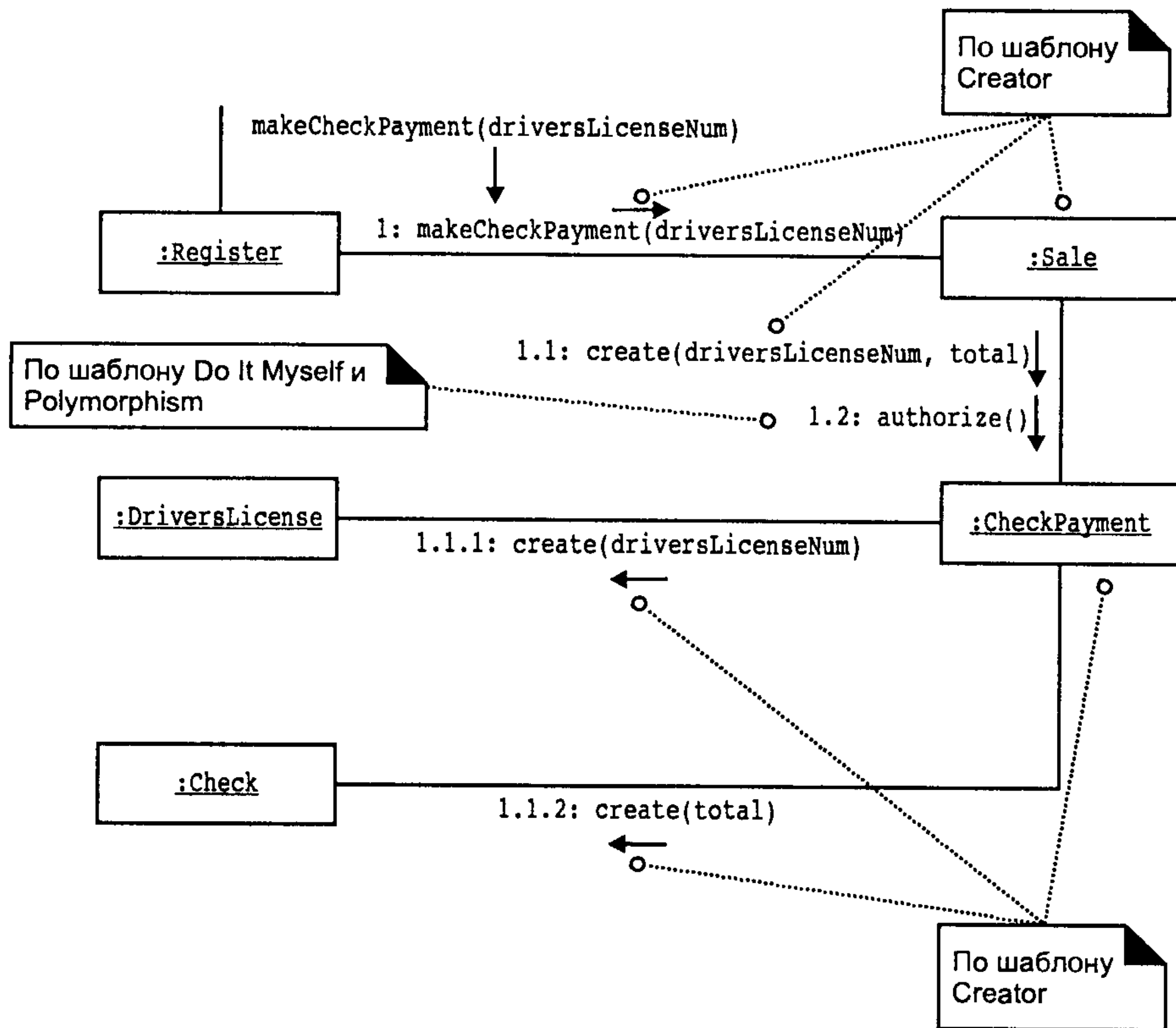



*Рис. 33.19. Создание экземпляра* `CheckPayment`

## Авторизация платежей по кредитной карточке

Для выполнения этой операции система должна взаимодействовать с внешней службой авторизации кредитов. Для поддержки такого взаимодействия уже создано проектное решение на основе объектов-адаптеров.

### Сведения из предметной области платежей по кредитной карточке

Рассмотрим некоторые моменты, в контексте которых строится проектное решение.

- POS-системы могут быть физически связаны с внешними службами авторизации несколькими способами, в том числе через телефонные линии (удаленное соединение) или выделенный канал Internet.
- Для авторизации используются различные протоколы уровня приложения и соответствующие им форматы данных, в частности SET (Secure Electronic Transaction). Со временем могут появиться новые протоколы, например XMLPay.
- Авторизацию платежей можно рассматривать как обычную синхронную операцию: поток POS-системы блокируется и ожидает ответа от удаленной службы (в течение ограниченного интервала времени).

- Все протоколы авторизации предполагают передачу идентификаторов магазина (ID магазина) и терминала POS-системы (ID терминала). В ответе содержится код подтверждения или отказа и уникальный идентификатор транзакции.
- В магазине для разных типов кредитных карточек могут использоваться различные внешние службы авторизации (одна для карточек Visa, другая — для MasterCard). Для каждой такой службы используется свой идентификатор магазина.
- Тип кредитной карточки можно определить по ее номеру. Например, номера, начинающиеся с цифры 5 относятся к системе MasterCard, а номера, начинающиеся с цифры 4 — к системе Visa.
- Реализации адаптеров защищают вышестоящие уровни системы от изменения типов авторизации платежей. Каждый адаптер отвечает за выполнение транзакции авторизации в соответствующем формате и за взаимодействие с внешними службами. Как было описано при рассмотрении предыдущей итерации, объект `ServicesFactory` отвечает за доставку соответствующей реализации интерфейса `ICreditAuthorizationServiceAdapter`.

### Сценарий разработки

На рис. 33.20 показана первая часть проектного решения, удовлетворяющего описанным требованиям.

После нахождения соответствующей реализации `ICreditAuthorizationServiceAdapter` на этот объект возлагается обязанность по выполнению авторизации (рис. 33.21).

После получения положительного ответа объект `CreditPayment` в соответствии с шаблоном Polymorphism и Do It Myself завершает свою работу (рис. 33.22).

Обратите внимание, что на этой диаграмме последовательностей некоторые объекты изображены в несколько ярусов. Это допускается синтаксисом UML, однако такую форму записи поддерживают не все CASE-средства. Тем не менее она достаточно удобна, если ширина листа ограниченна.

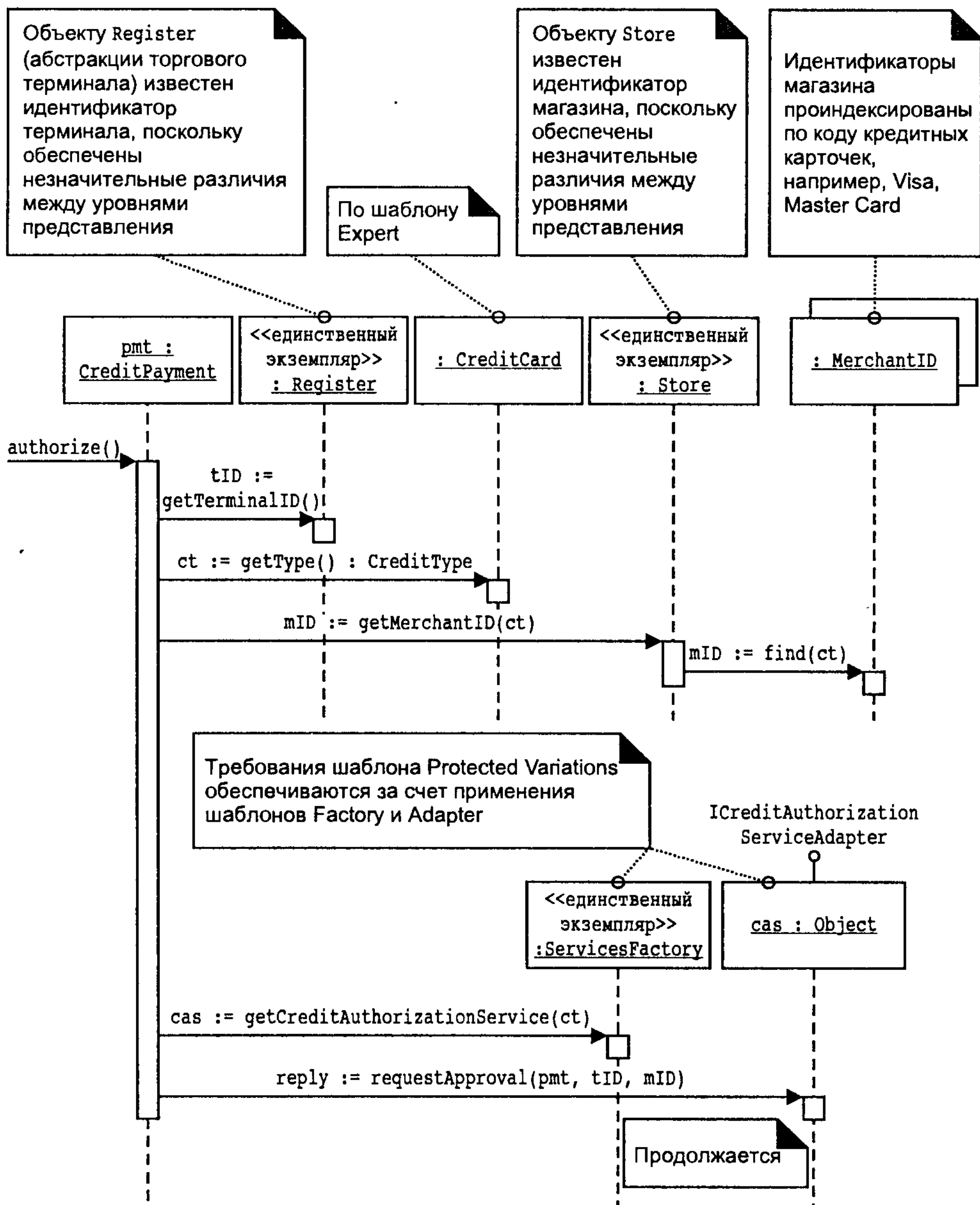



*Рис. 33.20. Обработка платежей по кредитной карточке*

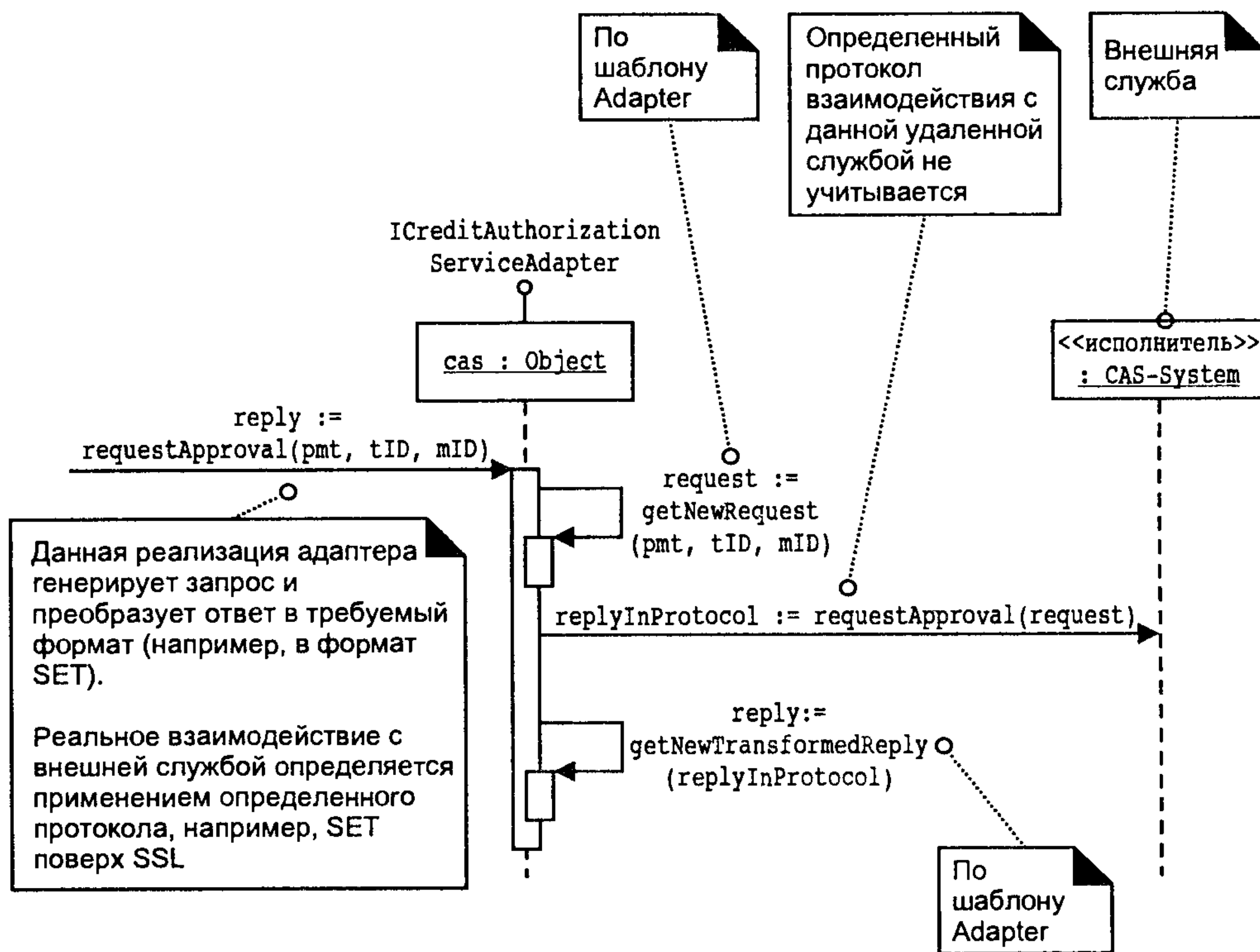



*Рис. 33.21. Завершение авторизации*

## 33.8. Заключение

В этом примере не ставилась задача показать корректное решение. Единственного правильного решения просто не существует. Автор уверен, что читатели смогут усовершенствовать предложенное проектное решение. Основной задачей было объектное проектирование на основе базовых принципов, таких как связывание и применение шаблонов.

### *Предупреждение: "злоупотребление" шаблонами*

В рассмотренном примере довольно часто применялись шаблоны GoF, что соответствовало задачам обучения читателей. Однако иногда разработчики злоупотребляют применением шаблонов и пытаются использовать их где надо и где не надо. Для умелого использования шаблонов необходимо изучить их. Для изучения шаблонов на практике в некоторых организациях (в обеденный перерыв или после работы) проводят деловые игры, в рамках которых участники группы разработчиков упражняются в применении шаблонов на тестовых приложениях.

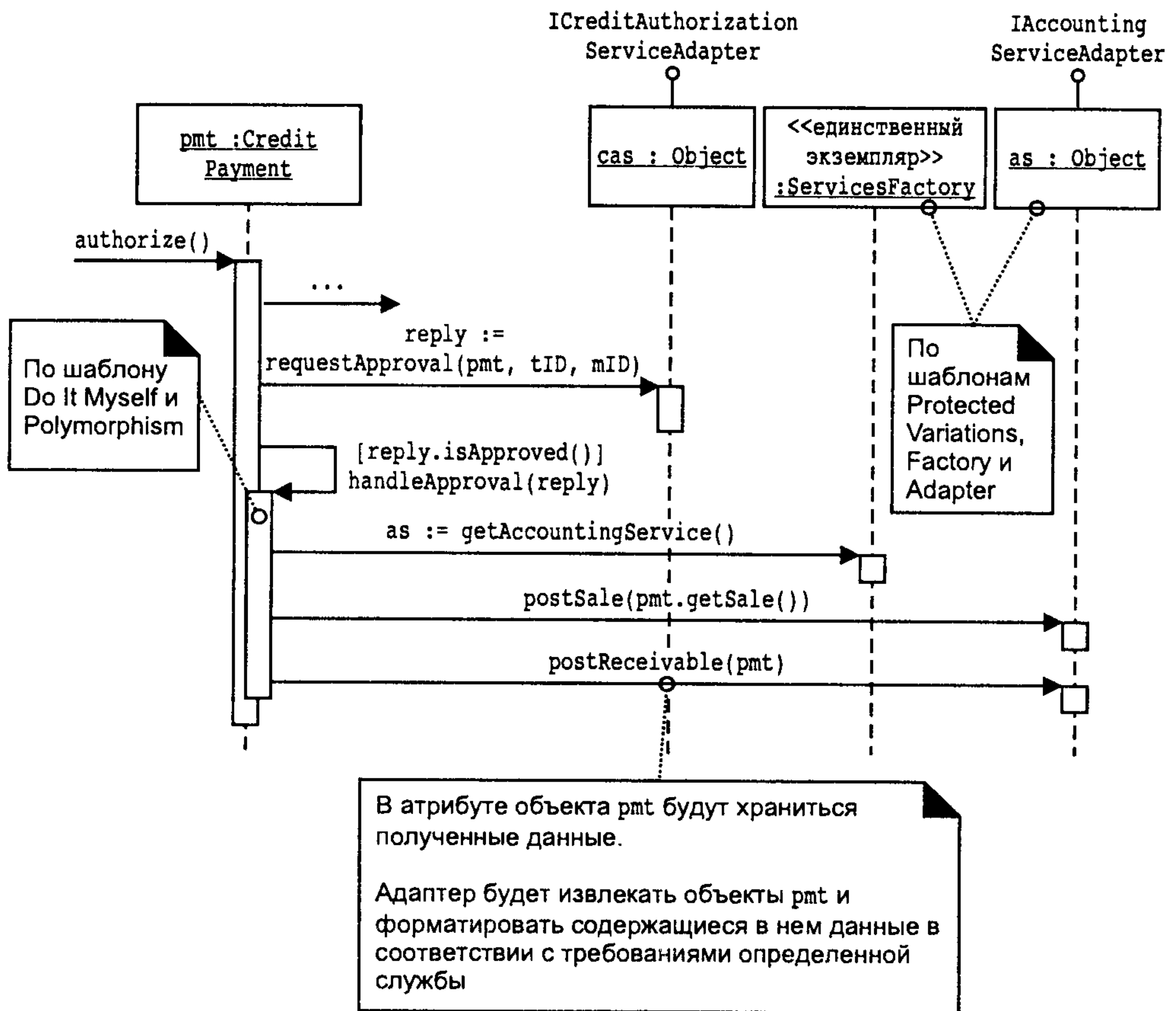



*Рис. 33.22. Завершение обработки платежа по кредитной карточке после получения подтверждения от службы авторизации*

Глава

# Проектирование контура взаимодействия с базой данных на основе шаблонов

*Время — великий учитель, но он, к сожалению, убивает всех своих учеников.*

*Гектор Берлиоз (Hector Berlioz)*

---

**Основные задачи**

- Спроектировать часть контура на основе шаблонов Template Method, State и Command.
- Ознакомиться с принципами преобразования информации из объектного представления в формат реляционной базы данных.
- Реализовать пассивную материализацию на основе шаблона Virtual Proxy.

---

## Введение

В приложении NextGen, как и в большинстве приложений, возникает необходимость хранить информацию на постоянных носителях, например, в реляционной базе данных. В этой главе рассматриваются вопросы объектно-ориентированного проектирования контура объектов, связанные с хранением информации на постоянных носителях.

Как правило, лучше приобрести готовую службу преобразования информации из объектного представления в формат реляционной базы данных или O-R-службу (object-relational service), чем самому приниматься за ее разработку. Такие службы продаются как в виде отдельных продуктов, так и в составе специализированных наборов компонентов, в частности для EJB или других Java-технологий. На разработку полноценной O-R-службы потребуется несколько лет, при этом многие вопросы придется выяснять дополнительно у специалистов по

базам данных. Более того, частные решения уже предлагаются в рамках многих технологий, к примеру основанных на спецификации JDO (Java Data Objects).

Поэтому в данной главе не ставится задача проектирования коммерческого контура взаимодействия с базой данных и не пропагандируется отказ от технологий на базе JDO. Контур взаимодействия с базой данных выбран лишь в качестве примера для изучения вопроса проектирования контуров на основе шаблонов. Здесь также показано, как отобразить процесс разработки средствами языка UML.

> Рассматриваемый пример предназначен для изучения принципов проектирования контуров и не претендует на проектное решение для коммерческой службы взаимодействия с базой данных.

## 34.1. Проблема: объекты, подлежащие постоянному хранению

Предположим, что в приложении NextGen экземпляры объекта `ProductSpecification` хранятся в реляционной базе данных и загружаются в оперативную память в процессе работы приложения. *Постоянно хранимыми объектами* (persistent objects) назовем объекты, требующие хранения на постоянном носителе, например экземпляры `ProductSpecification`.

### Механизмы хранения и постоянно хранимые объекты

**Объектные базы данных.** При хранении информации в объектной базе данных не требуется никаких дополнительных служб обеспечения взаимодействия с базой данных. В этом состоит одно из преимуществ такого подхода.

**Реляционные базы данных.** Для хранения информации чаще всего применяются реляционные базы данных. При этом возникает множество проблем, связанных с несоответствием объектно-ориентированного представления данных в системе и их представления в виде записей в базе данных. Эти проблемы будут рассмотрены несколько позже. Для работы с реляционными базами данных требуются специальные средства преобразования форм представления информации (O-R-преобразования).

**Другие базы данных.** Иногда желательно хранить информацию в другом виде, например в текстовых файлах, в формате XML, в файлах Palm OS PDB, иерархических базах данных и т.п. В этом случае возникают те же проблемы преобразования форм представления информации, что и при работе с реляционными базами данных. Как и при использовании реляционных баз данных, здесь возникает несоответствие между представлением постоянно хранимого объекта в системе и его представлением в не объектно-ориентированной структуре. Для работы с такими объектами требуются специальные службы.

## 34.2. Решение: контур интерфейса с базой данных

*Контур интерфейса с базой данных* (persistent framework) — многократно используемый и обычно расширяемый набор классов, обеспечивающий обслуживание постоянно хранимых объектов. Функции контура выполняет *служба интерфейса с базой данных* (или подсистема), создаваемая на базе классов этого контура. Служба интерфейса с базой данных обычно разрабатывается для взаимодействия с реляционными базами данных. В этом случае она называется

*службой O-R-преобразования* (O-R mapping service). Обычно служба интерфейса с базой данных выполняет преобразование объектной формы представления информации в форму записей (или другой структурированный формат данных, типа XML) и их сохранение в базе данных, а также обратные операции.

В терминах многоуровневой архитектуры приложения NextGen служба интерфейса с базой данных — это подсистема уровня технических служб.

## 34.3. Контуры

Рискуя примитивизировать это понятие, возьму на себя смелость утверждать, что *контур* (framework) — это расширяемый набор объектов для реализации взаимосвязанных функций. Типичным примером контура является контур графического интерфейса пользователя, в частности библиотека AWT или Swing для Java.

Хороший контур включает реализацию базовых и инвариантных функций, а также обеспечивает механизм подключения модифицируемых функций или их расширения.

Например, контур графического интерфейса Swing для Java содержит много классов и интерфейсов для основных функций графического интерфейса пользователя. Разработчики могут добавлять специализированные средства за счет реализации подклассов для классов Swing и перекрытия существующих методов. Разработчики могут также подключать разнообразные обработчики событий (например, для класса `JButton`), регистрируя слушателей или подписчиков на основе шаблона Observer. Это и есть контур.

В целом, контур — это набор классов:

- с высокой степенью зацепления, взаимодействующих между собой и обеспечивающих обслуживание ядра, т.е. неизменяемой части логической подсистемы;
- включающий конкретные и абстрактные классы, которые определяют интерфейс между объектами;
- обычно (но не всегда) требующий определения пользователем подклассов для существующих классов контура, их использования, настройки и расширения служб контура;
- имеющий абстрактные классы, которые могут содержать как абстрактные, так и конкретные методы;
- основанный на принципе Голливуда (Hollywood Principle) "Не звоните нам, мы сами с вами свяжемся". Это означает, что определенные пользователем классы (например, новые подклассы) будут получать сообщения от определенных ранее классов контура. Это обычно достигается путем реализации абстрактных методов суперкласса.

Следующий пример контура интерфейса с базой данных иллюстрирует это определение.

### Контуры предназначены для повторного использования

Контуры предоставляют очень хорошие возможности для повторного использования, гораздо более широкие, чем отдельные классы. Следовательно, если организация заинтересована (а кто в этом не заинтересован?) в повторном использовании своего программного кода, то необходимо сосредоточить внимание на создании контуров.

## 34.4. Требования к контуру интерфейса с базой данных

Для приложения NextGen необходим контур интерфейса с базой данных (persistence framework), обеспечивающий обслуживание постоянно хранимых объектов. Назовем его PFW (Persistence Framework). PFW — это упрощенный контур. Полнофункциональные контуры интерфейса с базой данных не относятся к вопросам, обсуждаемым в этой книге.

Рассматриваемый здесь контур должен обеспечивать реализацию следующих функций.

- Хранение объектов на постоянном носителе и извлечение этих объектов
- Завершение и отмена транзакций (`commit` и `rollback`)

Проектное решение должно быть расширяемым и поддерживать различные механизмы хранения данных, включая реляционную базу данных, обычные файлы, формат XML и т.д.

## 34.5. Ключевые идеи

В последующих разделах будут рассмотрены некоторые ключевые идеи.

- **Соответствие**. Между классом и его представлением на постоянном носителе должно существовать некоторое соответствие (например, таблица в базе данных). Поля записи базы данных должны соответствовать атрибутам. Таким образом должно существовать соответствие между этими двумя схемами.
- **Идентичность объектов**. Для упрощения связывания записей с объектами и во избежание неоправданного дублирования записи и объекты должны иметь уникальные идентификаторы объектов.
- **Преобразователи базы данных**. За материализацию и дематериализацию объектов, согласно шаблону Pure Fabrication, отвечает специальный класс-преобразователь базы данных.
- **Материализация и дематериализация**. Материализация — это действие по преобразованию необъектного представления данных (например, записей) в объектное представление. Дематериализация — это обратное действие (которое еще называют деактивацией).
- **Кэширование**. С целью повышения производительности специальные службы кэшируют материализованные объекты.
- **Состояние транзакции данного объекта**. Очень важно знать состояние объекта в терминах его отношения к текущей транзакции. Например, полезно знать, какие объекты были модифицированы, чтобы знать, нужно ли их снова записывать в базу данных.
- **Операции транзакции**. Операции `commit` и `rollback`.
- **Пассивная материализация**. Не все объекты должны материализоваться одновременно. Конкретный экземпляр должен материализоваться только в случае необходимости — по требованию.
- **Виртуальные посредники**. Пассивная материализация реализуется с использованием продуманных ссылок, получивших название *виртуального механизма посредника*.

## 34.6. Шаблон представления объектов в виде таблиц

Как установить соответствие объекта с его представлением в реляционной базе данных?

Шаблон Representing Objects as Tables (Представление объектов в виде таблиц) [25] предлагает для каждого класса объектов, подлежащих постоянному хранению, определить отдельную таблицу (при использовании реляционной базы данных), а атрибуты объекта, содержащие данные простых типов (числа, строки, логические переменные и т.д.), хранить в отдельных столбцах.

Если все атрибуты объекта являются данными простых типов, то установка такого соответствия не представляет сложностей. Однако реальность не столь проста, поскольку одни объекты могут содержать ссылки на другие сложные объекты, а реляционная модель базы данных требует, чтобы эти значения были простыми (первая нормальная форма) (рис. 34.1).

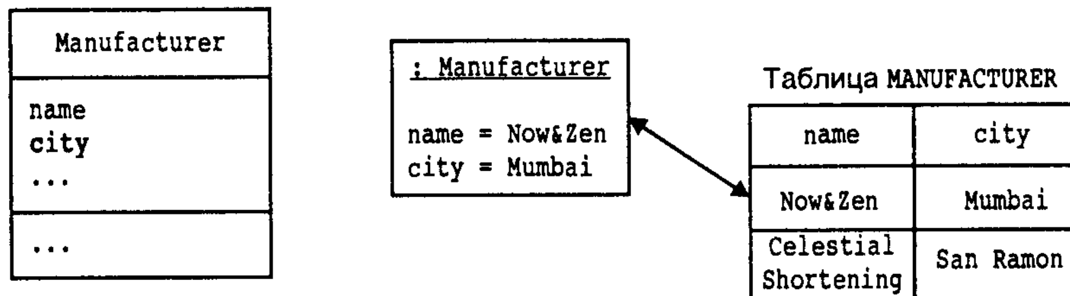



*Рис. 34.1. Соответствие объектов и таблиц*

## 34.7. Профиль моделирования данных UML

Если речь идет о хранении объектов в базе данных, то вполне естественно предположить наличие в рамках UML специальных обозначений для моделей данных. Напомним, что *модель данных* является одним из официальных артефактов UP и относится к дисциплине проектирования. Некоторые обозначения, применяемые в UML для моделирования данных, приводятся на рис. 34.2.

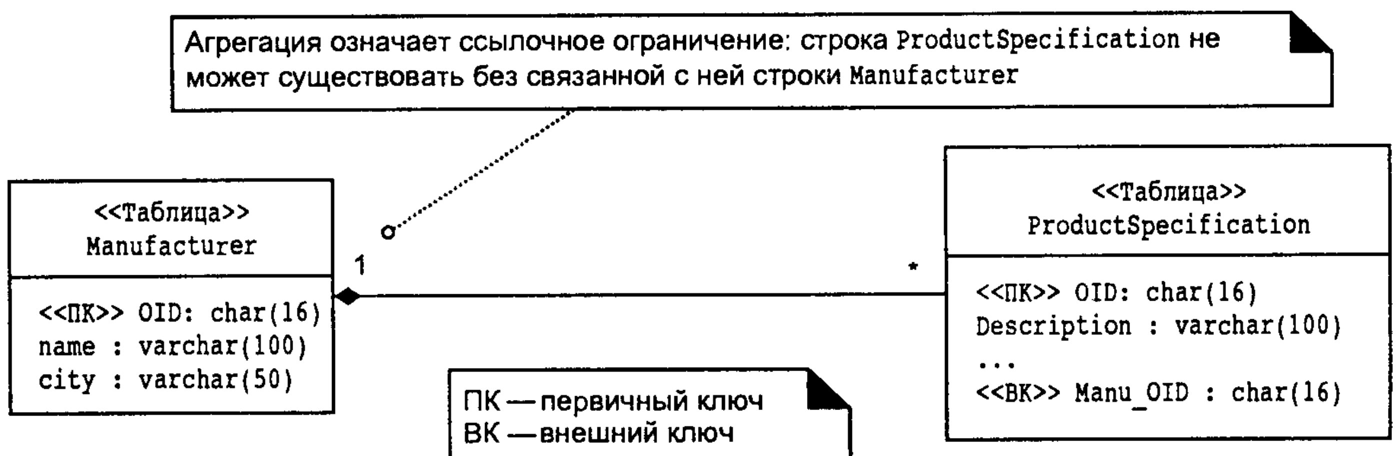



*Рис. 34.2. Пример модели данных в UML*

Представленные на рис. 34.2 стереотипы не относятся к ядру UML — они составляют часть расширения. В целом, в UML существует понятие *профиля UML* (UML profile), под которым понимается набор взаимосвязанных стереотипов, тегированных значений и ограничений, объединенных общим предназначе-

нием. На рис. 34.2 показана часть профиля моделирования данных UML, который на момент написания этой книги еще не был утвержден официально организацией OMG. На самом деле не все профили должны утверждаться официально, однако наиболее типичные (такие как профиль моделирования базы данных) предлагаются для утверждения.

## 34.8. Шаблон Object Identifier

Желательно разработать приемлемый способ взаимосвязи объектов с записями, а также гарантировать, что многократная материализация записей не приведет к дублированию объектов.

Согласно шаблону Object Identifier (Идентификатор объектов) [25], каждой записи и объекту (или объекту-посреднику) присваивается идентификатор объекта (OID — Object identifier).

Идентификатор объекта обычно представляет собой буквенно-цифровое значение. Такой идентификатор является уникальным для каждого конкретного объекта. Существуют различные подходы к формированию идентификаторов. Некоторые из них предполагают уникальность идентификатора в рамках одной базы данных, другие основаны на полной уникальности идентификаторов [3].

В рамках объектного подхода идентификатор объекта представляется интерфейсом или классом OID, инкапсулирующим реальное значение и его представление. В реляционных базах данных идентификатор представляет собой символьную строку фиксированной длины.

Каждая таблица реляционной базы данных в качестве первичного ключа использует идентификатор объекта, прямо или косвенно связанный с каждым объектом. Если каждый объект имеет свой идентификатор, используемый в качестве первичного ключа соответствующей таблицы, то каждому экземпляру объекта можно единственным образом сопоставить некоторую запись этой таблицы (рис. 34.3).

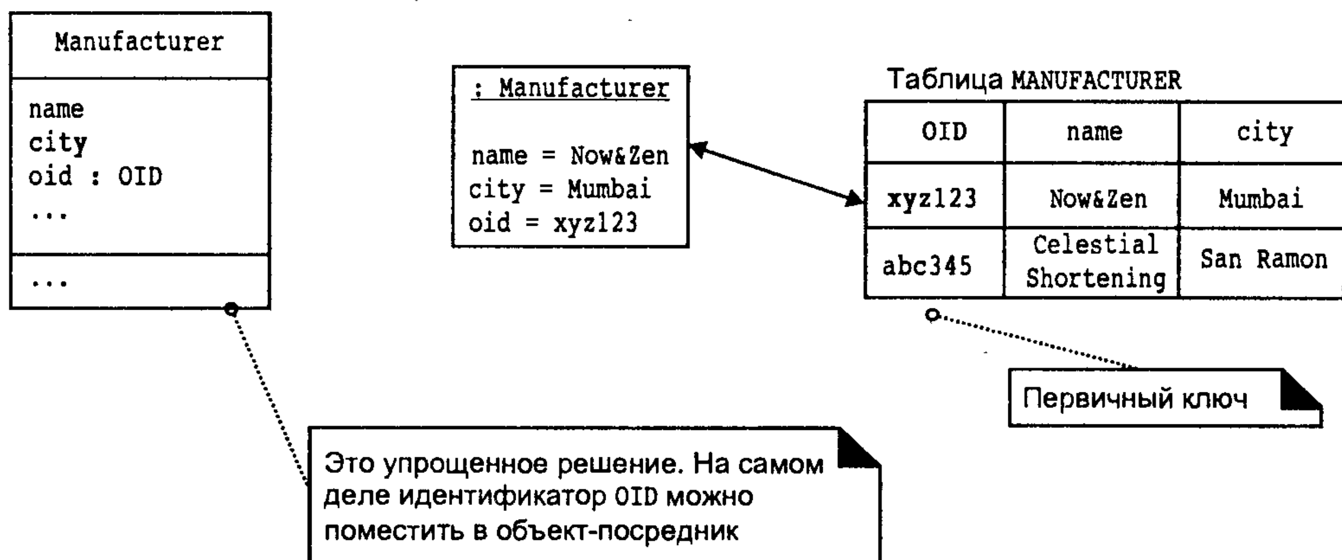



*Рис. 34.3. Объекты и записи связываются посредством идентификаторов объектов*

В этом состоит основная идея проектного решения. На самом деле идентификатор объекта не обязательно помещать в объект, предназначенный для постоянного хранения, хотя возможен и такой вариант. Его можно поместить в

объект-посредник (такой подход будет описан ниже). Конкретное проектное решение определяется также выбором языка программной реализации.

