АНАЛИЗ И ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ С ПОМОЩЬЮ UML 2.0

Title_Java_Sch.indd 1 30.05.2008 13:33:51

REQUIREMENTS ANALYSIS AND SYSTEM DESIGN

Third edition

LESZEK A. MACIASZEK



ADDISON-WESLEY

An imprint of Pearson Eduaction
Harlow, England • London • New York • Boston • San Francisco • Toronto • Sydney • Singapore • Hong Kong
Tokyo • Seoul • Taipei • New Delhi • Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

Title_Java_Sch.indd 2 30.05.2008 13:33:51

АНАЛИЗ И ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ С ПОМОЩЬЮ UML 2.0

Третье издание

ЛЕШЕК А. МАЦЯШЕК



Москва • Санкт-Петербург • Киев
2008

Title_Java_Sch.indd 3 30.05.2008 13:33:52

ББК 32.973.26-018.2.75
М36
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция канд. физ.-мат. наук *Д.А. Клюшина*

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

Мацяшек, Лешек А.

М36 Анализ и проектирование информационных систем с помощью UML 2.0, 3-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2008. — 816 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1430-9 (рус.)

Книга представляет собой новое издание популярного учебника Лешека Мацяшека по объектно-ориентированной разработке информационных систем. В книге подробно описаны методы анализа и проектирования промышленных информационных систем с использованием языка UML. Отличительной особенностью книги является обилие учебных примеров, упражнений, контрольных вопросов и многовариантных тестов. Уникальный характер книги обусловлен оптимальным сочетанием практического опыта и теоретических представлений.

Книга будет полезна системным аналитикам и архитекторам, программистам, преподавателям и студентам высших учебных заведений, а также всем специалистам по информационным технологиям.

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 Addison-Wesley UK.

Authorized translation from the English language edition published by Pearson Education Limited, Copyright © Pearson Education Limited 2001, 2005, 2007, 2007

The right of Leszek A. Maciaszek to be identified as author of this work has been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a licence permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2008

ISBN 978-5-8459-1430-9 (рус.)
ISBN 978-0-321-44036-5 (англ.)

© Издательский дом "Вильямс", 2008
© Pearson Education Limited 2001, 2005, 2007, 2007

Оглавление

Предисловие 25

Краткий обзор 34

Глава 1. Процесс разработки программного обеспечения 37

Глава 2. Определение требований 127

Глава 3. Основы визуального моделирования 199

Глава 4. Спецификация требований 259

Глава 5. Переход от анализа к проектированию 355

Глава 6. Процесс разработки программного обеспечения 429

Глава 7. Проектирование графического пользовательского интерфейса 523

Глава 8. Персистентность и проектирование баз данных 597

Глава 9. Управление качеством и изменениями 659

Глава 10. Систематизация и закрепление учебного материала 699

Приложение А. Основы объектной технологии 751

Библиография 793

Предметный указатель 803

req_analysis_system design.indb 5 05.06.2008 14:42:54

Содержание

Предисловие **25**

Краткое содержание книги **25**

Отличительные особенности книги **26**

Для кого предназначена книга 28

Структура книги 28

Изменения, внесенные во второе издание **29**

Изменения, внесенные в третье издание **30**

Вспомогательные материалы **31**

Обратная связь **32**

Благодарности **32**

Краткий обзор **34**

Глава 1. Процесс разработки программного обеспечения **37**

1.1. Характер процесса разработки программного обеспечения **38**

1.1.1. Инварианты разработки программного обеспечения 39

1.1.2. Второстепенные трудности разработки программного обеспечения 42

1.1.2.1. Участники проекта 42

1.1.2.2. Процесс 43

1.1.2.2.1. Итеративный и поступательный процесс 44

1.1.2.2.2. Модель технологической зрелости 45
1.1.2.2.3. Стандарт ISO 9000 47
1.1.2.2.4. Библиотека ITIL 48
1.1.2.2.5. Стандарт COBIT 50
1.1.2.3. Моделирование 52
1.1.2.3.1. UML 53
1.1.2.3.2. CASE-средства и совершенствование процесса 54
1.1.3. Разработка или интеграция? 55
1.2. Планирование систем 56
1.2.1. Подход SWOT 58
1.2.2. Подход VCM 60
1.2.3. Подход BPR 62
1.2.4. Подход ISA 64
1.3. Системы для трех уровней управления 65
1.3.1. Системы обработки транзакций 67
1.3.2. Системы аналитической обработки данных 68
1.3.3. Системы обработки знаний 70
1.4. Этапы жизненного цикла программного обеспечения 71
1.4.1. Подходы к разработке программного обеспечения 72
1.4.1.1. Структурный подход 73
1.4.1.2. Объектно-ориентированный подход 74
1.4.2. Этапы жизненного цикла 76
1.4.2.1. Бизнес-анализ 77
1.4.2.1.1. Этап установления требований 78
1.4.2.1.2. Этап спецификации требований 79
1.4.2.2. Проектирование систем 79
1.4.2.2.1. Этап архитектурного проектирования 80
1.4.2.2.2. Этап детализированного проектирования 81
1.4.2.3. Этап реализации 82
1.4.2.4. Этап интеграции и развертывания 82
1.4.2.5. Этап эксплуатации и сопровождения 83
1.4.3. Действия, выполняемые на протяжении всего жизненного цикла 84
1.4.3.1. Планирование проекта 85
1.4.3.2. Показатели 86
1.4.3.3. Тестирование 87

1.5. Модели и методы разработки программного обеспечения 89
1.5.1. Спиральная модель 90
1.5.2. Унифицированный процесс RUP 92
1.5.3. Архитектура, управляемая моделями 93
1.5.4. Ускоренная разработка программного обеспечения 95
1.5.5. Аспектно-ориентированная разработка программного обеспечения 97
1.6. Учебные примеры 100
1.6.1. "Зачисление в университет" 101
1.6.2. "Магазин видеокассет" 102
1.6.3. Управление взаимоотношениями с заказчиками 103
1.6.4. Прямой маркетинг по телефону 104
1.6.5. "Затраты на рекламу" 105
1.6.6. "Регистрация времени" 106
1.6.7. "Конвертация валют" 107
Резюме 108
Ключевые термины 110
Многовариантные тесты 113
Вопросы 115
Ответы на контрольные вопросы 117
Ответы к многовариантным тестам 119
Ответы на вопросы с нечетными номерами 119

Глава 2. Определение требований 127

2.1. Переход от бизнес-процессов к концепции решения 128
2.1.1. Моделирование иерархии процессов 129
2.1.1.1. Процессы и декомпозиция процессов 130
2.1.1.2. Диаграмма иерархии процессов 130
2.1.2. Моделирование бизнес-процессов 132
2.1.2.1. Потоковые и связующие объекты, дорожки и артефакты 132
2.1.2.2. Диаграмма бизнес-процессов 135
2.1.3. Выработка концепции решения 137
2.1.3.1. Процесс выработки концепции системы 137

2.1.3.2. Стратегии реализации и мандатная архитектура 139

2.2. Определение требований 141

2.2.1. Системные требования 143

2.2.1.1. Функциональные требования 143

2.2.1.2. Нефункциональные требования 144

2.2.2. Традиционные методы выявления требований 145

2.2.2.1. Интервьюирование заказчиков и экспертов в проблемной области 146

2.2.2.2. Анкетирование 148

2.2.2.3. Наблюдение 149

2.2.2.4. Изучение документов и программных систем 150

2.2.3. Современные методы выявления требований 151

2.2.3.1. Создание прототипов 151

2.2.3.2. "Мозговой штурм" 152

2.2.3.3. Совместная разработка приложений (метод JAD) 153

2.2.3.4. Быстрая разработка приложений (метод RAD) 155

2.3. Согласование и оценка требований 156

2.3.1. Требования, выходящие за рамки проекта 157

2.3.2. Матрица зависимости требований 157

2.3.3. Требования — риски и приоритеты 158

2.4. Управление требованиями 159

2.4.1. Требования — идентификация и классификация 160

2.4.2. Иерархии требований 161

2.4.3. Управление изменениями 161

2.4.4. Трассировка требований 162

2.5. Бизнес-модель требований 163

2.5.1. Модель границ системы 164

2.5.2. Модель бизнес-прецедентов использования 166

2.5.3. Бизнес-глоссарий 170

2.5.4. Модель бизнес-классов 171

2.6. Техническое задание 175

2.6.1. Шаблоны документа 175

2.6.2. Предварительные замечания к проекту 175

2.6.3. Системные сервисы 177

2.6.4. Системные ограничения 177
2.6.5. Проектные вопросы 178
2.6.6. Приложения 179
Резюме 180
Ключевые термины 181
Многовариантные тесты 184
Вопросы 185
Упражнения. Затраты на рекламу 186
Упражнения. Регистрация времени 187
Ответы на контрольные вопросы 188
Ответы к многовариантным тестам 189
Ответы на вопросы с нечетными номерами 190
Объяснение упражнений. Затраты на рекламу 193

Глава 3. Основы визуального моделирования 199

3.1. Ракурс прецедентов использования 201
3.1.1. Действующие лица 202
3.1.2. Прецеденты использования 203
3.1.3. Диаграммы прецедентов использования 204
3.1.4. Документирование прецедентов использования 206
3.2. Ракурс деятельности 208
3.2.1. Действия 209
3.2.2. Диаграммы деятельности 210
3.3. Ракурс структуры 213
3.3.1. Классы 214
3.3.2. Атрибуты 216
3.3.3. Ассоциации 218
3.3.4. Агрегация 219
3.3.5. Обобщение 220
3.3.6. Диаграммы классов 220
3.4. Ракурс взаимодействий 223
3.4.1. Диаграммы последовательностей 223

3.4.2. Диаграммы коммуникации 226
3.4.3. Методы классов 227
3.5. Ракурс конечных автоматов 229
3.5.1. Состояния и переходы 229
3.5.2. Диаграммы конечных автоматов 231
3.6. Ракурс реализации 233
3.6.1. Подсистемы и пакеты 233
3.6.2. Компоненты и диаграммы компонентов 235
3.6.3. Узлы и диаграммы развертывания 237
Резюме 238
Ключевые термины 240
Многовариантные тесты 242
Вопросы 243
Упражнения 244
Упражнения. Магазин видеокассет 246
Ответы на контрольные вопросы 248
Ответы к многовариантным тестам 249
Ответы на вопросы с нечетными номерами 249
Объяснение упражнений с нечетными номерами 251
Объяснение упражнений с нечетными номерами. Магазин видеокассет 254

Глава 4. Спецификация требований 259

4.1. Архитектурные прерогативы 261
4.1.1. Модель–представление–контроллер 261
4.1.2. Архитектура Core J2EE 263
4.1.3. Презентация–контроллер–компонент–посредник–сущность–ресурс 265
4.1.3.1. Уровни архитектуры PCBMER 266
4.1.3.2. Принципы PCBMER 268
4.2. Спецификация состояний 270
4.2.1. Моделирование классов 271

4.2.1.1. Выявление классов 271
4.2.1.2. Спецификация классов 280
4.2.2. Моделирование ассоциаций 290
4.2.2.1. Выявление ассоциаций 291
4.2.2.2. Спецификация ассоциаций 292
4.2.2.3. Пример спецификации ассоциации 293
4.2.3. Моделирование отношений агрегации и композиции 295
4.2.3.1. Выявление агрегаций и композиций 296
4.2.3.2. Спецификация агрегаций и композиций 297
4.2.3.3. Пример спецификации агрегации и композиции 297
4.2.4. Моделирование отношений обобщения 299
4.2.4.1. Выявление обобщений 300
4.2.4.2. Спецификация обобщений 300
4.2.4.3. Пример спецификации обобщений 300
4.2.5. Моделирование интерфейсов 302
4.2.5.1. Выявление интерфейсов 302
4.2.5.2. Спецификация интерфейсов 303
4.2.5.3. Примеры спецификации интерфейсов 303
4.2.6. Моделирование объектов 304
4.2.6.1. Спецификация объектов 305
4.2.6.2. Пример спецификации объектов 305
4.3. Спецификация поведения 306
4.3.1. Моделирование прецедентов использования 307
4.3.1.1. Спецификация прецедентов 308
4.3.1.2. Пример спецификации прецедентов использования 309
4.3.2. Моделирование деятельности 314
4.3.2.1. Выявление действий 316
4.3.2.2. Спецификация действий 316
4.3.2.3. Пример спецификации видов деятельности 316
4.3.3. Моделирование взаимодействий 318
4.3.3.1. Выявление последовательностей сообщений 318
4.3.2.2. Спецификация последовательностей сообщений 318
4.3.3.3. Пример спецификации последовательностей 319
4.3.4. Моделирование открытых интерфейсов 322
4.3.4.1. Выявление операций классов 322

4.3.4.2. Спецификация операций классов 323
4.3.4.3. Пример спецификации операций классов 323
4.4. Спецификации изменения состояний 325
4.4.1. Моделирование состояний объектов 326
4.4.1.1. Выявление состояний объектов 326
4.4.1.2. Спецификация состояний объектов 326
4.4.1.3. Пример спецификации диаграммы конечных автоматов 327
Резюме 329
Ключевые термины 329
Многовариантные тесты 332
Вопросы 333
Упражнения. Магазин видеокассет 336
Упражнения. Управление взаимоотношениями с заказчиками 338
Упражнения. Зачисление в университет 338
Ответы на контрольные вопросы 340
Ответы к многовариантным тестам 341
Ответы на вопросы с нечетными номерами 341
Объяснение упражнений. Зачисление в университет 349

Глава 5. Переход от анализа к проектированию 355

5.1. Углубленное моделирование классов 356
5.1.1. Механизмы расширения 357
5.1.1.1. Стереотипы 357
5.1.1.2. Комментарии и ограничения 358
5.1.1.3. Примечания и дескрипторы 361
5.1.2. Видимость и инкапсуляция 362
5.1.2.1. Защищенная видимость 363
5.1.2.2. Видимость унаследованных свойств классов 365
5.1.2.3. Видимость в пакетах и дружественных классах 366
5.1.3. Производная информация 370
5.1.3.1. Производный атрибут 370

5.1.3.2. Производная ассоциация 370
5.1.4. Квалифицированная ассоциация 371
5.1.5. Ассоциативный или материализованный класс 372
5.1.5.1. Модель с ассоциативным классом 373
5.1.5.2. Модель, использующая материализованный класс 374
5.2. Углубленное моделирование обобщения и наследования 376
5.2.1. Обобщение и заменимость 376
5.2.2. Наследование или инкапсуляция 377
5.2.3. Наследование интерфейса 377
5.2.4. Наследование реализации 379
5.2.4.1. Правильный способ использования наследования реализации — наследование посредством расширения 379
5.2.4.2. Проблематичный способ использования наследования реализации — наследование посредством ограничения 380
5.2.4.3. Неверный способ использования наследования реализации — удобное наследование 381
5.2.4.4. Недостатки наследования реализации 382
5.2.4.4.1. Изменчивый базовый класс 383
5.2.4.4.2. Замещение, нисходящие и восходящие вызовы 384
5.2.4.4.3. Множественное наследование реализации 387
5.3. Углубленное моделирование агрегации и делегирования 388
5.3.1. Расширение семантики агрегации 389
5.3.1.1. Агрегация ExclusiveOwns 389
5.3.1.2. Агрегация Owns 390
5.3.1.3. Агрегация Has 390
5.3.1.4. Агрегация Member 391
5.3.2. Агрегация как альтернатива обобщению 392
5.3.2.1. Делегирование и системы-прототипы 393
5.3.2.2. Сравнение делегирования или наследования 393
5.3.3. Агрегация и холоны — интеллектуальное орудие 394
5.4. Углубленное моделирование взаимодействий 396
5.4.1. Линии жизни и сообщения 397
5.4.1.1. Учет базовой технологии 399
5.4.1.2. Визуализация информации о технологии в моделях взаимодействия 400
5.4.2. Фрагменты 404

5.4.3. Использование взаимодействия 406
Резюме 408
Ключевые термины 409
Многовариантные тесты 410
Вопросы 411
Упражнения 412
Упражнение. Регистрация времени 413
Упражнение. Затраты на рекламу 415
Ответы на контрольные вопросы 416
Ответы к многовариантным тестам 417
Ответы на вопросы с нечетными номерами 417
Объяснение упражнений с нечетными номерами 422
Объяснение упражнений. Регистрация времени 425

Глава 6. Процесс разработки программного обеспечения 429

6.1. Распределенная физическая архитектура 431
6.1.1. Одноранговая архитектура 432
6.1.2. Ярусная архитектура 433
6.1.3. Архитектура, ориентированная на базы данных 435
6.2. Многоуровневая логическая структура 437
6.2.1. Архитектурная сложность 438
6.2.1.1. Пространственная когнитивная сложность 439
6.2.1.2. Структурная сложность 440
6.2.1.2.1. Структурная сложность сетей 440
6.2.1.2.2. Структурная сложность иерархий 441
6.2.2. Архитектурные шаблоны 444
6.2.2.1. Шаблон Фасад 445
6.2.2.2. Абстрактная фабрика 448
6.2.2.3. Цепочка обязанностей 451
6.2.2.4. Наблюдатель 453
6.2.2.5. Посредник 458
6.3. Архитектурное моделирование 462

6.3.1. Пакеты 462
6.3.2. Компоненты 464
6.3.2.1. Сравнение компонентов и пакетов 465
6.3.2.2. Сравнение компонентов с классами и интерфейсами 467
6.3.3. Узлы 468
6.4. Принципы разработки и повторного использования программ 470
6.4.1. Связность и связанность классов 470
6.4.1.1. Виды связанности классов 472
6.4.1.2. Закон Деметера 472
6.4.1.3. Методы доступа и бессмысленные классы 473
6.4.1.4. Динамическая классификация и связность классов со смешанными экземплярами 478
6.4.2. Стратегия повторного использования 482
6.4.2.1. Повторное использование инструментальных средств 482
6.4.2.2. Повторное использование каркасов 483
6.4.2.3. Повторное использование шаблонов 484
6.5. Моделирование кооперации 485
6.5.1. Кооперация 485
6.5.2. Композитная структура 487
6.5.3. Переход от прецедента использования к композитной кооперации 489
6.5.4. Переход от кооперации к взаимодействию 494
6.5.5. Переход от взаимодействия к композитной структуре 499
Резюме 500
Ключевые термины 502
Многовариантные тесты 504
Вопросы 505
Упражнения. Магазин видеокассет 507
Упражнения. Затраты на рекламу 510
Ответы на контрольные вопросы 513
Ответы к многовариантным тестам 514
Ответы на вопросы с нечетными номерами 514

Объяснение упражнений. Затраты на рекламу **518**

Глава 7. Проектирование графического пользовательского интерфейса 523

7.1. Принципы проектирования графического пользовательского интерфейса **525**

7.1.1. Переход от прототипа графического пользовательского интерфейса к его реализации 526

7.1.2. Руководящие принципы проектирования интерфейса, ориентированного на пользователя 529

7.1.2.1. Ориентация на пользователя 530

7.1.2.2. Согласованность 531

7.1.2.3. Индивидуализация и настройка 532

7.1.2.4. Толерантность 532

7.1.2.5. Обратная связь 533

7.1.2.6. Эстетичность и удобство 533

7.2. Проектирование оконного интерфейса **534**

7.2.1. Главные окна 535

7.2.1.1. Окно просмотра строк 537

7.2.1.2. Окно просмотра деревьев 538

7.2.2. Вторичное окно 539

7.2.2.1. Диалоговые окна 540

7.2.2.2. Папка с вкладками 541

7.2.2.3. Выпадающий список 542

7.2.2.4. Окна сообщения 543

7.2.3. Меню и панели инструментов 543

7.2.4. Кнопки и другие средства управления 545

7.3. Проектирование Web-интерфейса **546**

7.3.1. Технология реализации Web-приложений 548

7.3.2. Проектирование содержания 550

7.3.2.1. Web-сайт для континуума Web-приложений 551

7.3.2.2. Формы 552

7.3.3. Проектирование навигации 557

7.3.3.1. Меню и ссылки 558

7.3.3.2. Навигационные цепочки и панели 559

7.3.3.3. Кнопки 560
7.3.4. Использование моделей графических пользовательских интерфейсов для Web-проектирования 561
7.3.4.1. Дилемма MVC 562
7.3.4.2. Технология Struts 565
7.4. Моделирование навигации в графическом пользовательском интерфейсе 569
7.4.1. Раскадровка работы пользователя 569
7.4.2. Моделирование элементов UX 572
7.4.3. Функциональная кооперация UX 573
7.4.4. Структурная кооперация UX 577
Резюме 578
Ключевые термины 579
Многовариантные тесты 580
Вопросы 581
Упражнения. Управление взаимоотношениями с заказчиками 582
Упражнения. Прямой маркетинг по телефону 584
Ответы на контрольные вопросы 588
Ответы к многовариантным тестам 588
Ответы на вопросы с нечетными номерами 589
Объяснение упражнений. Управление взаимоотношениями с заказчиками 592

Глава 8. Персистентность и проектирование баз данных 597

8.1. Бизнес-объекты и персистентность 599
8.1.1. Инварианты разработки программного обеспечения 599
8.1.2. Уровни моделей данных 600
8.1.3. Интеграция приложений и моделирование баз данных 601
8.1.4. Отображение объектов в базу данных 603
8.2. Модель реляционной базы данных 604
8.2.1. Столбцы, домены и правила 605
8.2.2. Реляционные таблицы 606
8.2.3. Ссылочная целостность 608
8.2.4. Триггеры 610

8.2.5. Хранимые процедуры 612
8.2.6. Реляционные представления 614
8.2.7. Нормальные формы 615
8.3. Объектно-реляционное отображение 617
8.3.1. Отображение классов сущностей 617
8.3.2. Отображение отношений ассоциации 618
8.3.3. Отображение отношений агрегации 620
8.3.4. Отображение отношений обобщения 622
8.4. Шаблоны управления персистентными объектами 625
8.4.1. Поиск персистентных объектов 626
8.4.2. Загрузка персистентных объектов 628
8.4.3. Выгрузка персистентных объектов 630
8.5. Проектирование доступа к базам данных и транзакций 631
8.5.1. Уровни программирования на SQL 631
8.5.2. Проектирование транзакций 634
8.5.2.1. Короткие транзакции 634
8.5.2.1.1. Пессимистическое управление параллельным выполнением операций 635
8.5.2.1.2. Уровни изолированности 635
8.5.2.1.3. Автоматическое восстановление 636
8.5.2.1.4. Программируемое восстановление 638
8.5.2.1.5. Проектирование хранимых процедур и триггеров 639
8.5.2.2. Проектирование хранимых процедур и триггеров 640
Резюме 641
Ключевые термины 642
Многовариантные тесты 644
Вопросы 645
Упражнения. Управление взаимоотношениями с заказчиками 646
Упражнения. Прямой маркетинг по телефону 647
Ответы на контрольные вопросы 648
Ответы к многовариантным тестам 649
Ответы на вопросы с нечетными номерами 649

Объяснение упражнений. Управление взаимоотношениями с заказчиками **652**

Глава 9. Управление качеством и изменениями **659**

9.1. Управление качеством **661**

9.1.1. Поддержка качества 662

9.1.1.1. Контрольные списки, экспертиза и аудит 662

9.1.1.2. Разработка посредством тестирования 664

9.1.2. Контроль качества 666

9.1.2.1. Концепции и методы тестирования 667

9.1.2.2. Тестирование системных функций 671

9.1.2.3. Тестирование системных ограничений 672

9.1.2.3.1. Тестирование пользовательского интерфейса 673

9.1.2.3.2. Тестирование баз данных 674

9.1.2.3.3. Тестирование авторизации 675

9.1.2.3.4. Тестирование других ограничений 676

9.2. Управление изменениями **677**

9.2.1. Инструменты управления запросами на изменения 678

9.2.1.1. Отправка запроса на изменение 680

9.2.1.1. Отслеживание запросов на изменение 681

9.2.2. Трассируемость 682

9.2.2.2. Связи системных возможностей с тестовыми прецедентами и тестовыми требованиями 683

9.2.2.2. Связи планов тестирования с тестовыми спецификациями и тестовыми требованиями 684

9.2.2.2. Связи от UML-диаграмм с документами и требованиями 686

9.2.2.4. Связи между требованиями прецедентов использования и тестовыми требованиями 687

9.2.2.5. Связи между тестовыми требованиями и дефектами 687

9.2.2.6. Связи между требованиями прецедентов использования и усовершенствованиями 689

Резюме **690**

Ключевые термины **690**

Многовариантные тесты **692**

Вопросы **692**

Ответы на контрольные вопросы **693**

Ответы к многовариантным тестам **694**

Ответы на вопросы с нечетными номерами **694**

Глава 10. Систематизация и закрепление учебного материала **699**

10.1. Моделирование прецедентов использования **700**

10.1.1. Действующие лица 701

10.1.2. Прецеденты использования 702

10.1.3. Диаграммы прецедентов использования 704

10.1.4. Документирование диаграмм прецедентов использования 705

10.2. Моделирование деятельности **707**

10.2.1. Действия 707

10.2.2. Диаграмма деятельности 708

10.3. Моделирование классов **709**

10.3.1. Классы 709

10.3.2. Атрибуты 712

10.3.3. Отношения ассоциации 713

10.3.4. Отношения агрегации 714

10.3.5. Отношения обобщения 714

10.3.6. Диаграмма классов 715

10.4. Моделирование взаимодействия **716**

10.4.1. Диаграмма последовательностей 716

10.4.2. Диаграмма коммуникации 718

10.5. Моделирование конечных автоматов **721**

10.5.1. Состояния и переходы 721

10.5.2. Состояния и переходы 722

10.6. Модели реализации **723**

10.6.1. Подсистемы 723

10.6.2. Пакеты 725

10.6.3. Компоненты 727

10.6.4. Примечания 729

10.7. Разработка кооперации объектов **731**

Этап 22. Интернет-магазин 733
10.7.1. Проектные спецификации прецедентов использования 733
10.7.2. Создание прототипов пользовательского интерфейса 736
10.7.3. Диаграмма последовательностей 737
10.7.4. Проектная диаграмма классов 739
10.8. Проектирование навигации по окнам 740
10.8.1. Использование элементов UX 740
10.8.2. Функциональная кооперация UX 741
10.8.3. Структурная кооперация UX 742
10.9. Проектирование баз данных 743
10.9.1. Объектно-реляционное отображение 743
10.9.2. Проектирование ссылочной целостности 746
Резюме 746
Упражнения. Интернет-магазин 747

Приложение А. Основы объектной технологии 751

А.1. Аналогия с объектами реального мира 752
А.2. Объект-экземпляр 753
А.2.1. Объектная нотация 753
А.2.2. Как кооперируются объекты 754
А.2.3. Индивидуальность и коммуникация между объектами 756
А.2.3.1. Персистентная связь 756
А.2.3.2. Временная связь 757
А.2.3.3. Передача сообщений 758
А.3. Класс 759
А.3.1. Атрибуты 760
А.3.1.1. Тип атрибута, обозначающий класс 760
А.3.1.2. Видимость атрибутов 761
А.3.2. Операции 762
А.3.2.1. Операции в рамках кооперации объектов 762
А.3.2.2. Видимость операций 763
А.3.3. Объекты-классы 763
А.4. Переменные, методы и конструкторы 765

A.5. Ассоциации **766**
A.5.1. Степень ассоциации 768
A.5.2. Кратность ассоциации 768
A.5.3. Ассоциативная связь и объем ассоциации 770
A.5.4. Ассоциативный класс 770
A.6. Агрегация и композиция **771**
A.6.1. Скрытая ссылка 772
A.6.2. Внутренний класс 774
A.6.3. Делегирование 776
A.7. Обобщение и наследование **777**
A.7.1. Полиморфизм 779
A.7.2. Замещение и перегрузка 780
A.7.3. Множественное наследование 781
A.7.4. Множественная классификация 781
A.7.5. Динамическая классификация 782
A.8. Абстрактный класс **783**
A.9. Интерфейс **784**
A.9.1. Интерфейс и абстрактный класс 784
A.9.2. Реализация интерфейса 785
A.9.3. Использование интерфейса 787
Резюме **788**
Вопросы **789**
Ответы на вопросы с нечетными номерами **790**

Библиография **793**

Предметный указатель **803**

Предисловие

Краткое содержание книги

Разработка *информационной системы* — от начальной фазы до развертывания — состоит из трех последовательных и поступательных этапов: анализа, проектирования и реализации. В книге описываются методы и приемы, используемые на этапах *анализа* и *проектирования*. Вопросы, связанные с реализацией, включая примеры программ, затрагиваются только в той мере, в которой они необходимы на этапе проектирования. Управление качеством и изменениями рассматриваются отдельно в главе 9.

Настоящая книга посвящена *объектно-ориентированной разработке программного обеспечения*. Для описания артефактов моделирования используется язык UML (Unified Modeling Language — унифицированный язык моделирования). Основное внимание уделяется проблемам конструирования с помощью языка UML, который используется на протяжении всего жизненного цикла проекта. Аналитики, проектировщики и программисты используют один и тот же язык, хотя иногда могут пользоваться диалектами (профилями) языка.

На первых порах объектная технология применялась для разработки графических интерфейсов пользователя (graphical user interface — GUI) и была направлена на ускорение разработки новых систем и выполнения программ. В этой книге основной упор сделан на применение объектной технологии к разработке *промышленных информационных систем* (enterprise information systems — EIS). Основными проблемами на этом пути являются большие объемы и сложные структуры данных, совместный доступ к информации со стороны многих пользователей, обработка транзакций, изменяющиеся требования и т.д. Основным преимуществом применения объектной технологии для разработки промышленных информационных систем является повышение их *адаптивности*, которое выражается в понятности, легкости сопровождения и масштабируемости.

Разработка промышленной информационной системы представляет собой *крупномасштабный анализ и проектирование*. Ни один проект промышленной информационной системы не может быть успешным без выполнения строго определенного процесса разработки и понимания архитектуры программного обеспечения, лежащей в основе системы. Таким образом, разработка таких систем носит крупномасштабный, объектно-ориентированный, последовательный и поступательный характер.

В книге предложен детализированный подход к анализу и проектированию промышленных информационных систем с использованием языка UML. Эта разработка сводится к решению следующих задач.

- Анализ и моделирование бизнес-процессов.
- Преодоление сложности моделей крупных систем.
- Улучшение архитектуры программного обеспечения.
- Повышение адаптивности системы.
- Решение выявленных проблем, связанных в проектированием.
- Ясность графического пользовательского интерфейса.
- Управление качеством, изменениями и др.

Отличительные особенности книги

Книге присущи многие отличительные черты, сочетание которых делает ее уникальной. В основу изложения положено *обучение на примерах*. Основной текст содержит обсуждение семи учебных примеров, а также главу, предназначенную для закрепления знаний. *Учебные примеры* (case studies) относятся к семи предметным областям. Они отличаются уникальными особенностями и методическими аспектами. В книге рассмотрены следующие учебные примеры: “Зачисление в университет” (University Enrollment), “Магазин видеокассет” (Video Store), “Управление взаимоотношениями с заказчиками” (Contact Management), “Прямой маркетинг по телефону” (Telemarketing), “Расходы на рекламу” (Advertising Expenditures), “Регистрация времени” (Time Logging), “Конвертация валют” (Currency Converter). Для *закрепления знаний* используется пример “Интернет-магазин” (Online Shopping).

Чтобы облегчить читателю самостоятельное изучение материала, учебные примеры и методические материалы излагаются в виде *вопросов и ответов*, а также *задач и решений*. Примеры из практики дополняются и расширяются с помощью вопросов и примеров, приведенных в конце главы. Некоторые вопросы и задачи сопровождаются ответами и решениями. Кроме того, каждая глава содержит *резюмирующие вопросы* (review quizzes) и *многовариантные тесты* (multiple-choice tests).

В книге рассматриваются принципы, методы и приемы правильного анализа и проектирования. Особое внимание уделяется этапу проектирования, причем он не рассматривается как простое преобразование результатов анализа. Книга

описывает трудности и сложности, связанные с разработкой крупных объектно-ориентированных систем. Во многих отношениях книга содержит свежий взгляд на крупномасштабное, последовательное и поступательное проектирование больших систем, а также на возможности и ограничения, присущие инструментам и методам, предназначенным для создания крупных систем программного обеспечения.

Уникальный характер книги объясняется гармоничным сочетанием практических аспектов и теоретических представлений. Основной предпосылкой является стремление избежать лишней сложности, не утратив необходимой строгости. Книга носит практический характер. В нее не вошли вопросы, не имеющие прямого отношения к промышленным информационным системам либо представляющие исключительно академический интерес.

Книга отражает последние достижения в сфере информационной технологии. Она использует стандарт в области визуального моделирования систем — язык UML — и посвящена новейшим разработкам в технологиях Web и баз данных. В ней отражена тенденция перехода от "толстых клиентов" (т.е. мощных настольных компьютеров) к серверным вычислениям. Рассматриваемые в книге принципы анализа и проектирования в равной мере применимы как клиент-серверным решениям, так и к распределенным приложениям, созданным с использованием компонентного подхода.

Разработка программного обеспечения не сводится к однозначным решениям типа "черное–белое", "истина–ложь" или "нуль–один". Источником эффективных программных решений служат идеи толковых бизнес-аналитиков, системных проектировщиков и программистов, а не бездумное применение алгоритмов. Книга знакомит читателей с потенциальными трудностями, которые не могут быть полностью разрешены в рамках предлагаемого подхода. Благодаря этому можно надеяться, что читатели будут обдуманно применять приобретенные знания и не будут рассчитывать на то, что использование предлагаемого подхода не потребует от них усилий.

Итак, можно отметить следующие отличительные черты, свойственные книге.

- Книга устанавливает связь теории с практикой, используя для этого реалистичные задачи и ограничения, на которые следует обращать внимание, применяя предлагаемый подход.
- Особое внимание уделяется этапу проектирования. Проектирование *не* рассматривается как простое преобразование результатов анализа, при этом описываются трудности и сложности, присущие разработке крупномасштабных промышленных систем.
- Книга изобилует нетривиальными примерами, вопросами, упражнениями и многовариантными тестами. Большинство из них сопровождаются ответами и решениями, приведенными в приложениях на Web-сайте, предназначенном для преподавателей. Руководство для преподавателей не написано вдогонку — оно создавалось одновременно с книгой и тщательно продумано.

Для кого предназначена книга

С учетом растущей потребности в университетских курсах, более приближенных к практике, книга предназначена для студентов и профессионалов. Подготовка такой книги оказалась непростой задачей, но есть основания надеяться, что ее удалось успешно решить. Для того чтобы полученные знания оставались актуальными в течение более продолжительного времени, аспекты реализации, относящиеся к разработке программного обеспечения, рассматриваются независимо от конкретных программных продуктов, представленных сегодня на рынке (хотя при описании примеров используются коммерческие CASE-средства).

Книгу можно использовать как учебник при изложении курса по компьютерным наукам и информационным системам. Поскольку книга содержит как темы, относящиеся к системному моделированию, так и вопросы проектирования пользовательского интерфейса и баз данных, она может оказаться полезной для студентов, изучающих системный анализ, проектирование систем и программного обеспечения, базы данных и объектную технологию, а также выполняющих полномасштабную разработку программного обеспечения, в которых студенты проходят весь жизненный цикл проекта: начиная с определения требований и заканчивая реализацией пользовательского интерфейса и баз данных. Учебник рассчитан на один семестр, но может использоваться и на протяжении двух семестров, один из которых посвящен анализу, а другой — проектированию.