Идентификаторы объектов используются для взаимодействия с базой данных.

## 34.9. Доступ к службе взаимодействия с базой данных на основе шаблона Facade

Первым шагом на пути решения этой задачи является определение внешнего интерфейса для службы взаимодействия с базой данных (ее "фасада"). Напомним, что шаблон Facade обеспечивает унифицированный интерфейс для подсистем. Для начала нужна операция, позволяющая восстановить объект по его идентификатору. Однако помимо идентификатора объекта подсистеме нужно знать тип материализуемого объекта. Следовательно, параметром этой операции должен быть и класс объекта. На рис. 34.4 показаны некоторые операции, реализуемые согласно шаблону Facade для одного из адаптеров системы NextGen.

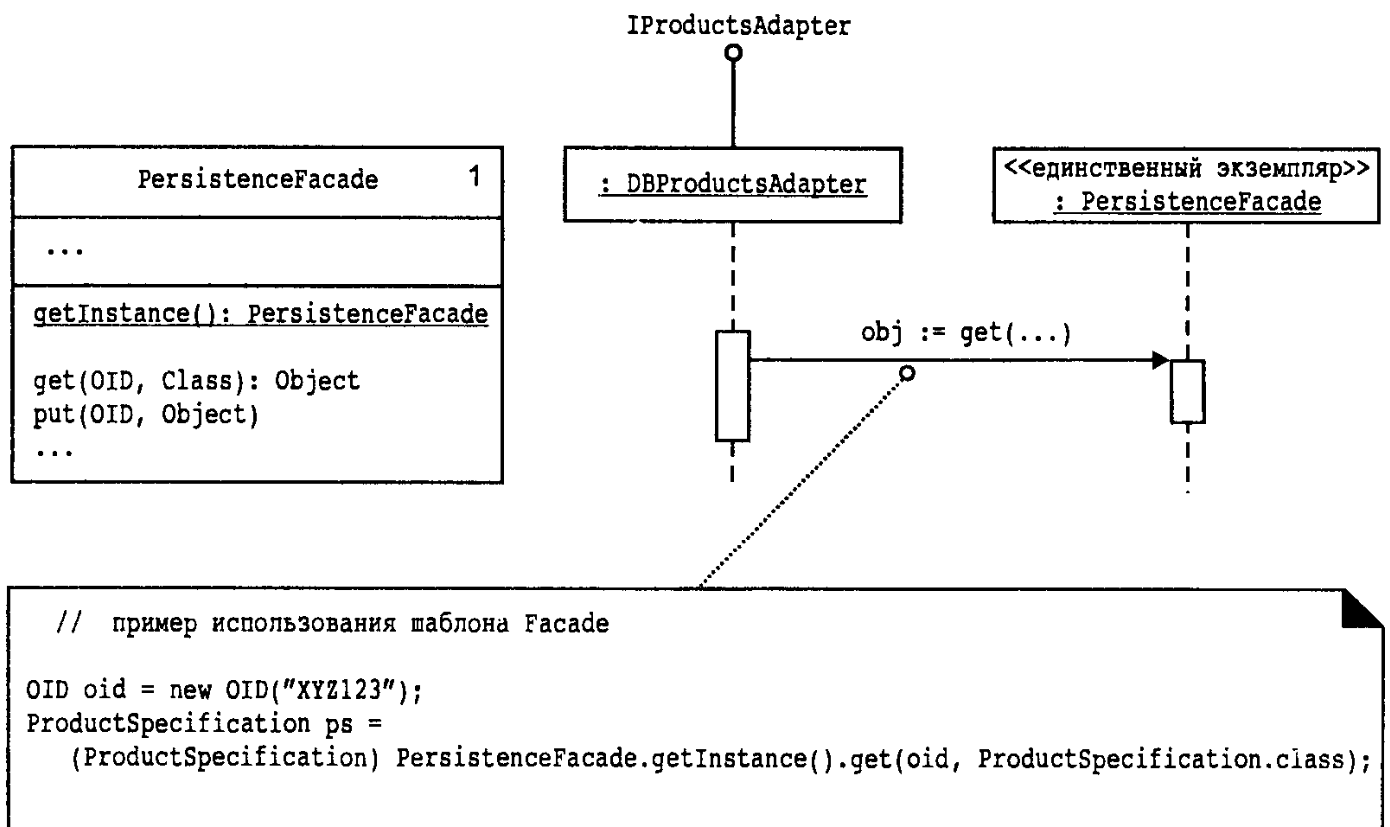



*Рис. 34.4. Применение шаблона Facade для взаимодействия с базой данных*

## 34.10. Объекты-преобразователи: шаблон Database Mapper или Database Broker

Объект `PersistenceFacade`, как и любой другой "фасадный" объект, сам не выполняет никаких действий, а делегирует запросы объектам подсистем.

Кто должен отвечать за материализацию и дематериализацию объектов (например, объекта `ProductSpecification`)?

Согласно шаблону Information Expert, эту обязанность можно возложить на сам класс (`ProductSpecification`), поскольку он обладает некоторой информацией (сохраняемыми данными), необходимой для выполнения этой обязанности.

Если объект, подлежащий постоянному хранению, содержит код для сохранения самого себя в базе данных, то такое проектное решение называется *прямым отображением* (direct mapping). Прямое отображение применимо в том случае, если код для взаимодействия с базой данных автоматически генерируется и внедряется в описание класса постпроцессором, и разработчику не требуется поддерживать этот код для своих классов.

Если же код для прямого отображения приходится добавлять и поддерживать вручную, то такой подход обладает рядом недостатков, в том числе следующими.

- Связывание класса объектов, подлежащих постоянному хранению, со структурой базы данных (с нарушением принципов шаблона Low Coupling).
- Сложные обязанности в новых и несвязанных между собой областях (в разрез с шаблоном High Cohesion). При этом обязанности технических служб смешиваются с обязанностями объектов уровня логики приложения.

В дальнейшем будет использован классический подход *непрямого отображения* (indirect mapping), при котором для преобразования объектов в структуру базы данных используются другие объекты.

В рамках этого подхода применяется шаблон Database Broker (Брокер базы данных) [24]. Согласно этому шаблону, создается класс, отвечающий за материализацию, дематериализацию и кэширование объектов. Этот шаблон называют также Database Mapper (Преобразователь базы данных) [48]. Это название автору кажется более предпочтительным, поскольку оно описывает обязанности класса, а термин "брокер" в распределенных системах [15] имеет другое устоявшееся значение.[1]

Для каждого класса объектов, подлежащих хранению в базе данных, определяется отдельный класс-преобразователь. Из рис. 34.5 видно, что для каждого объекта, подлежащего хранению в базе данных, может существовать собственный класс-преобразователь, и для различных механизмов хранения данных могут использоваться различные виды преобразователей. Приведем пример кода.

```
class PersistenceFacade
{
//...
public Object get(OID oid, Class persistenceClass)
{
    // объект IMapper выбирается на основе типа объекта,
    // подлежащего хранению в базе данных
    IMapper mapper = (IMapper) mappers.get( persistenceClass );

    // делегирование прав
    return mapper.get( oid );
}
//...
}
```

[1] В распределенных системах *брокером* называют внешний серверный процесс, делегирующий задачи внутренним серверным процессам.

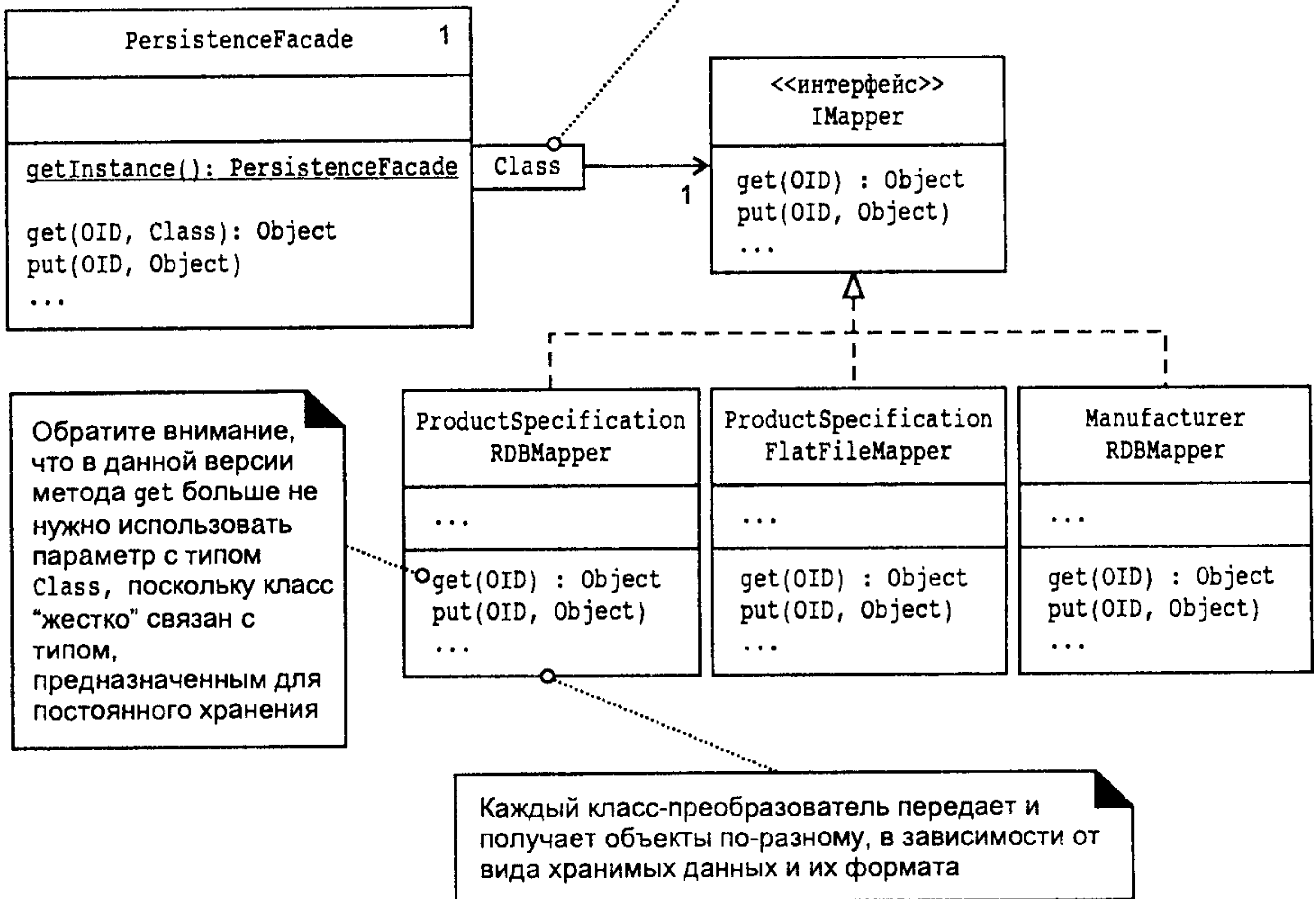



*Рис. 34.5. Преобразователи базы данных*

Хотя на этой диаграмме показаны два класса-преобразователя для объекта `ProductSpecification`, для каждой конкретной базы данных будет использован только один из них.

## Преобразователи на основе метаданных

Более гибким, но более сложным является подход к созданию преобразователей на основе *метаданных* (данных о данных). Чтобы не создавать каждый конкретный класс-преобразователь для каждого класса объектов, подлежащих хранению в базе данных, преобразователи на основе метаданных динамически генерируют отображение объектного представления в некоторое другое (например, в структуру реляционной базы данных). Это делается путем считывания описания преобразования из метаданных, например, "`TableX` отображается в `Class Y`, столбец `Z` отображается в свойство объекта `P`" (или еще более сложного описания). Такой подход применим для языков с возможностями рефлексивного программирования, таких как Java, C# или Smalltalk, а для остальных, например C++, — нет.

Преобразователи на основе метаданных позволяют изменять схему отображения для уже функционирующего приложения без модификации исходного

кода (что согласуется с шаблоном Protected Variations в смысле изменения схемы отображения).

Тем не менее, важнейшим свойством рассматриваемого здесь контура на основе обычных преобразователей или метаданных является инкапсуляция его реализации, т.е. возможность использования без изменения клиента.

## 34.11. Разработка контура на основе шаблона Template Method

В следующем разделе описываются важнейшие особенности проектирования преобразователей. Эти принципы проектирования основываются на шаблоне Template Method (Метод-шаблон) из группы GoF [52].[2] Этот шаблон является основным принципом проектирования контуров[3] и знаком большинству программистов если не по названию, то по содержанию.

Его идея состоит в определении в суперклассе метода (метода-шаблона), задающего скелет алгоритма и содержащего варьируемую и неизменную части. Метод-шаблон вызывает другие методы, часть из которых может перекрываться в производных классах. Таким образом, производные классы могут перекрывать варьируемые методы и добавлять собственную функциональность (рис. 34.6).

## 34.12. Материализация на основе шаблона Template Method

При создании двух или трех классов-преобразователей некоторые фрагменты кода неизбежно повторяются. Базовая структура алгоритма материализации объекта имеет следующий вид.

```
if (объект в буфере)
    возвратить его
else
    создать объект на основе его представления в базе данных
    сохранить объект в буфере
    возвратить его
```

Точкой вариации является конкретный способ создания объекта.

Создадим метод-шаблон `get` абстрактного суперкласса `AbstractPersistenceMethod`.

Проектное решение для такого метода-шаблона представлено на рис. 34.7.

---

[2] Этот шаблон никак не связан с шаблонами в C++, поскольку он описывает шаблон алгоритма.

[3] Более точно, контуров в виде "белого ящика". Под такими контурами обычно понимают иерархию классов и контуров, требующих от пользователя знания их внутренней структуры и проектных решений.

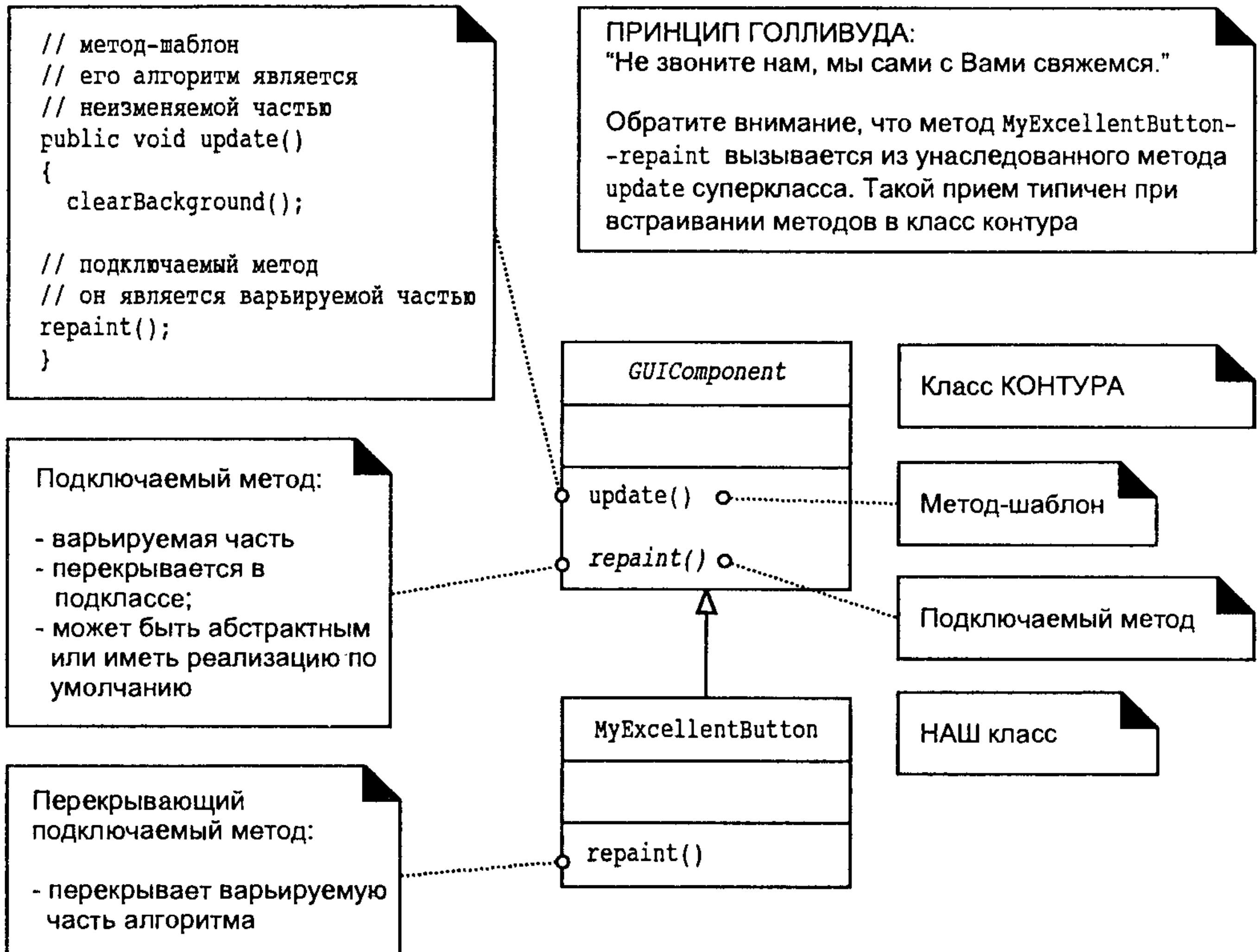



*Рис. 34.6. Шаблон Template Method в контуре графического интерфейса пользователя*

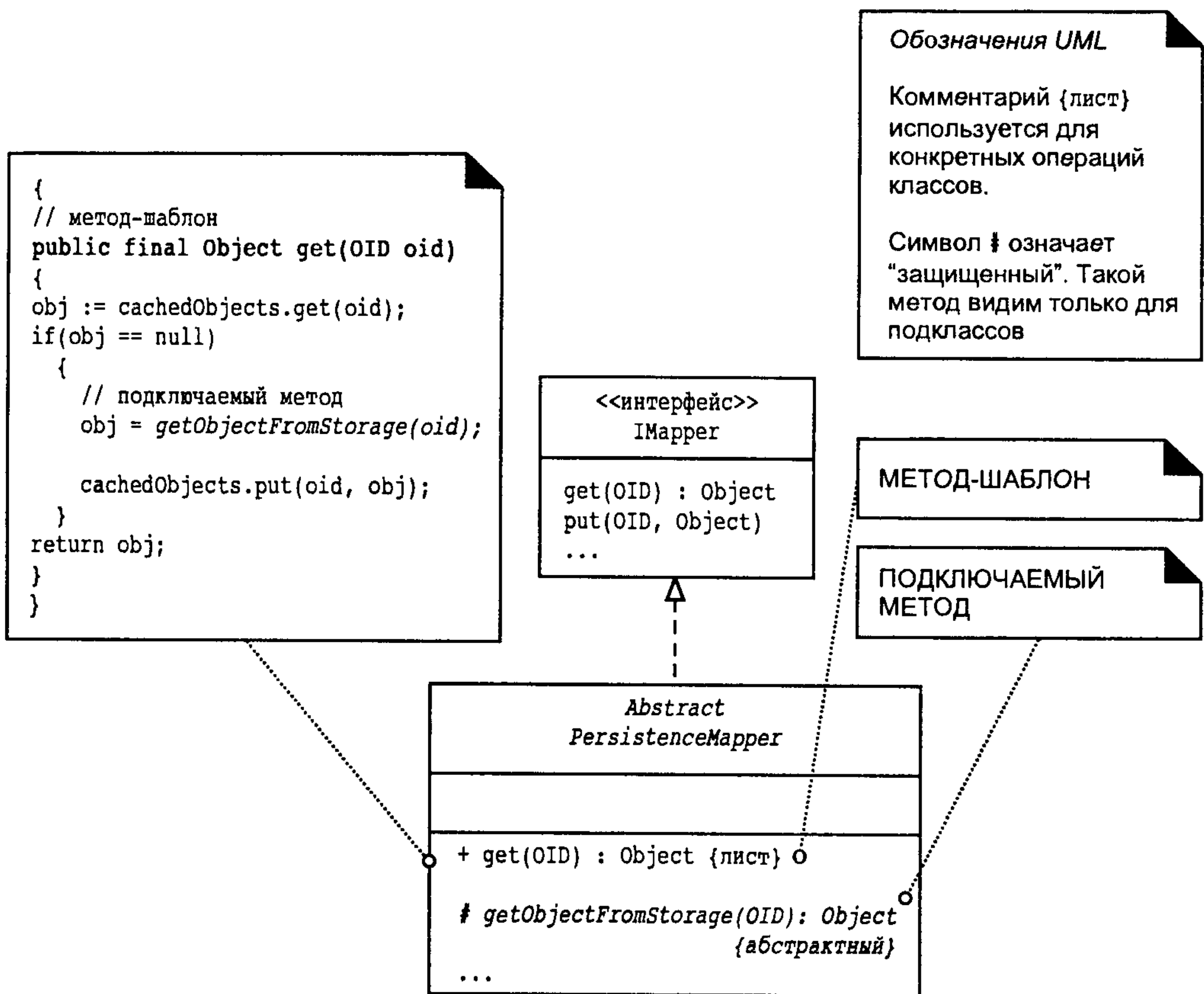



*Рис. 34.7. Применение шаблона Template Method для объектов-преобразователей*

Как видно из этого примера, метод-шаблон обычно является открытым (`public`), а подключаемый метод — защищенным (`protected`). Классы `AbstractPersistenceMapper` и `IMapper` являются частью контура взаимодействия с базой данных. Теперь разработчик может подключать свои методы в этот контур, добавляя собственные подклассы и перекрывая или реализуя метод `getObjectFromStorage`. Пример такой реализации показан на рис. 34.8.

Предположим, что в реализации подключаемого метода на рис. 34.8 начальная часть алгоритма, в которой выполняется SQL-оператор `SELECT`, одинакова для всех объектов. Варьируется только имя таблицы базы данных.[4] При таком предположении можно снова применить шаблон Template Method и разделить изменяемую и неизменную части алгоритма. На рис. 34.9 метод `AbstractRDBMapper--getObjectFromStorage` является подключаемым с точки зрения метода `AbstractPersistenceMapper--get`, но становится шаблоном для вновь подключаемого метода `getObjectFromRecord`.

[4] Зачастую все не так просто. Объект может извлекаться из нескольких таблиц или даже различных баз данных. В этом случае первая версия шаблона Template Method является более гибкой.

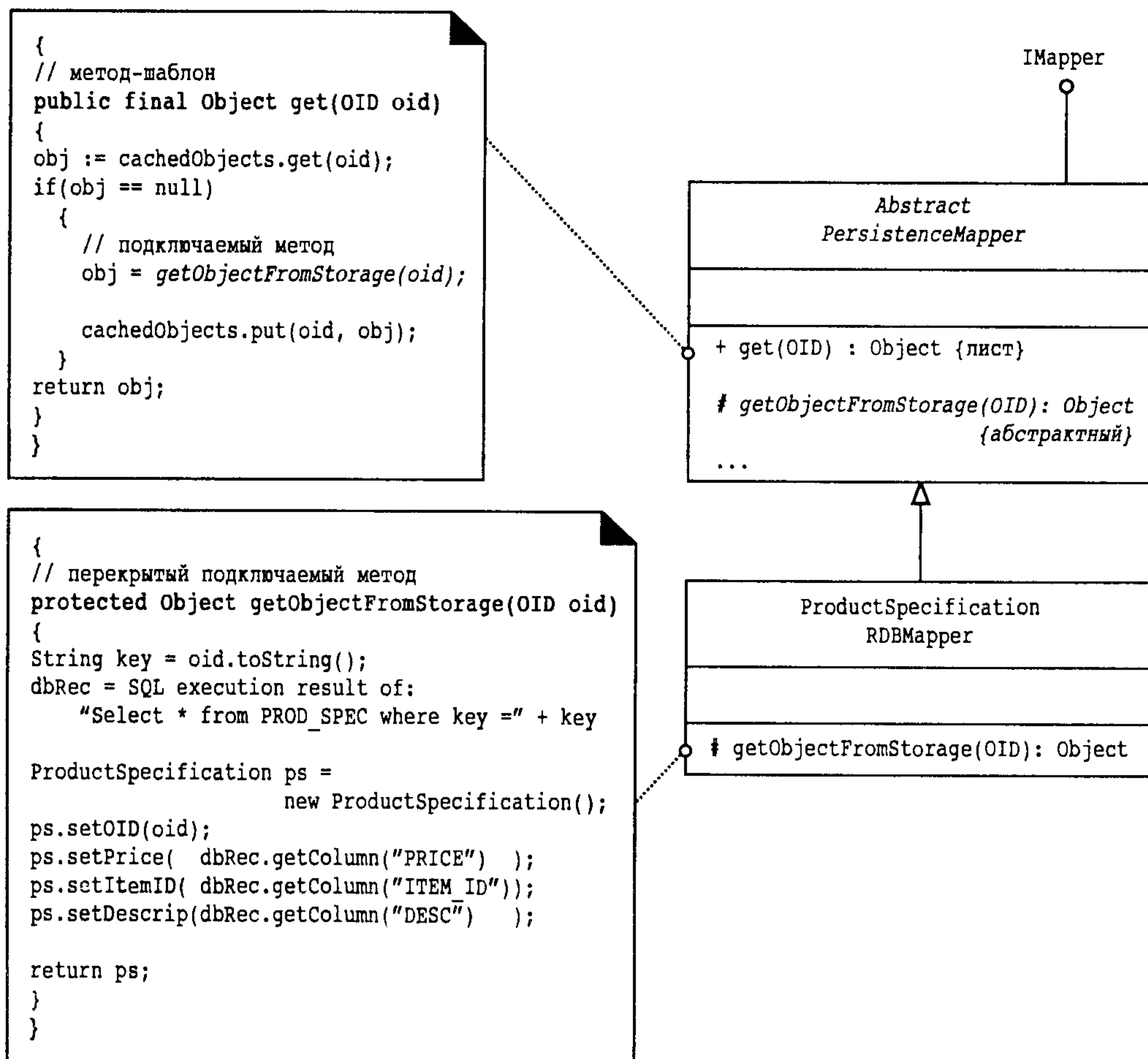



*Рис. 34.8. Перекрытие подключаемого метода*[5]

Обратите внимание на объявление конструкторов в UML. Стереотип здесь использовать необязательно, особенно если имя конструктора совпадает с именем класса.

Теперь в состав контура входят классы `IMapper`, `AbstractPersistenceMapper` и `AbstractRDBMapper`. Разработчику приложения остается добавить свои собственные производные классы, например `ProductSpecificationRDBMapper`, для соответствующей таблицы (имя которой должно быть передано конструктору объекта).

[5] В Java объект `dbRec`, возвращаемый после выполнения SQL-запроса, будет представлять собой результирующее множество JDBC.

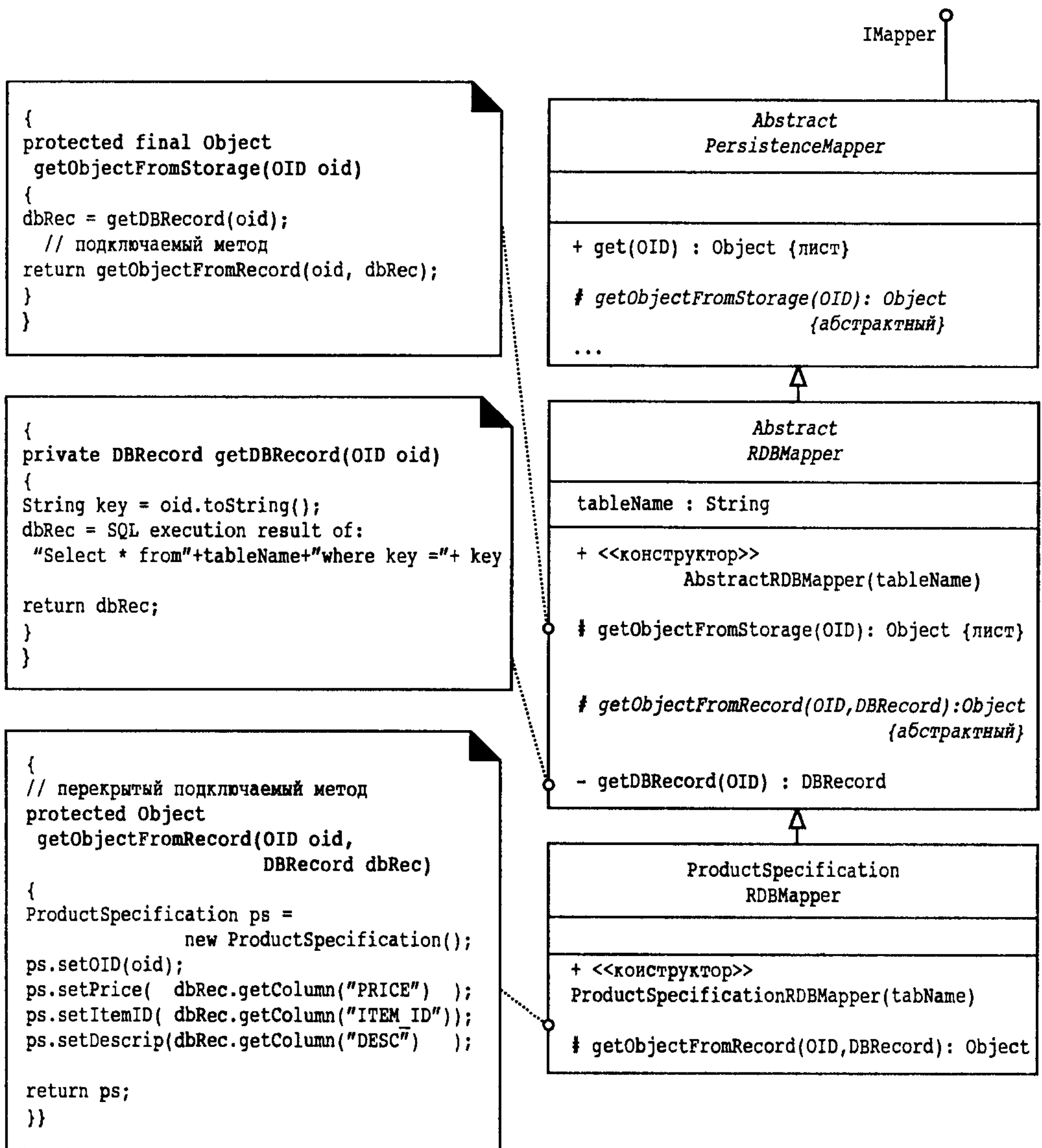



*Рис. 34.9. Повторное применение шаблона Template Method*

Иерархия классов-преобразователей базы данных составляет важную часть контура. Разработчики приложения могут добавлять свои производные классы в эту иерархию и настраивать свое приложение для работы с новыми механизмами хранения данных или новыми таблицами и файлами в рамках существующего механизма хранения. На рис. 34.10 показаны структуры некоторых пакетов и классов. Заметим, что классы, составляющие специфику приложения NextGen, не относятся к пакету общих технических служб `Persistence`. Эта диаграмма в сочетании с диаграммой на рис. 34.9 демонстрирует важность визуальных языков типа UML для описания частей программного обеспечения. Подобные диаграммы содержат много наглядной информации.

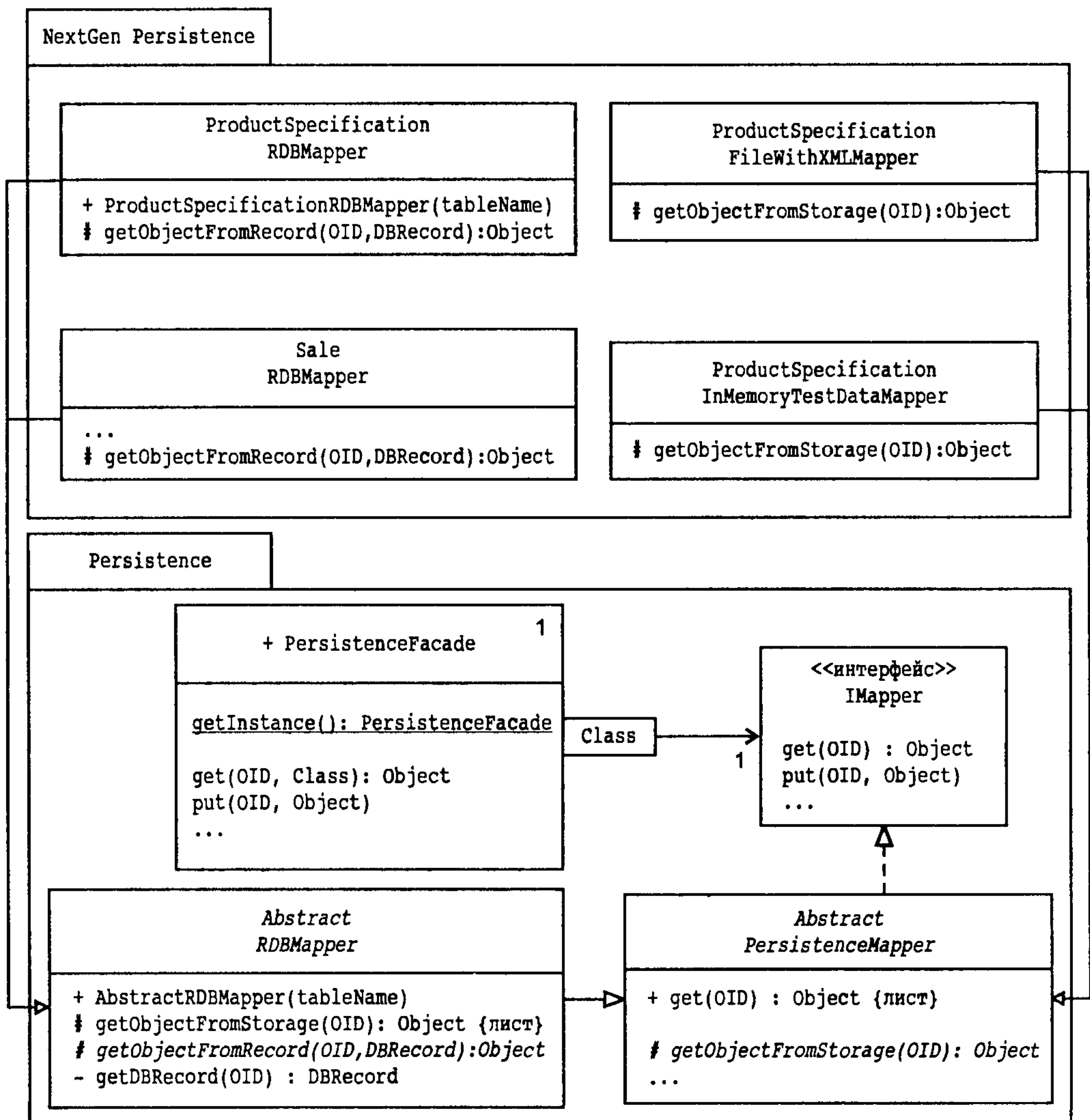



*Рис. 34.10. Контур взаимодействия с базой данных*

Обратите внимание на класс `ProductSpecificationInMemoryTestDataMapper`. Такие объекты можно использовать для тестирования программы без привлечения внешних баз данных.