Профессионалы найдут в книге описание реальных проблем. Источником большинства постановок задач, примеров и упражнений послужил опыт работы автора в качестве консультанта. Мы предупреждаем читателя о потенциальных трудностях и ограничениях, связанных с предлагаемым подходом. По нашему мнению, наибольшую пользу от книги могут получить специалисты по анализу деловой активности и системные аналитики, проектировщики, программисты, системные архитекторы, руководители и менеджеры проектов, специалисты по анализу решений, тестированию и подготовке технической документации, а также инструкторы, работающие в компаниях.

Структура книги

Книга полностью охватывает вопросы объектно-ориентированного анализа и проектирования информационных систем. Материал представлен в порядке, соответствующем современным процессам разработки. Книга состоит из десяти глав и приложения, посвященного основам объектной технологии. Содержание книги в одинаковой степени освещает вопросы анализа и проектирования.

Книга должна быть доступной для читателей с различным базовым уровнем подготовки. Последняя глава книги призвана закрепить знания читателей. Это предусмотрено требованиями, предъявляемыми к университетским курсам. Кроме того, принцип закрепления пройденного материала является краеугольным камнем любого продолжительного обучения.

Изменения, внесенные во второе издание

Несмотря на то что первое издание книги было хорошо принято читателями и переведено на несколько языков, всегда существует возможность для ее улучшения, особенно в части, посвященной такой технологичной области, как разработка систем. Основные изменения (и, как мы надеемся, улучшения), внесенные во второе издание, заключаются в следующем.

- В конце главы приведены ответы на вопросы и решения задач, имеющих нечетные номера. Это делает книгу более удобной для самообразования. Остальные ответы и решения доступны только для преподавателей на специальном Web-сайте вместе с другими учебными материалами.
- Учебные руководства, ранее включенные в главы 2 и 6, были выделены в отдельную главу 10, глубоко переработаны и сильно расширены, чтобы их можно было использовать для закрепления изложенного материала.
- Главы 2 и 3 переставлены местами. Это объясняется тем, что анализ требований (изложенный теперь в главе 2) не требует предварительных знаний об объекте и объектном моделировании (изложенных теперь в главе 3).
- Глава 3 называется "Объекты и объектное моделирование", что отражает переход от описания объектов в целом к описанию на основе языка UML и примеров.
- Глава 4 "Спецификация требований" дополнена объяснением важности системной архитектуры и сопровождения разрабатываемой системы. Соответственно, в эту главу включен материал, посвященный архитектуре, который обсуждается в последующих главах.
- Изменено содержание главы 6. В основном это выражается в перераспределении материала по другим главам. В частности, учебное руководство перенесено в главу 10, а вместо него включены разделы из главы 9, посвященные системной архитектуре. Обсуждение системной архитектуры значительно расширено. В итоге глава теперь называется "Архитектура системы и проектирование программ".
- Глава 7 "Разработка пользовательского интерфейса" модифицирована и расширена. В частности, в нынешнем изложении некоторых принципов разработки пользовательского интерфейса мы воспользовались преимуществами, предоставленными библиотекой Java Swing. Ранее модели навигации по окнам были основаны на диаграммах деятельности UML. В данном издании они заменены новым инструментом языка UML — хранилищем пользовательского опыта (UX storyboard).

- Глава 8 теперь называется "Персистентность и проектирование базы данных". Она расширена за счет обсуждения вопросов, связанных с проектированием персистентных объектов, включая шаблоны для управления персистентностью в рамках принятой архитектурной среды. Прежнее подробное обсуждение объектных и объектно-ориентированных баз данных удалено (из-за продолжающегося доминирования реляционных баз данных в промышленных информационных системах и трудности этого материала). Сэкономленное место заполнено разделами из главы 9, например темой, посвященной управлению транзакциями.
- После переноса материала из главы 9 в предыдущие главы прежняя глава 10 стала новой главой 9. Эта глава также была расширена. В частности, в нее был добавлен раздел, посвященный разработке посредством тестирования.

Изменения, внесенные в третье издание

Третье издание примерно на четверть больше второго. В каждую главу добавлены контрольные вопросы (с ответами), многовариантные тесты (также с ответами) и определение основных терминов. Изменения были внесены практически в каждую главу (за исключением главы 8, посвященной базам данных). Кроме того, улучшена структура некоторых глав. Перечислим основные изменения, внесенные в третье издание.

- Глава 1 пересмотрена полностью: добавлены новые разделы, посвященные средам управления решениями (ITIL и COBIT), аспектно-ориентированному проектированию и системной интеграции (чтобы подчеркнуть тот факт, что современные промышленные системы редко разрабатываются как самостоятельные приложения и, как правило, являются частью более крупных проектов).
- Глава 2 также изменена. В нее добавлен раздел, посвященный моделированию иерархических процессов, деловой активности и визуализации решений.
- Изменена глава 3. Раздел 3.1 из второго издания перемещен в приложение, а раздел 3.2 изменен и расширен с учетом новейших усовершенствований, внесенных в стандарт языка UML.
- Расширена глава 4. Теперь она учитывает изменения в языке UML и лучше описывает архитектурные решения и среды.
- В главу 5 добавлен раздел, посвященный усовершенствованному моделированию действий.
- Глава 6 расширена и модифицирована. Теперь в разделе 6.2, посвященном логической архитектуре, описываются вопросы, связанные с архитектурной сложностью. Кроме того, переработан раздел 6.5, посвященный шаблонам

и моделированию взаимодействия. Это позволило учесть изменения стандарта языка UML.

- В главу 7 включено обсуждение проектирования графического пользовательского интерфейса в среде Windows.
- Изменена структура главы 9.
- В главе 10 нашли отражение новшества, внесенные в стандарт языка UML.
- Как указывалось ранее, в книгу включено новое приложение "Основы объектной технологии".

Вспомогательные материалы

На Web-сайте, посвященном книге, в распоряжение читателей предоставляется обширный комплект вспомогательных материалов. Большинство Web-документов открыто для свободного доступа, но некоторые из них защищены паролем в интересах преподавателей, планирующих применять книгу в учебном процессе. Информационная страница книги одновременно поддерживается на следующих Web-сайтах.

```
http://www.booksites.net/maciaszek
http://www.comp.mq.edu.au/books/rasd3ed
```

Комплект вспомогательных материалов включает следующие Web-документы. Руководство преподавателя, которое содержит следующие материалы.

- Лекционные слайды в формате Microsoft Power Point.
- Ответы и решения, содержащие комментарии и решения для всех вопросов и упражнений, помещенных в конце каждой главы.
- Задания и проекты для студентов, сопровождаемые ответами и решениями.
- Экзаменационные задания, сопровождаемые ответами и решениями.

Материалы для студентов.

- Лекционные слайды в формате Acrobat Read.
- Файлы моделей, содержащие решения задач из учебных примеров, руководства и остальных глав (представленные с помощью средств Rational Rose, Magic Draw, Enterprise Architect, PowerDesigner и Visio Professional).
- Список опечаток, ошибок и упущений, встретившихся в книге.

Обратная связь

Ваши комментарии, исправления, предложения по улучшению и прочие пожелания будут приняты с большой благодарностью. Пожалуйста, направляйте вашу корреспонденцию автору.

Leszek A. Maciaszek
Department of Computing
Macquarie University
Sydney, NSW 2109
Australia
Электронный адрес: leszek@ics.mq.edu.au
Web-сайт: http://www.comp.mq.edu.au/~leszek/
Телефон: +61 2 98509519
Факс: +61 2 98509551
Почтовый адрес: North Ryde, Herring Road, Bld. E6A, Room 319

Благодарности

Я не смог бы написать эту книгу без общения с моими друзьями, коллегами, студентами, ведущими специалистами отрасли и многими другими людьми, которые вольно или невольно оказали влияние на формирование моих знаний о проблемной области. Я действительно в большом долгу перед ними. Попытка перечислить всех могла бы показаться неблагоразумной и невыполнимой, поэтому я благодарю всех.

Я хотел бы выразить благодарность лекторам и студентам университетов, а также профессионалам-практикам, взявшим на себя труд указать мне на недостатки, найденные в первом и втором изданиях книги. Эта обратная связь легла в основу третьего издания.

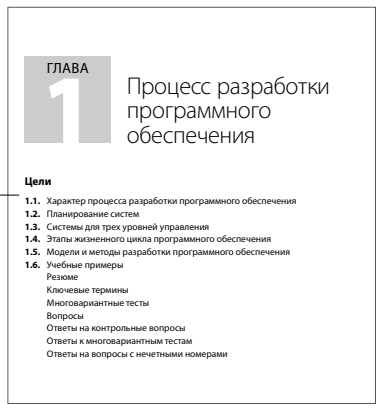
Я благодарен моим редакторам из издательства Pearson Education в Лондоне. Выражаю особую признательность Кейт Менсфилд (Keith Mansfield), бывшей рецензентом первого и второго изданий этой книги, а также другой моей книги — *Practical Software Engineering*. После того как Кейт покинула издательство Pearson, ее работу продолжил Симон Пламтри (Simon Plumtree). Он провел тщательный анализ конкурентоспособности предложенной книги, и в результате издательство приняло решение выпускать новое издание каждые несколько лет.

Во время работы над книгой *Practical Software Engineering* и вторым изданием книги *Requirements Analysis and System Design*, а также в ходе подготовки ее третьего издания я контактировал с издательством Pearson через Оуэна Найта (Owen

Knight). Его ответственность и открытость обеспечивали мне постоянную поддержку. Благодарю Оуэна от всего сердца.

Редактирование и подготовка книги к печати — удивительно сложный процесс, в котором участвует множество людей, неизвестных автору. Спасибо им всем. Особая ответственность в этом процессе лежит на техническом редакторе. Я благодарю Джорджину Кларк-Мазо (Georgina Clark-Mazo) за прекрасную работу, выполненную ею при подготовке третьего издания моей книги.

Краткий обзор



ГЛАВА 1

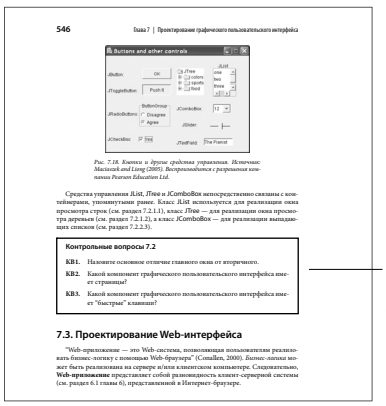
Процесс разработки программного обеспечения

Цель главы задает ее тему

Цели

1.1. Характер процесса разработки программного обеспечения
1.2. Планирование систем
1.3. Системы для трех уровней управления
1.4. Этапы жизненного цикла программного обеспечения
1.5. Модели и методы разработки программного обеспечения
1.6. Учебные примеры

Резюме
Ключевые термины
Многовариантные тесты
Вопросы
Ответы на контрольные вопросы
Ответы к многовариантным тестам
Ответы на вопросы с нечетными номерами



546 Глава 7 | Проектирование графического пользовательского интерфейса

Рис. 7.18. Кнопки и другие средства управления. Источник: Macintosh and Luing (2005). Воспроизводится с разрешения компании Pearson Education Ltd.

Средства управления *JList*, *JTree* и *JComboBox* непосредственно связаны с контейнерами, упомянутыми ранее. Класс *JList* используется для реализации окна просмотра строк (см. раздел 7.2.1.1), класс *JTree* — для реализации окна просмотра деревьев (см. раздел 7.2.1.2), а класс *JComboBox* — для реализации выпадающих списков (см. раздел 7.2.2.3).

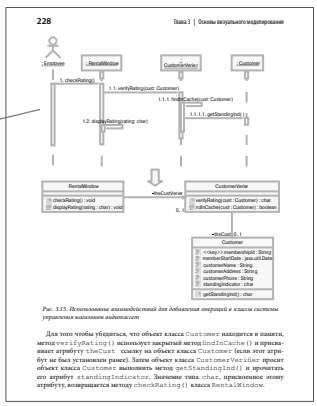
Контрольные вопросы 7.2

КВ1. Назовите основное отличие главного окна от вторичного.
КВ2. Какой компонент графического пользовательского интерфейса имеет странницы?
КВ3. Какой компонент графического пользовательского интерфейса имеет "быстрые" клавиши?

Контрольные вопросы в каждой главе позволяют проверить знания

7.3. Проектирование Web-интерфейса

"Web-приложение — это Web-система, позволяющая пользователям реализовать бизнес-логику с помощью Web-браузера" (Conallen, 2000). *Бизнес-логика* может быть реализована на сервере и/или клиентском компьютере. Следовательно, **Web-приложение** представляет собой разновидность клиент-серверной системы (см. раздел 6.1 главы 6), представленной в Интернет-браузере.

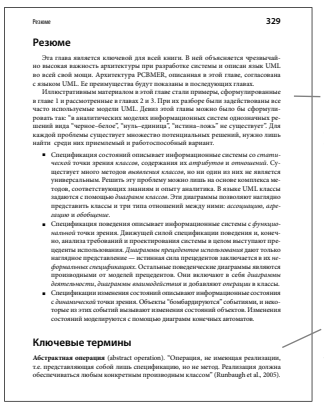


228 Глава 3 | Основы визуального моделирования

Широко используются диаграммы UML

Рис. 3.15. Использование взаимодействий для добавления операций в классы системы управления наказанием заключенных

Для того чтобы убедиться, что объект класса `Customer` находится в памяти, метод `verifyRating()` использует закрытый метод `findInCache()` и присваивает атрибуту `theCust` ссылку на объект класса `Customer` (если этот атрибут не был установлен ранее). Затем объект класса `CustomerVerifier` просит объект класса `Customer` выполнить метод `getStandingInd()` и прочитать его атрибут `standingIndicator`. Значение типа `char`, присвоенное этому атрибуту, возвращается методу `checkRating()` класса `RentalWindow`.



Резюме 329

Резюме

Эта глава является ключевой для всей книги. В ней объясняется чрезвычайно высокая важность архитектуры при разработке системы и описан язык UML во всей своей мощи. Архитектура PCBMER, описанная в этой главе, согласована с языком UML. Ее преимущества будут показаны в последующих главах.

Иллюстративным материалом в этой главе стали примеры, сформулированные в главе 1 и рассмотренные в главах 2 и 3. При их разборе были задействованы все часто используемые модели UML. Домен этой главы можно было бы сформулировать так: "в аналитических моделях информационных систем однозначных решений вида "черное–белое", "нуль–единица", "истина–ложь" не существует". Для каждой проблемы существует множество потенциальных решений, нужно лишь найти среди них применимый и работоспособный вариант.

- Спецификация состояний описывает информационные системы со *статической* точки зрения классов, содержащих их *атрибутов* и *отношений*. Существует много методов *выявления* классов, но ни один из них не является универсальным. Решить эту проблему можно лишь на основе комплекса методов, соответствующих знаниям и опыту аналитика. В языке UML классы задаются с помощью *диаграмм классов*. Эти диаграммы позволяют наглядно представить классы и три типа отношений между ними: *ассоциацию*, *агрегацию* и *обобщение*.
- Спецификация поведения описывает информационные системы с *функциональной* точки зрения. Движущей силой спецификации поведения и, конечно, анализа требований и проектирования системы в целом выступают прецеденты использования. *Диаграммы прецедентов использования* дают только наглядное представление — истинная сила прецедентов заключается в их *неформальном спецификации*. Остальные поведенческие диаграммы являются производными от моделей прецедентов. Они включают в себя *диаграммы деятельности*, *диаграммы взаимодействия* и добавляют *операции* в классы.
- Спецификации изменения состояний описывают информационные состояния с *динамической* точки зрения. Объекты "бомбардируются" *событиями*, и некоторые из этих событий вызывают изменения состояний объектов. Изменения состояний моделируются с помощью диаграмм конечных автоматов.

Резюме в конце каждой главы помогает прояснить основные факты

Ключевые термины

Абстрактная операция (abstract operation). "Операция, не имеющая реализации, т.е. представляющая собой лишь спецификацию, но не метод. Реализация должна обеспечиваться любым конкретным производным классом" (Rumbaugh et al., 2005).

В конце каждой главы перечисляются ключевые термины

В книге приводятся семь учебных примеров, иллюстрирующих концепции, лежащие в основе реальных приложений

Основные понятия иллюстрируются многочисленными снимками экрана, сопровождаемыми комментариями

На рис. 7.22 показана Web-форма для конвертации валют, предоставляемая пользователям как услуга Национального австралийского банка (National Australia Bank — NAB) в соответствии с требованиями, перечисленными в примере 7.4. Эта форма состоит из трех полей и двух командных кнопок. Два поля из этих полей дополнены выпадающим списком для выбора соответствующих опций.

Пример 7.5. Конвертация валют

Вернитесь к примеру 7.4. Разработайте Web-форму для вывода результатов пересчета. Эта форма должна содержать обменный курс, сумму денег для конвертации и дату обмена. Кроме того, она должна позволять пользователю вернуться к форме для ввода данных, чтобы выполнить следующий обмен.

Форма для вывода результатов пересчета конвертором валют банка NAB в соответствии с требованиями, перечисленными в примере 7.5, показана на рис. 7.23.



Рис. 7.23. Прототип формы для вывода результатов работы системы конвертации валют

Web-форма работает в браузере, и некоторые стандартные пункты меню и функции его панели инструментов могут оказывать на нее влияние, если не ограничить их средствами Web-приложения. Приложение должно ограничить следующие функции (Fowler and Stanwick, 2004):

- Кнопки для перехода вперед и назад по Web-страницам, если этот переход не контролируется приложением (такой переход может привести к потере данных, введенных пользователем, или прервать бизнес-операцию).
- Кнопки и пункты меню, закрывающие окно просмотра и приводящие к неконтролируемому выходу из приложения.

Формулировка ограничения (constraint statement). Ограничение, которому должна удовлетворять система.
Формулировка сервиса (service statement). Требование, определяющее услугу, ожидаемую от системы.
Функциональное требование (functional requirement). См. *Формулировка сервиса*.

Многовариантные тесты

МТ1. Что называется атомарным процессом в системе BPMN?
- **а.** Действие.
- **б.** Задача.
- **в.** Событие.
- **г.** Задание.

МТ2. Что называется потоком сообщений в системе BPMN?
- **а.** Потоковый объект.
- **б.** Дорожка.
- **в.** Артефакт.
- **г.** Коннектор.

МТ3. Что является элементом моделирования, определяющим бизнес-ценность функционального свойства в рамках концепции решения?
- **а.** Бизнес-прецедент использования.
- **б.** Прецедент решения.
- **в.** Прецедент возможности.
- **г.** Бизнес-прецедент.

МТ4. Какой метод выявления требований работает с понятием провозможности?
- **а.** Анкета.
- **б.** "Мозговой штурм".
- **в.** JAD.
- **г.** Ни один из перечисленных.

МТ5. Какой метод выявления требований работает с понятием триггерного вопроса?
- **а.** Анкета.
- **б.** RAD.
- **в.** JAD.
- **г.** Ни один из перечисленных.

Многовариантные тесты в конце каждой главы помогают закрепить пройденный материал

Общие вопросы в конце каждой главы служат для повторения материала

Упражнения в конце каждой главы уточняют учебные примеры

- Как правило, контракты заключаются на один год и ежегодно пересматриваются с учетом регистрационных записей о клиентах. Регистрационные записи позволяют определить, насколько часто клиент обращался в компанию и как часто у него возникали проблемы на протяжении года. Информация, содержащаяся в регистрационных записях, позволяет проводить переговоры быстро, в течение одного совещания.
- Счета клиентам выставляются раз в месяц. Содержание счета зависит от контракта и дополнительных услуг, не предусмотренных в контракте.
- Если контракт истек или счет не оплачен, то обслуживание клиентов может быть приостановлено, пока контракт не будет пересмотрен.
- Счета генерируются и рассылаются в конце каждого месяца. Оплата отслеживается. Детали счетов и оплаты записываются в отчеты.

Постройте диаграмму бизнес-процесса "Контракты и дебиторская задолженность".

У2. Рассмотрите задачу 5 из раздела 1.6.5. Сосредоточьте внимание только на бизнес-процессах. В частности, проанализируйте разделы "Управление контактами" и "Контракты и дебиторская задолженность" (см. рис. 2.2 в разделе 2.1.1.2).

Постройте контекстную диаграмму системы. Опишите модель.

У3. Рассмотрите задачу 5 из раздела 1.6.5. Сосредоточьте внимание только на бизнес-процессах. В частности, проанализируйте разделы "Управление контактами" и "Контракты и дебиторская задолженность" (см. рис. 2.2 в разделе 2.1.1.2).

Постройте диаграмму бизнес-прецедентов использования системы. Опишите модель.

У4. Рассмотрите задачу 5 из раздела 1.6.5.

Напишите бизнес-сценарий. Введите в него только понятия, имеющие отношение к оценке затрат на рекламу.

У5. Рассмотрите задачу 5 из раздела 1.6.5.

Постройте диаграмму бизнес-классов для системы оценки затрат на рекламу. Используйте бизнес-сценарий, разработанный в упражнении А4, и сконцентрируйте лишь бизнес-классы, имеющие отношение к оценке затрат на рекламу. Опишите модель.

Упражнения. Регистрация времени

У1. Рассмотрите задачу 6 из раздела 1.6.6.

Постройте высокоуровневую диаграмму прецедентов использования для системы регистрации времени. Опишите эту модель.

ГЛАВА 10

Систематизация и закрепление учебного материала

Цели

10.1. Моделирование прецедентов использования
10.2. Моделирование деятельности
10.3. Моделирование классов
10.4. Моделирование взаимодействия
10.5. Моделирование конечных автоматов
10.6. Модели реализации
10.7. Разработка кооперации объектов
10.8. Проектирование навигации по окнам
10.9. Проектирование баз данных
Резюме
Упражнения. Интернет-магазин

Руководство по обучению и закреплению материала подводит итог обучения

ГЛАВА 1

Процесс разработки программного обеспечения

Цели

1.1. Характер процесса разработки программного обеспечения
1.2. Планирование систем
1.3. Системы для трех уровней управления
1.4. Этапы жизненного цикла программного обеспечения
1.5. Модели и методы разработки программного обеспечения
1.6. Учебные примеры
Резюме
Ключевые термины
Многовариантные тесты
Вопросы
Ответы на контрольные вопросы
Ответы к многовариантным тестам
Ответы на вопросы с нечетными номерами

Цели

Данная глава посвящена изложению — на уровне обзора — стратегических вопросов, касающихся процесса разработки программного обеспечения. Она вводит читателя в процессы и подходы, лежащие в основе современной разработки программного обеспечения.

Прочитав эту главу, читатели будут

- понимать природу разработки программного обеспечения и его социальное значение, а также то, что бизнес-системы нельзя основывать лишь на строгих научно-технических принципах;
- знать стандарты разработки программного обеспечения (CMM, ISO 9000, ITIL и COBIT);
- разбираться в подходах к стратегическому планированию систем (SWOT, VCM, BPR, ISA), гарантирующих соответствие проектов информационных систем поставленным целям;
- понимать, что информационные системы очень разнообразны и зависят от уровня управления, которые они обслуживают, и преимуществ, которые они предоставляют;
- понимать сущность различий между структурным и объектно-ориентированным подходами к разработке программного обеспечения;
- знать стадии жизненного цикла разработки программного обеспечения и соответствующих видов деятельности;
- владеть информацией о современных моделях и методах разработки программного обеспечения (спиральной модели, унифицированном процессе компании IBM Rational, модельной архитектуре, ускоренной разработке, аспектно-ориентированной разработке);
- понимать смысл семи учебных примеров, описанных в книге.

1.1. Характер процесса разработки программного обеспечения

Литература, посвященная вопросам управления информационными системами, изобилует примерами провалившихся проектов, превышения сроков и бюджетов, ошибочных решений, систем, не пригодных для сопровождения, и т.д. Даже если широко цитируемый отчет Standish Chaos (утверждающий, что 70% программных проектов оказываются неудачными) преувеличивает их количество

(Glass, 2005), то нет никаких сомнений в том, что многие "успешные" системы (т.е. оплаченные и доставленные заказчику) страдают от недостатка надежности, производительности, безопасности, легкости сопровождения и других проблем.

Для того чтобы разобраться с этими проблемами, необходимо понять характер процесса разработки программного обеспечения. В своей ставшей уже классической статье Брукс (Brooks, 1987) определяет как основные (inherent), так и второстепенные (accidental) свойства, характерные для проектирования программного обеспечения. *Сущность* этого процесса кроется в трудностях, присущих самому программному обеспечению. Эти трудности можно только осознать — их нельзя ни преодолеть за счет какого-либо технологического прорыва, ни устранить взмахом волшебной палочки. Согласно статье Брукса, сущность проектирования программного обеспечения является следствием его изначальной сложности (complexity), согласованности (conformity), изменчивости (changeability) и невидимости (invisibility).

"Существенные трудности" создания программного обеспечения являются инвариантами процесса его разработки. Эти инварианты просто констатируют тот факт, что программное обеспечение является продуктом творческого акта — ремесла или искусства в том смысле, что эта деятельность совершается скорее ремесленником, чем художником. При обычном положении дел программное обеспечение не является результатом повторяющегося акта производства.

Поняв сущность инвариантов разработки программного обеспечения, перейдем к понятию *второстепенных трудностей* (accidental difficulties), присущих проектированию программного обеспечения и обусловленных человеческим фактором. Эти трудности можно разделить на три категории.

1. Участники проекта.
2. Процесс.
3. Моделирование.

1.1.1. Инварианты разработки программного обеспечения

Имеются существенные особенности программного обеспечения, не предполагающие вмешательства человека. Эти свойства остаются инвариантами во всех проектах, связанных с проектированием программного обеспечения. Их необходимо учитывать, но невозможно устранить. Основная задача при разработке программного обеспечения — не выпускать инварианты из-под контроля и не допускать их негативного влияния на проект.

Программное обеспечение по самой своей природе является *сложным*. В современных системах сложность зависит от размера программ (выраженного в виде количества строк в их коде), а также от взаимодействия компонентов.

Сложность программного обеспечения зависит от природы предметной области, для которой оно разрабатывается. Оказывается, предметные области, свя-

занные с интенсивными вычислениями, приводят к менее сложным системам, чем предметные области, для которых характерны большие объемы данных. К приложениям второго типа относятся системы электронной коммерции, которым посвящена данная книга. Такие системы обрабатывают огромные объемы данных и бизнес-правил, часто противоречивых или двусмысленных. Создание программного обеспечения, способного справиться с этими данными, правилами и особыми условиями, является трудным по своей природе, т.е. инвариантно.

Эти трудности усугубляются тремя другими существенными свойствами программного обеспечения, указанными Бруксом: *согласованностью*, *изменчивостью* и *невидимостью*. Прикладное программное обеспечение должно соответствовать конкретному аппаратному и программному обеспечению, для которого оно разрабатывается, а также допускать интеграцию с существующими информационными системами. Поскольку бизнес-процессы и требования постоянно изменяются, прикладное программное обеспечение должно допускать модификацию. Несмотря на то что результаты работы программного обеспечения являются вполне осязаемыми, код, создающий эти результаты, как правило, глубоко скрыт в недоступных фрагментах программ, в библиотеках и окружающем программном обеспечении.

Как мы уже отмечали, программное обеспечение скорее *разрабатывается*, а не *производится* (Pressman, 2005). Разумеется, невозможно отрицать, что достижения программной инженерии привнесли в практику разработки бо́льшую определенность, однако успех программного проекта гарантировать нельзя. Это проектирование программных систем отличается от проектирования традиционных инженерных дисциплин, в которых продукция (артефакты) конструируется с математической точностью, а затем создается (часто во многих вариантах) с помощью станков и конвейеров.

Однажды разработанное программное обеспечение можно скопировать (произвести) с минимальными затратами. Однако промышленные информационные системы практически никогда не воспроизводятся вновь. Каждая из таких систем уникальна и разрабатывается для конкретного предприятия. Трудности здесь возникают в проектировании, а не в производстве. Следовательно, основной вклад в стоимость программного обеспечения вносит проектирование.

Для того чтобы упростить процесс проектирования и снизить его стоимость, промышленность по производству программного обеспечения предлагает множество повторно используемых компонентов. Основная проблема в этом случае — связать воедино небольшие фрагменты решения в согласованную промышленную систему, соответствующую потребностям сложных бизнес-процессов.

Практика создания программного обеспечения способствует разработке систем из настраиваемых программных пакетов — решений на основе COTS-продуктов (*commercial of the shelf software* — готовое к использованию программное обеспечение) или ERP-систем (*enterprise resource planning system* — система планирования

корпоративных ресурсов). Однако программный пакет может обеспечить лишь функции стандартного бухгалтерского учета, производственной системы или системы управления кадрами. Эти решения необходимо адаптировать к конкретным бизнес-процессам. Для этого сначала необходимо определить бизнес-процессы и создать модели системы. Таким образом, акцент смещается от разработки "с чистого листа" к разработке путем "настройки программного обеспечения", однако в обоих вариантах природа процесса производства остается той же самой.

Для любой системы необходимо сначала создать *концептуальные конструкции* (модели) конечного решения, удовлетворяющие специфические потребности организации. После их создания функциональные возможности программного пакета настраиваются в соответствии с концептуальными конструкциями. Задачи программирования могут быть разными, но анализ требований и проектирование систем, связанные с этими конструкциями, не отличаются от процесса разработки системы с нуля. В общем, концептуальная конструкция (модель) остается неизменной, несмотря на всевозможные изменения ее представления (реализации).

Не менее важно и то, что организация вряд ли сможет найти готовый программный пакет для автоматизации ее *основных бизнес-процессов*. Например, основным бизнес-процессом в телефонной компании является обеспечение телефонной связи, а не управление кадрами или бухгалтерский учет. Следовательно, основное программное обеспечение этой системы, скорее всего, придется разрабатывать с нуля. Как отметил Шиперский (Szyperski, 1988): "Стандартные пакеты создают лишь гладкое игровое поле, а умение играть приходит совсем с другой стороны".

Конечно, в любом случае процесс разработки должен использовать преимущества *компонентной технологии* (Allen and Frost, 1998; Szyperski, 1998). **Компонент** (component) — это исполняемый модуль, реализующий четко определенные функции (сервисы) и коммуникационные протоколы (интерфейсы) взаимодействия с другими компонентами. Компоненты можно настраивать так, чтобы удовлетворить требования, предъявляемые к приложению. Наиболее влиятельными и конкурирующими друг с другом компонентными технологиями являются J2EE/EJB компании Sun и .NET компании Microsoft. Технология **SOA** (Service-Oriented Architecture) предлагает конструирование систем из **сервисов** (services), т.е. исполняемых экземпляров программ (в противоположность компонентам, которые необходимо сначала загрузить, инсталлировать, развернуть и инициализировать).

Пакеты, компоненты, сервисы и другие методы не изменяют сущности разработки программного обеспечения. В частности, принципы и задачи анализа требований и системного проектирования остаются неизменными. Конечный программный продукт может собираться из стандартных или разрабатываемых на заказ компонентов, но сам процесс "сборки" по-прежнему остается искусством. Как заметил Прессман (Pressman, 2005), у нас нет даже "запасных частей", чтобы заменить отказавшие компоненты системы.

1.1.2. Второстепенные трудности разработки программного обеспечения

Инварианты программного обеспечения определяют сущность его разработки и создают наибольшие трудности. Чрезвычайно важным фактом является то, что второстепенные аспекты проектирования программного обеспечения не увеличивают сложность и не снижают легкость дальнейшего сопровождения системы. **Легкость сопровождения** (*адаптивность*) определяется тремя свойствами системы: ясностью, легкостью сопровождения и масштабируемостью (расширяемостью) системы.

Второстепенные аспекты разработки программного обеспечения можно объяснить тем, что информационные системы носят социальный характер. Их успех или провал зависит от людей, их восприятия и особенностей эксплуатации системы, процессов, использованных при разработке системы, практики управления, приемов программирования и других факторов.

1.1.2.1. Участники проекта

Участники проекта (*stakeholders*) — это люди, заинтересованные в программном проекте. Любой человек, интересы которого так или иначе затрагиваются системой либо сам оказывающий влияние на разработку системы, является участником проекта. Участники проекта делятся на две основные группы.

- Заказчики (пользователи и владельцы системы).
- Разработчики (аналитики, проектировщики, программисты и т.д.).

В обиходе термин "пользователь" означает "заказчик". Однако следует подчеркнуть, что слово "заказчик" лучше отражает существо дела. Во-первых, заказчик — это лицо, оплачивающее разработку и принимающее решения. Во-вторых, даже если заказчик не во всем прав, разработчики не могут по собственной воле изменить или отвергнуть их требования, — любое противоречивое, невыполнимое или неправомерное требование должно быть предметом переговоров с заказчиками.

Информационные системы — это *социальные системы* (social systems). Они разрабатываются *людьми* (разработчиками) *для людей* (заказчиков). Успех программного проекта определяется социальными факторами, а роль технологии второстепенна. Есть немало примеров технически отсталых систем, которые работают и приносят пользу заказчикам. Обратное, как правило, неверно. От системы, не приносящей пользы (ожидаемой или реальной), заказчик рано или поздно откажется, независимо от того, насколько блестящей она является в техническом отношении.

Как правило, основные причины провала программных проектов можно приписать участникам проекта. С точки зрения *заказчиков*, провал проекта обусловлен рядом причин (Pfleeger, 1998):

- потребности заказчиков не поняты или не полностью зафиксированы;
- требования заказчиков изменяются слишком часто;
- заказчики не готовы выделить достаточно ресурсов для проекта;
- заказчики не желают сотрудничать с разработчиками;
- ожидания заказчиков нереалистичны;
- система оказывается бесполезной для заказчиков.

Проекты также заканчиваются неудачей потому, что *разработчики* оказываются не в состоянии справиться с поставленной задачей. В связи с увеличением сложности программного обеспечения растет понимание того, что критическим фактором разработки становятся опыт и знания разработчиков. Хорошие разработчики могут найти приемлемое решение. Высококлассные разработчики способны найти лучшее решение намного быстрее и дешевле. На эту тему существует довольно известная шутка Фреда Брукса (Brooks, 1987): "Великие проекты создаются великими разработчиками".

Мастерство и ответственность разработчиков являются факторами, вклад которых в достижение качества и продуктивности программного обеспечения трудно переоценить. Для того чтобы гарантировать успешную поставку программного обеспечения заказчику и, что более важно, обеспечить выгоды от его использования, организация-исполнитель должна следовать некоторым простым правилам (Brooks, 1987; Yourdon, 1994).

- Нанимать лучших разработчиков.
- Обеспечивать непрерывный процесс обучения и образования своих разработчиков.
- Поощрять обмен информацией и общение между разработчиками так, чтобы они стимулировали друг друга.
- Стимулировать разработчиков, устраняя препятствия и направляя их усилия на продуктивную работу.
- Создавать исключительно благоприятную рабочую атмосферу (зачастую это оказывается намного более важным, чем редкие прибавки к зарплате).
- Увязывать личные цели разработчиков со стратегией и задачами организации.
- Придавать особое значение коллективной работе.