## Унифицированный процесс и описание программной архитектуры

В рамках UP важное значение имеет документ с описанием программной архитектуры. Он позволяет разработчикам будущих систем ознакомиться с опытом своих предшественников и наиболее ценными архитектурными решениями. В этот документ обычно включаются диаграммы, подобные представленным на рис. 34.9 и 34.10.

### *Синхронизированные методы в UML*

Метод `AbstractPersistenceMapper--get` содержит не безопасный с точки зрения многопоточности фрагмент кода — один и тот же объект можно одновременно материализовать в различных потоках. Как и все подсистемы технических служб, службу взаимодействия с базой данных нужно реализовывать с учетом безопасности потоков. Действительно, функции системы могут выполняться в различных потоках или даже на различных компьютерах. Тогда объект `PersistenceFacade` становится удаленным серверным объектом, а подсистема может функционировать одновременно в нескольких потоках, обслуживая разных клиентов.

Поэтому данный метод должен контролировать параллельное функционирование потоков. В Java это достигается за счет использования ключевого слова `synchronized`. На рис. 34.11 показан фрагмент диаграммы классов, содержащий синхронизированный метод.

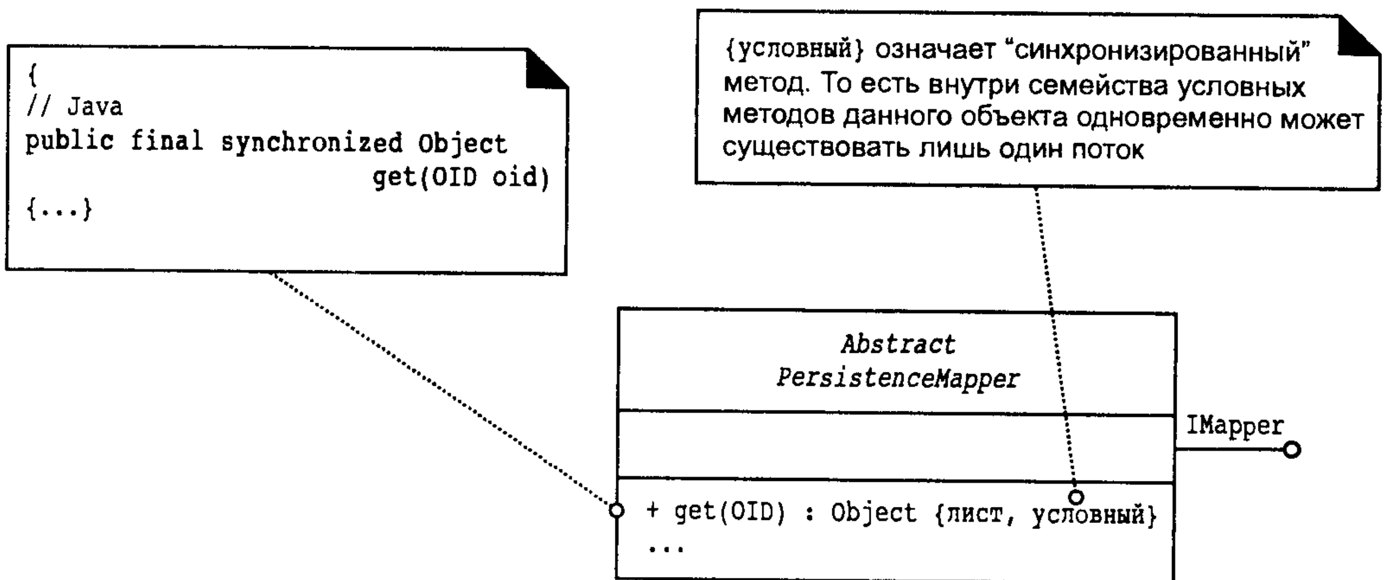



*Рис. 34.11. Синхронизированный метод в UML*

## 34.13. Настройка преобразователей с помощью объекта MapperFactory

Аналогично приведенному выше примеру использования шаблона Factory, объекты `IMapper` для "фасадного" объекта `PersistenceFacade` тоже можно конфигурировать с помощью объекта-фабрики `MapperFactory`. Однако в этом случае не желательно для каждой операции определять отдельный преобразователь. Например, следующий фрагмент нельзя считать удачным.

```
class MapperFactory
{
public IMapper getProductSpecificationMapper(){...}
public IMapper getSaleMapper(){...}
...
}
```

При таком подходе нарушается шаблон Protected Variations, поскольку не обеспечивается возможность добавления новых преобразователей. Поэтому более предпочтительным является следующий вариант.

```
class MapperFactory
{
public Map getAllMappers(){...}
...
}
```

Здесь классами являются ключи java.util.Map (реализованные, возможно, с помощью HashMap), а IMapper становятся экземплярами с конкретными значениями.

Поэтому фасадный объект может инициализировать коллекцию IMapper следующим образом.

```
class PersistenceFacade
{
private java.util.Map mappers =
    MapperFactory.getInstance().getAllMappers();
}
```

Значения объектам IMapper можно присвоить следующим образом. Объект-фабрика может считывать системные свойства для выяснения, какой из классов IMapper нужно инстанцировать. При использовании языка с возможностями рефлексивного программирования, например Java, инстанцирование может быть основано на считывании имен классов в виде строк и использовании операции Class.newInstance. Тогда набор преобразователей можно модифицировать без изменения исходного кода.

## 34.14. Шаблон Cache Management

С целью повышения производительности системы материализованные объекты желательно хранить в локальном буфере (поскольку материализация — это сравнительно медленная операция), обеспечивая при этом поддержку операций управления транзакциями типа commit.

Согласно шаблону Cache Management (Управление буфером) [25], обязанность по поддержке операций буферизации возлагается на классы-преобразователи базы данных. Если для каждого класса объектов, хранящихся в базе данных, используется отдельный преобразователь, то этот преобразователь поддерживает буферизацию своих объектов.

После материализации объекты помещаются в буфер, а в качестве ключа используется их идентификатор. При запросе на определенный объект преобразователь сначала выполняет поиск объекта в буфере, что позволяет избежать дополнительных операций материализации.

## 34.15. Объединение и сокрытие операторов SQL в одном классе

Кодирование операторов SQL в различных классах-преобразователях контура взаимодействия с базой данных не является преступлением. Однако такую реализацию можно улучшить. Примем следующие предположения.

- Существует один чисто синтетический класс (удовлетворяющий шаблону Singleton) RDBOperations, в котором сосредоточены все операторы SQL (SELECT, INSERT, ...).

- Классы-преобразователи взаимодействуют с этим объектом для получения одной или нескольких записей из базы данных (например, ResultSet).
- Интерфейс этого класса имеет следующий вид.

```
class RDBOperations
{
public ResultSet getProductSpecificationData(OID oid) {...}
public ResultSet getSaleData(OID oid) {...}
...
}
```

Тогда преобразователь должен быть реализован следующим образом.

```
class ProductSpecificationRDBMapper extends
AbstractPersistenceMapper
{
protected Object getObjectFromStorage(OID oid)
{
ResultSet rs =
    RDBOperations.getInstance().getProductSpecificationData(oid);

ProductSpecification ps = new ProductSpecification();
ps.setPrice(rs.getDouble("PRICE"));
ps.setOID(oid);
return ps;
}
}
```

Использование шаблона Pure Fabrication обеспечивает следующие преимущества.

- Простота поддержки и настройка производительности специалистами. Для оптимизации производительности SQL требуются эксперты именно в этой области, а не специалисты по объектному программированию. Если все операторы SQL сосредоточены в одном классе, то этот код можно предложить для оптимизации специалисту по SQL.
- Инкапсуляция деталей реализации. Например, операторы SQL, обеспечивающие получение данных, можно заменить вызовом хранимых процедур реляционной базы данных. Можно использовать и более тонкий подход, основанный на метаданных, и динамически генерировать операторы SQL по схеме описания метаданных, считываемой из внешнего источника.

С точки зрения архитектора, интересный аспект данного проектного решения состоит в том, что оно учитывает навыки разработчика. При этом достигается баланс между высокой степенью зацепления и удобством для специалиста. Не все проектные решения определяются только принципами проектирования программных систем, такими как связывание или зацепление.

## 34.16. Состояние транзакции и шаблон State

Вопросы поддержки транзакций достаточно нетривиальны, поэтому для простоты (чтобы сосредоточить внимание на применении шаблона State), примем следующие предположения.

- Объекты, подлежащие хранению в базе данных, можно добавлять, удалять или модифицировать.
- Действия над постоянно хранимыми объектами (например, их модификация) не приводят к немедленному изменению базы данных, а выполняются в рамках операции commit.

Кроме того, будем считать, что результат операции зависит от состояния транзакции для данного объекта. Пример результатов подобных операций показан на диаграмме состояний на рис. 34.12.

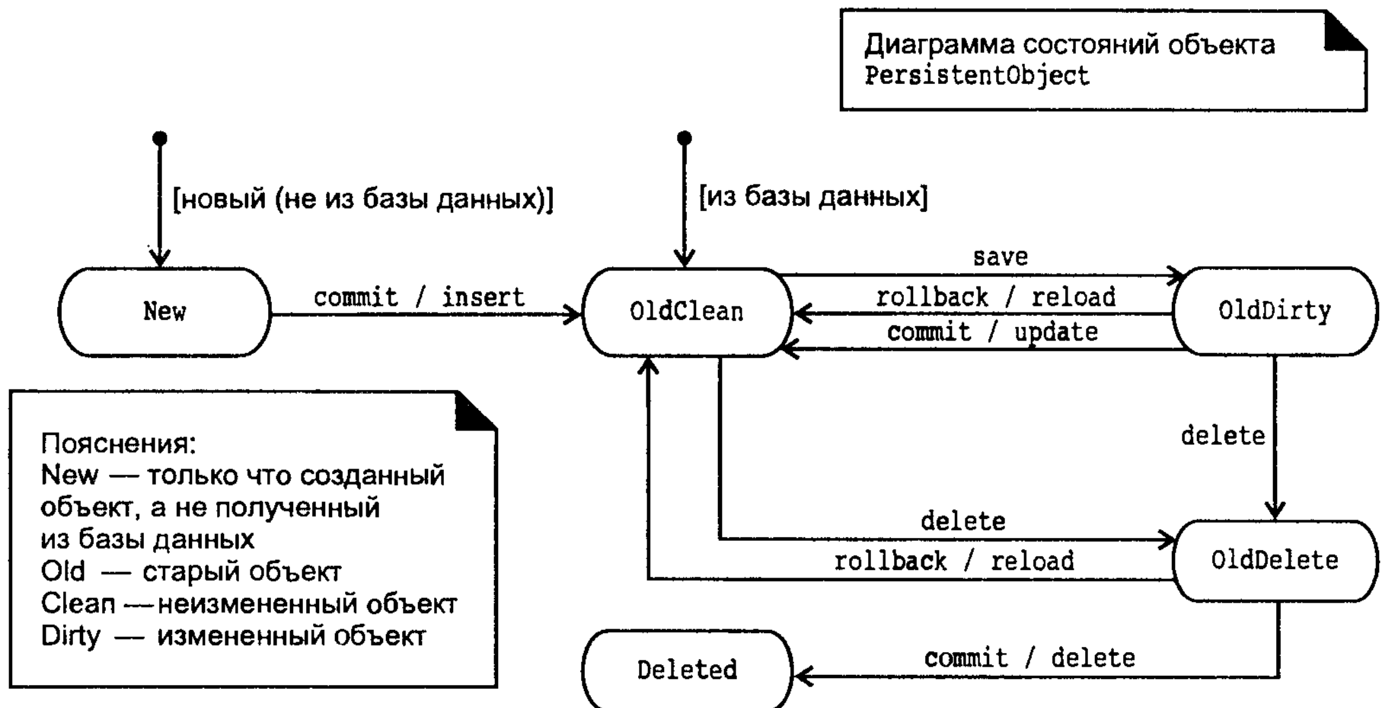



*Рис. 34.12. Диаграмма состояний для объекта* PersistentObject

Например, старым измененным объектом (OLD_DIRTY) будем считать объект, извлеченный из базы данных и модифицированный. При выполнении операции commit этот объект необходимо обновить в базе данных (в отличие от старого неизменного объекта (OLD_CLEAN), который после извлечения из базы данных не изменялся). При использовании контура взаимодействия с базой данных операции сохранения или удаления не выполняются немедленно. Просто подлежащие хранению объекты переходят в определенное состояние, а необходимые действия выполняются при завершении или отмене транзакции.

Для UML изменение состояний в рамках транзакций очень полезно отображать на диаграмме состояний.

Предположим, что все классы объектов, подлежащих хранению в базе данных, расширяют класс PersistentObject,[6] обеспечивающий реализацию общих технических служб для взаимодействия с базой данных[7]. Пример такой ситуации представлен на рис. 34.13.

---

[6] В [5] хорошо описан класс PersistentObject и уровни взаимодействия с базой данных, хотя идея использования такого класса появилась гораздо раньше.

[7] Некоторые вопросы расширения класса PersistentObject обсуждаются ниже. Если класс, представляющий понятие предметной области, расширяет класс технических служб, то это приводит к смешиванию архитектурных уровней (базы данных и логики приложения).

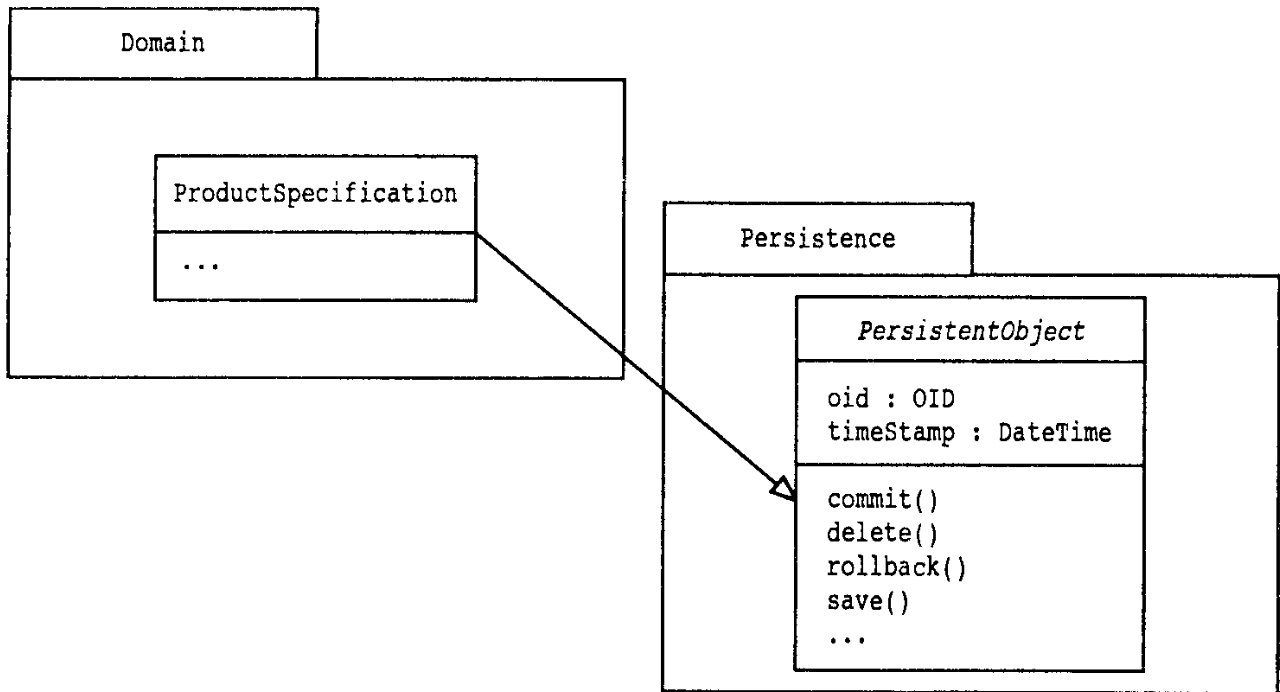



*Рис. 34.13. Объекты, подлежащие хранению в базе данных*

Заметим, что методы `commit` и `rollback` предусматривают одинаковую логику выполнения, основанную на состоянии транзакции. И хотя они выполняют различные действия, логическая структура этих методов одинакова.

```
public void commit()
{
switch ( state )
{
case OLD_DIRTY:
  // ...
  break;
case OLD_CLEAN:
  // ...
  break;
...
}
}
```

```
public void roolback()
{
switch ( state )
{
case OLD_DIRTY:
  // ...
  break;
case OLD_CLEAN:
  // ...
  break;
...
}
}
```

Чтобы многократно не использовать для решения поставленной задачи условные операторы, можно воспользоваться шаблоном State (Состояние) из набора GoF.

*Шаблон State*

*Контекст/Проблема*

Поведение объекта зависит от его состояния, а его методы содержат условную логику, отражающую выполнение различных действий в зависимости от состояния объекта. Существует ли альтернатива традиционной условной логике?

*Решение*

Для каждого состояния создать отдельный класс (класс состояния) со стандартным интерфейсом. Делегировать зависящие от состояния операции от контекстного объекта объекту его текущего состояния. Удостовериться, что контекстный объект всегда указывает на объект текущего состояния.

Рис. 34.14 иллюстрирует применение этого шаблона к подсистеме взаимодействия с базой данных.

Выполнение зависящих от состояния методов класса PersistentObject делегируется соответствующему объекту состояния. Если контекстный объект указывает на состояние OldDirtyState, значит, при реализации метода commit объект будет модифицирован в базе данных, а самому контекстному объекту будет присвоена ссылка на состояние OldCleanState. Если же контекстный объект указывает на состояние OldCleanState, то при реализации метода commit не будут выполнены никакие действия.

Обратите внимание, что диаграмма классов на рис. 34.14 соответствует диаграмме состояний на рис. 34.12. Шаблон State — один из механизмов программной реализации модели перехода состояний[8]. Он обеспечивает переход объекта в различные состояния в ответ на внешние события.

С точки зрения производительности, следует отметить, что классы состояний (по иронии судьбы) не зависят от состояния. Поэтому нет необходимости создавать несколько экземпляров объектов таких классов — каждый такой класс содержит по единственному экземпляру. Тысячи объектов, подлежащих хранению в базе данных, могут ссылаться на один и тот же экземпляр OldDirtyState.

## 34.17. Обработка транзакций на основе шаблона Command

В предыдущем разделе транзакции рассматривались несколько упрощенно. В этом разделе обсуждение транзакций будет продолжено, при этом будут освещены все вопросы, связанные с проектированием их обработки. Нестрого говоря, транзакция — это единица работы или набор задач, которые должны быть выполнены или отменены одновременно, т.е. транзакцию можно рассматривать как атомарную операцию.

В терминах службы взаимодействия с базой данных к задачам транзакции относятся добавление, удаление и модификация объектов. В рамках одной транзакции может быть добавлено два объекта, удалено три и один модифицирован. Для представления этих операций вводят класс Transaction [5][9]. Как указывается в [48], порядок выполнения задач в рамках одной транзакции может повлиять на ее успешное выполнение (и производительность).

Можно привести следующие примеры.

1. Допустим, база данных имеет ограничение целостности ссылок, которое сводится к следующему. При обновлении записи в таблице TableA, содержащей внешний ключ к записи в таблице TableB, соответствующая запись в TableB должна существовать.
2. Транзакция включает задачу INSERT, согласно которой в таблицу TableB добавляется запись, и задачу UPDATE по добавлению записи в таблицу TableA. Если операция UPDATE выполняется раньше, чем INSERT, то может возникнуть нарушение целостности ссылок.

---

[8] Существуют и другие механизмы, в том числе условная логика, интерпретаторы состояний и генераторы кода на основе таблиц состояний.

[9] В [48] такой класс назван UnitOfWork.

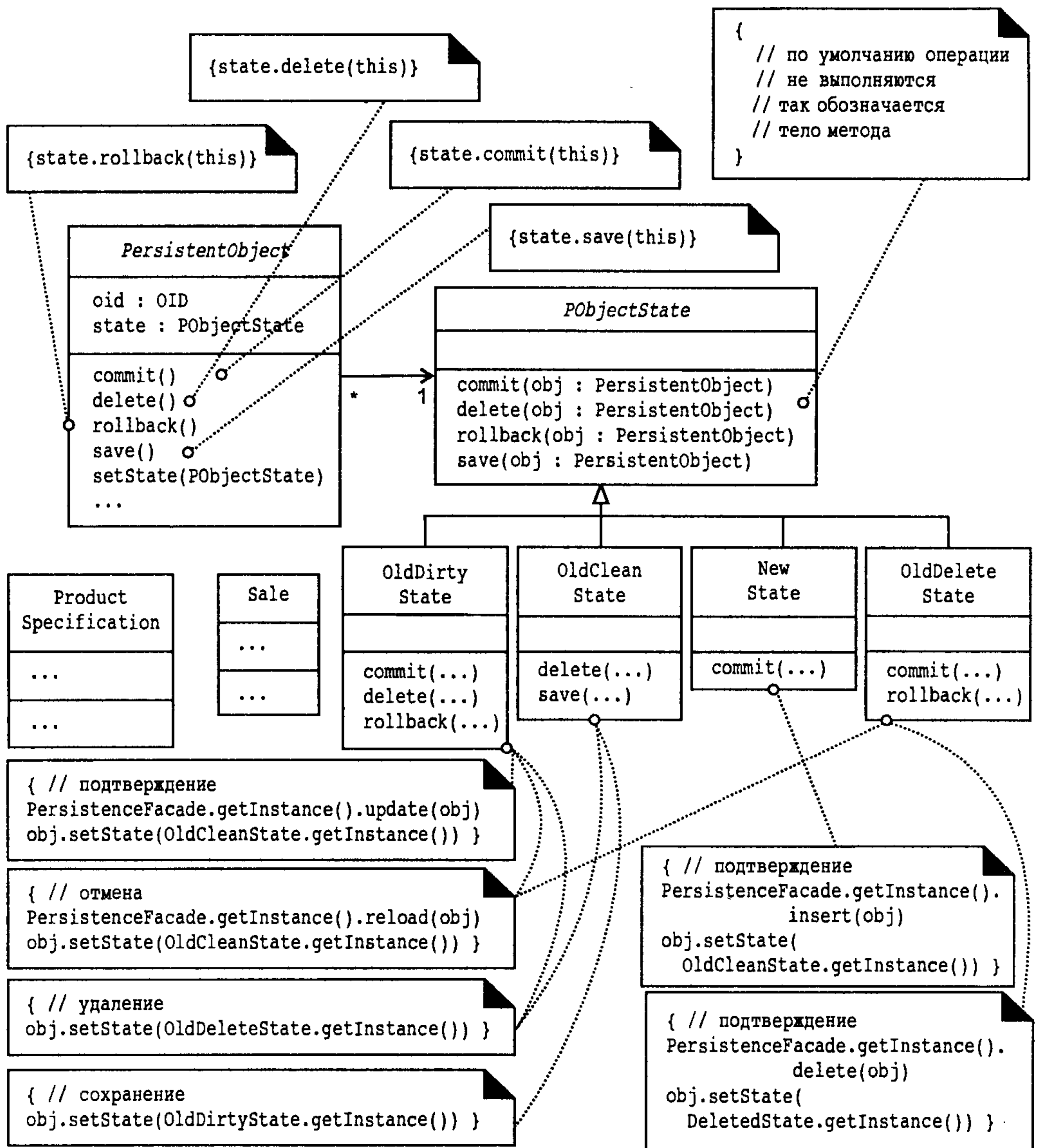



*Рис. 34.14. Применение шаблона State*[10]

Упорядоченность задач взаимодействия с базой данных может обеспечить значительные преимущества. Некоторые вопросы упорядочения зависят от структуры базы данных, но в целом стратегия такова: сначала выполняются операции добавления, затем обновления, и, наконец, удаления.

Нужно иметь в виду, что порядок задач в рамках транзакции может оказаться не оптимальным с точки зрения их выполнения. Эти задачи необходимо переупорядочить до начала выполнения.

[10] Класс `Deleted` не изображен на этой диаграмме с целью экономии места.

Такую проблему решает шаблон Command (Команда) из набора GoF.

> *Шаблон Command*
>
> *Контекст/Проблема*
>
> Как обрабатывать запросы или задачи, требующие предварительной сортировки (расстановки приоритетов), очередности, регистрации или задержки?
>
> *Решение*
>
> Для каждой задачи создать класс, реализующий общий интерфейс.

Это достаточно простой шаблон, имеющий множество полезных приложений. При его использовании действия становятся объектами, а значит, их можно сортировать, регистрировать, ставить в очередь и т.д. Например, на рис. 34.15 показаны классы для операций с базой данных, созданные на основе шаблона Command.

Обработка транзакций включает множество других вопросов, но основная идея этого раздела состоит в представлении каждой задачи или действия в виде объекта с полиморфным методом `execute`. Это повышает уровень гибкости при обработке запросов.

Замечательным примером применения шаблона Command является реализация команд графического интерфейса пользователя, подобных вырезанию и вставке. Например, метод `execute` объекта `CutCommand` выполняет вырезание, а метод `undo` — отменяет его. Для отмены операции объекту `CutCommand` требуются исходные данные. Все команды графического интерфейса пользователя поддерживают стек выполняемых действий, поэтому каждое действие можно отменить.

Еще одним примером использования шаблона Command является обработка запросов в серверной части приложения. При получении сообщения серверный объект создает объект-команду для этого запроса и передает ее специальному объекту `CommandProcessor` [15], который регистрирует, упорядочивает и выполняет команды.

## 34.18. Пассивная материализация на основе шаблона Virtual Proxy

Зачастую желательно отложить процесс материализации объекта на как можно более поздний срок обычно из соображений повышения производительности. Например, допустим, что объекты `ProductSpecification` ссылаются на объект `Manufacturer`, который должен извлекаться из базы данных очень редко. Информация о производителе требуется только в том случае, если этот производитель вводит скидки на свои товары.

Отсроченная во времени материализация "дочерних" объектов называется *пассивной материализацией* (lazy materialization) и может выполняться с использованием шаблона Virtual Proxy (Виртуальный объект-посредник) из набора GoF, представляющего собой одну из многих вариаций шаблона Proxy.

Виртуальный объект-посредник — это объект-посредник для другого объекта (*реального объекта* (real subject)), материализующий реальный объект при первом его использовании. Таким образом, реализуется пассивная материализация. Виртуальный объект-посредник — это "облегченный" объект, использую-

мый вместо реального объекта, который к настоящему времени может оказаться еще не материализованным.

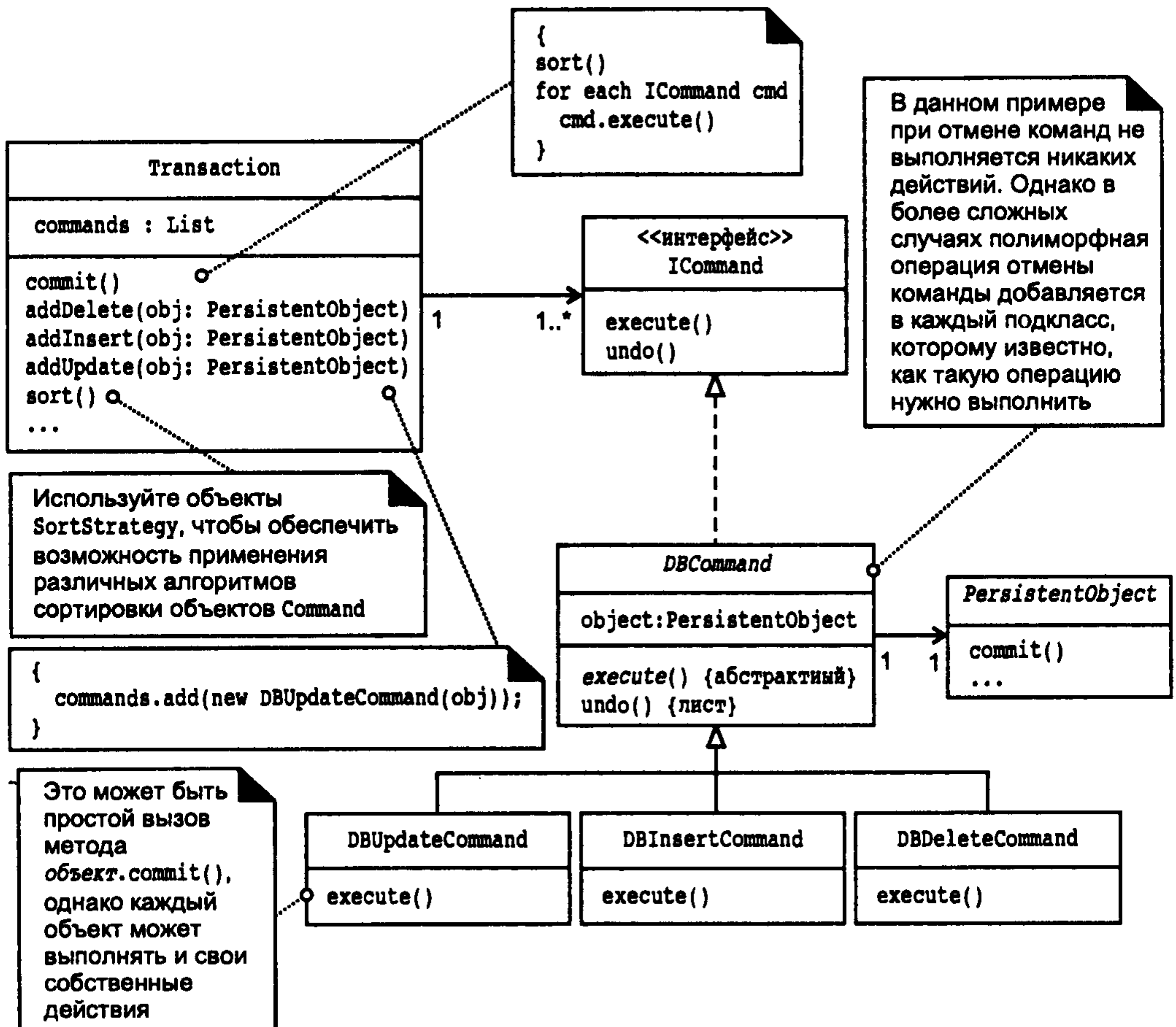



*Рис. 34.15. Классы команд для операций с базой данных*

Конкретный пример применения шаблона Virtual Proxy к классам ProductSpecification и Manufacturer представлен на рис. 34.16. Это проектное решение основано на предположении, что объекты-посредники знают идентификаторы своих реальных объектов и при необходимости материализации используют их для идентификации и извлечения реальных объектов.

Обратите внимание, что видимость между экземплярами ProductSpecification и IManufacturer обеспечивается посредством атрибутов. Имя производителя для данного экземпляра ProductSpecification может еще не быть материализовано в оперативной памяти. При отправке объектом ProductSpecification сообщения getAddress объекту-посреднику ManufacturerProxy (как материализованному объекту), объект-посредник материализует реальный экземпляр Manufacturer на основе его идентификатора.

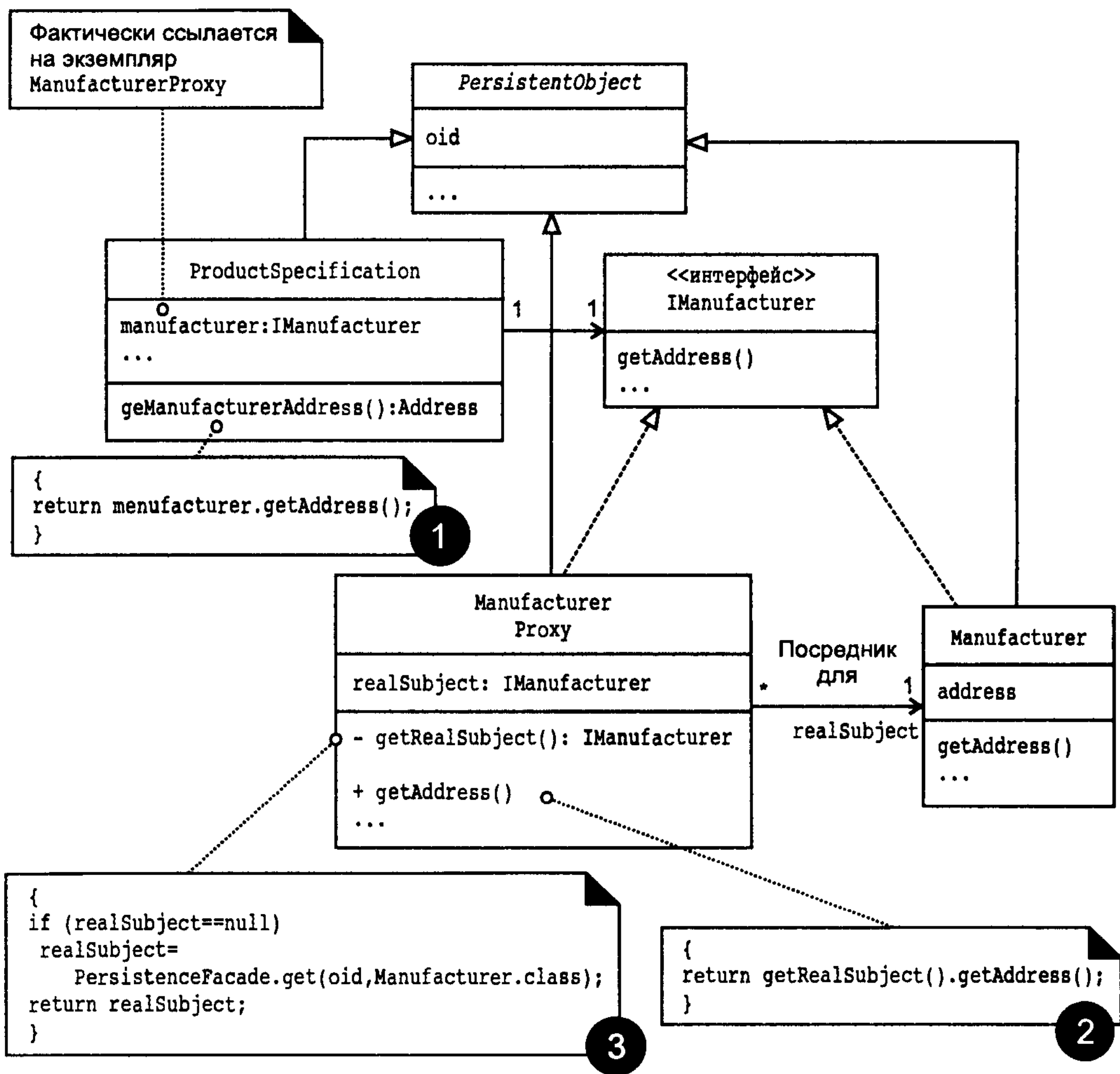



*Рис. 34.16. Виртуальный объект-посредник для класса* `Manufacturer`

## Реализация виртуального объекта-посредника

Реализация виртуального объекта-посредника зависит от языка программирования. Подробное рассмотрение этого вопроса не входит в задачи этой главы, поэтому рассмотрим лишь общие принципы его реализации на разных языках.

| Язык | Реализация виртуального объекта-посредника |
|---|---|
| C++ | Определяется шаблонный класс "интеллектуальных" указателей. Определение интерфейса `IManufacturer` не требуется |
| Java | Реализуется класс `ManufacturerProxy`. Определяется интерфейс `Imanufacturer`. Однако эти классы обычно не программируются вручную. Создается генератор кода, анализирующий реальные классы (например, `Manufacturer`), и генерируются объекты `IManufacturer` и `ProxyManufacturer`. Еще одной альтернативой для языка Java является использование программного интерфейса Dynamic Proxy |

| Язык | Реализация виртуального объекта-посредника |
|---|---|
| Smalltalk | Определяется виртуальный трансформируемый объект-посредник, который для преобразования в реальный объект использует определения `#doesNotUnderstand:` и `#become:`. Определения интерфейса `IManufacturer` не требуется |

### *Кто создает виртуальные объекты-посредники*

Из рис. 34.16 видно, что виртуальный объект-посредник `ManufacturerProxy` взаимодействует с объектом `PersistenceFacade` с целью материализации объектов. Но какой класс должен отвечать за создание объектов `ManufacturerProxy`? Класс-преобразователь в структуру базы данных для объекта `ProductSpecification`. Класс-преобразователь определяет момент материализации объекта, решает, какие из дочерних объектов должны быть материализованы немедленно, а какие подлежат пассивной материализации с использованием объектов-посредников.

Рассмотрим два альтернативных решения: активную и пассивную материализацию.

```
// Активная материализация объекта Manufacturer

class ProductSpecificationRDBMapper extends
AbstractPersistenceMapper
{
protected Object getObjectFromStorage(OID oid)
{
ResultSet rs =
    RDBOperations.getInstance().getProductSpecificationData(oid);

ProductSpecification ps = new ProductSpecification();
ps.setPrice(rs.getDouble("PRICE"));
    // основные операции выполняются здесь

String manufacturerForeignKey = rs.getString("MANU_OID");
OID manuOID = new OID(manufacturerForeignKey);
ps.setManufacturer( (IManufacturer)
    PersistenceFacade.getInstance().get(manuOID,
            Manufacturer.class);
...
}
```

Приведем решение на основе пассивной материализации.

```
// Пассивная материализация объекта Manufacturer

class ProductSpecificationRDBMapper extends
AbstractPersistenceMapper
{
protected Object getObjectFromStorage(OID oid)
{
ResultSet rs =
    RDBOperations.getInstance().getProductSpecificationData(oid);

ProductSpecification ps = new ProductSpecification();
ps.setPrice(rs.getDouble("PRICE"));
```

```
    // основные операции выполняются здесь

String manufacturerForeignKey = rs.getString("MANU_OID");
OID manuOID = new OID(manufacturerForeignKey);
ps.setManufacturer( new ManufacturerProxy(manuOID));
...
}
```

## 34.19. Представление отношений в таблицах

В предыдущем примере для ссылки на запись в таблице MANUFACTURER используется внешний ключ MANU_OID из таблицы PRODUCT_SPEC. Возникает вопрос: как представить взаимоотношения объектов в реляционной модели?

Ответ на этот вопрос дает шаблон Representing Object Relationships as Tables (Представление взаимосвязей объектов в виде таблиц) [25], который сводится к следующему.

- Ассоциации "один к одному".
  - Поместить идентификатор объекта OID в качестве внешнего ключа в одну или обе таблицы, представляющие взаимосвязанные объекты.
  - Либо создать ассоциативную таблицу с идентификаторами каждого объекта, участвующего во взаимоотношении.
- Ассоциации "один ко многим".
  - Создать ассоциативную таблицу с идентификаторами каждого объекта, участвующего во взаимоотношении.
- Ассоциации "многие ко многим".
  - Создать ассоциативную таблицу, содержащую идентификаторы каждого объекта, участвующего во взаимоотношении.

## 34.20. Суперкласс PersistentObject

Одним из типичных проектных решений при работе с постоянно хранимыми объектами является создание абстрактного суперкласса технических служб PersistentObject, от которого наследуют свои свойства все другие объекты, предназначенные для постоянного хранения (рис. 34.17). В таком классе обычно определяются атрибуты для обеспечения постоянного хранения, в том числе идентификатор объекта, а также методы его сохранения в базе данных.

Такой подход нельзя назвать неверным, однако он отличается сильным связыванием конкретного класса с классом PersistentObject, поскольку классы, реализующие понятия предметной области, наследуют свойства класса технических служб.

Такое проектное решение не соответствует идее разделения обязанностей: обязанности технических служб смешиваются с задачами объектов уровня логики приложения.

Однако "разделение обязанностей" — это не конечная цель, к которой необходимо стремиться любой ценой. Как указывалось при описании шаблона Protected Variations, разработчики должны выработать свою позицию и стараться

избежать неустойчивости проектного решения. Если в данном конкретном приложении наследование свойств класса PersistentObject приводит к простому и прозрачному решению и не создает проблем с поддержкой существующих классов, то почему бы его не использовать? Ответ на этот вопрос зависит от требований и проектного решения данного приложения. Определенное влияние на принятие такого решения оказывает выбор языка программирования: языки, поддерживающие одиночное наследование (такие как Java), допускают использование одного общего суперкласса.

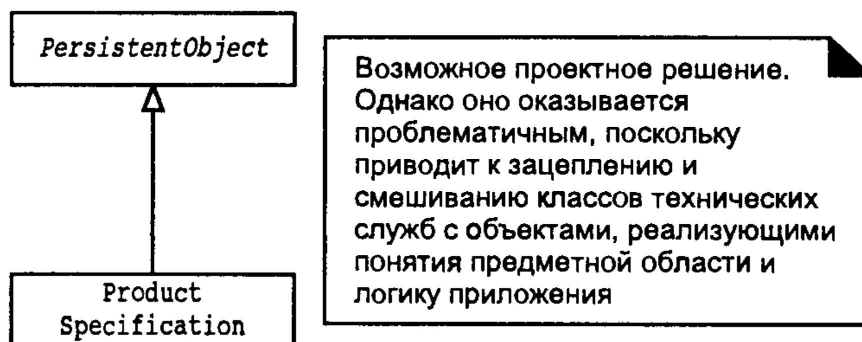



*Рис. 34.17. Проблемы, связанные с использованием суперкласса* PersistentObject

## 34.21. Нерешенные вопросы

В этой главе мы очень кратко рассмотрели проблемы и проектные решения по созданию контура взаимодействия с базой данных и соответствующих служб. Однако многие важные вопросы остались "за кадром", в том числе следующие.

- Дематериализация объектов.
  - В двух словах, этот процесс можно реализовать так. Конкретные преобразователи должны включать метод putObjectToStorage. При дематериализации сложных иерархических объектов необходимо обеспечить взаимодействие нескольких преобразователей и поддержку ассоциативных таблиц (при использовании реляционной базы данных).
- Материализация и дематериализация коллекций.
- Запросы на группы объектов.
- Полная обработка транзакций.
- Обработка ошибок при взаимодействии с базой данных.
- Принципы совместного использования данных многими пользователями и стратегии блокировки данных.
- Вопросы безопасности — контроль доступа к базе данных.

# Часть VI

# Специальные вопросы

Глава

# Средства создания диаграмм

*Мыльные пузыри не взрываются.*

*Бертранд Мейер (Bertrand Meyer)*

Основные задачи

- Ознакомиться с советами по построению диаграмм для проекта.
- Проиллюстрировать применение некоторых стандартных функций CASE-средств, связанных с UML.

## Введение

В реальном проекте анализ и проектирование с построением диаграмм UML происходит не столь систематично, как на страницах этой книги. Оно выполняется командой разработчиков программы наряду с другой деятельностью в офисе в процессе оживленных дискуссий. Диаграммы зачастую строятся на доске, иногда с помощью специальных средств, а порой разработчики стараются скорее приступить к программированию, не вдаваясь в детали построения диаграмм. Если процесс построения диаграмм слишком утомителен, неструктурирован или кажется менее важным, чем программирование, его стараются пропустить.

В этой главе предлагаются некоторые рекомендации по сбалансированию программирования и построения диаграмм, а также описываются некоторые специализированные средства, призванные облегчить проектирование и сделать его более полезным для всего проекта.

## 35.1. Умозрительное проектирование и визуальное мышление

Проектные решения, проиллюстрированные диаграммами UML, не представляют самостоятельной ценности, а служат лишь "трамплином" для программной реализации системы. Создание слишком большого количества диаграмм до начала программирования займет слишком много времени и не обеспечит построения готовой системы. Ничто, кроме реального кода не даст реального представления о системе. Бертранд Мейер (Bertrand Meyer) выразил эту мысль еще лучше: "Мыльные пузыри не взрываются".

Тем не менее, автор настоятельно рекомендует затратить *некоторые* усилия на построение диаграмм до перехода к программированию. Такая деятельность может оказаться полезной для формирования основных стратегий проектирования. Возникает интересный вопрос о том, сколько диаграмм нужно построить до начала программирования. Ответ на этот вопрос зависит от опыта и стиля работы разработчиков.

У некоторых людей очень развито пространственно-визуальное мышление, поэтому выражение проектного решения на наглядном языке диаграмм свойственно их натуре. Другие люди мыслят иначе. Большая часть мозга осуществляет обработку визуальной информации, а не смысловой (кода). Поэтому визуальные языки, такие как UML, играют важную роль в восприятии информации для большинства людей. Владение языком UML облегчает процесс проектирования системы. В целом, более опытные разработчики могут эффективно проектировать системы на языке диаграмм, не вдаваясь в пространные рассуждения, а лишь полагаясь на свой опыт и знания. Менее искушенные с помощью диаграмм могут скорее освоить тонкости проектирования, поскольку они позволяют абстрагироваться от лишних деталей и сконцентрироваться на общей картине.

Единственное исключение из общего правила составляют системы, которые более естественно рассматривать как конечные автоматы. Для таких систем может понадобиться более детальное моделирование. Некоторые CASE-средства обеспечивают полную генерацию кода на основе диаграмм состояний UML для всех классов системы. Однако такой подход применим не для всех предметных областей. В частности, в его рамки укладываются системы управления и телекоммуникационные системы, а экономические информационные системы строятся на других принципах.

## 35.2. Предложения по построению диаграмм UML в рамках процесса разработки

### *Трудозатраты*

В качестве рекомендации на каждой итерации предлагается выделить следующие периоды времени для построения диаграмм (этот вид деятельности удобно осуществлять парами разработчиков).

| | |
|---|---|
| Двухнедельная итерация: | полдня или день в начале итерации (в понедельник или во вторник). |
| Четырехнедельная итерация: | один или два дня в начале итерации. |

В обоих случаях процесс построения диаграмм не должен прекращаться после указанного срока. В течение итерации разработчики могут (в идеале попарно) проводить у доски краткие семинары и обсуждать очередные идеи проектных решений. При возникновении сложной проблемы в течение итерации можно провести еще один семинар (на полдня).

### *Другие рекомендации*

- Не занимайтесь построением диаграмм в одиночку. Тогда в дело вступит синергетика. Кроме того, разработчики смогут быстро перенять опыт друг друга и повысить свой профессиональный уровень. Очень сложно совершенствовать свои навыки проектирования в одиночку. Регулярно обменивай-

тесь идеями со своими коллегами и приобщайтесь к знаниям других разработчиков.

- В итеративном процессе (например, UP) программисты сами являются проектировщиками. Не существует отдельной команды разработчиков, занимающейся построением диаграмм и передающей их программистам. Разработчики вооружаются знанием UML и строят диаграммы. Затем они возвращаются к своему привычному "амплуа" и реализуют проектные решения, продолжая одновременно проектировать систему.
- Если группа разработчиков состоит из 10 человек, то одновременно заниматься построением диаграмм могут 5 пар специалистов. Если архитектор системы поочередно посещает каждый из семинаров, то он заметит "точки" расхождения, зависимости и повторяющиеся идеи. Поэтому архитектор системы может структурировать разработку и прояснять спорные моменты.
- Пригласите специалиста по техническому описанию систем, познакомьте его с системой обозначений UML и базовыми принципами объектно-ориентированного анализа и проектирования. Воспользуйтесь его помощью при работе с CASE-средствами, выполнении обратного проектирования диаграмм на основе кода, распечатке диаграмм на листах большого формата. Пусть разработчики занимаются своим делом (их время стоит дороже) — рисованием диаграмм на бумаге и программированием. Специалист по техническому описанию поможет им оформить диаграммы, а также выполнит свои основные обязанности по работе над документами для конечных пользователей. Этот подход известен под именем шаблона Mercenary Analyst (Наемный аналитик) [39].
- Оснастите свой офис несколькими большими досками и расположите их недалеко друг от друга.
- Вообще, постарайтесь увеличить рабочую площадь для удобного построения диаграмм на стенах помещения. Создайте дружественную среду для моделирования. Нельзя ожидать высокой культуры визуального моделирования в обстановке, где разработчики вынуждены довольствоваться маленькими досками, компьютерами с обычными мониторами или листами бумаги. Для удобства проектирования требуется большое открытое пространство — физическое или виртуальное.
- В дополнение к стационарным доскам стены можно оклеить тонкой белой бумагой, которая продается во многих магазинах. Ее можно использовать как доску для рисования диаграмм. При этом можно пользоваться карандашом и резинкой. Автору известна группа разработчиков, оклеившая такой бумагой всю комнату, — от пола до потолка. Разработчики считали, что такое решение способствует хорошему взаимопониманию.
- При построении диаграмм UML на доске обзаведитесь устройством (на рынке такие есть), позволяющим распознавать рукописные диаграммы и преобразовывать их в компьютерные графические файлы.
- При отсутствии такого устройства воспользуйтесь цифровым фотоаппаратом. Сфотографируйте нарисованные от руки диаграммы; большие рисунки можно заснять по частям. Такой прием построения компьютерных диаграмм применяется довольно часто.

- Еще одна технология — "печатающая доска". Это двусторонняя доска со сканером и подключенным принтером. Она тоже очень полезна при моделировании.
- Созданные вручную изображения введите в компьютер и распечатайте. Разместите их как можно ближе к рабочим местам программистов. Основная задача диаграмм — направлять и систематизировать мысли программистов. От сложенных в сейф диаграмм очень мало прока.
- При построении диаграмм UML вручную пользуйтесь простыми обозначениями. Это ускорит и облегчит процесс моделирования.
- Даже при построении диаграмм на доске, для создания диаграмм классов и пакетов используйте возможности CASE-средств по обратному проектированию на основе исходного кода в начале каждой следующей итерации. Затем рассматривайте эти диаграммы в качестве отправной точки для дальнейшего проектирования системы.
- Периодически распечатывайте "свежесгенерированные" сложные, интересные и изменяемые диаграммы классов в увеличенном масштабе на плоттере. Размещайте их на стене поближе к разработчикам для наглядности. Эту функцию может выполнять специалист по техническому описанию. Предлагайте разработчикам в процессе проектирования вносить изменения в эти чертежи.
- Далеко не все CASE-средства обеспечивают возможность обратного проектирования диаграмм последовательностей (а не только диаграмм классов) на основе исходного кода. По возможности обзаведитесь одним из таких средств для генерации диаграмм последовательностей архитектурно важных сценариев, распечатайте их в увеличенном масштабе на плоттере и предоставьте на всеобщее обозрение.
- При использовании CASE-средств, поддерживающих UML (и вообще для всех работ по программированию), используйте рабочую станцию со сдвоенным монитором (два обычных дисплея с плоскими экранами дешевле, чем один большой дисплей с плоским экраном). Современные операционные системы поддерживают сдвоенные (как минимум) видеоадаптеры, а значит, и два дисплея. Удобно разместите окна UML-средства на этих экранах. Вы спросите, зачем? Один небольшой монитор усложняет построение диаграмм, поскольку размер его рабочей области слишком мал. У разработчика может возникнуть желание завершать проектирование, поскольку на экране уже нет места.
- При проектировании системы на языке UML с помощью CASE-средств в небольших группах или попарно подключите два компьютерных проектора к двум видеоадаптерам компьютера и спроектируйте изображения на стену таким образом, чтобы обеспечить хороший обзор и большую рабочую область. Небольшая рабочая область и трудно различимые диаграммы психологически затрудняют взаимодействие участников группы.

## 35.3. CASE-средства (примеры)

Было бы неправильно не упомянуть о специализированных CASE-средствах для построения диаграмм UML — компьютерных программах для проектирования программных систем. Эта книга посвящена вопросам проектирования на языке UML, а для построения диаграмм используется либо доска, либо CASE-средства. В то же время невозможно охватить все средства. В этой книге автор не пытает-

ся дать сравнительную характеристику средств компьютерного моделирования. Важно уяснить следующее.

> В этой книге не рекламируется какое-либо CASE-средство, поддерживающее язык UML. Следующие примеры лишь иллюстрируют некоторые типичные и важные особенности CASE-средств.

## Неполная поддержка обозначений UML

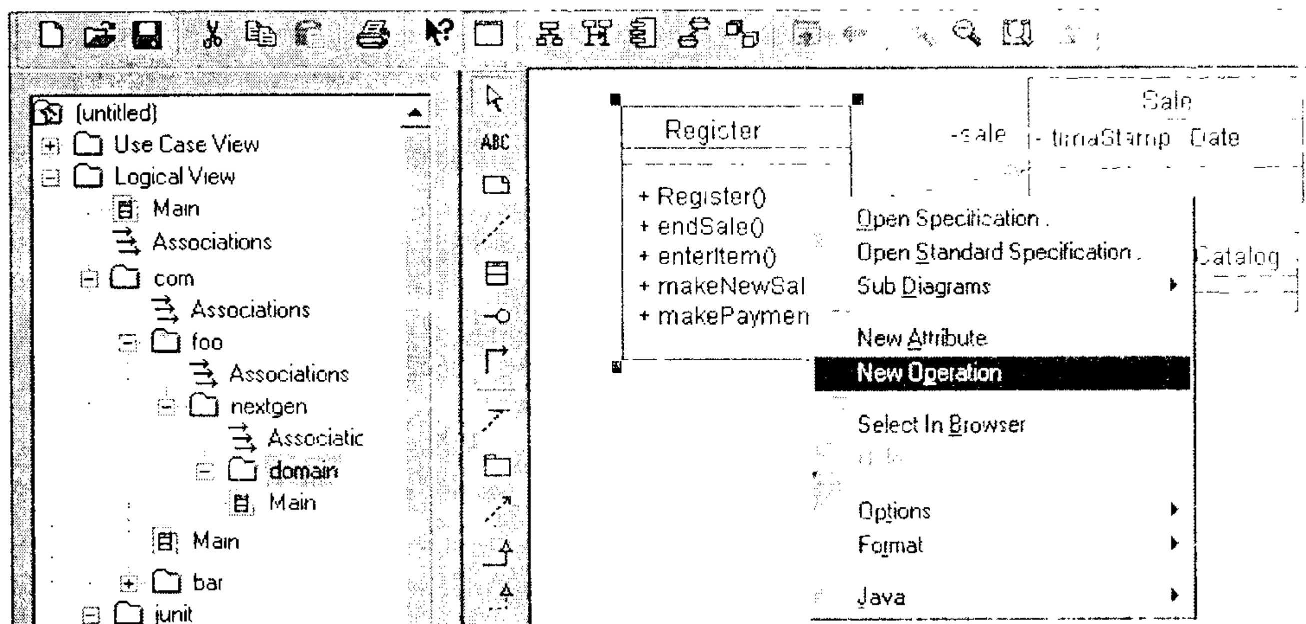
Лишь некоторые CASE-средства поддерживают весь набор обозначений UML и соответствуют текущей версии спецификации языка (или любой другой его версии). Однако это не главное при выборе средства проектирования - гораздо важнее их функциональные возможности и простота в использовании.

## Первый пример

Программу Together от компании TogetherSoft можно использовать для *прямого* и *обратного проектирования* (forward-engineering and reverse engineering). Именно эти функции отличают средства проектирования от обычных графических приложений.

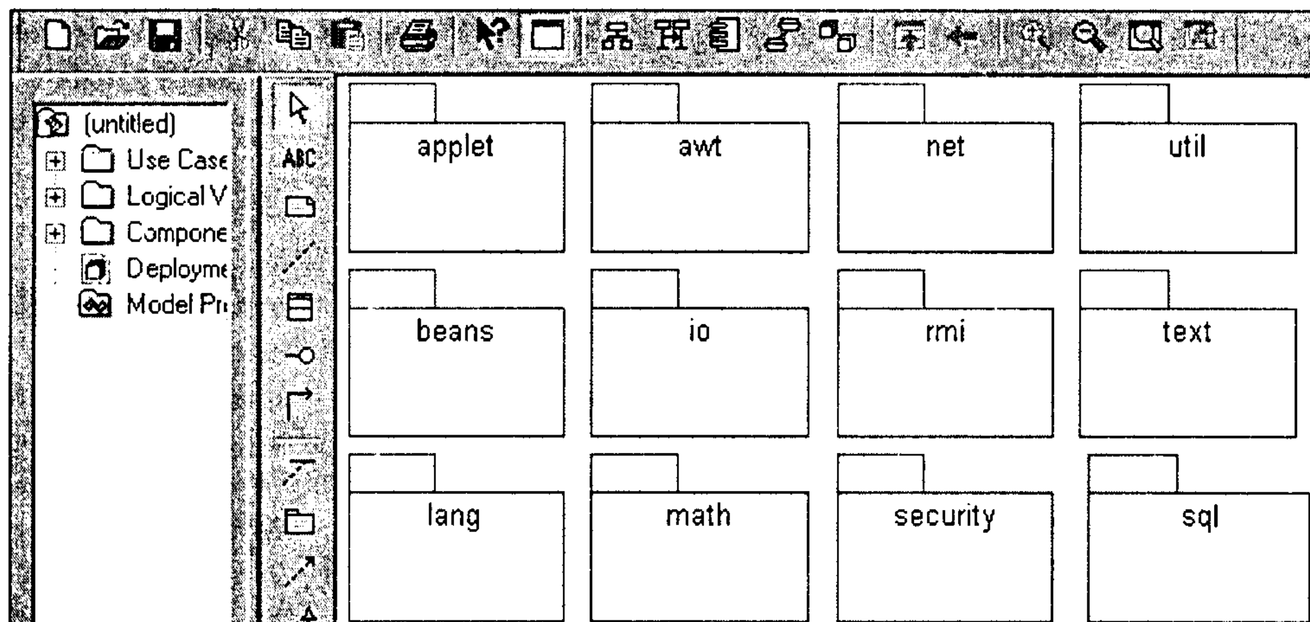
## Второй пример

На рис. 35.1 и 35.2 для иллюстрации некоторых других функций CASE-средств использовано приложение Rational Rose.




Создание диаграмм классов, используемых при генерации исходного кода, является ключевой особенностью CASE-средства с поддержкой UML.

*Рис. 35.1. Создание диаграммы классов*




Организация диаграмм в пакеты является одной из наиболее важных возможностей CASE-средств с поддержкой UML. Щелкнув на изображении пакета, можно увидеть его содержимое

*Рис. 35.2. Управление пакетами*

## Критерии выбора CASE-средства разработки

При выборе CASE-средства разработки автор предлагает руководствоваться следующими четырьмя критериями.

1. Корректная реализация современной системы обозначений UML.
2. Выбор той программы, которая активно используется командой разработчиков для проектирования самого CASE-средства (включая обратное проектирование диаграмм).
3. Использование N-версии CASE-средства для создания версии N+1.
4. Поддержка прямого и обратного проектирования диаграмм последовательностей (большинство CASE-средств поддерживает только диаграммы классов).

Компания Microsoft утверждает, что разработчики CASE-средств “едят пищу собственного приготовления”. Это очень хорошая метафора.

Глава

# Знакомство с итеративным планированием и проектированием

*Предсказание — очень трудное дело, особенно если оно касается будущего.*

*Автор неизвестен*

**Основные задачи**

- **Систематизировать требования и риски.**
- **Сопоставить и сравнить адаптивное и предиктивное планирование.**
- **Определить план разработки и план итерации в рамках UP.**
- **Ознакомиться со средствами отслеживания требований для итеративной разработки.**
- **Рассмотреть предложения по организации артефактов проекта.**

## Введение

Планирование проекта и вопросы управления — это достаточно емкие проблемы. Тем не менее не лишним будет кратко рассмотреть некоторые ключевые моменты итеративного процесса разработки и, в частности UP. К их числу относятся следующие.

- Что делать на следующей итерации?
- Как отслеживать изменение требований к системе в итеративном процессе разработки?
- Как организовать артефакты проекта?

## 36.1. Ранжирование требований

### *Основные критерии для начальных итераций: риск, охват, критичность, выработка навыков*

Что делать на начальных итерациях разработки? Ответ на этот вопрос сводится к следующему. Нужно систематизировать требования и спланировать итерации на основе рисков, охвата проблем и критичности элементов системы [72]. Понятие рисков включает техническую сложность и другие факторы, такие как неопределенность трудозатрат, плохую спецификацию, политические проблемы или полезность разработки. Ранжирование требований на основе рисков не сводится к ранжированию самих рисков (этот вопрос будет рассмотрен в последующих разделах).

Термин "охват проблем" применяется здесь в следующем смысле. На ранних итерациях должны быть затронуты все основные части системы — нужно выполнять широкомасштабную и поверхностную реализацию многих компонентов. "Критичность" подразумевает функции высокой экономической значимости. Это означает, что главные функции основных успешных сценариев должны быть хотя бы частично реализованы на начальных итерациях, даже если они не связаны со значительными техническими рисками.

В некоторых проектах на первый план выступает выработка навыков — группа разработчиков должна освоить новое направление, например объектные технологии. В таких проектах получение навыков является основным фактором, на основании которого строится последовательность итераций разработки: на начальных итерациях реализуются простейшие и наименее рискованные требования. При этом главной задачей является обучение сотрудников, а не снижение рисков проекта.

### *Что ранжировать*

Разработка в рамках UP выполняется на основе прецедентов. Поэтому необходимо ранжировать прецеденты и сценарии. Кроме того, некоторые требования формулируются в виде высокоуровневых свойств системы, не связанных с конкретным прецедентом. Такие требования обычно затрагивают множество прецедентов или общие службы, например службы регистрации. Подобные не связанные с прецедентами требования формулируются в дополнительной спецификации, таким образом в список для ранжирования включают и прецеденты, и высокоуровневые свойства.

| **Требование** | **Тип** | **...** |
|---|---|---|
| Оформление продажи | Прецедент | ... |
| Регистрация | Свойство | ... |
| ... | ... | ... |

### *Качественные методы ранжирования*

Требования ранжируются на основе указанных выше критериев, и наиболее приоритетные из них реализуются на начальных итерациях. Ранжирование может выполняться неформально или на качественном уровне, в процессе общего обсуждения группой разработчиков.

> *Совет*
>
> Чтобы неформально установить приоритеты для требований, задач или рисков на общем семинаре группы, воспользуйтесь методом последовательного "голосования фантами". Выпишите список элементов на доске. Затем каждому члену группы раздайте по 20 (к примеру) наклеек. После этого без дополнительного обсуждения (чтобы исключить взаимное влияние мнений) все члены группы должны распределить свои голоса (наклейки) между выписанными элементами. Каждый участник может "отдать" за любой элемент произвольное количество голосов. Подсчитайте результаты голосования и обсудите их. Затем проведите второй раунд "голосования фантами", чтобы отразить результаты первого раунда и обсуждения. Второй раунд обеспечит обратную связь и настройку общего мнения по результатам дискуссии.

Ранжирование рисков и требований нужно провести до начала первой итерации, затем перед началом второй итерации и т.д.

### Количественные методы ранжирования

Групповое обсуждение и голосование обеспечивают качественный (несколько нечеткий) подход к ранжированию требований и рисков. Для более тонкого (количественного) анализа используется следующий метод. Предлагаемый пример является чисто иллюстративным, числовые значения и весовые коэффициенты можно варьировать в зависимости от своих приоритетов.

| **Требование** | **Тип** | **АЗ** | **Риск** | **Критичность** | **Весовой коэффициент** |
|---|---|---|---|---|---|
| Оформление продажи | Прецедент | 3 | 2 | 3 | 15 |
| Регистрация | Свойство | 3 | 0 | 1 | 7 |
| Возврат товара | Прецедент | 1 | 0 | 0 | 2 |
| ... | ... | ... | ... | ... | ... |
| | | | | | |
| | | Вес | Диапазон | | |
| АЗ (архитектурная значимость) | | 2 | 0-3 | | |
| Риск технический, общий... | | 3 | 0-3 | | |
| Критичность высокая, экономическая | | 1 | 0-3 | | |

В любом проекте точные значения воспринимаются не слишком серьезно. Числовые значения помогают описать требования в терминах теории нечеткой логики с помощью градаций "высокий", "средний" и "низкий". Ясно, что на начальных итерациях необходимо реализовать прецедент `Оформление продажи`.

Числа говорят далеко не обо всем. Несмотря на то, что свойство регистрации связано с низким риском и является достаточно простым, его реализация является архитектурно значимой, поскольку требует интеграции с остальным кодом. Поэтому добавление этого свойства на последнем этапе может нарушить архитектурную целостность приложения.

### Ранжирование требований к POS-системе NextGen

Некоторые методы ранжирования позволяют сгруппировать требования на основе нечеткой логики. В терминах артефактов UP, результаты такого ранжирования описываются в "Плане разработки программы".

| Приоритет | Требование (прецедент или свойство) | Комментарий |
|---|---|---|
| Высокий | Оформление продажи | Высокое значение по всем критериям |
| | Регистрация | Сложно добавить впоследствии |
| | ... | ... |
| Средний | Поддержка пользователей | Влияет на подсистему безопасности |
| | Аутентификация пользователей | Важный, но не слишком сложный процесс |
| | ... | ... |
| Низкий | Сдача кассы | Легкий в реализации. Оказывает минимальное влияние на архитектуру |
| | Завершение работы | То же |
| | ... | ... |

### *Прецеденты Запуск системы и Завершение работы*

Теоретически все системы явно или неявно включают прецедент Запуск системы. И хотя по большинству критериев этот прецедент не имеет высокого приоритета, хотя бы простейшую его версию нужно реализовать на начальных итерациях, чтобы обеспечить инициализацию данных для других прецедентов. На каждой последующей итерации прецедент Запуск системы нужно постепенно дорабатывать в соответствии с требованиями других прецедентов. Сказанное относится и к прецеденту Завершение работы. В некоторых системах он достаточно сложен (например, завершение работы телекоммуникационной станции). Эти прецеденты необходимо упомянуть в плане итерации, указав, например, что "требуется реализация прецедентов Запуск системы и Завершение работы". Очевидно, что сложные версии этих прецедентов требуют более детальной проработки и тщательного планирования.

### *Предостережение: планирование проекта и задачи обучения*

Задачей этой книги является ознакомление читателя с вопросам анализа и проектирования, а не разработка POS-системы NextGen. Поэтому задачи для первых итераций разработки во многом выбирались с учетом требований обучения читателей, а не целей проекта.

## 36.2. Ранжирование рисков проекта

Полезным методом систематизации рисков проекта является оценка их вероятности и значения (в денежном, временном выражении или в трудозатратах). Оценки могут быть количественными (зачастую достаточно субъективными) или качественными (на основе группового обсуждения и голосования). Наиболее неприятными являются риски с высокой вероятностью и большим значением. Рассмотрим несколько примеров.

| Риск | Вероятность | Значение | Идеи по снижению |
|---|---|---|---|
| Недостаточное количество опытных разработчиков объектно-ориентированных систем | Высокая | Высокое | Прочтите эту книгу.<br>Пригласите временных консультантов.<br>Организуйте курсы.<br>Организуйте процесс попарного проектирования и программирования |

| Риск | Вероятность | Значение | Идеи по снижению |
|---|---|---|---|
| К предстоящему совещанию в Гамбурге не готова демонстрационная версия системы | Средняя | Высокое | Пригласить временных консультантов, специализирующихся на разработке POS-систем на языке Java.<br>Определите "показательные" требования, которые будут выгодно смотреться в демонстрационной версии и установите для них высокий приоритет.<br>Максимизируйте использование разработанных ранее компонентов |
| ... | ... | ... | ... |

В терминах артефактов UP, эти сведения помещают в "План разработки программной системы".

## 36.3. Адаптивное и предиктивное планирование

Одной из основных идей итеративной разработки является обеспечение адаптации за счет обратной связи, а не попытка составить детальный план реализации всего проекта. Следовательно, в рамках UP детальный план составляется только на *следующую* итерацию. Задачи остальных итераций определяются адаптивно с течением времени (рис. 36.1). Помимо обеспечения гибкости, причиной такого планирования является то, что при итеративной обработке сразу определяются не все требования, и вначале разработки не вырабатывается конкретное проектное решение.[1] План составляется с учетом мнения всей команды в процессе разработки. Допустим, в начале проекта был разработан хорошо продуманный план реализации, а в процессе его выполнения стало ясно, как этот проект можно усовершенствовать. На первый взгляд, такую ситуацию можно рассматривать как провал проекта, хотя на самом деле это совсем не так.

Тем не менее, необходимо определить основные цели и этапы проекта. При итеративной разработке не предполагается хаотического движения. Группа разработчиков должна знать основные этапы и задачи, однако детали реализации проекта могут варьироваться. Например, для приложения NextGen можно поставить задачу: за три месяца завершить реализацию прецедентов `Оформление продажи`, `Возврат товара` и `Аутентификация пользователей`, а также разработать процедуру регистрации пользователей и подключения новых правил ценообразования. Однако детальные планы реализации этих прецедентов разрабатываются по ходу проекта. Порядок действий и задачи для каждой итерации на эти три месяца не фиксируются. Подробно планируется только следующая двухнедельная итерация, и так, шаг за шагом, группа движется к поставленной цели и намеченному сроку ее реализации. Естественно, порядок выполнения проекта определяется внутренними зависимостями компонентов и ресурсов, однако не все виды деятельности детально планируются на начальной стадии проекта.

Заинтересованные лица должны одобрить план разработки (задачи на три месяца). Детальное планирование лучше всего возложить на плечи группы разработчиков, которые могут адаптивно учитывать новые обстоятельства (рис. 36.1).

---

[1] Точное проектное решение и детализированные требования в начале проекта не известны даже в рамках "каскадной" разработки, хотя она предполагает детальное планирование всего проекта.