1.1.2.2. Процесс

Процесс (process) разработки программного обеспечения определяет действия и организационные процедуры, используемые в ходе его производства и сопровождения. Цель процесса — обеспечить управление коллективом разработчиков и усилить сотрудничество между его членами, чтобы поставить заказчику качественную продукцию и эффективно поддерживать его работу в дальнейшем. На модель процесса возлагаются следующие функции.

- Определение порядка выполнения действий.
- Определение состава и времени поставки артефактов
- Распределение заданий и артефактов между разработчиками.
- Формулировка критериев мониторинга проекта, определение средств измерения результатов и планирования будущих проектов.

В отличие от языков моделирования и программирования процесс разработки программного обеспечения невозможно стандартизировать. Каждая организация должна разработать свою собственную модель процесса или принять некоторый общий шаблон процесса, например шаблон *Rational Unified Process®* (RUP®) (Kruchten, 2003). Процесс разработки программного обеспечения (software process), принятый организацией, является важной частью ее общего *бизнес-процесса* и определяет ее уникальный характер и место на рынке.

Процесс разработки программного обеспечения, принятый организацией, должен быть увязан с ее развитием, культурой, социальной динамикой, знаниями и опытом разработчиков, практикой руководства, ожиданиями заказчиков, масштабами проектов и даже характером предметных областей. Поскольку все перечисленные факторы подвержены изменениям, может возникнуть необходимость внести разнообразие в ее модель процесса и предусмотреть разные варианты для каждого отдельного программного проекта. Например, если разработчики слабо владеют методами и средствами моделирования, процесс может предусматривать специальные курсы обучения.

Вероятно, самое большое влияние на процесс оказывает *масштаб проекта* (project size). В небольших проектах (в которых принимают участие не более десяти разработчиков) формальный процесс может вообще не потребоваться. Такая небольшая бригада скорее всего способна общаться и реагировать на изменения неформально. В более крупных проектах возможностей неформального общения может оказаться недостаточно и потребуется строго определенный процесс управляемой разработки.

1.1.2.2.1. Итеративный и поступательный процесс

Современные процессы разработки программного обеспечения непременно являются *итеративными* (iterative) и *поступательными* (incremental). Модели системы уточняются и преобразуются в ходе анализа, проектирования и реализации. Новые детали добавляются в проект путем **итераций**, а изменения и усовершенствования вводятся с помощью выпуска *последовательных версий* программных модулей. Это позволяет удовлетворять потребности пользователей и обеспечивать обратную связь, необходимую для продолжения разработки модулей.

Конструкции (builds) исполняемого кода поставляются по частям, каждая из которых является совершеннее предыдущей. Наращивание системы не выходит за *рамки* проекта (project's scope), очерченные набором функциональных требований. *Итерации* должны быть короткими — в течение недель, но не месяцев.

Обратная связь с заказчиком должна быть частой, а планирование — непрерывным. Кроме того, неотъемлемыми частями жизненного цикла проекта являются управление изменениями, а также регулярный сбор показателей и анализ риска, образующие основу для последующих версий.

Существует несколько вариантов описанного итеративного и поступательного процесса, описанных в работе Мацяшека и Лионга (Maciaszek and Liong, 2005).

- Спиральная модель.
- Унифицированный процесс (rational unified process — RUP).
- Архитектура, управляемая моделями (model-driven architecture — MDA).
- Ускоренный процесс разработки.
- Аспектно-ориентированная разработка программного обеспечения.

Спиральная модель (spiral model), предложенная Боэмом (Boehm, 1988), была отправной точкой для следующих трех моделей, перечисленных в списке. Процесс RUP характеризуется относительной гибкостью и создает основу для разработки системы (называемой платформой RUP), состоящую из разнообразных шаблонов документов, объяснений понятий, идей разработки и т.п. Процесс *MDA* основан на идее исполняемых спецификаций, которая заключается в генерировании программного обеспечения с помощью моделей и компонентов. *Ускоренный* процесс образует среду, в которой сотрудничество отдельных людей и коллективов играет более важную роль, чем планирование, документирование и другие формальности. *Аспектно-ориентированное проектирование* вводит ортогональную идею пересекающихся понятий (аспектов) и предполагает создание отдельных модулей программного обеспечения на основе этих понятий. Каждый из аспектов (например, безопасность, надежность или параллельность) разрабатывается в виде отдельного модуля, который затем тщательно сплетается с остальными частями системы.

Успех итеративного и поступательного процесса основывается на раннем выявлении *архитектурных модулей системы* (system's architectural modules). Модули должны быть примерно одинаковыми по размерам, обладать сильной внутренней связностью и иметь минимум взаимных пересечений (внешних связей). Важен также порядок реализации модулей. Некоторые модули невозможно реализовать в виде исполняемой версии, если они зависят от информации или результатов вычислений, производимых другими модулями, которые еще предстоит разработать. Если итеративная и поступательная разработка не подразумевает планирование и управление, то в отсутствие контроля процесс может превратиться в набор сиюминутных решений.

1.1.2.2.2. Модель технологической зрелости

Главной проблемой для любой организации, занятой производством программного обеспечения, является усовершенствование процесса разработки. Естественно, чтобы ввести усовершенствования, организация должна знать существующие

проблемы, связанные с существующим процессом. Одним из наиболее известных методов оценки и усовершенствования процессов является так называемая *модель технологической зрелости* (capability maturity model — CMM) (CMM, 1995).

Модель CMM была разработана в центре Software Engineering Institute (SEI) при Питтсбургском университете Карнеги–Меллона (Carnegie Mellon University), США. Первоначально эта модель использовалась министерством обороны США для оценки возможностей организаций в области информационных технологий, участвовавших в конкурсах на получение оборонных заказов. В настоящее время она находит широкое применение в индустрии информационных технологий как в Америке, так и за ее пределами.

По существу модель CMM — это *анкета* (questionnaire), которую заполняет организация, специализирующаяся в области информационных технологий. После анкетирования следуют верификация и аттестация процесса, по результатам которых организацию относят к одному из пяти уровней модели CMM. Чем выше уровень, тем более зрелым является процесс разработки ПО в организации. На рис. 1.1 определены уровни, даны краткие описания главных особенностей каждого из уровней и приведены основные сферы улучшения процесса, необходимые для достижения организацией более высокого уровня технологической зрелости.

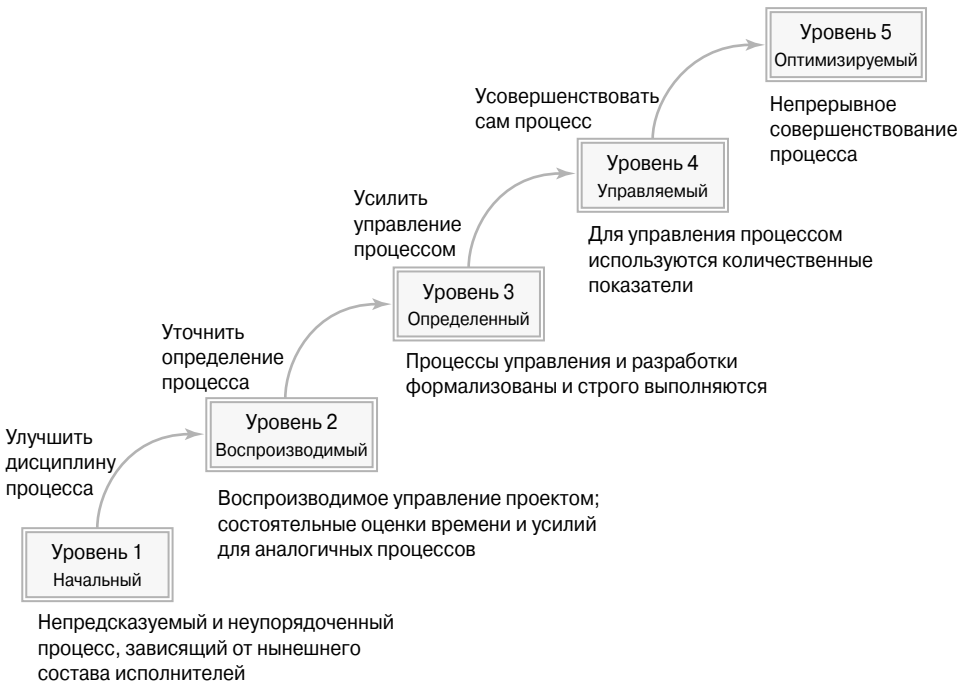


Рис. 1.1. Уровни зрелости процессов в модели CMM

Артур (Arthur, 1992) называет уровни зрелости процесса "лестницей, ведущей к совершенству в области программного обеспечения". Пять ступенек на этой лестнице соответствуют хаосу, управляемому проекту, применению методов и средств организации процесса, измерению процесса и непрерывному повышению качества. Как показывает опыт, переход с одного уровня технологической зрелости на другой занимает несколько лет. Большинство организаций находятся на уровне 1, некоторые — на уровне 2; известно очень немного организаций, находящихся на уровне 5. Приводимые ниже вопросы показывают, насколько трудновыполнима эта задача. Организация, которая стремится достичь 2-го уровня, должна добиться положительных ответов на все приведенные ниже вопросы (а также на некоторые другие) (Pfleeger, 1998).

- Имеется ли в организации канал управления и отчетности по обеспечению качества программного обеспечения, отдельный от управления проектам?
- Предусмотрена ли для каждого проекта функция управления конфигурацией программного обеспечения, затрагивающая его разработку?
- Использует ли руководство формальный процесс для анализа каждой программной разработки прежде, чем взять на себя договорные обязательства?
- Используется ли для составления календарного плана разработки программного обеспечения формальная процедура?
- Применяются ли формальные процедуры для оценки затрат на разработку программного обеспечения?
- Собирается ли статистика, касающаяся программного кода и ошибок, выявленных при тестировании?
- Есть ли в распоряжении высшего руководства механизм для регулярного анализа состояния проектов по разработке программного обеспечения?
- Используется ли определенный механизм для контроля изменений требований к программному обеспечению?

1.1.2.2.3. Стандарт ISO 9000

Кроме модели CMM, существуют и другие модели усовершенствования процесса создания программного обеспечения. Особый интерес представляет серия стандартов качества ISO 9000, разработанная Международной организацией по стандартизации (International Organization for Standardization). Эти стандарты ISO применяются к управлению *качеством* и *процессу* производства качественной продукции. Стандарты носят общий характер — они применимы для любой отрасли и всех видов бизнеса, включая разработку программного обеспечения.

В основе серии стандартов ISO 9000 лежит предположение о том, что если процесс организован надлежащим образом, то и результат процесса (товар или услуга) также будет обладать надлежащим качеством. "Цель управления качеством — сделать качество неотъемлемым свойством товара, а не проверять, насколько качественным является товар" (Schmauch, 1994).

Возвращаясь к предыдущему обсуждению особенностей процесса разработки программного обеспечения, заметим, что стандарты ISO не навязывают конкретного процесса и не уточняют его характеристик. Стандарты дают модель того, *что* должно быть достигнуто, однако не говорят о том, *как* именно должна осуществляться та или иная деятельность. Организация, которая стремится получить сертификат ISO (это также называется регистрацией), должна рассказать, что она делает; доказать, что слова у нее не расходятся с делом; и показать, чего она уже достигла (Schmauch, 1994).

Критерием, позволяющим проверить, насколько организация заслуживает сертификата ISO, может служить ее способность создать качественный товар или обеспечить качественное обслуживание даже в том случае, если заменить весь персонал. С этой целью необходимо *оформить документально и зафиксировать* все виды своей деятельности. Для каждого вида деятельности должна быть определена письменная процедура, включая действия, которые необходимо выполнить в случае нарушения процесса или жалоб клиента.

Так же как в случае модели CMM, сертификат ISO может быть официально предоставлен только после *аудита, проведенного на месте* (on-site audit) регистрационным бюро ISO. Подобные аудиты затем повторяются через определенное время на регулярной основе. Организации вынуждены проходить через подобную систему из-за давления конкурентных факторов, обусловленных требованиями со стороны потребителей, чтобы поставщики товаров и услуг были сертифицированы. Многие страны приняли стандарты ISO 9000 в качестве национальных. Особенно широко эти стандарты распространены в Европе.

1.1.2.2.4. Библиотека ITIL

С точки зрения бизнеса программное обеспечение (или информационная технология в целом) представляет собой инфраструктуру, которая быстро становится товаром. Информационные технологии являются основным источником конкурентного преимущества компаний, которым удается внедрить их раньше других, но время, в течение которого можно использовать это преимущество, намного короче, чем хотелось бы. Причины этого явления разнообразны. В частности, к ним относятся существование источников открытого кода, бесплатные лицензии на коммерческое программное обеспечение, предоставляемые образовательным учреждениям, короткие циклы итеративной и поступательной разработки программного обеспечения и др.

Со временем программное обеспечение становится просто бизнес-решением, предоставляющим определенные услуги. Сила взаимного влияния бизнеса и программного обеспечения настолько велика, что в настоящее время следует говорить о *поставке решения (услуги)*, а не просто о поставке программного обеспечения или системы. Поставляемое программное обеспечение нуждается в дальнейшем сопровождении. *Управление решением* (solution management) относится ко всем аспектам управления разработкой программного обеспечения. Этот факт лег в основу библи-

отеки инфраструктуры информационных технологий (IT Infrastructure Library — ITIL®) — широко используемой и общепринятой коллекции наилучших решений, принятых в области управления информационными технологиями (ITIL, 2004).

"Менеджеры в области информационных технологий должны координировать работу и поддерживать сотрудничество с заказчиками, для того чтобы предоставлять высококачественные информационные услуги. Для этого необходимо сократить стоимость владения (total cost of ownership — TCO) и увеличить частоту, сложность и объем изменений... Управление информационными технологиями касается производительности и эффективности использования четырех П-факторов — персонала, процессов, продуктов (инструментов и технологий) и партнеров (поставщиков, продавцов и субподрядчиков)" (ITIL, 2004).

На рис. 1.2 подход ITIL к осуществлению управления решением представлен в виде программы непрерывного улучшения качества услуг (continuous service improvement programme — CSIP). Программа CSIP начинается с определения абстрактных целей бизнеса, после чего осуществляется проверка пунктов календарного плана (поставок) и поддержка развития проекта путем консолидации достигнутых улучшений и продолжения цикла разработки.



Рис. 1.2. Подход ITIL к управлению решениями (услугами)

Абстрактные *задачи бизнеса* (business objectives) определяются в рамках стратегического планирования и моделирования. Эти задачи анализируются в контексте текущего состояния организации и должны заполнять пробелы, существующие в ее деятельности. Оценка *зрелости организации с точки зрения информационных технологий* (IT organizational maturity) предусматривает внутренние исследования, внешнее тестирование и верификацию процессов в соответствии с промышленными стандартами (ITIL, CMM и ISO 9000).

Завершив исследование пробелов в деловой активности, необходимо разработать бизнес-прецедент для программы CSIP. Этот прецедент должен описывать "быстрые победы" (если они существуют), но прежде всего он должен описывать *измеримые цели* (measureable targets), представляющие собой краткосрочные цели, согласованные со стратегическим направлением работы компании. Зная эту информацию, организация может разработать *план улучшения процесса* (process improvement plan), в котором определены масштаб, подход и процесс, а также сроки проекта.

Прогресс и эффективность программы CSIP оцениваются в сравнении с календарным планом, сроками поставки, критическими факторами успеха (critical success factors — CSF) и ключевыми индикаторами производительности (key performance indicators — KPI). Несмотря на важность включения *показателей* (measurements and metrics), непосредственно связанных с бизнесом, необходимо помнить, что улучшение бизнес-процессов зависит от качества программного обеспечения.

1.1.2.2.5. Стандарт COBIT

В отличие от библиотеки ITIL, ориентированной на операционную сторону поставки решений и управления, стандарт COBIT® (Control Objectives for Information and related Technology) относится к управлению решениями (COBIT, 2000).

ITIL, CMM и ISO 9000 — это *стандарты процессов* (process standards). Они устанавливают требования, регламентирующие, как организация должна управлять процессом, чтобы обеспечить качество продукции или услуги. Стандарт COBIT, наоборот, является *стандартом продукции* (product standard) и определяет, *что* организация должна делать, а не то, *как* она должна это делать.

Целевая аудитория стандарта COBIT не включает проектировщиков программного обеспечения. Стандарт COBIT предназначен для старших менеджеров по информационным технологиям, а также высших руководителей и аудиторов. "Благодаря высокому уровню разработки и широкому охвату материала, основанного на наилучших из существующих методов, стандарт COBIT часто называют "интегратором", объединяющим в одно целое несопоставимые приемы и, что не менее важно, помогающим связать разные информационные технологии с требованиями бизнеса" (COBIT, 2005).

Стандарт COBIT является довольно инструктивным. Он разделяет деятельность, связанную с информационными технологиями, на четыре *области* (domains).

- Планирование и организация.
- Приобретение и реализация.
- Поставка и поддержка.
- Мониторинг.

В каждой из этих областей существуют свои *задачи управления* (control objectives). Всего существует 34 задачи управления высокого уровня. В свою очередь, с этими задачами связаны *принципы аудита* (audit guideline), позволяющие сравнить информационные процессы с 318 задачами управления, рекомендованными стандартом COBIT. Эти задачи определяют цель для управления гарантиями и направления улучшения системы.

Первая область — *планирование и организация* — связана с планированием деловой активности (раздел 1.3). В ней информационные технологии рассматриваются как часть стратегического и тактического планирования. В результате планирования и организации работ следует выяснить, как лучше использовать информационные технологии для достижения поставленных бизнес-целей. Кроме того, к этой категории относится краткосрочное планирование работы информационных систем. Для этого необходимо осуществить анализ существующих систем, провести исследования, распределить ресурсы, построить информационную **архитектурную** модель, оценить направления технологического развития, системную архитектуру и стратегии перехода на новое оборудование, провести организационное распределение информационных функций, определить владельцев данных и системы, распределить персонал, составить бюджет, предназначенный для внедрения информационных технологий, установить прогноз затрат и прибыли, оценить риск, внедрить управление качеством и др.