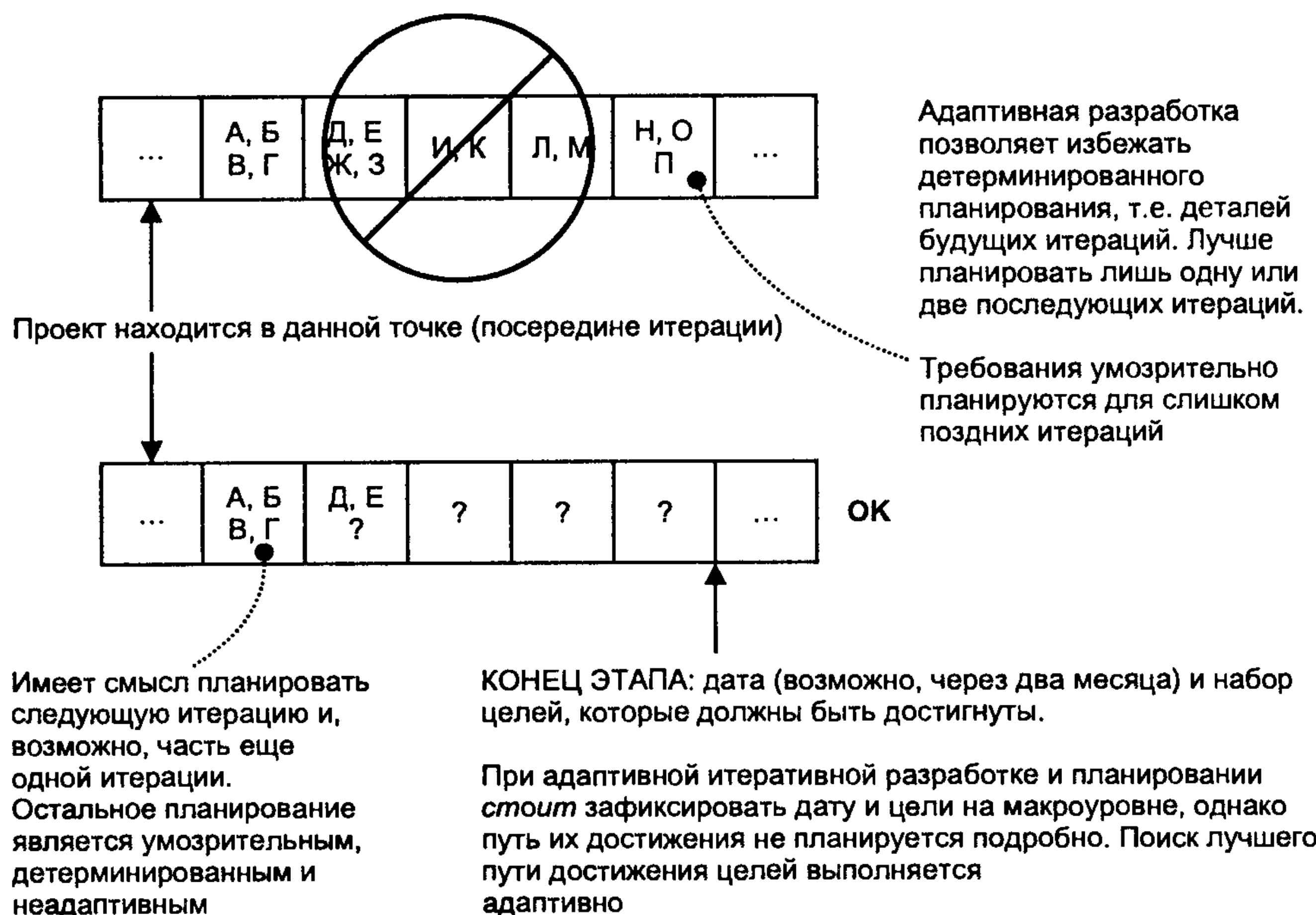



*Рис. 36.1. Важно определить основные этапы проекта, но не следует составлять детальный план на слишком длительный срок*

## 36.4. Планы для фазы и итерации

На макроуровне можно определить сроки и задачи основных этапов, на микроуровне проработать план следующей итерации (например, на четыре недели вперед). Эти два плана в UP описываются в "Плане фазы разработки" и "Плане итерации" — разделах "Плана разработки программного обеспечения". План фазы определяет основные задачи и сроки их выполнения, в том числе дату завершения данной фазы и даты получения промежуточных результатов. План итерации определяет задачи текущей и последующей итерации, но не всех итераций сразу (рис. 36.2).

На начальной стадии сроки основных этапов являются весьма приблизительными. На стадии развития эти оценки уточняются. Одной из задач фазы развития является получение реалистичной информации о сроках реализации основных задач и целях проекта.

## 36.5. План итерации: что делать на следующей итерации

Унифицированный процесс разработки строится на основе прецедентов, поэтому их реализации подчинена вся деятельность разработчиков. На каждой итерации нужно реализовать один или несколько прецедентов или сценариев, если прецедент достаточно сложен для реализации на одной итерации. Поскольку некоторые требования не отражаются в прецедентах (например, регистрация и под-

ключение правил ценообразования), то эти свойства тоже должны быть реализованы на одной или нескольких итерациях (рис. 36.3).

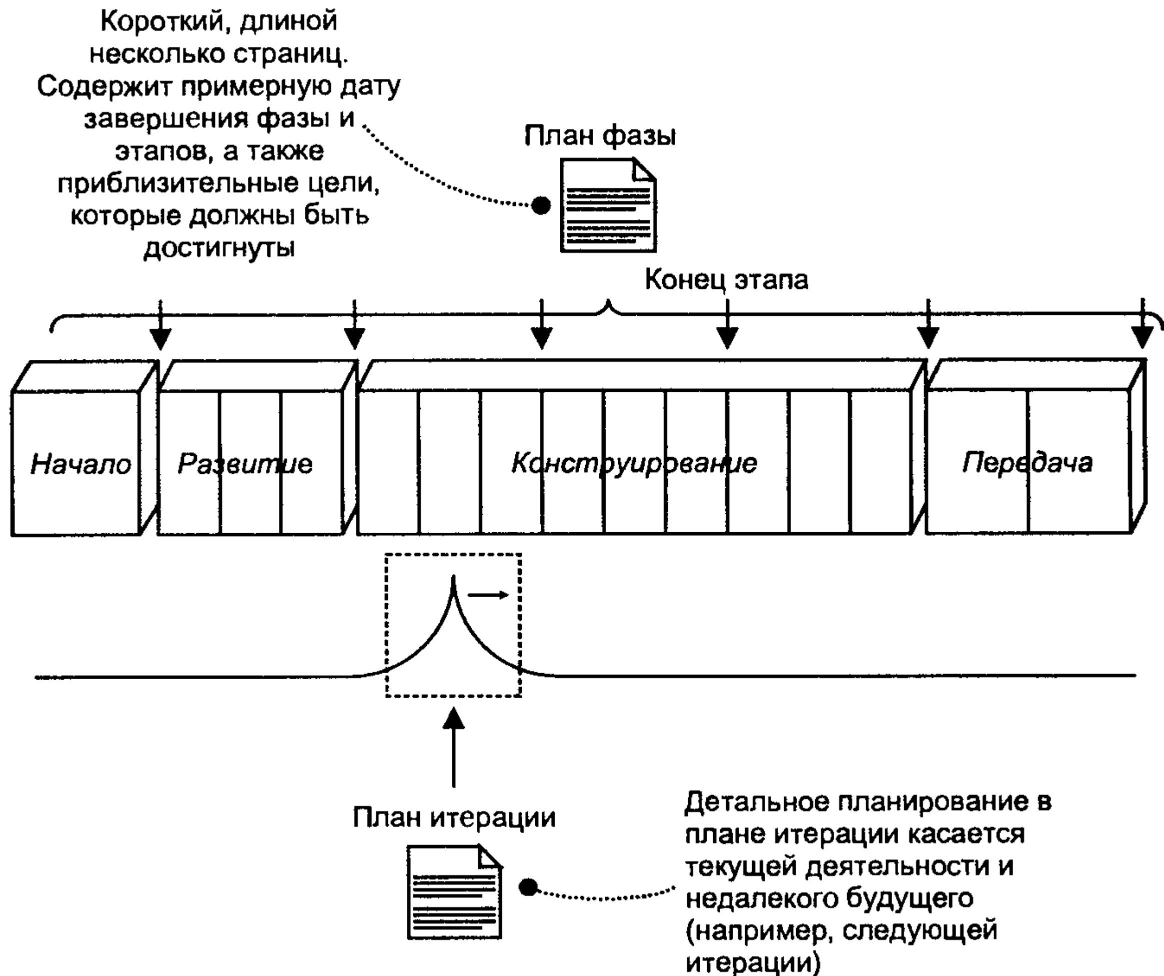



*Рис. 36.2. Планы для фазы и итерации*

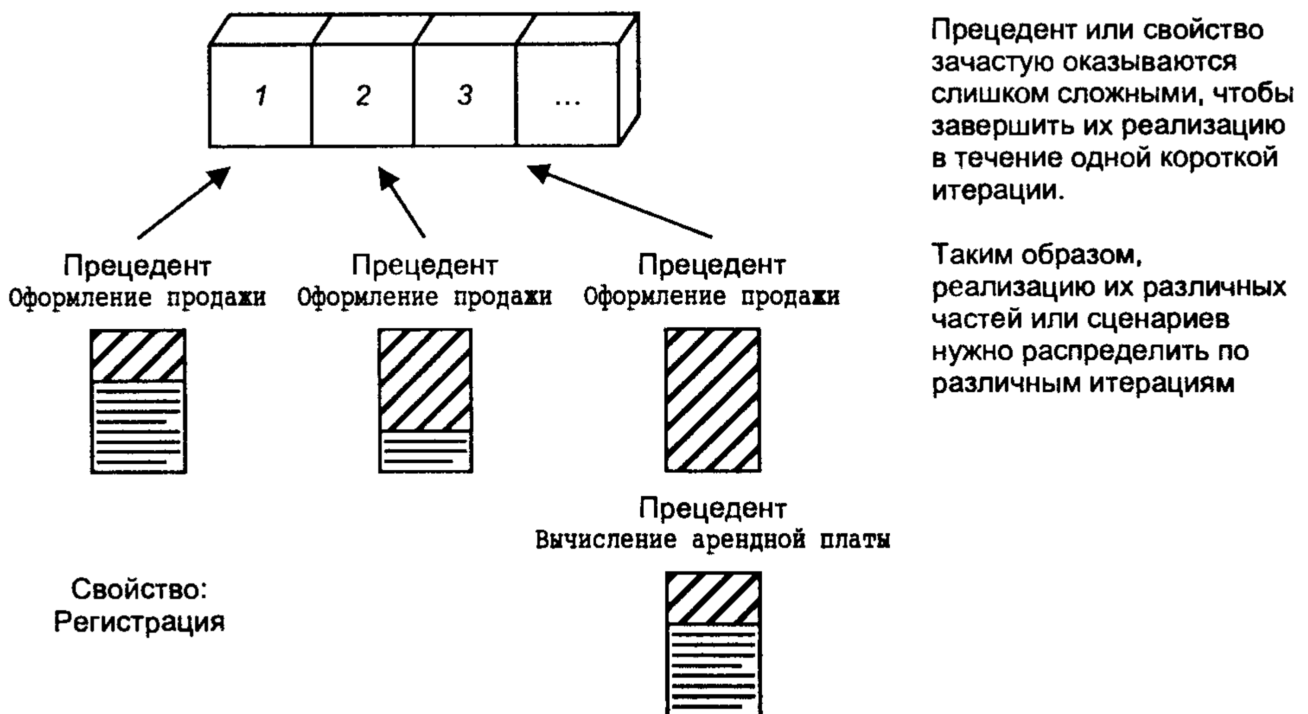



*Рис. 36.3. Распределение работы по итерациям*

Обычно на первой итерации фазы развития приходится решать множество вспомогательных задач: устанавливать программное обеспечение и компоненты, уточнять требования и т.д.

Задачи первых итераций определяются по результатам систематизации требований. Например, прецедент Оформление продажи, безусловно, имеет очень высокий приоритет. Следовательно, к нему нужно приступать на первой итерации. Тем не менее, на этой итерации реализуются не все сценарии этого прецедента, а лишь самые простые успешные сценарии, реализация которых затрагивает базовые элементы системы, например оплату наличными.

На следующих итерациях можно реализовать другие архитектурно важные требования, связанные с этим прецедентом. При этом разработчики приступят к реализации многих архитектурных аспектов системы: основных уровней, базы данных, интерфейса пользователя и интерфейса между основными подсистемами. Таким образом, уже на ранних итерациях будут реализованы многие важные части системы — в этом и состоит задача фазы развития.

## 36.6. Отслеживание требований в процессе итераций

Задача создания плана первой итерации приводит к осознанию важного аспекта итеративной разработки, отраженного на рис. 36.3.

Как следует из предыдущего раздела, на первой итерации реализуются не все сценарии прецедента Оформление продажи. Для реализации сложного прецедента может потребоваться полгода, т.е. множество двухнедельных итераций. На каждой итерации нужно реализовывать новые сценарии или их части.

Если реализовать все сценарии прецедента на одной итерации невозможно, возникает проблема отслеживания требований. Как записать, что уже реализовано, какие требования находятся в процессе реализации, а какие еще ждут своего часа? Одно из решений этой проблемы: использование специальных средств отслеживания требований.

Примером такого средства является приложение RequisitePro компании Rational. Рассмотрим как работают подобные средства. Это не реклама данного программного продукта, а лишь иллюстрация решения очень важной проблемы отслеживания требований.

### *Пример работы средства управления требованиями*

Программа RequisitePro интегрируется с Microsoft Word, поэтому требования можно редактировать в Word, а затем выделить нужную фразу и пометить ее как отслеживаемое требование в RequisitePro.

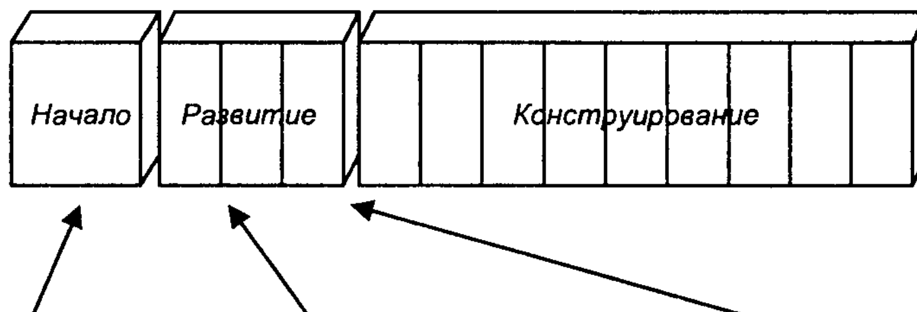
Каждое требование может иметь различные атрибуты, такие как статус, риск и т.д. При наличии такого средства проблема отслеживания требований снимается.

Все пункты основного и дополнительных сценариев прецедента можно рассматривать как отдельные требования, имеющие свой статус (предложено, одобрено и т.д.). Их можно контролировать отдельно.

## 36.7. Приблизительность начальных оценок

Что посеешь, то и пожнешь. Оценки, построенные на основе ненадежной и нечеткой информации, сами являются нечеткими и ненадежными. В рамках UP однозначно постулируется, что оценки, полученные на начальной стадии, не являются окончательными (это касается всех методов, но в UP это явно провоз-

глашается). Ранние оценки лишь дают общее представление о целесообразности проекта, при реализации которого можно будет получить более точные оценки. Более реалистичная информация появляется уже после первой итерации фазы развития. Оценки, полученные после второй итерации, уже вполне заслуживают доверия (рис. 36.4).




Оценки, полученные на начальной фазе, *не* используются для определения продолжительности и результатов всего проекта. Они лишь дают реалистичную оценку целесообразности выполнения проекта

В конце первой итерации фазы развития появляется более правдоподобная оценка

После двух итераций фазы развития и на последующих итерациях формируются достаточно реалистичные оценки возможности успешного завершения всего проекта и соответствующие сроки

*Рис. 36.4. Оценки и фазы проекта*

Это не значит, что не нужно стремиться к точности оценок с самого начала. Если это возможно — очень хорошо. Однако в большинстве случаев это не реально из-за использования новых технологий, методик и других аспектов. Поэтому в UP рекомендуется приступать к планированию проекта и бюджета по истечении нескольких итераций разработки.

## 36.8. Организация артефактов проекта

В UP артефакты организованы в терминах дисциплин. Модель прецедентов и дополнительная спецификация являются частью дисциплины определения требований. План разработки относится к дисциплине управления проектом и т.д. Следовательно, в рабочем каталоге можно организовать папки, имена которых соответствуют названиям дисциплин, и помещать в них соответствующие артефакты (рис. 36.5).

Такая организация применима для большинства артефактов, не относящихся к стадии реализации. Некоторые артефакты стадии реализации, такие как база данных или выполняемые файлы, обычно располагаются в других каталогах.

> *Совет*
>
> После каждой итерации используйте средство контроля версий для пометки и фиксации состояния всех элементов папок проекта (включая исходный код). Назовите эти версии артефактов “Развитие-1”, “Развитие-2” и т.д. Впоследствии эти метки позволят оценить скорость реализации проекта и объем работ, выполненный на каждой итерации.

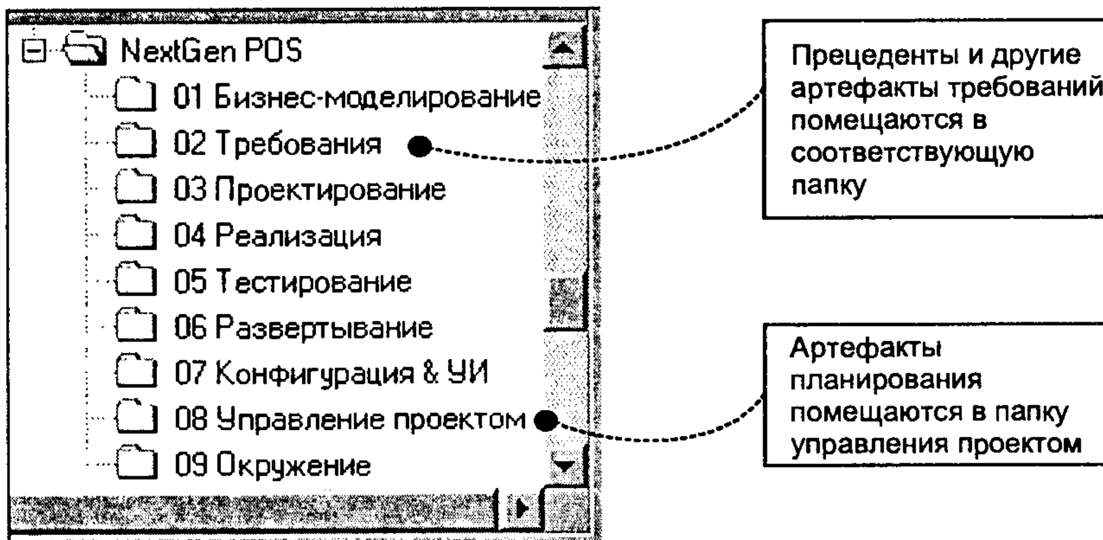



*Рис. 36.5. Организация артефактов UP в папки в соответствии с дисциплинами*

## 36.9. Некоторые вопросы составления графика работ

Большие проекты обычно разбиваются на параллельные процессы, выполняемые разными группами разработчиков. Эти процессы можно организовать по уровням и подсистемам. Еще один способ распараллеливания — по задачам. Этот подход обычно хорошо согласуется с архитектурой системы.

Например, можно выделить следующие группы.

- Группа разработчиков уровня предметной области (или подсистемы предметной области)
- Группа интерфейса пользователя
- Группа локализации
- Группа разработки технических служб (базы данных и т.д.)

Иногда разработка подсистем (например, службы взаимодействия с базой данных) требует длительного времени (особенно на начальных стадиях). Поэтому, не пытаясь "увязать" длительность всех итераций, для разных групп разработчиков можно назначить разные сроки итераций. Тогда для "быстрых" групп итерация будет длиться две недели, а для "медленных" — вдвое дольше (рис. 36.6).

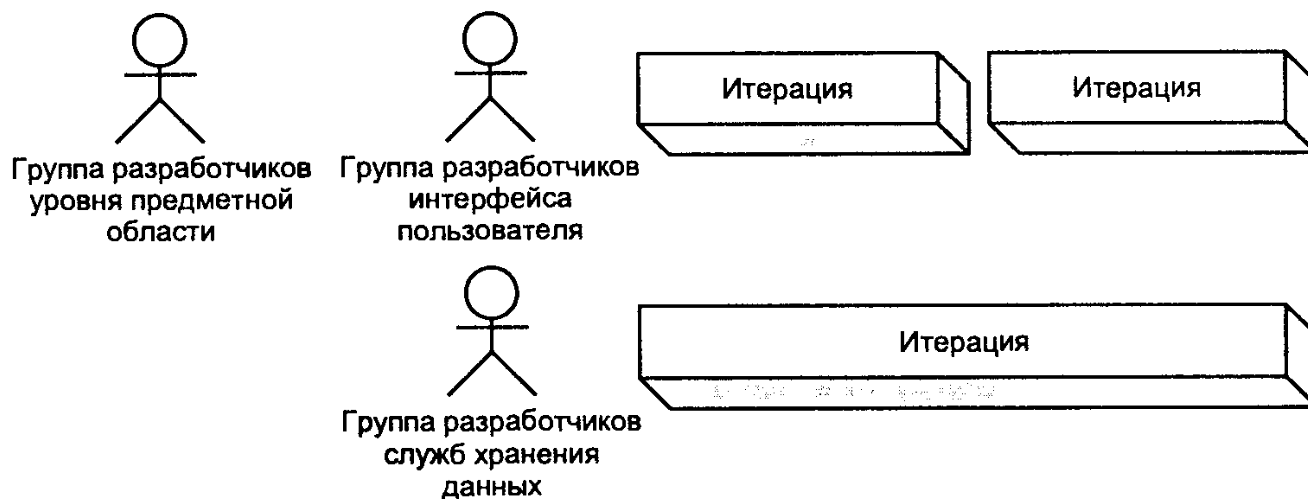



*Рис. 36.6. Переменная длительность итерации*

### *Скорость групповой работы и постепенная настройка процесса*

Большая длительность итерации определяется не только числом разработчиков в группе (чем больше людей, тем длиннее итерация), но и скоростью и опытом групповой работы. Если большинство участников только осваивают новые технологии, то работа, естественно, пойдет медленнее, и каждая итерация должна быть длиннее (например, не три, а четыре недели). Для менее опытных команд лучше несколько уменьшить количество итераций, удлинив каждую из них.

Заметим, что итеративный процесс предоставляет механизм для точной оценки скорости работы: реальный прогресс на ранних итерациях определяет скорость реализации последующих.

С этим вопросом связана стратегия *постепенной настройки процесса* (incremental process adoption). На ранних итерациях менее опытные группы разработчиков выполняют небольшое число задач. По мере совершенствования мастерства объемы выполняемых ими работ увеличиваются. Например, на начальных итерациях команда может один раз в день скомпоновать и протестировать всю систему. На последующих итерациях она может обеспечивать непрерывную интеграцию элементов в систему и тестирование системы (многократно в течение дня) с помощью средства непрерывной интеграции, например Cruise Control (`cruisecontrol.sourceforge.net`).

## 36.10. Вы не поняли принципов планирования в UP, если...

- Пытаетесь детально спланировать все итерации сразу, указав задачи каждой итерации.
- Рассчитываете на надежность начальных оценок и строите на их основе долгосрочные планы; считаете оценивание затрат тривиальной или не слишком сложной задачей.
- На ранних итерациях решаете простые проблемы, не связанные с высокими рисками.

Если в вашей организации процесс планирования выглядит примерно так, как описано ниже, значит, вы не поняли основных принципов планирования.

1. На начальной стадии планирования на высоком уровне определяются новые системы и средства, например "Web-система для управления финансами".
2. Менеджер по техническим вопросам в сжатые сроки оценивает затраты и длительность сложных, дорогостоящих и рискованных проектов, разрабатываемых с привлечением новых технологий.
3. План и бюджет проекта составляются на год.
4. Заинтересованные лица привлекаются в том случае, если реальный ход проекта не соответствует первоначальным оценкам. Тогда осуществляется переход к п. 1.

Такой подход не обладает итеративностью и реалистичностью, присущими унифицированному процессу разработки.

## 36.11. Дополнительная литература

В работе [72] несколько глав посвящены планированию и управлению проектами в рамках UP.

В качестве предостережения нужно отметить, что в некоторых книгах за словами "итеративная разработка" и "унифицированный процесс проектирования" кроется предиктивный или каскадный подход к планированию.

В работе [80] дается прекрасный обзор многих методик и вопросов планирования и управления проектом и рисками.

Глава

# Комментарии к итеративной разработке и UP

*Итеративную разработку нужно применять только в тех проектах, в которых вы рассчитываете на успех.*

*Мартин Фовлер (Martin Fowler)*

---

**Основные задачи**

- Ознакомиться с некоторыми дополнительными вопросами UP.
- Ознакомиться с приемами итеративной разработки.
- Оценить, как итеративный жизненный цикл помогает решить проблемы разработки.

---

## 37.1. Дополнительные приемы и понятия UP

Основная идея UP сводится к реализации проекта в рамках последовательных итераций фиксированной длительности — адаптивной разработке. Кроме того, в рамках UP используются следующие важные методики и ключевые понятия.

- **Реализация важных моментов с высоким риском на начальных итерациях.** Например, при разработке сервера приложений, призванного обслуживать 2000 клиентов одновременно при времени отклика порядка долей секунды, для реализации требований с высокой степенью риска не нужно ждать несколько месяцев или лет. Наоборот, сначала нужно сконцентрировать внимание на разработке, программировании и тестировании основных программных компонентов и архитектурно значимых элементов, оставив тривиальные вопросы на последующие итерации. Идея состоит в том, чтобы все принципиальные проблемы разрешить на начальных итерациях, избежав краха проекта накануне его завершения, что свойственно каскадным принципам разработки. Если уж проекту суждено провалиться, то пусть это лучше случится на ранних итерациях. Поэтому говорят, что в UP управление осуществляется *на основе рисков*. И наконец, заметим, что существуют различные формы рисков: недостаток ресурсов или квалификации,

технические риски, удобство в использовании, политические и т.д. Все эти риски должны быть устранены на ранних итерациях.

- **Постоянное привлечение пользователей.** Итеративная разработка и UP предполагают выполнение небольших шагов с получением обратной связи. Это требует постоянного привлечения заинтересованных лиц и экспертов для уточнения задач проекта. Однако большинство проектов проваливаются из-за недостаточного привлечения пользователей [100]. Только они могут точно сформулировать свои потребности. Если пользователями продукта может стать кто угодно (например, при разработке Web-узла), то следует выделить группу сторонних наблюдателей, которые должны высказать свои оценки.
- **Привлечение внимания к построению ядра системы, функции которого отличаются высоким зацеплением, на начальных итерациях.** Другими словами, UP сконцентрирован на архитектуре. Данный вопрос связан с реализацией высоких рисков на ранних итерациях, поскольку наиболее критичные элементы обычно составляют ядро системы. На ранних итерациях обычно выполняется всесторонняя реализация всех базовых элементов архитектуры, внимание уделяется основным проектным решениям, обязанностям и интерфейсам подсистем. Выполняется глубокая проработка особо критичных и сложных требований, например, обеспечение времени отклика на уровне долей секунды при параллельной работе 2000 клиентов.
- **Постоянная проверка качества.** В данном контексте под качеством подразумевают выполнение требований, возможность поддержки и масштабирования системы. Причиной постоянного тестирования системы начиная с первых итераций является тот факт, что стоимость ошибки нелинейно возрастает во времени. Поэтому итеративная разработка является адаптивной и основывается на обратной связи. Серьезное тестирование и реалистичная оценка системы позволяют получить эффективную обратную связь. В этом состоит отличие от каскадного процесса проектирования, когда важные функции системы реализуются в самом конце разработки, когда выявленные недостатки уже сложно или невозможно устранить. В UP проверка качества выполняется постоянно с самого начала, поэтому по завершении реализации проекта разработчиков не будут ожидать сюрпризы.
- **Применение прецедентов.** Неформально, прецеденты — это записанные рассказы о функционировании системы. Они обеспечивают механизм для определения функциональных требований, отличных от принятого ранее списка "система должна...". В UP рекомендуется рассматривать прецеденты как требования к системе и строить на их основе процесс планирования, проектирования, тестирования и написания документации для пользователей.
- **Визуальное моделирование системы.** Большая часть человеческого мозга вовлечена в процессы обработки визуальной или графической информации [103]. Поэтому при разработке системы важно применять не только текстовые языки (обычные описания и код), но и графические, пространственно-ориентированные языки диаграмм, в том числе UML. Это повышает интеллектуальный потенциал обработки информации.[1] Кроме того, в процессе

[1] По этой же причине стоит создавать цветные диаграммы (если только разработчики не страдают дальтонизмом) [27].

разработки важную роль играет *абстракция*, поскольку она позволяет выделить главное и отвлечься от незначительных деталей. Визуальные языки, такие как UML, обеспечивают визуализацию и построение абстрактной модели, воплощающей главные идеи разработки. Однако, как будет показано позднее, необходимо соблюдать "золотую средину" и не стараться уделить построению диаграмм слишком много или слишком мало внимания.

- **Внимательное управление требованиями.** Это не означает возврат к каскадному принципу проектирования и точное определение и фиксацию всех требований на первой фазе проекта. Наоборот, нужно проявить гибкость и внимательно записывать, систематизировать, отслеживать требования в рамках жизненного цикла разработки, применяя для этого специализированные средства. Это кажется очевидным, но на деле все далеко не так просто. Плохое управление требованиями — один из типичных факторов риска для проектов [100].
- **Управление изменениями.** Этот тезис включает несколько идей. Во-первых, нужно определять необходимость изменений. Хотя итеративный процесс допускает возможность изменений, он не допускает хаоса. При появлении новых требований нельзя восклицать: "Конечно, нет проблем!" и приступать к их реализации. Сначала нужно оценить необходимые затраты и влияние на систему и лишь после принятия соответствующего решения вносить изменения в график выполнения проекта. Этот тезис подразумевает также отслеживание состояния всех изменений (требуется, выполняется и т.д.). Во-вторых, нужно осуществлять управление конфигурацией. Средства конфигурирования применяются для поддержки частой (в идеале, как минимум ежедневной) интеграции и тестирования системы, параллельной разработки, разделения рабочего пространства разработчиков и контроля версий с самого начала проекта. В UP все аспекты проекта (за исключением кода) должны подлежать конфигурированию и контролю версий.

## 37.2. Фазы конструирования и передачи

### *Фаза конструирования*

*Фаза развития* завершается после разрешения всех высоких рисков, разработки архитектурного "остова" системы и формулирования большинства требований. По завершении фазы развития можно более реалистично оценить оставшиеся затраты и сроки реализации проекта.

За ней следует *фаза конструирования* (construction phase), задачей которой является завершение разработки приложения, альфа-тестирование, подготовка к бета-тестированию (на стадии передачи), подготовка к развертыванию путем написания руководства пользователя и создания интерактивной справки. Эти виды деятельности иногда сравнивают с украшением "остова", разработанного на стадии развития. Если стадия развития характеризуется построением критичного и архитектурно важного ядра системы, то на стадии конструирования создается остальная часть системы. На этой стадии, как и ранее, разработка выполняется в рамках последовательных итераций фиксированной длительности. На стадии развития рекомендуется задействовать небольшой мобильный коллектив, а затем расширить его на этапе конструирования. На этой стадии работа команды еще больше "распараллеливается".

### *Стадия передачи*

Стадия конструирования завершается после подготовки системы к развертыванию, доработки всех вспомогательных материалов (руководства пользователя, обучающих материалов и т.п.). За ней следует стадия передачи (transition phase), задачей которой является ввод системы в промышленную эксплуатацию. На этой стадии выполняется бета-тестирование, настройка, обучение пользователей, допускается параллельная работа старой и новой системы и т.п.

## 37.3. Другие интересные приемы

Рассмотрим некоторые интересные приемы, не документированные явно в UP, но оказывающие влияние на итеративный процесс разработки. Этот список не является исчерпывающим.

- **Шаблон реализации процесса SCRUM** [6]. Описан в Internet по адресу `www.controlchaos.com`. В рамках этого шаблона ежедневно проводится "пятнадцатиминутка", на которой главный разработчик проекта выясняет у каждого участника, что он сделал за предыдущий день и что планирует выполнить сегодня. Можно также попросить поделиться наиболее важными новостями с командой разработчиков. Такие встречи позволяют отслеживать ход процесса разработки, обеспечивают высокий уровень общения, адаптивность разработки и быструю обратную связь. Другие принципы этого шаблона освобождают разработчиков от всех дополнительных обязанностей, чтобы они могли сконцентрироваться на выполнении проекта. Все проблемные вопросы решает менеджер.
- **Шаблон Extreme Programming** (XP) [11] предполагает написание модульного теста *до* создания самого кода, а также написание тестов для всех классов. При работе на Java часто используют бесплатный контур модульного тестирования JUnit (`www.junit.org`). Пишется небольшой тест, затем небольшой фрагмент кода, код тестируется и процесс повторяется снова. Основное внимание в рамках этого подхода уделяется первоначальному написанию теста.
- **Непрерывная интеграция** — это еще один прием XP. Более подробная информация содержится в [11] и по адресу `www.martinfowler.com`. В UP предполагается полная интеграция системы на каждой итерации. Однако зачастую полную компоновку системы проводят ежедневно. Непрерывная интеграция подразумевает объединение всех компонентов системы каждые несколько часов. Это можно делать вручную, но лучше процесс автоматизировать с помощью фонового процесса (daemon). Он периодически "просыпается" (например, каждые две минуты) и выполняет поиск модифицированного кода, затем перестраивает систему и запускает сценарий тестирования. Популярная бесплатная система непрерывной интеграции для Java называется Cruise Control. Ее (вместе с исходным кодом) можно найти по адресу `cruisecontrol.sourceforge.net`.

## 37.4. Планирование длительности итерации

Существует, как минимум, четыре причины для фиксации длительности итераций.

Во-первых, это *закон Паркинсона*. Паркинсон открыл закон, согласно которому "работа расширяется, заполняя все время, отведенное для ее выполне-

ния" [87]. Удаленная или неточная дата завершения (например, через 6 месяцев) усиливает этот эффект. В начале проекта кажется, что еще вполне достаточно времени для его выполнения. Однако если каждая итерация длится всего две недели, по истечении которых нужно иметь работающую часть системы, то команда может сконцентрироваться и сразу приступить к решению задачи.

Во-вторых, итерации фиксированной длительности обеспечивают возможность *систематизации и принятия решений* о приоритетности работы и степени риска, выделения наиболее высоких технических и экономических рисков. Например, если длительность итерации составляет четыре недели, то необходимо конкретно определить, что реально можно сделать за этот период.

В-третьих, *команда получает удовлетворение* от своей деятельности. Короткие итерации позволяют быстро повышать профессиональный уровень и получать готовый результат. Разработчики привыкают за две–четыре недели достигать некоторого результата, а не трудиться в течение нескольких месяцев, не видя результатов своего труда. Эти психологические факторы играют важную роль для каждого участника и для консолидации команды в целом.

В-четвертых, *повышение доверия заинтересованных лиц*. Если команда регулярно открыто отчитывается о выполненной работе через короткие интервалы времени, например каждые две недели, то к ней и ее проекту повышается доверие со стороны спонсоров и других заинтересованных лиц.

## 37.5. Последовательный цикл каскадной разработки

В отличие от итеративного жизненного цикла UP, ранее использовался последовательный, линейный или так называемый каскадный жизненный цикл [95], связанный с предиктивным планированием и процессом разработки. Каскадный жизненный цикл характеризуется следующими моментами.

1. Четкая формулировка, запись и фиксация окончательных требований.
2. Проектирование системы на основе этих требований.
3. Реализация на базе проектного решения.
4. Интеграция отдельных модулей.
5. Тестирование и оценивание на предмет соответствия требованиям.