Вторая область — *приобретение и реализация* — связана с реализацией информационной стратегии. Она определяет автоматизированные решения, способствующие достижению бизнес-целей и удовлетворению потребностей пользователей. Кроме того, выполнив эту работу, менеджер может определить, можно ли купить готовое информационное решение или его следует разработать. Деятельность, связанная с приобретением и реализацией информационных технологий, предусматривает изучение не только прикладного программного обеспечения, но и оценку технологической архитектуры. Эта часть стандарта регламентирует процедуры разработки, тестирования, инсталляции и сопровождения, а также требования, связанные с обслуживанием и обучением, и приемы управления изменениями.

Третья область — *поставка и сопровождение* — охватывает вопросы, связанные с оказанием информационных услуг, обработку **данных** прикладными системами (управление приложениями), а также процедуры поддержки информационных систем. Эта часть стандарта регламентирует уровень обслуживания, услуги субподрядчиков, вопросы безопасности систем, связь с системами ведения отчетности,

вопросы обучения и образования, консультации и рекомендации, конфигурацию систем, устранение неисправностей, управления данными, средства и операции обслуживания.

Четвертая область — *мониторинг* — связана с оценкой качества информационных технологий во времени и соответствия требованиям, предъявляемым к управлению. Эта часть стандарта регламентирует мониторинг процесса, а также оценку достижения целей и уровня удовлетворенности пользователей, адекватности механизмов внутреннего контроля, независимых гарантий безопасности, производительности системы, согласованности с законодательством, нормами профессиональной этики и т.д.

1.1.2.3. Моделирование

Участники проекта и процесс составляют две вершины "треугольника", обеспечивающего успех программного проекта. Третья вершина — это *моделирование программного обеспечения* (software modeling), рассматриваемое как моделирование артефактов программирования. **Модель** — это абстракция реальности. Реализованная и работающая система также является моделью реальности.

Подлежащие моделированию артефакты необходимо выразить (с помощью языка) и документально зафиксировать (с помощью инструментов). Разработчикам необходим *язык* для построения визуальных и других моделей, а также для обсуждения их с заказчиками и коллегами. Язык должен позволять строить модели на различных уровнях абстракции, чтобы представлять предлагаемые решения на различных уровнях детализации. Язык должен обладать мощным *визуальным компонентом* (visual component), поскольку "лучше один раз увидеть, чем сто раз услышать". Язык моделирования должен также обладать мощной *декларативной семантикой* (declarative semantics), т.е. позволять фиксировать "процедурное" значение в форме "декларативных" предложений. Иначе говоря, мы должны иметь возможность сказать, что именно необходимо сделать, а не распространяться о том, как это должно быть сделано.

Кроме того, разработчикам необходимы *инструменты* (tools) для автоматизированного проектирования или более сложные интегрированные среды для создания программ, например CASE-средства (computer assisted software engineering — CASE). CASE-средства позволяют хранить и получать модели через центральный репозиторий, а также манипулировать этими моделями на экране компьютера в графическом и текстовом режимах. В идеале репозиторий должен обеспечивать многим пользователям одновременный доступ к моделям многих пользователей (разработчиков). Приведем перечень типичных функций *CASE-репозитория* (CASE-repository).

- Координация доступа к моделям.
- Помощь в организации взаимодействия разработчиков.
- Хранение нескольких версий моделей.

- Идентификация различий между версиями.
- Возможность совместного использования одних и тех же концептов в различных моделях.
- Проверка непротиворечивости и целостности моделей.
- Генерация проектных отчетов и документов.
- Генерация структур данных и программного кода (конструирование программного обеспечения).
- Генерация моделей по существующей реализации (реконструкция программного обеспечения) и т.д.

Следует заметить, что зачастую программа, сгенерированная с помощью CASE-средств, представляет собой на самом деле всего лишь скелет кода — вычислительный алгоритм, который необходимо дорабатывать.

1.1.2.3.1. UML

"Язык UML (unified modeling language — унифицированный язык моделирования) — это универсальный визуальный язык моделирования, используемый для спецификации, визуализации, конструирования и документирования артефактов программной системы. Он описывает решения и представления о конструируемых системах и используется для понимания, анализа, просмотра, конфигурирования, поддержки и контроля информации о системе. Предназначен для использования в сочетании с любыми методами разработки, на любых стадиях **жизненного цикла** проекта, в любых предметных областях и в любых средах" (Rumbaugh et al., 2005).

Язык UML был разработан компанией Rational Software Corporation для унификации лучших свойств, которыми обладали более ранние методы и способы нотации. В 1997 году организация OMG (Object Management Group — группа управления объектами) признала его в качестве стандартного языка моделирования. С тех пор UML получил дальнейшее развитие и широкое признание в отрасли информационных технологий.

Язык UML не зависит от применяемого процесса разработки программного обеспечения, хотя позже компания Rational предложила процесс, соответствующий этому языку, под названием *Rational Unified Process* (*RUP*: Krushten, 2003). Совершенно очевидно, что процесс, в котором в качестве базового языка принят UML, должен поддерживать *объектно-ориентированный подход* к созданию ПО. Язык UML не подходит для устаревших структурных подходов, результатом которых являются системы, реализованные с помощью процедурных языков программирования наподобие языка COBOL.

Язык UML также не зависит от технологий реализации (поскольку они являются объектно-ориентированными). Это ограничивает язык UML в отношении поддержки этапа детализированного проектирования жизненного цикла программного обеспечения. В то же время это делает язык UML более устойчивым к частой смене платформ реализации.

Конструкции языка UML позволяют моделировать статическую структуру и динамическое поведение системы. Система представляется в виде взаимодействующих **объектов** (программных модулей), реагирующих на внешние события. Действия объектов позволяют выполнить определенные задачи или получить клиентам (пользователям) системы некоторый полезный результат. Отдельные модели отображают определенные стороны системы и пренебрегают другими сторонами, которые охватывают другие модели. Взятые в комплексе, модели обеспечивают полное описание системы.

Модели, создаваемые с помощью языка UML, можно разделить на три группы.

- *Модели состояний*. Описывают статические структуры данных.
- *Модели поведения*. Описывают взаимодействие объектов.
- *Модели изменения состояний*. Описывают допустимые состояния системы, в которые она может переходить с течением времени.

Язык UML также содержит несколько *архитектурных конструкций* (architectural constructs), позволяющих придать системе модульную структуру, используемую в процессе итеративной и поступательной разработки. Однако язык UML не ориентируется на какую-либо конкретную *архитектурную среду*, которой система должна или обязана соответствовать. Такая среда должна была бы определять структурные слои системных компонентов и определять способы связи между ними. Это противоречило бы универсальному характеру языка UML.

1.1.2.3.2. CASE-средства и совершенствование процесса

Совершенствование процесса — это нечто большее, чем просто введение новых методов и средств. В действительности введение новых методов и средств в организации, находящейся на низком уровне зрелости процесса разработки, может принести больше вреда, чем пользы.

Рассмотрим соответствующий пример. CASE-средства, применяемые в виде *интегрированной среды*, позволяют нескольким разработчикам взаимодействовать друг с другом и совместно использовать проектную информацию для выработки новых проектных артефактов. Для того чтобы воспользоваться преимуществами этой технологии, коллектив разработчиков должен подчиняться определенным правилам, поскольку CASE-средства налагают на процессы некоторые ограничения. Однако, если разработчики не настолько квалифицированны, чтобы усовершенствовать процесс разработки, чрезвычайно маловероятно, чтобы они смогли воспринять процесс, диктуемый CASE-средствами. В результате потенциальные возможности роста продуктивности и качества, обещанные новой технологией, так и не будут реализованы.

Рассмотренные выше особенности применения CASE-средств не означают, что CASE-технология — рискованное дело. Она может не дать ожидаемых выгод, только если попытаться использовать ее в некомпетентной команде. Однако те же методы и CASE-средства безусловно могут обеспечить повышение личной

производительности и качества работы отдельных разработчиков, использующих технологию на своих локальных рабочих станциях. Моделировать программные артефакты с помощью карандаша и бумаги уместно только в аудитории, но не в реальном проектировании.

1.1.3. Разработка или интеграция?

Разработка новых систем программного обеспечения, автоматизирующих исключительно ручные процессы, в настоящее время практически не осуществляется. Большинство проектов, связанных с разработкой программного обеспечения, заменяют или расширяют существующие программные системы либо интегрируют их в более крупные системы, обеспечивающие новый уровень автоматизации. Соответственно *интеграция* (integration development), в противоположность *прикладной разработке* с нуля, также производится с помощью тех же самых итеративных подходов и приводит к одинаковым программным моделям. Отличия заключаются лишь в уровне интеграции и технологии.

Интеграционные подходы разделяются на три общие категории (Hohpe and Woolf, 2003; Linthicum, 2004).

- Информационно-ориентированный и/или ориентированный на порталы.
- Ориентированный на интерфейсы.
- Ориентированный на процессы.

Информационно-ориентированная интеграция (information-oriented integration) основана на обмене **информацией** между источником и целевым приложением. Она происходит на уровне баз данных или интерфейсов прикладного программирования (application programming interface — API), открывающих доступ к информации для других приложений.

Интеграция, ориентированная на порталы (portal-oriented integration), представляет собой разновидность информационно-ориентированной интеграции, в рамках которой информация поступает от многочисленных внешних программных систем в общий пользовательский интерфейс, как правило, на портал Web-браузера. Отличие заключается в том, что информационно-ориентированная интеграция фокусируется на обмене информацией в реальном времени, а интеграция, ориентированная на порталы, требует вмешательства пользователей, чтобы информация поступила к адресату.

Интеграция, ориентированная на интерфейсы (interface-oriented integration), связывает между собой интерфейсы приложений (т.е. *сервисы*, определенные с помощью абстракции интерфейса). Интерфейсы предоставляют услуги, выполняемые одним приложением для другого. Интеграция, ориентированная на интерфейсы, не требует визуализации подробных бизнес-процессов, протекающих в приложениях. Поддержка услуг может быть *информационной* (когда поставляются данные) или *операционной* (когда выполняется часть функций). В первом слу-

чае интеграция становится разновидностью информационно-ориентированного подхода, однако второй вариант требует изменений источника и целевых приложений и может привести к созданию нового приложения (составного).

Интеграция, ориентированная на процессы (process-oriented integration), связывает приложения с помощью определения нового яруса процессов поверх набора процессов и данных в существующих приложениях. Вероятно, такая интеграция представляет собой наиболее ярко выраженное решение, в котором логика нового процесса отделена от логики участвующих приложений и возникает новая система, автоматизирующая задачи, которые ранее выполнялись вручную. Интеграция, ориентированная на процессы, начинается с построения модели нового процесса и предполагает полную видимость внутренних процессов интегрируемых приложений. Этот подход имеет стратегическую составляющую, нацеленную на усиление существующих бизнес-процессов и достижение конкурентного преимущества.

Контрольные вопросы 1.1

КВ1. Являются ли второстепенные трудности инвариантами разработки программного обеспечения?

КВ2. Назовите две основные группы участников проектов по разработке программного обеспечения.

КВ3. Добавляет ли каждая новая версия новые функциональные возможности разрабатываемого программного обеспечения?

КВ4. К чему относится стандарт COBIT— к продукции или процессам?

КВ5. Является ли компоновка, ориентированная на порталы, разновидностью информационно-ориентированной компоновки?

1.2. Планирование систем

Проекты по разработке информационных систем должны быть заранее спланированы. Информационные системы необходимо идентифицировать, классифицировать, ранжировать и выбирать для первоначальной разработки, для усовершенствования или, возможно, для ликвидации. Вопрос состоит в следующем: “Какие информационные технологии и приложения принесут наибольшие деловые выгоды? В идеале решение, которому необходимо следовать, должно базироваться на *бизнес-стратегии* и тщательном и методичном планировании” (Bennett et al., 2002; Hoffer et al., 2002; Maciaszek, 1990).

Бизнес-стратегию можно определить с помощью различных процессов: *стратегического планирования* (strategic planning), *бизнес-моделирования* (business modeling), *модернизации бизнес-процессов* (business process re-engineering), *стратегического выравнивания* (strategic alignment), *управления информационными ресурсами* (information resource management) и т.п. Все эти подходы предусматривают изучение фундаментальных бизнес-процессов в организации, целью которых является определение долгосрочных перспектив с последующим назначением приоритетов различным проблемам, которые могут быть разрешены с помощью информационных технологий.

Существует много организаций — в частности, небольших компаний, — не имеющих ясной стратегии ведения бизнеса. Подобные организации обычно принимают решение разработать информационную систему, просто выявляя наиболее насущные деловые проблемы, требующие немедленного решения. При изменении внешней деловой среды или внутренних условий ведения бизнеса существующая информационная система подлежит модификации и даже замене. Этот подход позволяет небольшим организациям быстро перестраиваться в соответствии с текущей ситуацией, извлекать выгоду из открывающихся возможностей и отвести новые угрозы.

Крупные организации не могут позволить себе постоянно изменять направления деловой активности. В действительности они часто диктуют направление работы другим организациям, работающим в той же сфере бизнеса, а также в определенной мере могут формировать среду для удовлетворения своих *текущих* потребностей. Однако крупные организации должны внимательно всматриваться в *будущее*, используя плановый подход при выборе проектов, связанных с разработкой. Как правило, это масштабные проекты, на выполнение которых требуется много времени. Они слишком громоздки, чтобы их можно было легко изменить или заменить, и должны быть способны адаптироваться или даже быть устремлены навстречу будущим возможностям и угрозам.

Существует много способов **планирования систем** (system planning). Один из традиционных подходов получил название **SWOT** (strengths, weaknesses, opportunities, threats — сильные стороны, слабые стороны, благоприятные возможности, угрозы). Еще одна популярная стратегия базируется на модели **VCM** (value chain model — модель цепочек ценности). Более современные варианты подходов к разработке бизнес-стратегий известны как **BPR** (business process reengineering — модернизация бизнес-процессов). Информацию, необходимую для деятельности организации, также оценивают с использованием проектных шаблонов **ISA** (information system architecture — архитектура информационных систем). Подобные проектные шаблоны можно получить, проводя аналогию с описательными схемами, успешно зарекомендовавшими себя в дисциплинах, отличных от информационных технологий (например, в строительной промышленности).

Все подходы к планированию разработки систем обладают общим свойством — они направлены скорее на достижение *эффективности* (делать то, что нужно), чем на *производительность* (делать так, как нужно). Наиболее эффективное решение означает, что работник выполняет прежнее задание быстрее с помощью существующих ресурсов или дешевле, если затрачивается меньшее количество ресурсов. Иначе говоря, повышение эффективности означает использование альтернативных ресурсов и идей, чтобы лучше выполнять работу. Кроме того, наиболее эффективное решение может означать достижение *конкурентного преимущества* за счет внедрения инноваций.

1.2.1. Подход SWOT

Подход SWOT позволяет идентифицировать, классифицировать, ранжировать и выбирать проекты по разработке информационных систем таким образом, чтобы они были увязаны с сильными и слабыми сторонами организации, а также с возможностями и угрозами. Это подход "сверху вниз", применение которого начинается с определения миссии организации.

Формулировка миссии (mission statement) фиксирует уникальный характер организации и определяет ее видение того, где она хотела бы оказаться в будущем. Верно сформулированная миссия отводит главное место потребностям клиентов, а не товарам или услугам, которые предоставляет организация.

Формулировка миссии и разрабатываемая на ее основе бизнес-стратегия учитывают то, какими *внутренними сильными и слабыми сторонами* обладает компания в таких областях, как управление, производство, кадровое обеспечение, финансы, маркетинг, исследования и разработка и т.д. Эти сильные и слабые стороны должны быть детально обсуждены и согласованы, после чего им назначаются приоритеты. Преуспевающие организации способны в любой момент идентифицировать текущий набор сильных и слабых сторон, которые направляют разработку их бизнес-стратегий.

К *сильным сторонам* относятся следующие факторы.

- Владение торговыми марками и патентами.
- Хорошая репутация среди заказчиков и поставщиков.
- Эксклюзивный доступ к ресурсам или технологии.
- Более низкие затраты благодаря объему производства, владению ноу-хау, эксклюзивным правам или партнерству.

Часто *слабой стороной* считается отсутствие потенциальной силы. К слабым сторонам относятся следующие факторы.

- Ненадежный денежный поток.
- Слабый опыт персонала и ненадежность ключевых сотрудников.
- Неудачное расположение компании.

Выявление внутренних сильных и слабых сторон компании — необходимое, но недостаточное условие для успешного делового планирования. Организация функционирует не в вакууме — ее деятельность зависит от внешних экономических, социальных, политических и технологических факторов. Она должна знать о *внешних благоприятных возможностях*, из которых необходимо извлечь выгоду, а также о *внешних угрозах*, которых следует избегать. Этими факторами организация не в состоянии управлять, однако осведомленность в отношении их играет существенную роль при определении целей и задач организации.

К *возможностям* относятся следующие факторы.

- Новые, менее строгие регуляторные правила, устранение торговых барьеров.
- Стратегический союз, совместное предприятия или слияние.
- Интернет как новый рынок.
- Коллапс конкурента или открытие рынка.

Любые негативные изменения среды являются угрозами. К ним относятся следующие факторы.

- Потенциальная ценовая война с конкурентами.
- Технологические изменения, выходящие за пределы возможностей компании.
- Новые налоговые барьеры, мешающие продвижению товаров или услуг.