Основанный на таком жизненном цикле процесс разработки связан со следующими особенностями.

- Тщательное и полное определение каждого артефакта (например, требований или проектного решения) до перехода к следующему шагу.
- Фиксация подробного набора требований.
- Отход от требований или проектного решения в процессе разработки не допускается. В следующий раз нужно лучше продумывать требования!

Каскадный процесс напоминает инженерный подход к построению зданий или мостов. Разработка программного обеспечения становится более структурированной. Некоторое время назад этот подход применялся повсеместно большинством разработчиков программного обеспечения, менеджеров и преподавателей, не подвергаясь критическому исследованию.

Некоторые объекты необходимо строить по принципу зданий, например сами здания, но не программные системы.

В списке причин, приводящих к провалу программных проектов, опубликованном по результатам двухлетнего исследования в [78], каскадный процесс разработки занимает первое место.

### *Некоторые проблемы каскадного жизненного цикла*

> "Аналогия с построением зданий изжила себя. Пришло время изменить подход. Если разрабатываемая система слишком сложна для точного описания и безошибочной разработки, значит, необходимо применить принципиально другой подход (итеративную, инкрементальную разработку)".
>
> Фредерик Брукс (Frederick Brooks), автор книги *Мифический человеко-месяц*

Определение некоторых требований до начала проектирования и проектирование части элементов до программной реализации неизбежно и оправдано. Более того, последовательный жизненный цикл применим для краткосрочных проектов длительностью не более двух месяцев. Каждая итерация напоминает небольшой каскадный проект.

Однако при расширении временных рамок возникают трудности. Повышается сложность проекта, отсутствие обратной связи не позволяет выявить элементы с высокой степенью риска.

Большие итерации приводят к тому, что многие решения реализуются без обратной связи и реалистичной оценки и тестирования. На уровне двухнедельного мини-проекта (одной итерации) линейная последовательность "требования–проектирование–реализация" вполне применима. Однако с увеличением временных масштабов повышается риск неудачи.

Каскадный процесс разработки влечет за собой следующие проблемы.

- Позднее выявление рисков и проблем
- Негибкость требований и проектного решения
- Высокая сложность
- Низкая степень адаптации

### *"Смягчение" некоторых проблем в рамках каскадного жизненного цикла*

Итеративная разработка — это не "серебряная пуля" для решения всех проблем разработки программных систем. Однако она позволяет "смягчить" некоторые проблемы, возникающие при использовании каскадного жизненного цикла.

#### Проблема: позднее выявление рисков и проблем

Риски могут быть вызваны разными причинами: неудачным проектным решением, плохо сформулированными требованиями, неподходящей политической обстановкой, недостатком ресурсов или квалификации и т.д.

При каскадном жизненном цикле нет реальной возможности для раннего выявления рисков. Например, неудачная архитектура Web-узла может привести к его низкой производительности. При каскадной разработке оценивание системы производится намного позже формулировки требований после принятия всех проектных решений уже на стадии реализации. То есть через полгода или год после начала проекта (рис. 37.1). Всем известны случаи, когда группы разработчиков параллельно создавали свои подсистемы в течение длительного времени, а

затем пытались интегрировать их при завершении работы над проектом с заранее прогнозируемым отрицательным результатом.

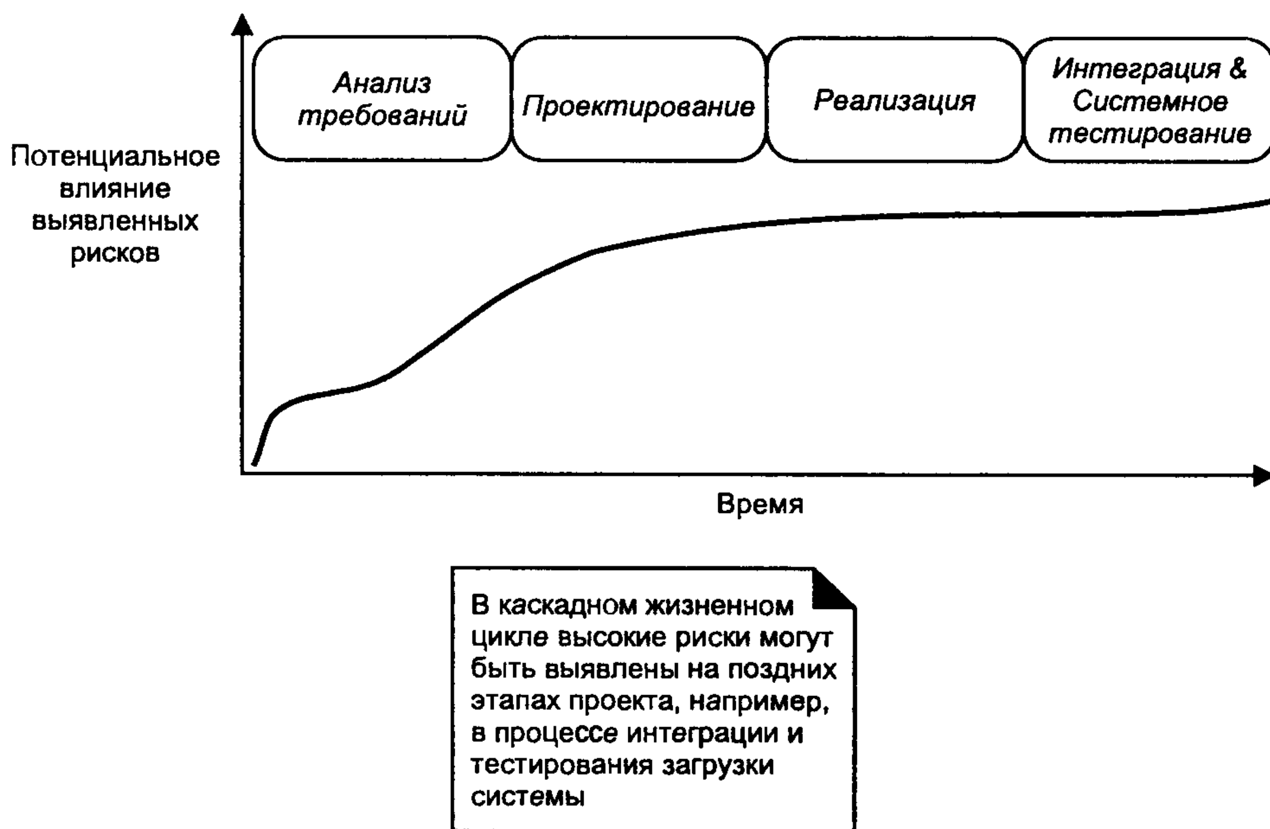



*Рис. 37.1. Каскадный жизненный цикл и риски*

***Смягчение рисков***

Задачей итеративной разработки, наоборот, является раннее выявление рисков. Высокие риски связаны с базовой архитектурой системы, удобством ее интерфейса, удовлетворенностью заинтересованных лиц. Эти риски нужно выявить в первую очередь. Как показано на рис. 37.2, риски выявляются на начальных итерациях. Возвращаясь к примеру с разработкой Web-узла, можно утверждать следующее. В рамках итеративной разработки после проектирования и реализации базовой архитектуры системы сразу станут ясны возможности системы. Если они окажутся неудовлетворительными, будет еще не поздно переделать ядро системы.

## Проблема: негибкость требований

Основной принцип каскадного процесса разработки сводится к формулировке и фиксированию требований к проекту на начальной стадии разработки. В таких проектах сначала выполняется анализ требований, разрабатывается набор соответствующих артефактов, эти документы подписываются и остаются неизменными до конца разработки.

Это неизбежно вызывает проблемы при дальнейшей разработке. Попытка определить все требования в самом начале, до проектирования и реализации, приводит к созданию новых сложностей. Она затрудняет обратную связь в проекте, учет новых возможностей нового программного обеспечения.

Правда, в некоторых проектах такой подход необходим. Это касается проектов по созданию аппаратно-программных комплексов, например, медицинских устройств или летательных аппаратов. Но даже в этом случае следует применять итеративный принцип разработки в процессе проектирования и реализации.

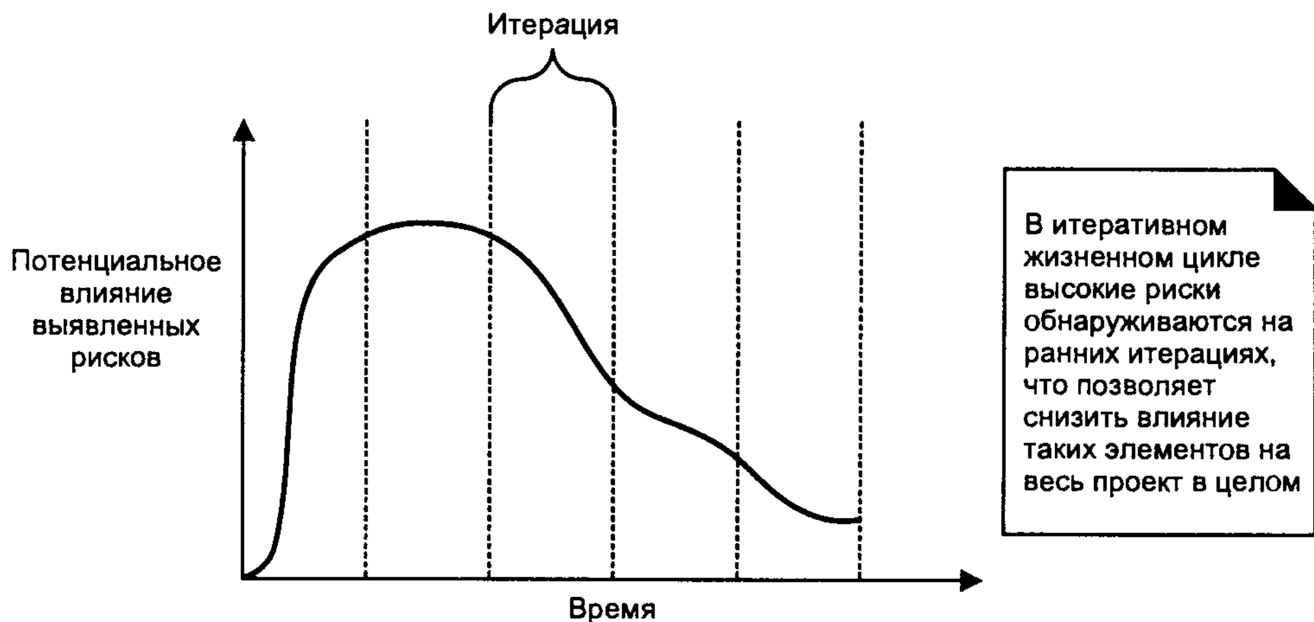



*Рис. 37.2. Итеративный жизненный цикл и риски*

Неистребимый миф о возможности успешной разработки на базе каскадного принципа "берет свое начало" из [68]. Как показано на рис. 37.3, при изучении 6700 проектов оказалось, что эволюция требований — существенный факт жизнеспособности программной разработки. В средних проектах требования изменяются примерно на 25%, а в крупных — на 50%. Аналогичные заключения приводятся в [20]. Поэтому каскадный принцип разработки, выступающий против этого неизбежного изменения, не учитывает реалии большинства проектов.

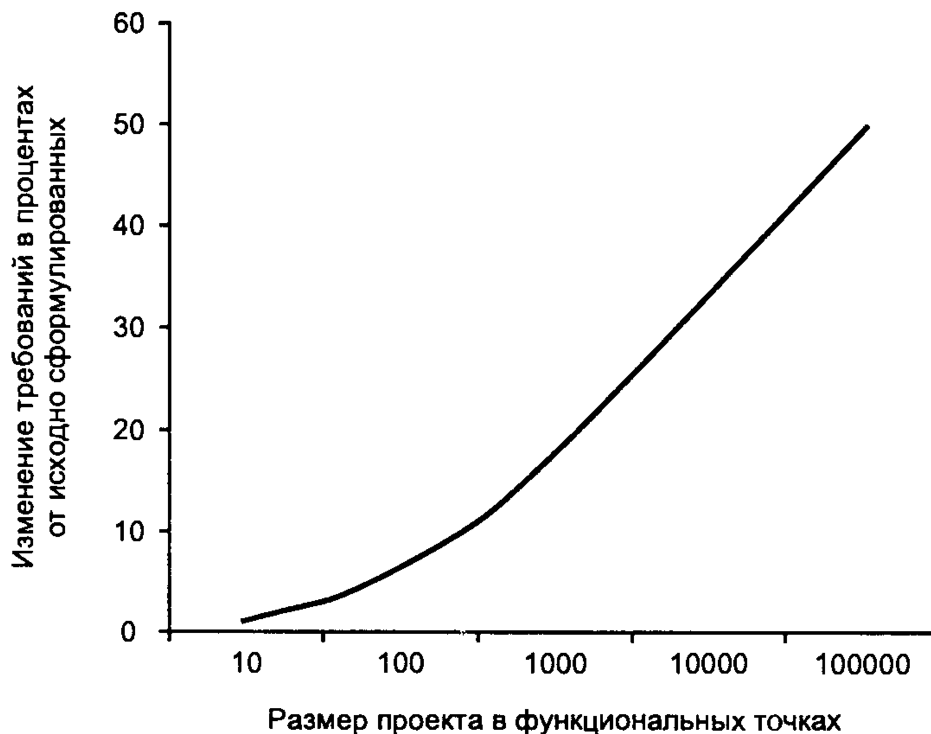



*Рис. 37.3. Изменение требований — "движущая сила" процесса разработки*[2]

---

[2] Функциональные точки описывают сложность системы в независимой от языка программирования метрике (см. www.ifpug.org).

"Незыблемые константы" изменяются по следующим причинам.

- Меняется точка зрения заинтересованных лиц либо они сразу не могут точно сформулировать, как они представляют разрабатываемую систему.[3]
- Происходят изменения на рынке программного обеспечения.
- Детализированная, корректная и точная спецификация дает психологический и организационный толчок к развитию мысли большинства заинтересованных лиц [72].

Следовательно, при каскадном проектировании возникают прогнозируемые проблемы, в том числе следующие.

- Как указывалось ранее, риски выявляются достаточно поздно.
- Команда пребывает в негативном эмоциональном состоянии, когда объективные изменения идут вразрез с разработкой проекта.
- Слишком большие средства вкладываются в "неверное" проектирование и реализацию (поскольку они основываются на некорректных требованиях).
- Отсутствует возможность изменения системы в соответствии с пожеланиями пользователей и потребностями рынка.

#### *"Смягчение" проблем*

При итеративной разработке не ставится задача сформулировать и зафиксировать требования к системе до начала проектирования и реализации. Это происходит лишь после нескольких итераций.

Набор базовых требований определяется, например, в течение двухдневного семинара. Затем часть из этих требований реализуется (наиболее критичных и значимых). После четырехнедельной итерации проводится еще один одно- или двухдневный семинар, на котором интенсивно обсуждается разработанная часть системы, уточняются и модифицируются требования. После второй (более короткой) итерации заинтересованные лица встречаются третий раз и снова уточняют требования. Теперь требования достаточно устойчивы и точно отражают пожелания заинтересованных лиц. Значит, можно составлять реалистичный план разработки и оценивать объем оставшейся работы. Этот период можно считать частью фазы развития UP.

Позднее тоже допускается некоторое изменение требований. Однако это происходит на основе обратной связи, когда реализация части системы позволяет уточнить новые требования.

### Проблема: негибкость проектного решения

Другая основная идея каскадной разработки состоит в том, что следующим этапом является полное проектирование системы и создание ее архитектуры. В таких проектах стараются как можно полнее описать архитектуру системы, проектное решение, интерфейс пользователя, структуру базы данных и лишь после этого приступать к реализации. С этим связаны следующим проблемы.

1. При изменении требований должно измениться и проектное решение.

---

[3] Барри Боем (Barry Boehm) сформулировал этот эффект так: "Я буду знать чего хочу, только когда увижу это".

2. Не опробованные средства и компоненты вносят долю риска в разработку системы. На этапе реализации их применение может оказаться неоправданным, поскольку "не это закладывалось в функции сервера приложений".
3. В целом основной проблемой является отсутствие обратной связи, подтверждающей или опровергающей правильность проектного решения.

*Устранение проблемы*

Эти проблемы устраняются в процессе итеративной разработки, при которой быстро разработанная часть системы служит для подтверждения правильности проектного решения и тестирования компонентов сторонних производителей.

## 37.6. Проектирование интерфейса пользователя с учетом удобства использования

Самое большое противоречие между важностью проблемы и невозможностью ее формального описания возникает при разработке интерфейса пользователя и оценке *удобства использования* системы (usability). Хотя эти вопросы не относятся к основам объектно-ориентированного анализа и проектирования и выходят за рамки UP, унифицированный процесс включает соответствующие этапы. Модель пользовательского интерфейса и удобство его применения относятся к дисциплине определения требований. В терминологии UP, в описании прецедентов, можно абстрактно описать элементы интерфейса и способы навигации между ними.

Эти вопросы описаны в книге Константина и Локвуда [28].

## 37.7. Модель анализа UP

К числу артефактов UP относится модель анализа. Этот артефакт не является обязательным и применяется редко. Имя артефакта выбрано не очень удачно, поскольку на самом деле это часть модели проектирования. В обычном смысле слова ([34, 46, 81]) модель анализа представляет собой модель объектов предметной области, построенную после исследования этой области. Однако в рамках UP модель анализа описывает взаимодействие между программными объектами и их обязанностями, поэтому ее можно отнести к модели проектирования. Можно даже привести цитату: "Модель анализа — это абстракция или обобщение проектного решения" [72]. И еще одна цитата: "Модель анализа можно рассматривать как первый элемент модели проектирования" [67].

Группа разработчиков RUP подчеркивает, что этот артефакт является не обязательным и его можно специально не создавать, поскольку эта модель редко используется опытными архитекторами и специалистами по методологии.

## 37.8. Продукт RUP

Продукт RUP представляет собой хорошо составленный набор Web-документов (HTML-страниц), разработанный компанией Rational Software и описывающий процесс Rational Unified Process — обновленную и детализированную версию UP. В нем описаны все артефакты, процессы и роли, приводятся рекомендации и шаблоны для многих артефактов.

Унифицированный процесс разработки описан в этой и многих других книгах. Поэтому нет необходимости покупать продукт RUP. Тем не менее многие организации находят его полезным для своей корпоративной сети (поскольку он содержит шаблоны артефактов). Переход к новому процессу разработки требует наличия дополнительной литературы и обучающих средств. Этой цели и служит набор документации для Web, включенный в продукт RUP.

## 37.9. Несколько слов о повторном использовании

Процесс UP разрабатывался для объектной технологии проектирования, одной из задач которой является возможность *повторного использования* элементов программы. Повторное использование — это важная, но сложнодостижимая цель. Ее реализация зависит не только от применения объектной технологии и корректного написания классов, но и от многих технических, организационных и социальных моментов. Естественно, замечательным примером повторного использования кода служат библиотеки классов и технических служб, например для Java, однако хотелось бы обратить внимание на сложности повторного использования своего кода, а не стандартных библиотек.

В организациях, использующих объектные технологии проектирования, проводился специальный опрос, в рамках которого нужно было указать основные преимущества таких технологий. Интересно, что повторное использование кода оказалось на одном из последних мест [42]. Для опытных разработчиков это не секрет: они знают, что повторное использование кода в рамках объектно-ориентированного подхода — это своего рода миф. Это не означает, что такую задачу не нужно ставить вообще, в какой-то мере повторное использование кода наблюдается. Однако вовсе не в такой степени, как это описывается в некоторых книгах и публикациях. Многие опытные разработчики могут рассказать леденящие душу истории о попытках некоторых организаций создать мощные библиотеки повторно используемых компонентов или служб для своей компании, на которые ушло несколько лет и миллионы долларов, но они закончились провалом.

Значит ли это, что объектные технологии не нужно применять вообще? Вовсе нет. Однако значение этой технологии не сводится к повторному использованию кода. Гораздо важнее гибкость, управление сложностью и простота модификации. По результатам того же опроса [42] основными преимуществами объектной технологии являются экономия средств и простота поддержки приложения. Хорошо спроектированные объектные системы легче модифицировать и расширять, чем системы, созданные без использования объектных технологий. Это очень важно, поскольку многие организации считают, что основные средства уходят не на разработку, а на поддержку и доработку системы. И хотя желательно снизить стоимость самой разработки, гораздо важнее сэкономить средства на ее поддержку и сопровождение, поскольку эти затраты гораздо существеннее. Объектные технологии позволяют решить именно эту проблему, а также обеспечивают создание мощных и “элегантных” сложных систем.

Глава

# Дополнительные обозначения UML

Основная задача

■ Ознакомиться с дополнительными обозначениями языка UML.

## 38.1. Символы общего назначения

### *Зависимости*

Зависимости могут существовать между любыми элементами, но чаще всего они используются на диаграммах пакетов (рис. 38.1).

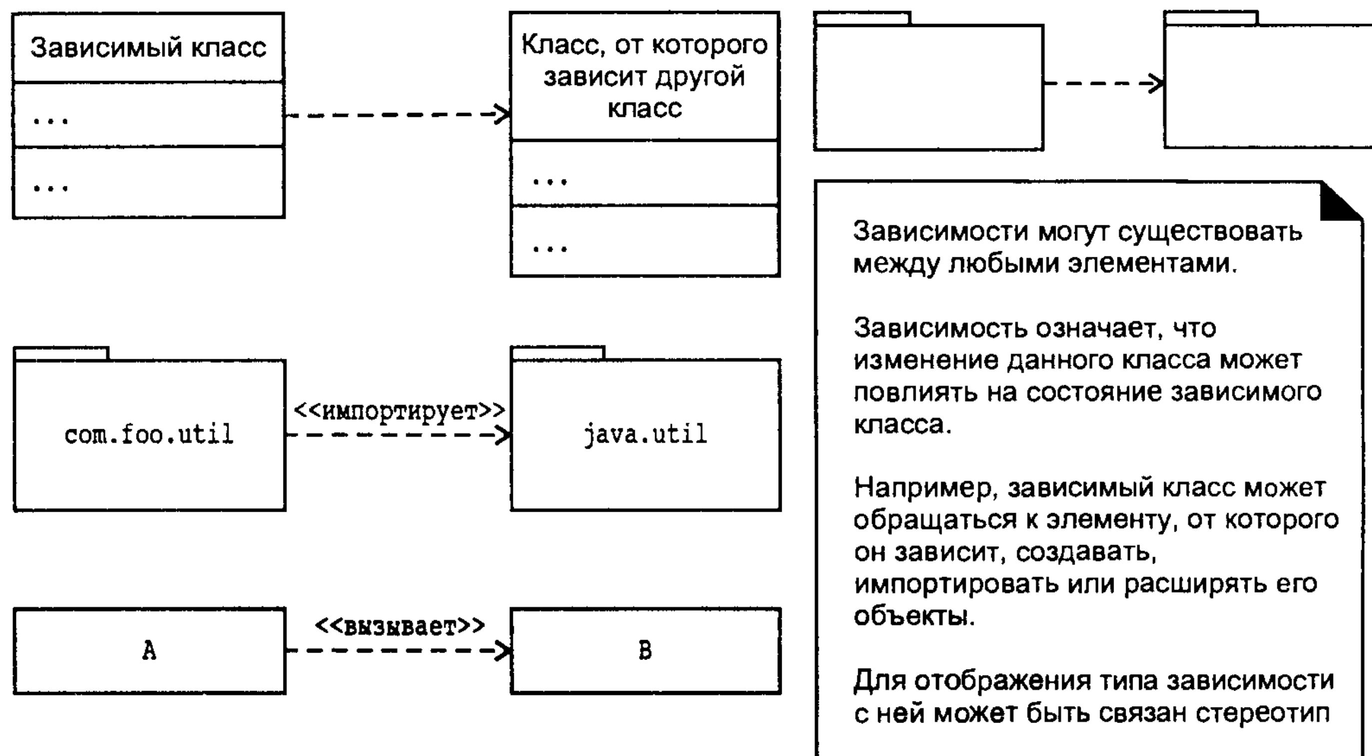



*Рис. 38.1. Зависимости*

### *Стереотипы и обозначения свойств с помощью тегированных значений*

Стереотипы используются в UML для классификации элементов (рис. 38.2).

*Рис. 38.2. Стереотипы и свойства*

### *Интерфейсы пакетов*

Пакет можно изобразить как реализацию интерфейса (рис. 38.3).

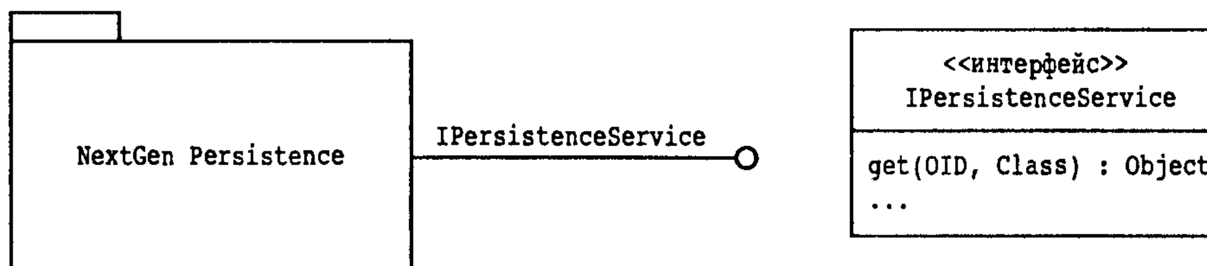



*Рис. 38.3. Интерфейс пакета*

## 38.2. Диаграммы реализации

В UML определены несколько типов диаграмм, которые можно использовать для иллюстрации деталей реализации. Чаще всего применяются диаграммы развертывания, иллюстрирующие процесс развертывания компонентов и обработки элементов.

### Диаграммы компонентов

Вспомним цитату: "*Компонент* (component) представляет собой отдельно размещаемую и заменяемую часть системы, инкапсулирующую реализацию нескольких интерфейсов" [86]. Это может быть исходный, двоичный или исполняемый код. Примерами компонентов являются броузеры, серверы HTTP, базы данных, библиотеки DLL или файлы JAR (для Enterprise Java Bean). Компоненты UML обычно отображаются на диаграммах развертывания, а не на специализированных диаграммах компонентов. Типичные обозначения показаны на рис. 38.4.

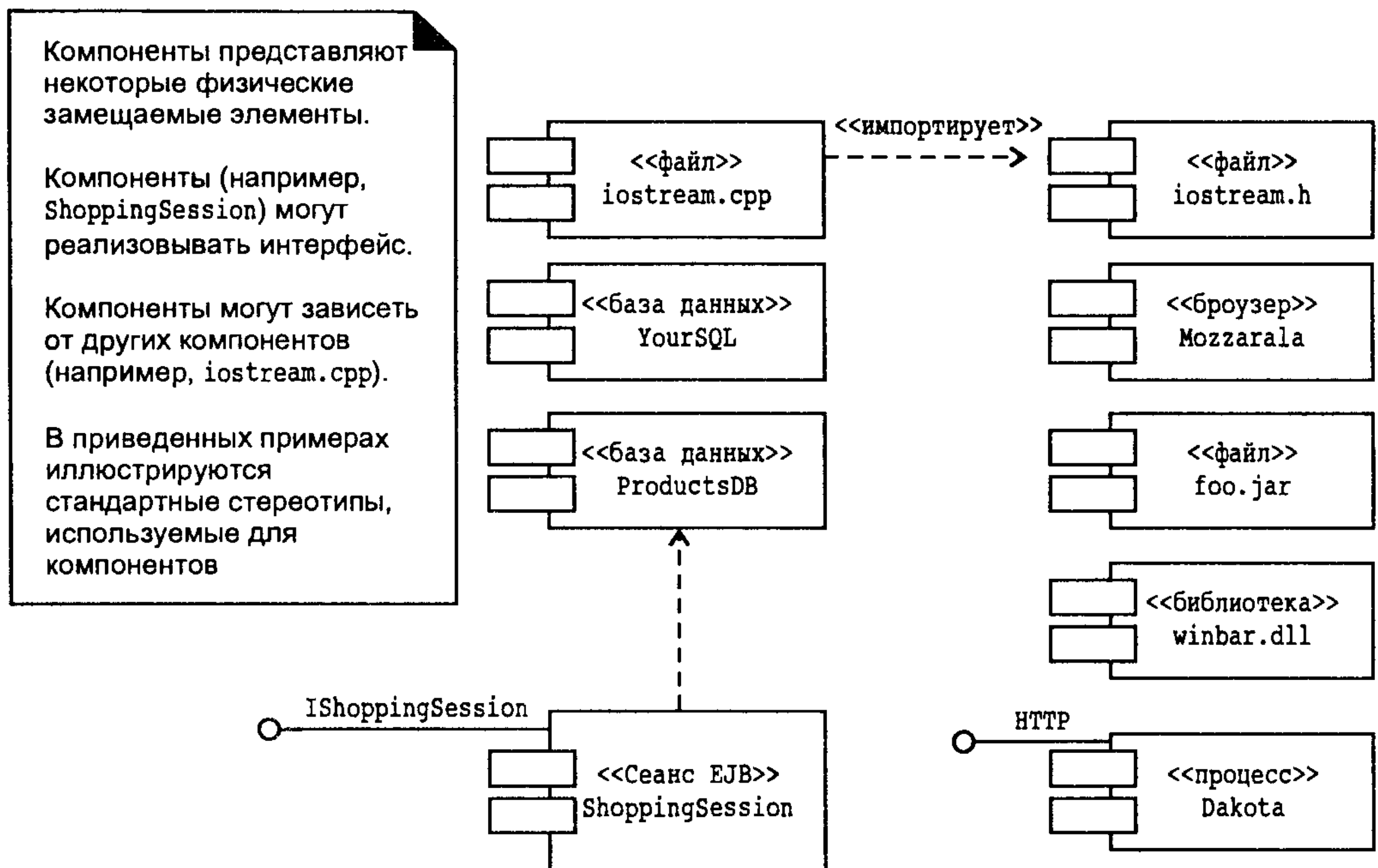



*Рис. 38.4. Компоненты UML*

### Диаграммы развертывания

Диаграммы развертывания показывают конфигурацию экземпляров компонентов и процессов во время работы приложения (рис. 38.5).

Можно показать, что компонент использует некоторый интерфейс, реализуемый другим компонентом

Заметим, что компонент может содержать другие компоненты

: Клиент

<<броузер>>
: Mozzarala

: Сервер

HTTP

<<процесс>>
:Dakota

<<сервер EJB>>
: JSlave

IShopping Session

<<Сеанс EJB>>
:ShoppingSession

:DBServer

<<база данных>>
products:YourSQL

SQL

Узел или обрабатывающий элемент

*Рис. 38.5. Диаграмма развертывания*

## 38.3. Классы шаблонов (параметризированные или родовые)

Классы шаблонов и их инстанцирование показаны на рис. 38.6.

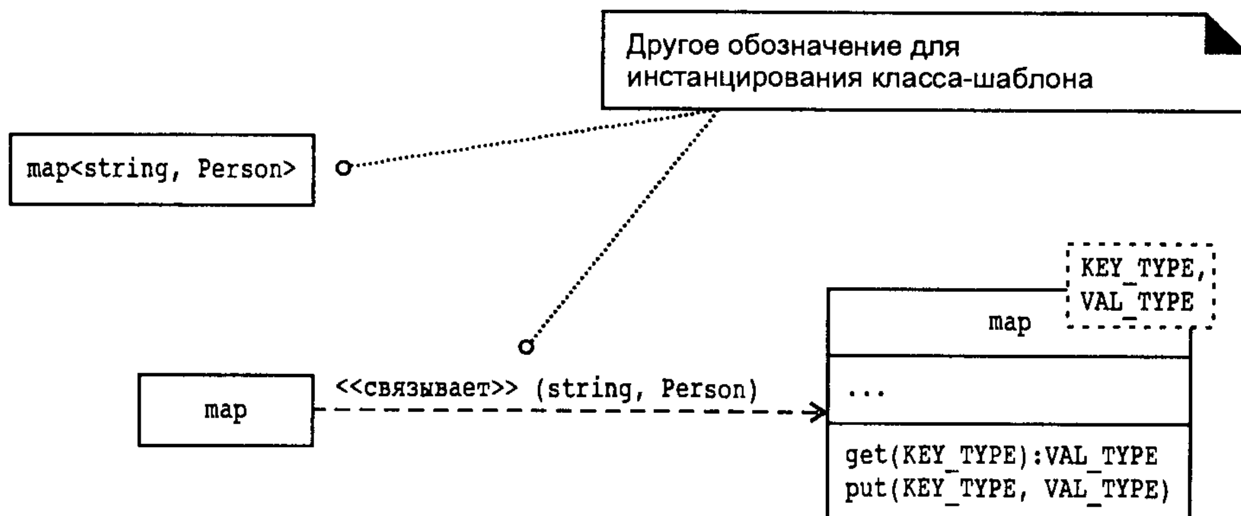



*Рис. 38.6. Классы шаблонов*

Некоторые языки, в частности C++, поддерживают параметризированные или родовые классы — классы шаблонов. Эта же возможность вскоре появится в языке Java. Например, в C++ класс `map<string, Person>` — это класс шаблона с ключевым значением типа `string` и значением типа `Person`.

## 38.4. Диаграммы видов деятельности

В UML *диаграммы видов деятельности* (activity diagrams) обеспечивают систему обозначений для последовательности видов деятельности. Эти обозначения можно использовать в различных целях (в том числе для визуализации шагов компьютерного алгоритма), однако они особенно полезны для построения диаграмм прецедентов и графиков выполнения проекта. Одной из дисциплин UP является бизнес-моделирование. Ее задача — сформировать и отразить "структуру и динамику организации, в которой развертывается система" [97]. Основным артефактом этой дисциплины является модель бизнес-объектов (прототип модели предметной области UP), в рамках которой с помощью диаграмм классов, последовательностей и видов деятельности UML обычно отображается экономическая деятельность предприятия. Таким образом, диаграммы видов деятельности чаще всего строятся в процессе дисциплины бизнес-моделирования.

Некоторые новые обозначения, включая распараллеливание видов деятельности, взаимодействие объектов и действий, показаны на рис. 38.7. Эта диаграмма построена на основе [49, 86]. Формально диаграммы видов деятельности являются частным случаем диаграмм состояний UML, когда состояниями являются виды деятельности, а переход в новые состояния означает завершение некоторых видов деятельности.