На любом заданном отрезке времени организация преследует одну цель или очень небольшое количество *целей*. Цели обычно имеют долговременный характер (три–пять лет), а иногда относятся к "вечным проблемам". К типичным примерам целей относятся повышение уровня удовлетворенности потребителей, введение новых видов услуг, преодоление конкурентных угроз, усиление контроля над поставщиками и т.д. Каждая стратегическая цель должна быть связана с определенными задачами, которые обычно принимают форму годовых заданий. Например, цель "повысить уровень удовлетворенности потребителей" может поддерживаться задачей более быстрого выполнения заказов клиентов — скажем, в течение двух недель.

Для решения задач и достижения целей необходимо применять *стратегии управления* и проводить конкретную *политику* для их реализации. Подобные методы управления позволяют привести в соответствие организационные структуры, распределить ресурсы и определить проекты, связанные с разработкой, в том числе информационных систем.

На рис. 1.3 показаны отношения и правила, связывающие концепции, на которых основан SWOT-анализ. Матрица SWOT определяет позицию организации на рынке и позволяет сопоставить ее возможности с конкурентной средой, в которой она действует.



Рис. 1.3. Подход SWOT

1.2.2. Подход VCM

Подход VCM (метод ценностных цепочек) позволяет оценить конкурентные преимущества с помощью анализа всей цепочки видов деятельности в организации, начиная с получения сырья и кончая конечной продукцией, продаваемой и доставляемой потребителям. В рамках этой модели товар или услуга рассматривается как средство, с помощью которого организация доставляет *ценности* своим заказчикам. Метафора цепочки подчеркивает тот факт, что единственное слабое звено приводит к разрыву всей цепи. Модель VCM позволяет определить, какая именно конфигурация цепочки приносит наибольшее конкурентное преимущество. Следовательно, проекты, связанные с разработкой информационных систем, могут быть нацелены на те сегменты, операции, каналы распределения, маркетинговые подходы и др., приносящие наибольшие выгоды.

В первоначальном варианте метода VCM (Porter, 1985) организационные функции были разделены на *основные* (primary activity) и *вспомогательные виды деятельности* (support activity). Основные виды деятельности создают или добавляют ценность конечному продукту. Они разделяются на пять итеративных этапов.

1. *Принимающая логистика* (inbound logistics). Получение данных для производства товаров или услуг
2. *Обработка* (operations). Использование данных для создания товаров или услуг.

3. *Отправляющая логистика* (outbound logistics). Распределение товаров или услуг среди потребителей.
4. *Сбыт и маркетинг* (sales and marketing). Стимулирование клиентов к потреблению товаров и услуг.
5. *Обслуживание* (services). Поддержка или повышение ценности товаров и услуг.

Эти пять категорий деятельности используют соответствующие информационные технологии и обслуживают разные категории информационных систем.

1. Принимающая логистика реализуется в системах складирования.
2. Обработка является основным видом деятельности на производстве компьютеров.
3. Отправляющая логистика реализуется в системах, связанных с доставкой товаров и расписанием движения.
4. Сбыт и маркетинг образуют основу систем обслуживания заказов и счетов.
5. Обслуживание реализуется в системах эксплуатации оборудования.

Поддерживающие виды деятельности не добавляют ценности, по крайней мере, прямо. Они также играют существенную роль, но не "обогащают" продукт. Поддерживающие виды деятельности включают: администрирование и обслуживание инфраструктуры, управление кадрами, исследования и разработку и, вполне естественно, разработку информационных систем.

Несмотря на то что модель VCM представляет собой полезный инструмент для планирования информационных систем, вездесущая компьютеризация может способствовать переменам в способах ведения бизнеса, что, в свою очередь, создает конкурентные преимущества. Иначе говоря, *информационные технологии могут трансформировать ценностные цепочки организации*. Таким образом, между информационными технологиями и моделью VCM может быть установлена положительная обратная связь.

Портер и Миллар (Porter and Millar, 1985) выделяют пять шагов, которые должна предпринять организация, чтобы воспользоваться преимуществами, предоставляемыми информационными технологиями.

1. Оценить информационную емкость продуктов и процессов.
2. Оценить роль информационных технологий в отраслевой структуре.
3. Выявить и ранжировать способы, с помощью которых информационные технологии создают конкурентное преимущество.
4. Рассмотреть, каким образом информационные технологии могут создать новое направление в бизнесе.
5. Разработать план, направленный на извлечение выгод от использования информационных технологий.

1.2.3. Подход BPR

Подход к планированию разработки информационных систем с использованием методов модернизации бизнес-процессов (BPR-методов) основан на допущении, что современные организации должны реконструироваться и отказаться от функциональной декомпозиции, иерархических структур и принципов приоритетности повседневных нужд, которые они сегодня используют.

Рассматриваемая концепция была введена в 1990 году Хаммером (Hammer), а также Дэвенпортом (Davenport) и Шортом (Short) и сразу обратила на себя внимание, вызвав полемику. Расширенное описание метода BPR можно найти в книгах его основателей (Davenport, 1993; Hammer and Champy, 1993a).

В настоящее время большинство организаций имеют структуру с *вертикальным подчинением подразделений*, сосредоточенных на функциях, товарах или регионах. Эти структуры и методы работы прослеживаются вплоть до восемнадцатого века и берут свое начало в принципе разделения труда и последующей фрагментации работы, сформулированном еще Адамом Смитом. Никто из работников или подразделений не отвечает за *бизнес-процесс*, который определяется как "…совокупность видов деятельности, получающих на входе один или несколько типов ресурсов и создающих на выходе продукцию, представляющую ценность для потребителя" (Hammer and Champy, 1993a).

Методология модернизации бизнес-процессов ставит под сомнение индустриальные принципы Смита о разделении труда, иерархическом управлении и экономии на масштабах. В современном мире организация должна быть способна адаптироваться к быстрым изменениям рынка, новым технологиям, конкурентным факторам, требованиям потребителей и т.д.

Жесткие организационные структуры, в которых бизнес-процессы разделены между многими подразделениями, устарели. Организации должны сосредоточиваться на бизнес-процессах, а не на отдельных задачах, заданиях, специалистах и функциях подразделений. Эти процессы разделены *по горизонтали* между видами деятельности и завершаются в точках контакта с потребителями. "Наиболее видимое различие между предприятием, ориентированным на бизнес-процессы, и традиционной организацией состоит в существовании у каждого из процессов "хозяина" (Hammer and Stanton, 1999).

Основная цель модернизации бизнес-процессов состоит в радикальном пересмотре бизнес-процессов в организации (поэтому подход BPR зачастую называется *перепроектированием процессов*). Бизнес-процессы необходимо идентифицировать, оптимизировать и усовершенствовать. Поведение процессов фиксируется в виде *диаграмм потоков работ* (workflow diagrams) и изучается в рамках *анализа потоков работ* (workflow analysis). Потоки работ охватывают потоки событий, документов и информации в рамках бизнес-процессов и могут использоваться для подсчета времени, ресурсов и финансовых средств, необходимых для этих видов деятельности.

Дэвенпорт и Шорт (1990) рекомендуют выполнять пять этапов в рамках BPR-подхода.

1. Определить концепцию бизнеса и цели процесса (концепция бизнеса проистекает из формулировки миссии, а цели учитывают уменьшение затрат и времени, улучшение качества, повышение компетентности персонала, приобретение знаний и т.д.).
2. Определение модернизируемого процесса.
3. Анализ и оценка существующих процессов (чтобы не повторять прошлых ошибок и найти прочную основу для модернизации и улучшения бизнес-процесса).
4. Идентифицировать механизмы информационной технологии и выяснить их влияние на модернизацию и улучшение процесса.
5. Разработать и создать *прототип* (prototype) нового процесса (прототип представляет собой рабочую систему, создаваемую в рамках итеративного и поступательного подхода).

Основное препятствие на пути реализации BPR-подхода в организации заключается в необходимости внедрения горизонтального процесса в традиционную вертикальную структуру управления. Серьезная инициатива по внедрению BPR-подхода требует опоры на коллективы проектировщиков как основные организационные единицы. Эти коллективы отвечают за один или несколько сквозных бизнес-процессов.

Иногда радикальные изменения неприемлемы. Традиционные структуры не могут быть изменены в одночасье. Радикальные шаги могут встретить сопротивление, и потенциальные выгоды от внедрения BPR-подхода могут быть подвергнуты риску. В данных обстоятельствах организация все же может выиграть от моделирования бизнес-процессов и попыток просто усовершенствовать их, а не подвергать полной переделке. Для характеристики подобного начинания используется термин "усовершенствование бизнес-процессов" (business process improvement — BPI) (Allen and Frost, 1998).

После того как бизнес-процессы определены, владельцы процесса могут потребовать поддержки со стороны информационных технологий с целью дальнейшего повышения их производительности. Результирующий проект по разработке информационной системы должен сосредоточиться на реализации выявленных потоков работ. Сочетание *эффективности*, получаемой от применения BPR-подхода, с *производительностью*, являющейся результатом применения информационных технологий, может привести к поразительному улучшению всех современных показателей деятельности организации, таких как уровень качества и обслуживания, скорость, затраты, цена, конкурентные преимущества, гибкость и т.д.

1.2.4. Подход ISA

В отличие от уже описанных моделей подход ISA (information systems architecture — ISA) основан на проектировании "снизу вверх" и предлагает использовать нейтральную архитектурную концептуальную схему для проектных решений по созданию информационных систем, которая может подходить для различных бизнес-стратегий. По существу, подход с использованием ISA не включает методологии планирования разработки систем. Он просто предлагает схему, которая может служить в качестве рычага для достижения целей большинства бизнес-стратегий.

Подход на основе схемы ISA был впервые введен Захманом (Zachman, 1997) и впоследствии развит Совой (Sowa) и Захманом (1987). Незначительно видоизмененный вариант оригинальной статьи вновь был опубликован Захманом (1998).

Схема ISA представляет собой таблицу из тридцати ячеек, организованных в виде пяти строк (помеченных от 1 до 5) и шести столбцов (помеченных от A до F). Строки представляют различные *ракурсы* (perspectives), используемые при конструировании сложных инженерных продуктов, таких как информационные системы. Эти ракурсы отражают точки зрения пяти основных "игроков" — пяти участников разработки ИС.

1. Планировщик (определяет границы системы).
2. Владелец (определяет концептуальную модель предприятия).
3. Проектировщик (задает физическую модель системы).
4. Конструктор (обеспечивает детализированное технологическое решение).
5. Субподрядчик (поставляет компоненты системы).

Шесть столбцов представляют шесть различных *описаний* (descriptions) или *архитектурных моделей* (architectural model), с которыми работает каждый участник. Подобно ракурсам, описания значительно отличаются друг от друга, но в то же время они внутренне связаны между собой. Описания призваны дать ответ на шесть вопросов в отношении моделируемых сущностей, которыми задается каждый из участников.

- **A.** *Что* представляет собой созданная сущность (например, в случае информационных систем сущность — это данные).
- **B.** *Как* функционирует эта сущность (т.е. бизнес-процесс).
- **C.** *Где* расположена сущность (иначе говоря, как расположены обрабатывающие компоненты).
- **D.** *Кто* работает с сущностью (кто является пользователем).
- **E.** *Когда* с сущностью что-то происходит (т.е. как события и состояния распределены во времени).
- **F.** *Зачем* нужна сущность (т.е. мотивация предприятия).

Комбинация точек зрения и описаний, представленных в тридцати ячейках таблицы Захмана, представляет собой мощную систематизацию, на основе которой можно выстроить полную архитектуру для разработки информационной системы. Расположенные по вертикали ракурсы могут отличаться степенью детализации, но, что более важно, они отличаются по существу и используют различные представления модели. Различные модели отражают различные взгляды участников. Аналогично расположенные по горизонтали описания подготовлены, исходя из различных соображений. Каждое из этих описаний призвано ответить на один из шести вопросов.

Наиболее привлекательной стороной подхода ISA является то, что он предлагает схему, которая, вполне вероятно, может оказаться достаточно гибкой для адаптации к будущим изменениям в условиях ведения бизнеса и в ресурсах, которыми располагает предприятие. Это является следствием того, что решения на основе ISA не базируются на какой-либо определенной бизнес-стратегии. Это просто схема для полного описания информационной системы. Сама схема получена на основании опыта, накопленного более устоявшимися дисциплинами, некоторые из которых насчитывают тысячи лет (например, классическая архитектура).

Контрольные вопросы 1.2

КВ1. Назовите основную цель планирования системы.

КВ2. Являются ли задачи следствием поставленных целей в SWOT-анализе, или наоборот?

КВ3. К какой категории действий в рамках модели VCM относятся "продажа и маркетинг" — к основным или поддерживающим?

КВ4. В чем заключается основное различие между промышленным процессом и традиционной организацией в рамках модели BPR?

КВ5. Перечислите пять ракурсов модели ISA.

1.3. Системы для трех уровней управления

Планирование системы основано на понимании, что в организации существуют три уровня управления.

1. Стратегический.
2. Тактический.
3. Оперативный.

Эти три уровня управления характеризуются собственным уровнем принимаемых решений, различным набором необходимых информационных приложений и специфическими требованиями к поддержке со стороны информационных подразделений. Одна из задач, возникающих при планировании разработки системы, состоит в определении комплекса информационных приложений и решений, который является наиболее эффективным для организации в определенный момент времени. В табл. 1.1 приведены решения, связанные с разными уровнями принятия решения (Benson and Standing, 2002; Jordan and Machesky, 1990; Robson, 1994).

Таблица 1.1. Поддержка различных уровней принятия решений со стороны информационной системы

Уровень принятия решений	Направленность принимаемых решений	Типичные приложения информационных систем	Типичные решения	Ключевая концепция
Стратегический (исполнительный уровень и верхние эшелоны управления)	Стратегии, направленные на достижение долгосрочных целей организации	Анализ рынка и продаж, производственное планирование, оценка производительности	Сбор данных, управление знаниями	**Знания**
Тактический (уровень линейного руководства)	Политика, направленная на достижение краткосрочных целей и оптимальное распределение ресурсов	Анализ бюджета, прогнозирование зарплат, складской учет, обслуживание клиентов	Хранилище данных, аналитическая обработка данных, электронные таблицы	**Информация**
Оперативный (уровень оперативного управления)	Повседневная работа и поддержка производства	Выплата зарплат, выставление счетов-фактур, оплата покупок, бухгалтерский учет	База данных, обработка операций, генераторы приложений	**Данные**

Информационные приложения и решения, способные обеспечить наибольшие доходы, относятся к *стратегическому уровню*. Однако эти решения наиболее трудно реализовать — они используют передовые технологии и требуют очень квалифицированного и специализированного проектирования. Тем не менее именно эти системы способны обеспечить организации конкурентное преимущество на рынке.

На противоположном конце шкалы располагаются системы, поддерживающие *оперативный уровень управления*. Эти системы отличаются однообразием действий

и процедур, используют традиционные технологии баз данных и зачастую собираются из готовых к применению пакетных программных решений. Данные системы неперспективны с точки зрения обеспечения конкурентного преимущества, однако без них организация не способна функционировать надлежащим образом.

Любая современная организация имеет в своем распоряжении полный комплект систем оперативного управления, но только организации, которые достигли больших высот в искусстве управления, обладают интегрированным набором информационных приложений стратегического уровня. Основная технология, которая используется для хранения и обработки данных в процессе принятия высокоуровневых стратегических и тактических решений, известна как технология *хранилищ данных* (Kimball, 1996).

Последний столбец в табл. 1.1 содержит три ключевые концепции, связанные с информационными системами и уровнями принятия решений, — знания, информация и данные. Приведем их определение.

- *Данные* — необработанные факты, представляющие собой числа, понятия и события, относящиеся к деловой активности.
- *Информация* — факты, имеющие дополнительную ценность; новые свойства и тенденции, вытекающие из обработанных и суммированных данных.
- *Знания* — понимание информации, полученное в результате опыта или изучения, порождающее способность эффективно и производительно действовать и хранящееся в мозгу (*подразумеваемое знание*) или в документированном виде.

Например, телефонный номер представляет собой фрагмент *данных*. Группа телефонных номеров, распределенных по географическим регионам или заказчикам, образуют *информацию*. Понимание способов использования этой информации для проведения телефонного маркетинга — это **знание**. Как шутливо заметили Бенсон и Стэндинг (Benson and Standing, 2002), решение не звонить другу посреди ночи — это *мудрость*. Говоря более серьезно, мудрость иногда рассматривается как сверхценная информационная концепция. Она представляет собой способность использовать знания для выработки суждений и принятия решений.

1.3.1. Системы обработки транзакций

Основной класс систем на оперативном уровне принятия решений состоит из систем оперативной обработки транзакций (online transaction processing — OLTP). **Транзакция** (transaction) — это логическая единица работы, выполняемой при решении конкретной бизнес-задачи и гарантирующей целостность базы данных после ее завершения (Maciashek and Liong, 2005). С точки зрения обработки транзакции *целостность* базы данных означает, что после ее завершения данные остаются согласованными и корректными.

Системы обработки транзакций тесно связаны с технологиями управления базами данных. *База данных* (database) — это центральное хранилище данных предприятия и ключевой стратегический ресурс любой компании. На *систему управления базами данных* (database system) возлагается основная ответственность за обеспечение одновременного доступа к данным многочисленным пользователям и прикладным программам. Этот доступ к данным ограничен понятием деловой транзакции, регламентирующим открытие или закрытие данных для одновременного доступа, а также определяющим условия, при которых данные можно изменять.

Размер базы данных измеряется в гигабайтах (GB — 10^9 байт) и даже терабайтах (TB — 10^{12} байт). Соответственно, базы данных хранятся в постоянных (*персистентных*) внешних носителях, таких как магнитные или оптические диски. Эти данные называются персистентными, поскольку они хранятся на дисках постоянно, независимо от того, подключены ли они к электропитанию или нет.