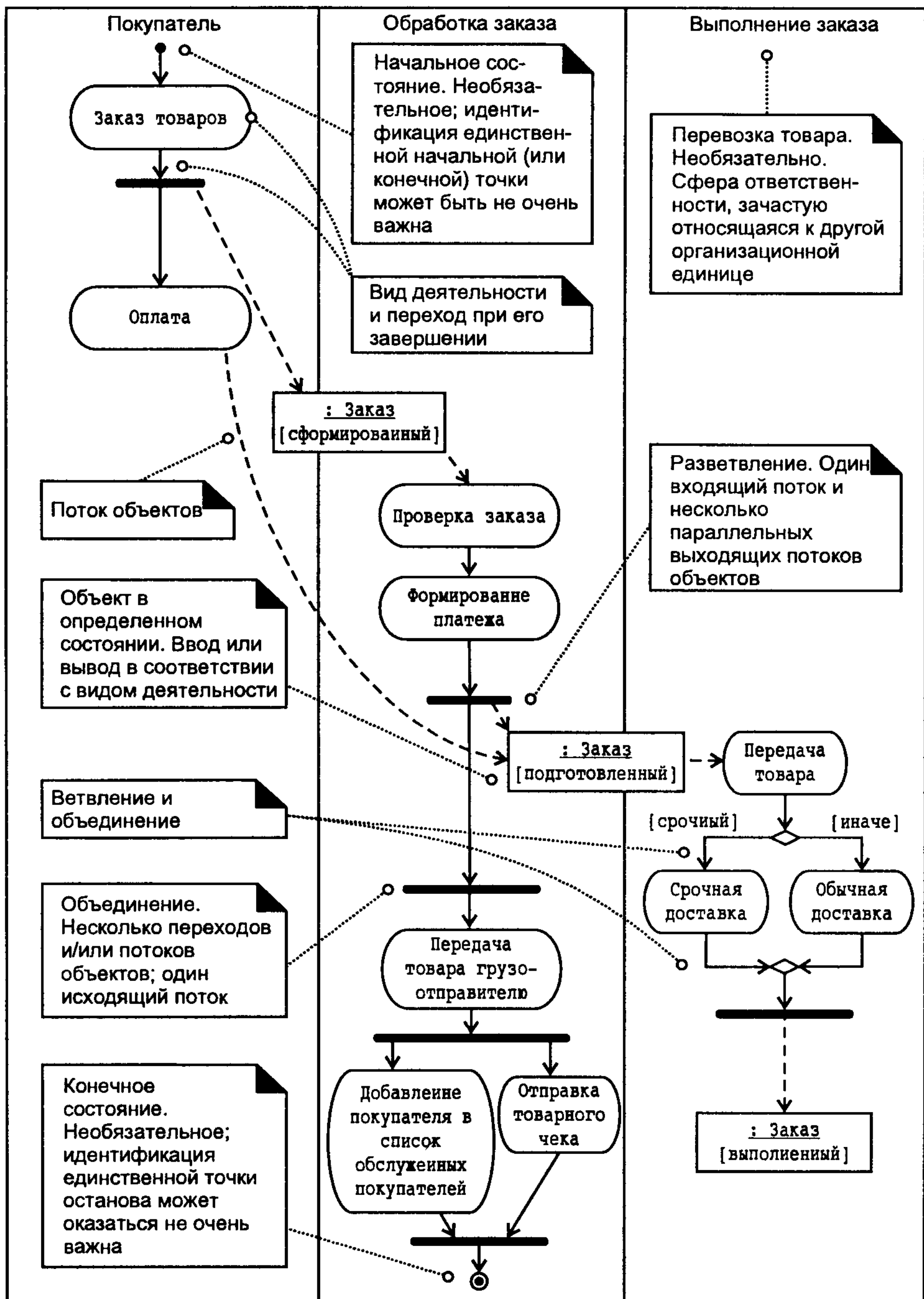


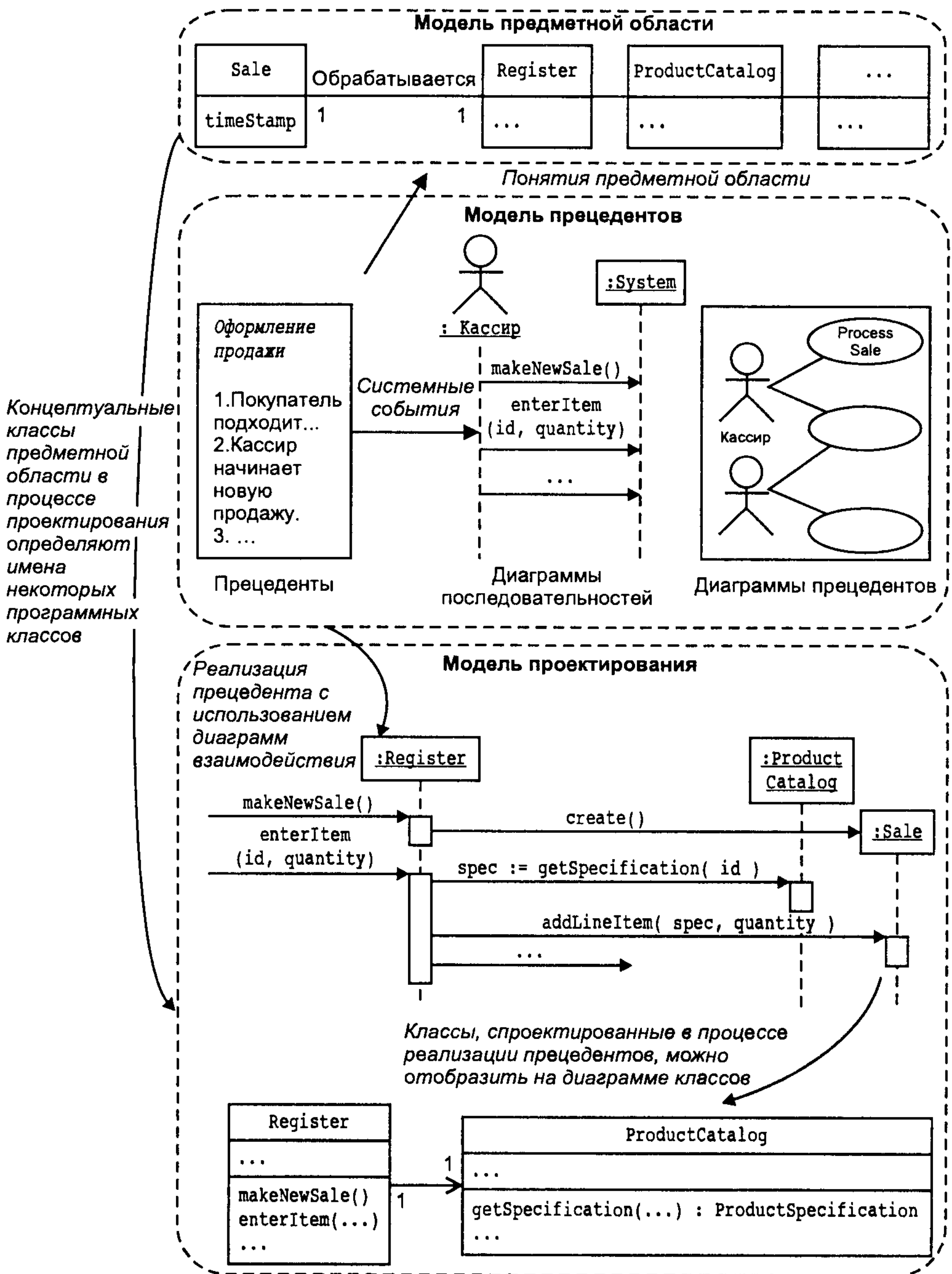

*Рис. 38.7. Диаграмма видов деятельности*

# Артефакты унифицированного процесса, шаблоны GRASP и условные обозначения языка UML

***Примерный набор артефактов унифицированного процесса и время их создания (н — начало, р — развитие)***

| Дисциплина | Артефакт<br>Итерация→ | Начало<br>I1 | Развитие<br>E1..En | Конструирование<br>C1..Cn | Передача<br>T1..Tn |
|---|---|---|---|---|---|
| Бизнес-моделирование | Модель предметной области | | н | | |
| Требования | Модель прецедентов | н | р | | |
| | Видение системы | н | р | | |
| | Дополнительная спецификация | н | р | | |
| | Словарь терминов | н | р | | |
| Проектирование | Модель проектирования | | н | р | |
| | Описание архитектуры | | н | | |
| | Модель данных | | н | р | |
| Реализация | Модель реализации | | н | р | р |
| Управление проектом | План разработки | н | р | р | р |
| Тестирование | Модель тестирования | | н | р | |
| Окружение | Набор документов | н | р | | |

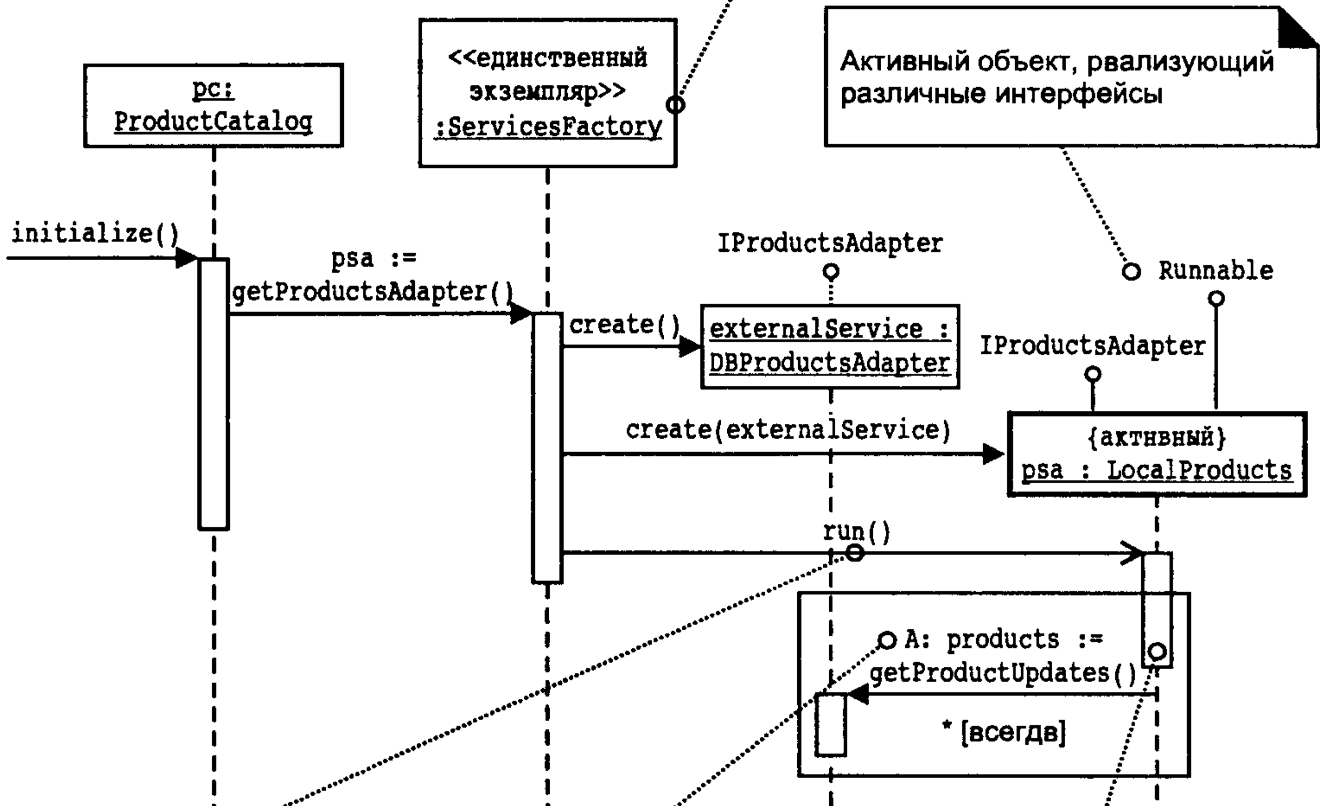
## Пример взаимосвязи артефактов унифицированного процесса



## *Основные шаблоны распределения обязанностей (шаблоны GRASP)*

| Шаблон | Описание |
|---|---|
| Information Expert | Какой класс обычно должен отвечать за выполнение обязанностей.<br>Эта обязанность назначается информационному эксперту — классу, имеющему информацию, которая необходима для выполнения обязанности |
| Creator | Кто должен отвечать за создание экземпляров классов<br>Классу В назначается обязанность по созданию экземпляров класса А, если выполняется одно из следующих условий.<br>◆ Класс В *содержит* объекты А<br>◆ Класс В *агрегирует* объекты А<br>◆ Класс В *обладает данными инициализации* для объекта А<br>◆ Класс В *записывает* экземпляры объектов А<br>◆ Класс В *активно использует* объекты А |
| Controller | Какой класс должен обрабатывать системные события.<br>Обязанность по обработке системных сообщений назначается классу, удовлетворяющему одному из следующих условий.<br>1. Класс представляет всю систему, устройство или подсистему в целом (внешний контроллер).<br>2. Класс представляет некоторый сценарий прецедента, в процессе выполнения которого происходят системные события (контроллер прецедента или сеанса) |
| Low Coupling (оценочный) | Как обеспечить низкую зависимость классов и повысить возможность повторного использования.<br>Обязанности распределяются таким образом, чтобы степень связанности оставалась низкой |
| High Cohesion (оценочный) | Как обеспечить возможность управления сложностью.<br>Обязанности распределяются таким образом, чтобы степень зацепления оставалась высокой |
| Polymorphism | Кто должен выполнять обязанности, если поведение зависит от типа.<br>Если поведение объектов одного типа (класса) может изменяться, обязанности распределяются для различных вариантов поведения с использованием полиморфных операций для этого класса |
| Pure Fabrication | Если у разработчика возникли проблемы, кто должен обеспечить реализацию шаблонов High Cohesion и Low Coupling.<br>Создается искусственный класс, не представляющий конкретного понятия из предметной области, т.е. синтезируется искусственная сущность для поддержки высокого зацепления, слабого связывания и повторного использования |
| Indirection | Кто должен обеспечить отсутствие прямого связывания.<br>Вводится промежуточный объект для обеспечения связи между другими компонентами или службами, которые не связаны между собой напрямую |
| Protected Variations | Как распределить обязанности между объектами, подсистемами и системами так, чтобы вариации или нестабильность их элементов не оказывала отрицательного влияния на другие элементы.<br>Идентифицируются конкретные нестабильные точки. Обязанности распределяются таким образом, чтобы вокруг этих точек был создан "стабильный" интерфейс |

## Диаграммы последовательностей




В Jввв вызов методв run интерфейса Runnable рвссматриввется как всинхронное сообщение. Такие сообщения обозначвются с помощью стрелки другого вида

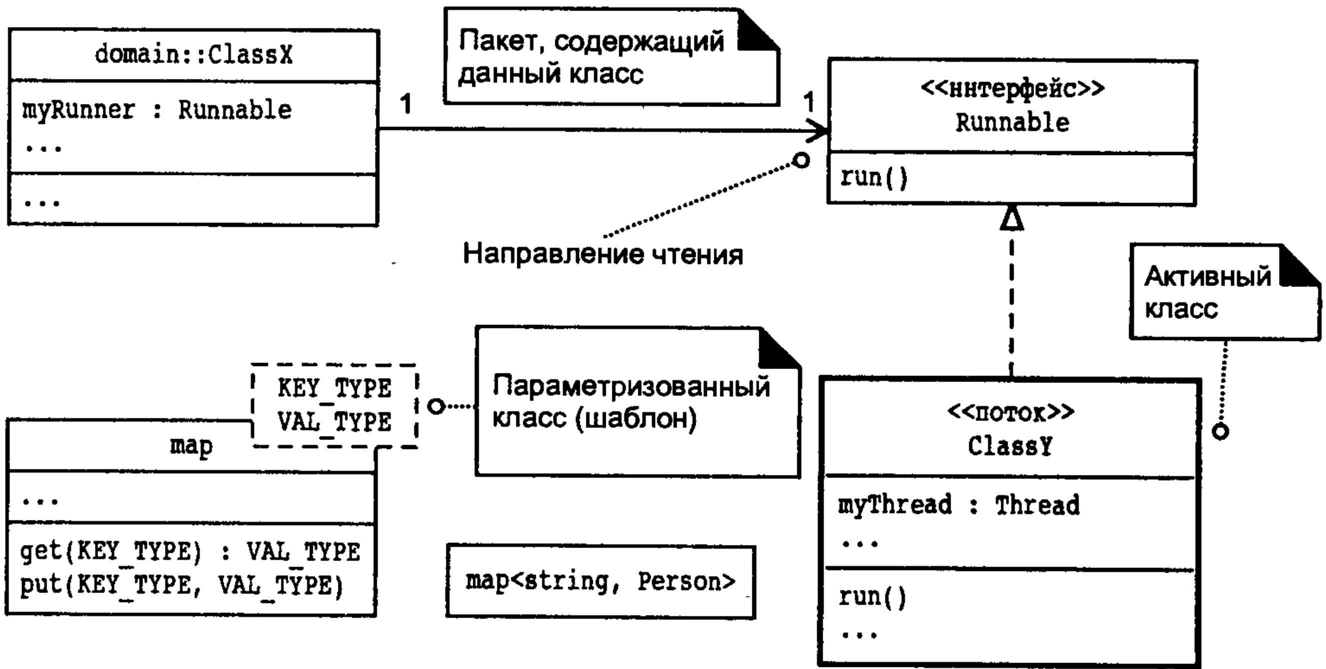
Если методы звпускаются в другом потоке, соответствующее выражение UML нв дивгрвмме последоввтельностей может нвчинвться с имени или символьного обозначения потока. Например, все сообщения, обрабвтываемые в потоке LocalProducts, будут помечаться символом A

```
//это сообщенне обрабатывается
//в своем собственном потоке
{
всегда выполняется циклическн:
  - "засыпает" на n минут
  - обновляет буфер
}
```

## Диаграммы кооперации

Итервция над всеми элементами коллекции

## *Дополнительные обозначения диаграмм классов*



ИмяКласса

атрибутКласса
+ открытыйАтрибут
- закрытыйАтрибут
атрибутБезЗаданияОбластиВидимости
атрибут1: тип
burgers: List of VeggieBurger
атрибут2: тип = начальное значение
finalConstantAttribute: int=5 {фиксированный}
/унаследованныйАтрибут

методКласса()
+ <<конструктор>> ИмяКласса(int)
<<сигнал>> CaughtException1()
методБезЗаданияОбластиВидимости()
методВозвращающийЗначение(): Foo
*абстрактныйМетод()*
абстрактныйМетод2() {абстрактный}
// дополнительное обозначение
+ открытыйМетод()
- закрытыйМетод()
# защищенныйМетод()
~ видимыйМетодПакета()
конечныйМетод() {лист}
методБезПобочныхЭффектов() {запрос}
методСинхронизации() {защищенный}
метод1СПараметрами(in parm1:String, inout parm2:int)
метод2СПараметрами(parm1:String, parm2:float)
метод3СПараметрами(parm1, parm2)
метод4СПараметрами(String, int)
методСПараметрамиИВозвращаемым Значением(parm1:String) : Foo
методСНезаданнымиПараметрами(...) : Foo
методСНезаданнымиПараметрами ИВозвращаемымЗначением()

**Исключения**
ГенерируемоеИсключение1

Класс СНезаданными Атрибутами | foo() ...

Класс СНезаданными Методами | bar : int ...

Примечание. Класс языка UML может использоваться для моделирования, либо понятий предметной области, либо программных классов

КлассБез Атрибутов | | foo()

КлассС Неуказанными Атрибутами | ... | foo()

*Абстрактный Класс* | ...

<<интерфейс>> ИнтерфейсА | операция1()

Обобщение

Реализация интерфейса

КлассБ | операция1() ...

Альтернативное Обозначение Абстрактного Класса {абстрактный} | ...

КонечныйКласс {лист} | ...

Целое

1

Агрегация

*

Часть

Альтернативное Обозначение Реализации ИнтерфейсаА | операция1()

ИнтерфейсА

**Ассоциации**

КлассА — 1 Имя ассоциации ▸ 1 — КлассБ

роль-1 роль-2

АссоциативныйКласс

**Кратность**

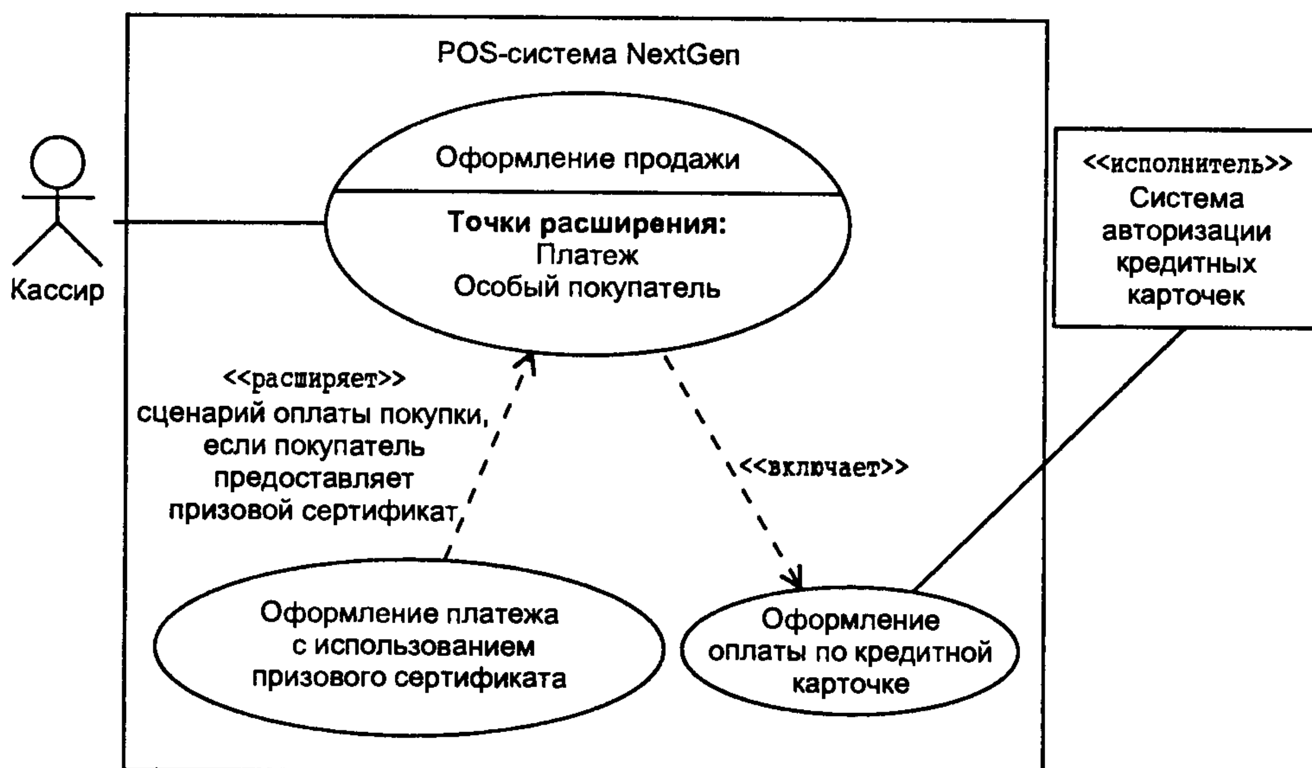
* Класс — 0 или больше; "много"

1..40 Класс — От одного до сорока

1..* Класс — Один или больше

5 Класс — Ровно пять

## *Диаграммы прецедентов*



# Литература


1. Abbot R. *Program Design by Informal English Descriptions*, Communications of the ACM, vol. 26(11), 1983.
2. Alexander C., Ishikawa S. and Silverstein M. *A Pattern Language — Towns-Building-Construction*, Oxford University Press, 1977.
3. Ambler S. *The Unified Process — Elaboration Phase*, Lawrence, KA.: R&D Books, 2000.
4. Ambler S., Constantine L. Enterprise-Ready Object IDs, *The Unified Process — Construction Phase*, Lawrence, KA.: R&D Books, 2000.
5. Ambler S. *The Design of a Robust Persistence Layer For Relational Databases*, www.ambysoft.com, 2000.
6. Beedle M., Devos M., Sharon Y., Schwaber K. and Sutherland J. SCRUM: A Pattern Language for Hyperproductive Software Development. *Pattern Languages of Program Design*, vol. 4, MA.: Addison-Wesley, 2000.
7. Beck K. and Cunningham W. *Using Pattern Languages for Object-Oriented Programs*, Tektronix Technical Report, CR-87-43, 1987.
8. Beck K. and Cunningham W. A Laboratory for Object-Oriented Thinking, *Proceedings of OOPSLA 89*, SIGPLAN Notices, vol. 24, N 10, 1989.
9. Bass L., Clements P. and Kazman R. *Software Architecture in Practice*, MA.: Addison-Wesley.
10. Beck K. *Patterns and Software Development*, Dr. Dobbs Journal, Feb. 1994.
11. Beck K. *Extreme Programming Explained — Embrace Change*, MA.: Addison-Wesley, 2000.
12. Beck K., Fowler M. *Planning Extreme Programming*, MA.: Addison-Wesley, 2000.
13. Bjorner D. and Jones C. *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science, vol. 61, Springer-Verlag, 1978.
14. Booch G., Jacobson I. and Rumbaugh J. *The UML specification documents*, Santa Clara, CA.: Rational Software Corp. (Спецификацию можно найти по адресу www.rational.com), 1997.
15. Buschmann F., Meunier R., Rohnert H., Sommerlad P. and Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, West Sussex, England: Wiley, 1996.
16. Boehm B. A *Spiral Model of Software Development and Enhancement*, IEEE Computer, May 1988.
17. Boehm B. *et al. Software Cost Estimation with COCOMO II*, Englewood Cliffs, NJ.: Prentice-Hall, 2000.
18. Booch G. *Object-Oriented Analysis and Design*, Redwood City, CA.: Benjamin/Cummings, 1994.

19. Booch G., *Object Solutions: Managing the Object-Oriented Project*, Menlo Park, CA.: Addison-Wesley, 1996.
20. Boehm B. and Papaccio P. *Understanding and Controlling Software Costs*, IEEE Transactions on Software Engineering, Oct. 1988.
21. Booch G., Rumbaugh J. and Jacobson I. *The Unified Modeling Language User Guide*, MA.: Addison-Wesley, 1999.
22. Brooks F. *The Mythical Man-Month*, MA.: Addison-Wesley, 1975.
23. Brown K. (Описание шаблона Convert Exception можно найти в Internet по адресу `http://c2.com`.)
24. Brown K. and Whitenack B., Crossing Chasms *A Pattern Language for Object-RDBMS Integration*, Knowledge Systems Corp, 1995.
25. Brown K. and Whitenack B. Crossing Chasms. *Pattern Languages of Program Design*, vol. 2, MA.: Addison-Wesley, 1996.
26. Cook S. and Daniels J. *Designing Object Systems*, Englewood Cliffs, NJ.: Prentice-Hall, 1994.
27. Coad P., De Luca J., Lefebvre E. *Java Modeling in Color with UML*, Englewood Cliffs, NJ.: Prentice-Hall, 1999.
28. Constantine L. and Lockwood L. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*, MA.: Addison-Wesley, 1999.
29. Constantine L., Myers G. and Stevens W. *Structured Design*, IBM Systems Journal, vol. 13 (No. 2, 1974), pp. 115–139, 1974.
30. Coad P. *Object Models: Strategies, Patterns and Applications*, Englewood Cliffs, NJ.: Prentice-Hall, 1995.
31. Cockburn A. *Using Natural Language as a Metaphoric Basis for Object-Oriented Modeling and Programming*, IBM Technical Report TR-36.0002, 1992.
32. Cockburn A. *Structuring Use Cases with Goals*. Journal of Object-Oriented Programming, Sep-Oct and Nov-Dec. SIGS Publications, 1997.
33. Cockburn A. *Writing Effective Use Cases*, MA.: Addison-Wesley, 2001.
34. Colemann D. *et al. Object-Oriented Development: The Fusion Method*, Englewood Cliffs, NJ.: Prentice-Hall, 1994.
35. Constantine L. Segmentation and Design Strategies for Modular Programming. In Barnett and Constantine (eds.), *Modular Programming: Proceedings of a National Symposium*, Cambridge, MA.: Information & Systems Press, 1968.
36. Constantine L. *Essentially Speaking*, Software Development, CMP Media, 1994.
37. Conway M. *Proposal for a Universal Computer-Oriented Language*, Communications of the ACM. 5-8 vol. 1, N 10, October, 1958.
38. Coplien J. *The History of Patterns*, 1995. (См. `http://c2.com/cgi/wiki?HistoryOfPatterns`.)

39. Coplien J. *A Generative Development-Process Pattern Language*, Pattern Languages of Program Design, vol. 1, MA.: Addison-Wesley, 1995.
40. Coplien J. and Schmidt D. et al. *Pattern Languages of Program Design*, vol. 1, MA.: Addison-Wesley, 1995.
41. Cunningham W. *EPISODES: A Pattern Language of Competitive Development*, Pattern Languages of Program Design, vol. 2, MA.: Addison-Wesley, 1996.
42. Cutter Group. *Report: The Corporate Use of Object Technology*, 1997.
43. Corbato F. and Vyssotsky V. *Introduction and overview of the Multics system*, AFIPS Conference Proceedings 27, pp. 185–196, 1965.
44. Dijkstra E. *The Structure of the THE-Multiprogramming System*, Communications of the ACM, 11(5), 1968.
45. Eck D. *The Most Complex Machine*, A K Paters Ltd, 1995.
46. Fowler M. *Analysis Patterns: Reusable Object Models*, MA.: Addison-Wesley, 1996.
47. Fowler M. *Put Your Process on a Diet*, Software Development, Dec., CMP Media, 2000.
48. Fowler M. *Draft patterns on object-relational persistence services*, 2001. (См. `www.martin-fowler.com`.)
49. Fowler M. and Scott. *UML Distilled*, MA.: Addison-Wesley, 2000.
50. Schulte R. *Three-Tier Computing Architectures and Beyond*, Published Report Note R-401-134, Gartner Group, 1995.
51. Gemstone Corp. (Описание набора архитектурных шаблонов можно найти по адресу `www.javasuccess.com`.)
52. Gamma E., Helm R., Johnson R. and Vlissides J. *Design Patterns*, MA.: Addison-Wesley, 1995.
53. Gilb T. *Principles of Software Engineering Management*, MA.: Addison-Wesley, 1988.
54. Guiney E. and Kulak D. *Use Cases: Requirements in Context*, MA.: Addison-Wesley, 2000.
55. Goldberg A. and Kay A. *Smalltalk-72 Instruction Manual*, Xerox Palo Alto Research Center, 1976.
56. Guthrie R. and Larman C. *Java 2 Performance and Idiom Guide*, Englewood Cliffs, NJ.: Prentice-Hall, 2000.
57. Grady R. *Practical Software Metrics for Project Management and Process Improvement*, Englewood Cliffs, NJ.: Prentice-Hall, 1992.
58. Grosso W. (Описание шаблона The Name The Problem Not The Thrower можно найти в Internet по адресу `http://c2.com`.)
59. Gause D. and Weinberg G. *Exploring Requirements*, NY, NY.: Dorset House, 1989.
60. Harrison N. *Patterns for Logging Diagnostic Messages*, Pattern Languages of Program Design, vol. 3, MA.: Addison-Wesley, 1998.

61. Hay D. *Data Model Patterns: Conventions of Thought*, NY, NY.: Dorset House, 1996.
62. Highsmith J. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, NY.: Dorset House, 2000.
63. Hofmeister C. Nord R. and Soni D. *Applied Software Architecture*, MA.: Addison-Wesley, 2000.
64. Jacobson M. *Software Requirements and Specification*, NY.: ACM Press, 1995.
65. Jacobson I. *et al. Object-Oriented Software Engineering: A Use Case Driven Approach*, MA.: Addison-Wesley, 1992.
66. Jeffries R. Anderson A., Hendrickson C. *Extreme Programming Installed*, MA.: Addison-Wesley, 2000.
67. Jacobson I. Booch G. and Rumbaugh J. *The Unified Software Development Process*, MA.: Addison-Wesley, 1999.
68. Jones C. *Applied Software Measurement*, NY.: McGraw-Hill, 1997.
69. Jones C. *Estimating Software Costs*, NY.: McGraw-Hill, 1998.
70. Kay A. *FLEX, a flexible extensible language*, M.Sc. thesis, Electrical Engineering, University of Utah. May (Univ. Microfilms), 1968.
71. Kovitz B. *Practical Software Requirements*, Greenwich, CT.: Manning, 1999.
72. Kruchten P. *The Rational Unified Process — An Introduction*, 2nd edition, MA.: Addison-Wesley, 2000.
73. Kruchten P. *The 4+1 View Model of Architecture*, IEEE Software 12(6), 1995.
74. Lakos J. *Large-Scale C++ Software Design*, MA.: Addison-Wesley, 1996.
75. Lieberherr K., Holland I. and Riel A. *Object-Oriented Programming: An Objective Sense of Style*. OOPSLA 88 Conference Proceedings, NY.: ACM SIGPLAN, 1988.
76. Liskov B. *Data Abstraction and Hierarchy*, SIGPLAN Notices, 23, 5 (May, 1988).
77. Leffingwell D. and Widrig D. *Managing Software Requirements: A Unified Approach*, MA.: Addison-Wesley, 2000.
78. MacCormack A. *Product-Development Practices That Work*, MIT Sloan Management Review, vol. 42, N 2, 2001.
79. Martin R. *Designing Object-Oriented C++ Applications Using the Booch Method*, Englewood Cliffs, NJ.: Prentice-Hall, 1995.
80. McConnell S. *Rapid Development*, Redmond, WA.: Microsoft Press, 1996.
81. Martin J. and Odell J. *Object-Oriented Methods: A Foundation*, Englewood Cliffs, NJ.: Prentice-Hall, 1995.
82. Moreno A.M. Object Oriented Analysis from Textual Specifications. *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, Madrid, June 17-20 (1997).
83. McMenamin S. and Palmer J. *Essential Systems Analysis*, Englewood Cliffs, NJ.: Prentice-Hall, 1984.

84. *The Merriam-Webster Dictionary*, Springfield, MA.: Merriam-Webster, 1989.
85. Nixon R. *Six Crisis*, NY.: Touchstone Press, 1990.
86. *Object Management Group*, 2001. OMG Unified Modeling Language Specification. www.omg.org.
87. Parkinson N. *Parkinson's Law: The Pursuit of Progress*, London, John Murray, 1958.
88. Parnas D. *On the Criteria To Be Used in Decomposing Systems Into Modules*, Communications of the ACM, vol. 5, N 12, Dec. 1972. ACM.
89. Putnam L. and Myers W. *Measures for Excellence: Reliable Software on Time, Within Budget*, Yourdon Press, 1992.
90. Pree W. *Design Patterns for Object-Oriented Software Development*, MA.: Addison-Wesley, 1995.
91. Renzel K. *Error Handling for Business Information Systems: A Pattern Language*. Статью можно найти по адресу http://www.objectarchitects.de/arcus/cookbook/exhandling/.
92. Rising L. *Pattern Almanac 2000*, MA.: Addison-Wesley, 2000.
93. Rumbaugh J. Jacobson I. and Booch G. *The Unified Modeling Language Reference Manual*, MA.: Addison-Wesley, 1999.
94. Ross R. *The Business Rule Book: Classifying, Defining and Modeling Rules*, Business Rule Solutions Inc., 1997.
95. Royce W. *Managing the Development of Large Software Systems*, Proceedings of IEEE WESCON, Aug. 1970.
96. Rumbaugh J. *et al. Object-Oriented Modeling and Design*, Englewood Cliffs, NJ.: Prentice-Hall, 1991.
97. Интерактивная документация по унифицированному процессу RUP, распространяемая компанией Rational Corp.
98. Rumbaugh J. *Models Through the Development Process*, Journal of Object-Oriented Programming, May 1997, NY.: SIGS Publications.
99. Shaw M. *Some Patterns for Software Architectures*, Pattern Languages of Program Design, vol. 2. MA.: Addison-Wesley, 1996.
100. Jim Johnson. *Chaos: Charting the Seas of Information Technology*, Published Report. The Standish Group, 1994.
101. Schneider G. and Winters J. *Applying Use Cases: A Practical Guide*, MA.: Addison-Wesley, 1998.
102. Tsichiritzis D. and Klug A. *The ANSI/X3/SPARC DBMS framework: Report of the study group on database management systems*. Information Systems, 3 1978.
103. Tufte E. *The Visual Display of Quantitative Information*, Graphics Press, 1992.
104. Vlissides J. *et al. Patterns Languages of Program Design*, vol. 2, MA.: Addison-Wesley, 1996.

105. Wirfs-Brock R. *Designing Scenarios: Making the Case for a Use Case Framework*, Smalltalk Report, Nov-Dec 1993, NY: SIGS Publications.

106. Warmer J. and Kleppe A. *The Object Constraint Language: Precise Modeling with UML*, MA.: Addison-Wesley, 1999.

107. Wirfs-Brock R. Wilkerson B. and Wiener L. *Designing Object-Oriented Software*, Englewood Cliffs, NJ.: Prentice-Hall, 1990.

# Словарь терминов

| Термин | Определение |
|---|---|
| **OID** | Идентификатор объекта. |
| **Private** | Спецификатор закрытой области. Обычно в закрытой области объявляются все атрибуты и некоторые методы. |
| **Public** | Спецификатор открытой области. Обычно в открытой области объявляются некоторые методы, но не атрибуты. |
| **Абстрактный класс** (abstract class) | Класс, который можно использовать только в качестве суперкласса для некоторых других классов. Такой класс не может иметь экземпляров, кроме экземпляров производных классов. |
| **Абстракция** (abstraction) | Выделение важных или общих свойств подобных понятий, а также выделение в результате важных характеристик сущности. |
| **Агрегация** (aggregation) | Свойство ассоциации, представляющей отношение "целое–часть", и (обычно) ограничение времени жизни частей временем жизни целого. |
| **Активный объект** (active object) | Объект, для которого существует отдельный поток управления. |
| **Анализ** (analysis) | Исследование предметной области, которое приводит к созданию моделей, описывающих статические и динамические характеристики ее объектов. Этот процесс призван ответить на вопрос "Что?", а не на вопрос "Как?". |
| **Архитектура** (architecture) | Описание организации и структуры системы. При разработке программных систем рассматривается несколько различных уровней архитектуры, начиная с физической или аппаратной архитектуры и заканчивая логической архитектурой контуров приложения. |
| **Ассоциация** (association) | Описание набора однородных связей между объектами двух классов. |
| **Атрибут** (attribute) | Именованная характеристика или свойство класса. |
| **Атрибут класса** (class attribute) | Характеристика или свойство, общее для всех экземпляров класса. Эта информация обычно хранится в определении класса. |
| **Видимость** (visibility) | Способность видеть другой объект или ссылаться на него. |
| **Делегирование** (delegation) | Передача сообщения другому объекту при получении сообщения. Таким образом, первый объект делегирует свои обязанности второму объекту. |
| **Закрытая область** (private) | Механизм ограничения доступа к членам класса со стороны других объектов. Обычно в закрытой области объявляются все атрибуты и некоторые методы. |

| | |
|---|---|
| **Идентичность объекта** (object identity) | Свойство объекта, состоящее в том, что существование объекта не зависит от любых значений, связанных с этим объектом. |
| **Иерархия классов** (class hierarchy) | Описание отношений наследования между классами. |
| **Инкапсуляция** (encapsulation) | Механизм, используемый для сокрытия данных, внутренней структуры и деталей реализации некоторого элемента, такого как объект или подсистема. Все взаимодействие с объектом выполняется через открытый интерфейс операций. |
| **Инстанцирование** (instantiation) | Создание экземпляра класса. |
| **Интерфейс** (interface) | Набор сигнатур открытых операций. |
| **Квалифицированная ассоциация** (qualified association) | Ассоциация, участники которой определяются значением квалификатора. |
| **Класс** (class) | В языке UML это "описание набора объектов с общими атрибутами, операциями, методами, взаимосвязями и поведением" [93]. Данный термин можно использовать для представления программных или концептуальных элементов. |
| **Классификация** (classification) | Определяет отношение между классом и его экземплярами. Соответствие классификации (classification mapping) устанавливает расширение класса. |
| **Композитный класс** (composition) | Класс, каждый экземпляр которого состоит из объектов других классов. |
| **Конкретный класс** (concrete class) | Класс, который может иметь экземпляры. |
| **Конструктор** (constructor) | Специальный метод, вызываемый при создании экземпляра класса в языке C++ или Java. Зачастую конструктор выполняет инициализацию объекта. |
| **Контейнерный класс** (container class) | Класс, разработанный для поддержки выполнения операций с набором объектов. |
| **Контур** (framework) | Набор взаимодействующих абстрактных и конкретных классов, который можно использовать в качестве шаблона для решения группы взаимосвязанных проблем. Контур обычно дополняется производными классами с конкретным поведением. |
| **Кооперация** (collaboration) | Взаимодействие двух или нескольких объектов в рамках отношения "клиент/сервер" для реализации некоторой службы. |
| **Кратность** (multiplicity) | Допустимое число объектов, принимающих участие в некоторой ассоциации. |
| **Метамодель** (metamodel) | Модель, определяющая другие модели. Метамодель UML определяет типы элементов языка UML, такие как классификатор. |

| | |
|---|---|
| **Метод** (method) | В UML это конкретная реализация алгоритма выполнения операции для некоторого класса. Неформально это программная процедура, которая выполняется в ответ на сообщение. |
| **Метод класса** (class method) | Метод, определяющий поведение самого класса, а не его экземпляров. |
| **Метод экземпляра** (instance method) | Метод, действие которого распространяется на один экземпляр. Реализуется путем передачи сообщения данному экземпляру. |
| **Модель** (model) | Описание статических или динамических характеристик системы, представленное с различных точек зрения (обычно в текстовом виде или в виде диаграмм). |
| **Наследование** (derivation) | Процесс определения нового класса на основе свойств существующего класса и добавления к нему новых атрибутов и методов. Существующий класс при этом называется суперклассом, а новый — подклассом или производным классом. |
| **Наследование** (inheritance) | Возможность объектно-ориентированных языков программирования, благодаря которой можно создавать специализированные подклассы на основе более общих суперклассов. Атрибуты и методы суперклассов автоматически присваиваются производным классам. |
| **Обобщение** (generalization) | Выявление общих свойств понятий и определение принципов взаимодействия суперкласса (общего понятия) и подклассов (специализированных понятий). Это способ классификации понятий, который затем отражается в иерархии классов. Концептуальные подклассы и суперклассы связаны отношением "обобщение/специализация". |
| **Объект** (object) | В UML это экземпляр класса, инкапсулирующий определенное состояние или поведение. Неформально это пример некоторой сущности. |
| **Объект, подлежащий постоянному хранению** (persistent object) | Объект, время жизни которого может превышать время жизни процесса или потока, в котором он был создан. Такие объекты существуют до момента их явного удаления. |
| **Объектно-ориентированное проектирование** (object-oriented design) | Определение логики программного решения в терминах программных объектов, а именно — их классов, атрибутов, методов и их взаимодействия. |
| **Объектно-ориентированный анализ** (object-oriented analysis) | Исследование предметной области или системы в терминах понятий предметной области, таких как концептуальные классы, ассоциации и изменение состояний. |
| **Объектно-ориентированный язык программирования** (object-oriented programming language) | Язык программирования, поддерживающий принципы инкапсуляции, наследования и полиморфизма. |

| | |
|---|---|
| **Обязанность** (responsibility) | Услуга или набор услуг, обеспечиваемых некоторым элементом (например, классом или подсистемой). Обязанность охватывает одну или несколько задач или обязательств элемента. |
| **Ограничение** (constraint) | Ограничение или условие, налагаемое на элемент. |
| **Операция** (operation) | В UML это "спецификация преобразования или запроса, выполняемого объектом" [93]. Операция имеет сигнатуру, определяемую ее именем и параметрами, и выполняется посредством передачи сообщения. Метод — это реализация операции со специфическим алгоритмом. |
| **Описание операции** (contract) | Описание обязанностей и постусловий, реализуемых при выполнении операции или метода. Используется также для обозначения множества всех условий, связанных с интерфейсом. |
| **Открытая область** (public) | Механизм обеспечения доступа к членам класса со стороны других объектов. Обычно в открытой области объявляются некоторые методы, но не атрибуты, поскольку открытость атрибутов нарушает принцип инкапсуляции. |
| **Переменная экземпляра** (instance variable) | Атрибут экземпляра, используемый в Java или Smalltalk. |
| **Переход** (transition) | Взаимосвязь между состояниями, активизирующаяся в результате некоторого события или выполнения некоторых условий. |
| **Переход состояний** (state transition) | Изменение состояния объекта. Нечто, о чем сигнализируют события. |
| **Подкласс** (subclass) | Класс, являющийся специализацией другого класса (суперкласса). Подкласс наследует атрибуты и методы суперкласса. |
| **Подтип** (subtype) | Концептуальный подкласс. Специализация другого типа (супертипа), согласованная с супертипом. |
| **Полиморфизм** (polymorphism) | Возможность разной реакции различных классов или объектов на одно и то же сообщение, основанная на использовании полиморфных операций. Возможность определять полиморфные операции. |
| **Полиморфная операция** (polymorphic operation) | Операция, которая по-разному реализуется различными классами. |
| **Получатель** (receiver) | Объект, которому отправляется сообщение. |
| **Понятие** (concept) | Категория идей или предметов. В данной книге используется для определения понятий реального мира, в отличие от программных сущностей. Понятие определяется своими атрибутами, операциями и семантикой. Понятие в широком смысле (concept's extension) — это набор экземпляров или примеров объектов, описываемых данным понятием. Зачастую используется в качестве синонима термина "класс". |

| | |
|---|---|
| **Понятие в широком смысле** (extension) | Набор объектов, к которым применимо данное понятие. Объекты расширенного понятия — это примеры или экземпляры понятия. |
| **Постоянный носитель** (persistence) | Место постоянного хранения состояний объекта. |
| **Постусловие** (post-condition) | Ограничение, которое должно соблюдаться после выполнения операции. |
| **Предметная область** (domain) | Формальная область, определяющая объект или сферу интересов. |
| **Предусловие** (pre-condition) | Ограничение, которое должно соблюдаться до начала выполнения операции. |
| **Проектирование** (design) | Процесс разработки спецификации для реализации системы на основе результатов анализа. Логическое описание принципов работы системы. |
| **Рекурсивная ассоциация** (recursive association) | Ассоциация, в которой классы начального и конечного объектов совпадают. |
| **Роль** (role) | Именованный конец ассоциации, указывающий ее задачу. |
| **Связывание** (coupling) | Зависимость между элементами (обычно — классами, пакетами или подсистемами), вызываемая взаимодействием между элементами для реализации службы. |
| **Связь** (link) | Наличие связи между двумя объектами; экземпляр ассоциации. |
| **Событие** (event) | Важное действие или стечение обстоятельств. |
| **Сообщение** (message) | Средство “общения” между объектами; обычно — запрос на выполнение метода. |
| **Состояние** (state) | Содержание объекта между событиями. |
| **Суперкласс** (superclass) | Класс, от которого другие классы наследуют атрибуты и методы. |
| **Супертип** (supertype) | Концептуальный суперкласс. Задает более общий тип в отношении “общее–частное”; объект, имеющий подтипы. |
| **Сущность** (intension) | Определение понятия. |
| **Чистые данные** (pure data values) | Типы данных, для которых не играют роли идентичность каждого отдельного экземпляра, например числа, логические значения и строки. |
| **Шаблон** (pattern) | Именованное описание проблемы, ее решения, области применения этого решения и способов его применения в новых ситуациях. |
| **Экземпляр** (instance) | Отдельный член класса, в UML называемый объектом. |

# Предметный указатель


A

Abstract
  class, *406*; *603*
  use case, *390*
Abstract Factory, шаблон, *512*
Abstraction, *603*
Accessing method, *295*
Activation box, *219*
Active object, *500*; *603*
Activity diagram, *587*
Actor, *75*; *95*
Adapter, шаблон, *337*; *348*; *496*
Addition use case, *390*
Aggregation, *235*; *412*; *603*
Agile process, *55*
Analysis, *36*; *603*
Architectural
  analysis, *440*
  baseline, *131*
  design, *440*
  investigation, *440*
  proof-of-concept, *492*
  prototype, *131*
  synthesis, *492*
Architecturally significant requirement, *478*
Architecture, *603*
Architecture view, *478*
Artifact, *50*
Association, *169*; *603*
Associative class, *411*
Attribute, *181*; *603*
  visibility, *287*

B

Base use case, *390*
Black-box use case, *76*
Bloated controller, *249*
Boundary object, *248*
Business
  actor, *100*
  modeling, *50*
  use-case, *100*

C

Cache Management, шаблон, *539*
CASE-средство, *310*; *554*
  Rational Rose, *557*
  Together, *557*
Centralized Error Logging, шаблон, *505*
Class, *604*
  attribute, *603*
  hierarchy, *408*; *604*
  method, *605*
Classification, *604*
Classifier, *163*
Cohesion, *239*
Collaboration, *604*
  diagram, *39*; *210*
Command, шаблон, *252*; *545*
Composite, *412*
Composite aggregation, *413*
Composite, шаблон, *362*
Composition, *413*; *604*
Concept, *606*
Conceptual class partition, *401*
Concrete
  class, *604*
  use case, *94*; *390*
Conflict resolution strategy, *363*
Constraint, *265*; *606*
Construction, *49*
Construction phase, *573*
Constructor, *604*
Container, *488*
Container class, *604*
Context object, *358*

Contract, *606*
Control object, *248*
Controller, объект, *244*
Controller, шаблон, *227*; *243*; *259*; *331*; *451*; *591*
Convert Exceptions, шаблон, *502*
CORBA, спецификация, *507*
Coupling, *236*; *607*
Coverage, *132*
CRC card, *252*
create, сообщение, *295*
Creator, шаблон, *227*; *233*; *260*; *331*; *591*
Cruise Control, средство интеграции, *569*

## D

Data dictionary, *121*
Data type, *182*
Database Broker, шаблон, *529*
Database design, *36*
Delegation, *603*
Delegation Event Model (DEM), *381*
Dependency relationship, *300*
Deployment view, *478*
Derivation, *605*
Derived attribute, *188*
Design, *36*; *50*; *607*
Design by contract, *200*
Design class diagram, *292*
Design model, *206*
Development Case, *53*
Diagnostic Logger
  шаблон, *506*
Direct mapping, *530*
Discipline, *50*
Domain, *607*
  model, *38*
Domain Object Factory, шаблон, *472*
Don't Talk to Strangers, шаблон, *342*
Dynamic view, *39*

## E

Eager initialization, *355*
Eiffel, *200*
Elaboration, *49*
Elementary Business Process (EBP), *86*
Encapsulation, *604*
Entity object, *248*
Error, *500*
Error Dialog, шаблон, *506*
Essential use case, *94*
Event, *431*; *607*
Evolution point, *344*; *486*
Executable architecture, *131*
Expert, шаблон, *331*
Extension, *150*
Extreme Programming, *57*; *316*; *574*

## F

Facade controller, *247*
Facade, шаблон, *371*
Factory, шаблон, *352*; *496*
Failure, *500*
Fault, *500*
Foreign key attribute, *186*
Framework, *525*; *604*
FURPS+, модель, *69*; *76*; *478*

## G

Generalization, *390*; *397*; *605*
Generalization-specialization type hierarchy, *397*
Global visibility, *290*
Glossary, *107*; *121*
GoF, шаблон, *347*
GRASP, шаблоны проектирования, *223*; *226*; *257*

## H

Heavy process, *54*
High Cohesion, шаблон, *227*; *239*; *331*; *591*

## I

Inception, *49*
Incremental process adoption, *569*
Indirect mapping, *530*

Indirection, шаблон, *331*; *338*; *591*
Information Expert, шаблон, *227*; *228*; *262*; *591*
Inheritance, *407*; *605*
Initial domain object, *274*
Instance, *607*
Instance method, *605*
Instance variable, *606*
Instantiation, *604*
Intension, *150*
Interaction diagram, *206*; *209*
Interface, *333*; *604*
Is-a Rule, *400*
ISO 9126, стандарт, *480*
Issue card, *483*
Iteration, *44*
  plan, *55*
Iterative development, *44*

## J

Java Messaging Service(JMS), *487*
Java Native Interface (JNI), *512*
JavaPOS, *511*
Junit, контур модульного тестирования, *317*; *574*

## L

Law of Demeter, шаблон, *342*
Layers, шаблон, *252*; *488*
Lazy
  initialization, *355*
  materialization, *545*
Link, *213*; *607*
Local visibility, *288*
Low Coupling, шаблон, *227*; *236*; *331*; *591*

## M

Mediator, шаблон, *339*
Message, *607*
Metamodel, *604*
Method, *199*; *605*
Microsoft Transaction Service, *488*
Model, *605*
Model-View Separation, шаблон, *463*
Multiobject, *217*
Multiplicity, *173*; *604*

## N

Name The Problem Not The Thrower, шаблон, *503*
Navigability, *297*
new, оператор, *295*
Note, *265*

## O

Object, *605*
  design, *36*
  identifier, *528*
  identity, *604*
  lifeline, *220*
Object Constraint Language (OCL), *196*; *199*; *265*
Object Identifier, шаблон, *528*
Object Management Group (OMG), *40*
Object Modeling Technique (OMT), метод, *40*
Object-oriented
  analysis, *605*
  design, *605*
  programming, *37*
  programming language, *605*
Object-Oriented Software Engineering (OOSE), метод, *40*
Observer, шаблон, *374*
OCP, принцип, *345*
Offstage actor, *95*
OID, *603*
Open-Closed Principle, *345*
Operation, *199*; *606*
  specification, *199*
O-R mapping service, *525*

## P

Package, *373*
Parameter visibility, *287*

Partition, *458*
Pattern, *34*; *207*; *213*; *225*; *607*
    of assigning responsibilities, *224*
Persistent
    framework, *524*
    object, *524*; *605*
Phase plan, *55*
Polymorphic operation, *606*
Polymorphism, *606*
Polymorphism, шаблон, *331*; *591*
Postcondition, *82*; *607*
Precondition, *82*; *607*
Predictive process, *54*
Primary actor, *95*
Proof-of-concept programming, *206*
Protected variations, шаблон, *331*; *339*; *484*; *591*
Proxy, шаблон, *507*
Publish-Subscribe, шаблон, *375*
Pure data values, *607*
Pure Fabrication, шаблон, *252*; *331*; *335*; *591*

## Q

Qualified association, *419*; *604*
Qualifier, *419*
Quality attribute, *71*; *112*

## R

Rational Software, компания, *580*
Rational Unified Process (RUP), *43*
Receiver, *606*
Recursive association, *607*
Reference attribute, *310*
Reflexive association, *420*
Remote Proxy, шаблон, *507*
Representing Objects as Tables, шаблон, *527*
Requirement, *50*; *69*
RequisitePro, средство отслеживания требований, *566*
Resource adapter, *348*
Responsibility, *76*; *206*; *224*; *606*
Reverse engineering, *310*
Risk, *132*
RMI, *507*
Role, *172*; *297*; *607*
    name, *312*
RUP, продукт, *580*
RUP, унифицированный процесс, *43*

## S

Scenario, *75*
SCRUM, шаблон реализации процесса, *574*
Secure Electronic Transaction (SET), *518*
SEI, институт, *71*
Sequence diagram, *140*; *210*
Sequence number, *215*
Shared aggregation, *413*
Signature, *199*
Simple attribute, *182*
Singleton, шаблон, *290*; *353*
Software Architecture Document (SAD), *475*
Software Engineering Institute (SEI), *479*
State, *431*; *607*
State transition, *606*
State, шаблон, *199*; *434*; *540*
State-independent object, *435*
Static view, *39*
Stereotype, *97*
Strategy, шаблон, *357*
Subclass, *397*; *606*
Subfunctional goal, *88*
Subtype, *606*
Superclass, *397*
Supplementary specification, *107*
Supporting actor, *90*; *95*
SWEBOK, *71*
Symbol, *150*
System
    behavior, *140*
    event, *244*
    feature, *118*
    sequence diagram, *140*; *427*
System operation contract, *191*
System use-case, *100*

## T

Technical memo, *483*
Template Method, шаблон, *532*
Test-first programming, *57*; *316*
Three-tier architecture, *461*
Transition, *49*; *432*; *606*
Transition phase, *574*
Two-tier architecture, *462*

## U

UML profile, *527*
Unified Modeling Language (UML), *34*; *40*
Unified Process (UP), *43*
UnifiedPOS, *511*
UP Business Object Model, *166*
UP, унифицированный процесс, *43*
Usability, *580*
Use case, *34*; *75*
  controller, *247*
  instance, *75*
  state diagram, *433*
Use-case driven development, *100*
Use-Case Model, *73*
usecases.org, шаблон описания прецедента, *77*
User goal, *87*

## V

Value object, *184*
Variation point, *344*
Vienna Development Method (VDM), *202*
Virtual Proxy, шаблон, *545*
Visibility, *263*; *286*; *603*

## W

Workflow, *50*

## А

Абстрактный
  класс, *406*; *603*
  класс-фабрика, *513*
Абстрактный прецедент, *390*
Абстракция, *573*; *603*
Агрегация, *235*; *412*; *603*
Адаптер ресурсов, *348*
Активная инициализация, *355*; *499*
Активный объект, *500*; *603*
Анализ, *36*; *603*
  архитектурный, *440*; *476*
  требований, *34*
Артефакт, *50*; *67*
Архитектура, *603*
  многоуровневая, *442*
  системы, *440*
  трехуровневая, *461*
Архитектурная основа, *131*
Архитектурно значимое требование, *478*
Архитектурное предложение, *483*
Архитектурное представление, *478*; *490*
Архитектурный
  анализ, *440*; *476*
  прототип, *131*
  синтез, *492*
  фактор, *477*
  шаблон, *440*
Асинхронное
  событие, *389*
  сообщение, *315*; *503*
Ассоциативный класс, *411*
Ассоциация, *169*; *603*
  имя, *175*
  кратность, *173*
  обозначение, *170*
  рефлексивная, *420*
  роль, *172*
Атрибут, *181*; *603*
  внешнего ключа, *186*
  качества, *71*; *112*
Атрибут-ссылка, *310*

## Б

Бизнес-исполнитель, *100*
Бизнес-моделирование, *50*; *587*
Бизнес-правило, *113*
Бизнес-прецедент, *100*

Блок активации, *219*

В

Видимость, *263*; *285*; *300*; *603*
  глобальная, *290*
  локальная, *288*
  посредством атрибутов, *287*; *297*
  посредством параметров, *287*
Виртуальная машина, *461*
Внешнее событие, *436*
Внешний контроллер, *243*; *247*
Внутреннее событие, *436*
Временное событие, *436*
Вспомогательный исполнитель, *90*

Г

Гибкий процесс, *55*
Границы системы, *132*
Группа OMG, *40*

Д

Двухуровневая архитектура, *462*
Делегирование, *603*
Дематериализация, *526*
Детерминированный процесс, *54*
Диаграмма
  взаимодействия, *39*; *206*; *209*; *254*
  видов деятельности, *587*
  классов, *39*; *148*; *206*; *228*; *307*; *310*
    ассоциации, 298
  классов проектирования, *292*
  кооперации, *210*
  пакетов, *442*; *583*
  последовательностей, *140*; *210*; *427*
  прецедентов, *95*
  развертывания, *491*; *585*
  состояний, *431*
  состояний прецедента, *433*
Динамическое представление, *39*
Дисциплина определения требований, *567*
Документ SAD, *475*
Дополнительная спецификация, *107*; *475*; *560*; *567*
Дополнительный прецедент, *390*

З

Зависимость, *583*
Задача пользователя, *87*
Закон Паркинсона, *574*
Закрытая область, *603*
Зацепление, *239*

И

Идентификатор объекта, *528*
Идентичность объекта, *604*
Идиома проектирования, *441*
Иерархия
  классов, *604*
  обобщения-специализации классов, *397*
  программных классов, *408*
Имя
  ассоциации, *175*
  роли, *312*
Инвариант, *200*
Инкапсуляция, *296*; *604*
Инстанцирование, *604*
Институт программных технологий, *479*
Интерфейс, *333*; *604*
Информационный эксперт, *229*
Информация о навигации, *297*
Исключение, *315*; *502*
Исполнитель, *75*; *95*
  вспомогательный, *95*
  закулисный, *95*
  основной, *95*
Исполняемая архитектура, *131*
Исследование архитектуры, *440*
Исходный специальный объект, *274*
Итеративная разработка, *44*; *47*
Итеративный процесс разработки, *309*; *559*; *574*
Итерация, *44*
  длительность, *48*

## К

Карта CRC, *252*
Каскадный
  архитектурный анализ, *489*
  жизненный цикл, *575*
  принцип разработки, *571*
  цикл разработки, *55*
Квалифицированная ассоциация, *604*
Класс, *163*; *604*
  абстрактный, *603*
  ассоциаций, *409*
  концептуальный, *147*; *257*
  параметризованный, *586*
  понятий, *148*
  программный, *257*
  проектирования, *163*
Классификатор, *163*
Классификация, *604*
  шаблонов, *440*
Класс-фабрика, *353*
Композитная агрегация, *413*
Композитный
  класс, *604*
  объект, *412*
Компонент, *585*
Конкретный
  класс, *604*
  прецедент, *390*
Конструктор, *604*
Контейнер, *488*
Контейнерный класс, *604*
Контекст, *286*
Контекстный объект, *358*
Контроллер
  прецедента, *244*; *247*
  сеанса, *244*
Контроль версий, *573*
Контур, *525*; *604*
  интерфейса с базой данных, *524*
  модульного тестирования JUnit, *574*
Концептуальный класс, *147*; *150*; *163*; *257*
Кооперация, *604*
Кратность, *173*; *604*

## Л

Линия жизни объектов, *220*
Логика приложения, *60*
Логическая архитектура, *442*
Логическое представление, *445*; *478*; *490*

## М

Материализация, *526*
  по требованию, *545*
Метамодель, *604*
Метод, *199*; *605*
  OOSE, *40*
  выбора имени, *295*
  доступа, *295*
  класса, *605*
  OMT, *40*
Метод-шаблон, *532*
Многоуровневая архитектура, *442*
Модель, *605*
  DEM, *381*
  FURPS+, *69*; *76*; *478*
  анализа, *580*
  бизнес-объектов, *587*
  данных, *527*
  концептуальная, *148*
  предметной области, *38*; *147*; *149*; *156*; *166*; *298*
  прецедентов, *73*; *567*
  проектирования, *206*; *292*; *490*
  развертывания, *456*
  реализации, *307*; *459*
  сетевых протоколов, *454*
Модульный принцип проектирования, *439*
Мультиобъект, *217*

## Н

Наследование, *407*; *605*
Начальная фаза проекта, *66*
Независимый от состояния объект, *435*
Непрямое отображение, *530*
Непрямой объект, *342*

## О

Область видимости, *286*
Обобщение, *397*; *605*
Обратное проектирование, *310*; *555*
Объединение, *413*
Объект, *605*
Объект значений, *184*
Объект-декоратор, *488*
Объект-коллекция, *296*
Объект-контейнер, *296*
Объект-контроллер, *244*
Объектная бизнес-модель UP, *166*
Объектная модель анализа, *148*
Объектное проектирование, *36*
Объектно-ориентированное программирование, *37*
Объектно-ориентированное проектирование, *39*; *605*
Объектно-ориентированный анализ, *605*
Объектно-ориентированный язык программирования, *605*
Объектный анализ, *36*
Объектный язык ограничений (OCL), *196*; *199*
Объект-посредник, *507*; *545*
Объект-фабрика, *274*
Обязанность, *76*; *224*; *606*
  распределение, *206*
Ограничение, *265*; *606*
Оператор new, *295*
Операция, *199*; *606*
Описание
  архитектурных подходов, *483*
  операции, *606*
  системной операции, *191*
Опорный объект, *507*
Основной прецедент, *390*
Основные принципы проектирования архитектуры, *486*
Отказ, *500*
Открытая область, *606*
Отношение
  включения, *388*
  зависимости, *300*
  обобщения, *390*; *392*
  расширения, *391*
Оценка удобства использования системы, *580*
Ошибка, *500*

## П

Пакет, *373*; *469*; *584*
Параметризированный класс, *586*
Пассивная
  инициализация, *355*; *499*
  материализация, *526*; *545*
Переменная экземпляра, *606*
Переход, *432*; *606*
План
  итерации, *55*
  разработки, *567*
Поведение системы, *140*
Повреждение, *500*
Повторное использование, *581*
Повышение устойчивости пакета, *470*
Пограничный объект, *248*
Подкласс, *397*
Подлежащий постоянному хранению объект, *605*
Подтип, *606*
Полиморфизм, *333*; *606*
Полиморфная операция, *606*
Получатель, *606*
Понятие, *606*
Понятие-спецификация, *159*
Постепенная настройка процесса разработки, *569*
Посткомпилятор, *488*
Постоянно хранимый объект, *524*
Постусловие, *82*; *193*; *607*
Правило
  100%, *400*
  Is-a, *400*
Предиктивное планирование, *575*
Предметная область, *607*
Представление
  архитектуры, *490*
  процессов, *490*
  развертывания, *478*
Предусловие, *82*; *193*; *607*

Прецедент, *34*; *73*; *75*; *390*; *572*
базовый, *94*
конкретный, *94*
методы выделения, *89*
основной успешный сценарий, *83*
расширение, *83*
реализация, *254*
системный, *100*
специальные требования, *84*
типа черный ящик, *76*
форматы, *77*
Примечание, *265*
Принцип OCP, *345*
Проверка
архитектурной концепции, *492*
качества, *572*
Программирование
на основе тестирования, *316*
с предварительным тестированием, *57*
Программный класс, *163*; *257*
Проектирование, *36*; *50*; *607*
архитектуры, *440*
базы данных, *36*
на основе данных, *340*; *353*
на основе описаний, *200*
Производный
атрибут, *188*
класс, *605*
Простой
атрибут, *182*
тип, *183*
Прототип, *67*
Профиль UML, *527*
Прямое отображение, *530*
Прямой объект, *342*

## Р

Развернутый прецедент, *77*
Раздел, *458*
Разделение концептуального класса, *401*
Раздутый контроллер, *249*
Разработка
на основе прецедентов, *560*
под управлением прецедентов, *100*
Ракурс модели, *162*
Ранжирование требований, *560*
Распределение обязанностей, *35*; *206*; *224*
Расширение, *150*
прецедента, *83*
Реализация прецедента, *254*
Рекурсивная ассоциация, *607*
Риск, *131*; *132*; *560*; *572*
Роль, *172*; *297*; *416*; *607*

## С

Свободный прецедент, *77*
Связывание, *607*
Связь, *213*; *607*
Сжатый прецедент, *77*
Сигнал, *503*
Сигнатура, *199*
Символ, *150*
Системная операция, *244*
Системное событие, *244*
Системные свойства, *118*
Системный интерфейс, *192*
Словарь
предметной области, *149*
терминов, *107*; *121*
Сложный объект, *217*
Служба O-R-преобразования, *525*
Событие, *431*; *607*
внешнее, *436*
внутреннее, *436*
временное, *436*
Совместная агрегация, *413*
Содержание, *150*
Сообщение, *214*; *607*
create, *214*; *295*
порядковый номер, *215*
условное, *216*; *220*
Составная ассоциация, *419*
Состояние, *431*; *607*
Спецификатор, *419*
Спецификация, *199*
CORBA, *507*
Средство отслеживания требований, *566*
Стандарт

JavaPOS, *511*
UnifiedPOS, *511*
Статическое представление, *39*
Степень
внутреннего связывания пакетов, *468*
связанности, *236*
Стереотип, *97*; *333*; *584*
Стратегия разрешения конфликтов, *363*
Структурный подход, *151*
Суперкласс, *397*; *605*
Сценарий, *75*
реализации качественных требований, *479*

## Т

Тематическая карта, *483*
Техническое описание, *483*
Точка
вариации, *344*
эволюции, *344*; *486*
Требование, *50*; *69*
категории, *70*
функциональное, *71*
Трехуровневая архитектура, *461*
Тяжелый процесс, *54*

## У

Укрупненный план проекта, *55*
Унифицированный процесс, *34*; *43*; *49*; *100*
RUP, *43*
UP, *43*
дисциплины, *50*
фазы, *49*
Управление
конфигурацией, *573*
на основе рисков, *571*
Управляющий объект, *248*
Уровень, *458*
архитектурный, *458*
данных, *462*
интерфейса, *461*
логики приложения, *462*
Условное сообщение, *216*; *220*

## Ф

Фаза развития, *129*; *573*
Фасадный объект сеансов, *450*
Функциональная точка, *578*

## Ч

Чистые данные, *607*

## Ш

Шаблон, *34*; *207*; *213*; *225*; *441*; *607*
Cache Management, *539*
Centralized Error Logging, *505*
Convert Exceptions, *502*
Database Broker, *529*
Diagnostic Logger, *506*
Do It Myself, *516*
Domain Object Factory, *472*
Error Dialog, *506*
Extreme Programming, *574*
Layers, *252*; *488*
Model-View Separation, *463*
Name The Problem Not The Thrower, *503*
Object Identifier, *528*
Representing Objects as Tables, *527*
SCRUM, *574*
State, *199*
анализа, *152*
архитектурный, *440*
Layers, *441*
проектирования, *441*
распределения обязанностей, *224*
Шаблон GoF, *347*
Abstract Factory, *512*
Adapter, *337*; *348*; *496*
Command, *252*; *545*
Composite, *362*
Facade, *371*; *449*
Factory, *352*; *496*
Mediator, *339*
Observer, *374*

Proxy, *507*
Publish-Subscribe, *375*
Singleton, *290*; *353*
State, *540*
Strategy, *357*
Template Method, *532*
Virtual Proxy, *545*
Шаблон GRASP, *226*; *257*; *331*
Controller, *243*; *259*; *451*; *591*
Creator, *233*; *260*; *591*
Don't Talk to Strangers, *342*
High Cohesion, *239*; *591*
Indirection, *338*; *350*; *591*
Information Expert, *228*; *262*; *591*
Low Coupling, *236*; *455*; *591*
Polymorphism, *332*; *591*
Protected Variations, *339*; *350*; *455*; *591*
Pure Fabrication, *252*; *335*; *591*

## Э

Экземпляр, *607*
прецедента, *75*
Экстремальное программирование, *57*
Элементарный бизнес-процесс, *86*

*Научно-популярное издание*

**Крэг Ларман**

# Применение UML и шаблонов проектирования. 2-е издание

Литературный редактор *Е.П. Перестюк*
Верстка *О.В. Линник*
Обложка *Е.П. Дынник*
Рисунки *А.Ю. Шелестов*
Корректоры *Л.А. Гордиенко, Т.А. Корзун, О.В. Мишутина*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 06.04.04. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 50,3. Уч.-изд. л. 35.
Доп. тираж 2000 экз. Заказ № 2300.

Отпечатано с фотоформ в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.