Кроме *параллельного* управления, понятие транзакции подразумевает, что при сбоях программного или аппаратного обеспечения базу данных всегда можно восстановить. Любое *восстановление* (recovery) после сбоев должно гарантировать возврат данных в корректное состояние, существовавшее до сбоя. Помимо этого, нарушенную транзакцию можно выполнить заново, чтобы достичь нового корректного состояния, определенного логикой обработки транзакций.

Несмотря на то что состояние базы данных в общем и целом определяется логикой обработки разнообразных транзакций, эта логика требует тщательного контроля бизнес-правил, принятых в компании. Следует отличать *логику приложения* (application logic), управляемую транзакциями, от *делового регламента* (business logic), управляемого другими механизмами программирования баз данных, в частности механизмами защиты целостности и триггерами. Например, если деловой регламент требует, чтобы студент, поступающий на курс, сдал вступительный экзамен, ни одна прикладная транзакция не может нарушить это правило, чтобы обеспечить незаконное поступление.

Как уже указывалось, база данных представляет собой основной стратегический ресурс любого предприятия. Вследствие этого технология управления базами данных должна обеспечивать механизмы для гарантии *безопасности* (security) данных, разрешая доступ и выполнение внутренних операций лишь пользователям и прикладным программам, прошедшим аутентификацию и авторизацию.

1.3.2. Системы аналитической обработки данных

Основной класс систем на уровне принятия тактических решений состоит из систем аналитической обработки данных в реальном времени (online analytical processing — OLAP). В отличие от оперативной обработки транзакций, которая, как правило, обеспечивает изменение данных, аналитическая обработка связана с *анализом* архивных данных при выработке правильных решений. Следовательно, вместо запроса детальных данных о продажах система аналитической обра-

ботки данных в реальном времени попробует ответить на вопрос: "Какую прибыль получила компания в прошлом месяце в результате рекламной распродажи товаров, предназначенных для женщин, по сравнению с прибылью от обычных продаж в этом же месяце в прошлом году?" Чем больше архивных данных имеет компания, тем надежнее будет принятое решение.

Системы аналитической обработки данных в реальном времени связаны с технологией управления **хранилищами данных** (data warehouse), которые обычно создаются с помощью накапливания последовательных копий данных, находящихся в одной или нескольких оперативных базах данных. Новые данные всегда добавляются в хранилище, а старые данные никогда оттуда не удаляются. Таким образом, хранилища данных быстро увеличиваются в размерах и становятся во много раз больше, чем исходные оперативные базы данных. Крупные хранилища измеряются в терабайтах (ТВ — 10^{12} байт) и даже петабайтах (РВ — 10^{15} байт). Размер хранилища данных представляет собой большую проблему, но, к счастью, эти данные являются статическими, т.е. не подвергаются изменениям (за исключением изменений, которые вносятся после обнаружения несогласованности, некорректности или неполноты данных).

В соответствии с требованиями менеджеров, принимающих тактические решения, системы аналитической обработки данных в реальном времени должны поддерживать интенсивные запросы. Это условие предъявляет особые требования к технологиям управления хранилищами данных. С одной стороны, менеджерам необходима технология доступа к данным, не требующая программирования, с другой — хранилище данных должно обеспечивать легкий доступ к подробным архивным данным, разделенным, резюмированным и упакованным

Резюмирование и упаковка данных представляет собой уникальную особенность хранилищ, предназначенных для обогащения данных, извлеченных из сторонних источников. *Резюмирование* (aggregation) подразумевает отбор, объединение и группировку данных, предназначенных для вычисления количественных показателей и трендов. *Упаковка* (packaging) подразумевает преобразование оперативных и резюмированных данных в более удобный формат, такой как графики, диаграммы, электронные формы и анимационные картинки. Для *разделения* (partitioning) информации используются технические средства и профильные данные о пользователях, позволяющие сократить объем данных, которые система должна просмотреть, чтобы ответить на запрос.

Поскольку создание крупного и монолитного хранилища всех данных компании представляет собой серьезную проблему, для ее решения были предложены специальные технологии. В частности, весьма популярным практичным решением стали **информационные лавки** (data marts). Подобно хранилищам данных, информационные лавки — это специализированные базы данных, предназначенные для аналитической обработки. В отличие от хранилищ данных информационные лавки хранят лишь часть данных компании, касающуюся конкретного подразде-

ления или отдельной производственной функции. Кроме того, информационные лавки в основном хранят резюмированные архивные данные, оставляя оперативные данные в исходных источниках.

В последнее время все бо́льшую популярность получают **Web-хранилища данных** (data webhause), представляющие собой "распределенные хранилища данных, реализованные с помощью сети WWW и не имеющие центрального репозитария" (Connoly and Begg, 2005). Web-хранилища данных естественным образом устраняют сложности, связанные с извлечением, очисткой и загрузкой крупных объемов потенциально противоречивых данных из многих источников в одно место накопления данных. Web-хранилище предлагает альтернативную технологию для аналитической обработки в рамках внутренних промышленных сетей. Их потенциальное использование в Интернете ограничено требованиями конфиденциальности и безопасности данных, являющихся наиболее защищаемым стратегическим активом компании. За исключением этих ограничений Web-хранилища данных могут оказаться очень полезным инструментом анализа данных, характеризующих поведение пользователей Интернета (*данные о переходах между Web-страницами* (clickstreams)).

1.3.3. Системы обработки знаний

Основным классом систем на стратегическом уровне принятия решений являются системы обработки знаний. Под **знаниями** (knowledge) обычно понимают "ноу-хау", т.е. интеллектуальный капитал, накопленный благодаря опыту. Как указывалось в работе Рус и Линдвал (Rus and Lindval, 2002), "Основная проблема, связанная с интеллектуальным капиталом, заключается в том, что он ежедневно изменяется. Опыт сменяет неопытность, и наоборот. Вольно или невольно, многие организации, занимающиеся разработкой программного обеспечения, сталкиваются с необходимостью поддерживать определенный уровень компетентности, необходимый для получения контрактов и выполнения заключенных договоров".

Для поддержки интеллектуального капитала, накопленного в информационной системе предприятия (enterprise information system — EIS), "ноу-хау" должно быть управляемым. *Управление знаниями* (knowledge management) необходимо для того, чтобы помочь организации найти, организовать и применить знания, хранящиеся в информационной системе. **Интеллектуальный анализ данных** (data mining) — это область управления знаниями, связанная с исследованиями данных с целью выявления скрытых зависимостей и шаблонов (особенно ранее неизвестных или забытых), необходимых для получения новых знаний, восстановления старых и принятия решений.

Основные цели интеллектуального анализа данных приведены ниже (Kifer et. al., 2006; Oz, 2004).

- *Ассоциация* (*анализ троп*). Поиск шаблонов в данных, в которых одно событие приводит к другому, например, позволяет предсказать, какие арендаторы склонны отказаться от аренды и приобрести в ближайшем будущем собственный дом.
- *Классификация*. Проверка, попадают ли определенные факты в заранее определенную категорию, например, предсказание, какие клиенты проявляют наименьшую лояльность и могут перейти к другому оператору мобильной связи.
- *Кластеризация*. Задача, похожая на классификацию, но категории в этом случае заранее не известны и выявляются с помощью алгоритма кластеризации без помощи аналитика, например, предсказание ответа на предложение, поступившее в ходе прямого маркетинга по телефону.

Как и в системах OLAP, основными источниками данных для интеллектуального анализа являются хранилища, а не оперативные базы данных. Интеллектуальный анализ данных расширяет возможности систем OLAP при выработке стратегических решений. Он создает *прогностические*, а не *ретроспективные* модели. Кроме того, интеллектуальный анализ использует методы искусственного интеллекта, позволяющие выявлять тенденции, корреляции и шаблоны в данных. Он пытается найти скрытые и неожиданные знания, которые имеют большое значения для выработки стратегических решений.

Контрольные вопросы 1.3

КВ1. Какой уровень принятия решений поддерживают хранилища данных?

КВ2. Какие две основные функции выполняют механизмы выполнения транзакций в системах OLTP?

КВ3. Какая разновидность технологии OLAP предназначена для поддержки работы отдельных подразделений или выполнения производственных функций, а также хранит только резюмированные архивные данные?

КВ4. На какой технологии основаны системы обработки знаний?

1.4. Этапы жизненного цикла программного обеспечения

Разработка программного обеспечения подчиняется определенному *жизненному циклу* (lifecycle). Жизненный цикл — это упорядоченный набор видов деятельности, осуществляемый и управляемый в рамках каждого проекта по разработке

программного обеспечения. Процессы и методы — это механизмы реализации жизненного цикла. Жизненный цикл состоит из следующих компонентов.

- Прикладное моделирование.
- Этапы, на протяжении которых программное обеспечение проходит ряд преобразований от начального пункта до законченного продукта.
- Метод (методология) и связанные с этим процессы разработки.

Как правило, жизненный цикл проекта начинается с *анализа* текущей ситуации и предложенного решения. Затем аналитические модели становятся основой для более подробного *проектирования*. После проектирования наступает очередь программирования (*реализации*). Реализованные части системы *интегрируются* и *развертываются* у заказчика. В этот момент система становится *оперативной* (т.е. способной поддерживать ежедневные деловые операции). Для успешной работы система должна *сопровождаться*.

Настоящая книга посвящена анализу и проектированию систем, а вопросы, связанные с реализацией, лишь упоминаются в ней. Грубо говоря, на этапе анализа деловых процессов разработчики изучают, что должна делать система, на этапе проектирования системы — как это осуществить на основе доступных технологий, а на этапе реализации воплощают проект в виде программного обеспечения, установленного у заказчика.

1.4.1. Подходы к разработке программного обеспечения

Революция в программном обеспечении вызвала ряд значительных изменений в способах работы программных продуктов. В частности, значительно увеличились *интерактивные возможности* программ. Задачи, выполняемые программами, и их поведение могут динамически адаптироваться к запросам пользователей.

Процедурная логика программ, написанных в прошлом на языках, подобных языку COBOL, не отличалась гибкостью и слабо реагировала на неожиданные события. После запуска программа выполнялась до завершения в более или менее детерминированном режиме. Иногда программа могла запрашивать некоторую информацию от пользователя и следовать по различным выполняемым ветвям. Однако в общем случае взаимодействие с пользователем было ограничено, а количество различных выполняемых ветвей заранее фиксировано. Контроль оставался за программой, а не за пользователем.

С появлением современного графического пользовательского интерфейса (graphical user interface — GUI) все кардинально изменилось. Программы на основе интерфейса GUI управляются событиями, ход их выполнения носит случайный и непредсказуемый характер и диктуется инициируемыми пользователем событиями, источником которых являются клавиатура, мышь и другие устройства ввода.

В среде интерфейса GUI пользователь контролирует (по большей части) выполнение программы, а не наоборот. За каждым событием стоит программный

объект, который знает, как обслужить данное событие при текущем состоянии хода выполнения программы. После завершения обслуживания управление вновь возвращается пользователю.

Различные стили программирования требуют разных подходов к разработке программного обеспечения. При разработке традиционного программного обеспечения хорошо зарекомендовал себя *структурный подход* (structural approach). Современные системы на основе графического интерфейса требуют объектного программирования, и поэтому *объектный подход* является наилучшим способом проектирования таких систем.

1.4.1.1. Структурный подход

Структурный подход (structured approach) к разработке систем получил широкое распространение (и был признан стандартом де-факто) в 1980-х годах. Этот подход основан на двух методах.

- Диаграммы потоков данных (data flow diagrams — DFD) для моделирования процессов.
- Диаграммы отношений логических объектов (entity relationship diagrams — ERD) для моделирования данных.

Структурный подход является *ориентированным на процессы* (processcentric) и рассматривает диаграммы DFD в качестве движущей силы разработки программного обеспечения. В рамках этого подхода система разделяется на управляемые модули в ходе *функциональной декомпозиции* (functional decomposition). Кроме того, система иерархически разделяется на бизнес-процессы, связанные потоками данных.

Позднее, в результате широкого распространения моделей реляционных баз данных, значение диаграмм DFD в структурной разработке снизилось, и подход стал больше ориентироваться на данные, и, соответственно, акцент в разработке сместился на диаграммы ERD

Сочетание диаграмм DFD и ERD позволяет разрабатывать относительно полные модели анализа, которые фиксируют все функции и данные системы на требуемом уровне абстракции независимо от особенностей аппаратного и программного обеспечения. Затем модель анализа преобразуется в проектную модель, которая обычно выражается в понятиях реляционных баз данных. После этого следует этап реализации.

Структурный подход к анализу и проектированию отличается рядом особенностей, и некоторые из них не очень хорошо увязываются с современными методами конструирования программного обеспечения.

- Структурный подход скорее является *последовательным* (sequential) и *трансформационным* (transformational), чем итеративным и поступательным подходом (т.е. этот подход не способствует непрерывному процессу разработки,

осуществляемому посредством итеративной детализации и пошаговой поставки программного обеспечения с расширяющимися возможностями).

- Структурный подход направлен на поставку негибких решений, которые способны удовлетворить набор определенных бизнес-функций, но затрудняют масштабирование и расширение в будущем.
- Структурный подход предполагает разработку с чистого листа и не поддерживает повторное использование уже существующих компонентов.

Трансформационный характер структурного подхода является источником повышенного риска неправильно истолковать исходные требования пользователей по ходу разработки. Такой риск усиливается за счет необходимости постепенной замены относительно декларативной семантики моделей анализа процедурными решениями для проектных моделей и программного кода реализации (это является следствием того, что модели анализа семантически богаче, чем базовые проектные и реализационные модели).

1.4.1.2. Объектно-ориентированный подход

Объектно-ориентированный подход (object-oriented approach) подразумевает разделение системы на компоненты с разной степенью детализации. В основе этой декомпозиции лежат классы объектов. Классы связаны разнообразными отношениями и обмениваются сообщениями, вызывающими операции над объектами.

Несмотря на то что объектно-ориентированные языки программирования (Simula) существовали еще в начале 1970-х годов, объектно-ориентированный подход к проектированию систем получил распространение лишь в 1990-х годах. Позднее ассоциация производителей программного обеспечения Object Management Group утвердила в качестве стандартного средства моделирования для этого подхода язык UML (Unified Modeling Language — унифицированный язык моделирования).

По сравнению со структурным подходом объектно-ориентированный подход в большей степени *ориентирован на данные* — он развивается вокруг моделей классов. На этапе анализа для классов не требуется определять операции — только атрибуты. Однако возрастающее значение использования в языке UML прецедентов способствует незначительному смещению акцентов от данных к функциям.

Существует понимание того, что разработчики используют объектный подход благодаря техническим преимуществам объектной парадигмы, таким как абстракция, инкапсуляция, повторное использование, наследование, передача сообщений, полиморфизм и т.д. Эти технические свойства могут привести к более высокому уровню повторного использования программного кода и данных, сокращению времени разработки, росту продуктивности труда программистов, повышению качества программного обеспечения, большей понятности программ и т.д. При всей свой привлекательности эти преимущества объектной технологии до сих пор не нашли своего практического воплощения. Тем не менее мы продолжаем использовать сегодня

объекты и будем использовать их завтра. Одной из причин этого является необходимость использовать *программирование, управляемое событиями* (event-driven), требуемое современными интерактивными графическими интерфейсами.

Другой причиной популярности объектного подхода является возможность удовлетворить требования *вновь возникающих типов приложений* и находить наилучшие способы борьбы с ростом *дефицита ресурсов* (application backlog), который проявляется либо в недостаточном количестве программ, либо в нарушении сроков поставок в ходе расширения и компоновки существующих систем программного обеспечения. К двум наиболее важным новым категориям приложений, для которых требуется объектная технология, относятся *групповые вычисления* (workgroup computing) и *мультимедийные системы* (multimedia systems). Идея уменьшить дефицит ресурсов посредством концепции *упаковки объектов* (object wrapping) доказала как свою привлекательность, так и работоспособность.

Объектный подход к разработке систем полностью соответствует требованиям итеративного и поступательного процесса разработки. На этапах анализа, проектирования и реализации разрабатывается единая модель (и единый проектный документ), которая постепенно уточняется, изменяется и усовершенствуется, а выпуски программных модулей с расширенными возможностями поддерживают высокий уровень удовлетворенности пользователей и обеспечивают обратную связь, необходимую для продолжения разработки модулей.

Разработка с помощью последовательной детализации становится возможной благодаря тому, что все создаваемые в ходе разработки модели (анализа, проектирования и реализации) обладают семантическим богатством и базируются на одном и том же "языке" — базовый словарь этих моделей существенно не отличается (классы, атрибуты, методы, наследование, полиморфизм и т.д.). Следует, однако, заметить, что если в основу реализации положена реляционная база данных, то необходимость в сложной и связанной с риском трансформации по-прежнему сохраняется (поскольку базовая семантика реляционной модели сравнительно беднее).

Объектный подход устраняет большинство из наиболее значительных недостатков структурного подхода, однако служит источником некоторых новых проблем.

- Во-первых, этап анализа проводится на еще более высоком уровне абстракции, и если реализация сервера предполагает использование реляционной базы данных, то может возникнуть значительный *семантический разрыв* (semantic gap) между концепцией и ее реализацией. Несмотря на то что анализ и проектирование могут проводиться итеративно и последовательно, в конце концов разработка достигает этапа реализации, на котором решение должно быть воплощено в реляционную базу данных. Если в качестве платформы реализации используется объектная или объектно-реляционная база данных, трансформация проекта проходит значительно легче.
- Во-вторых, затрудняется *управление проектом*. Менеджеры измеряют степень продвижения разработки, используя четко определенные структу-

ры декомпозиции, элементы комплекта поставки и календарный план. При объектно-ориентированной разработке с помощью детализации не существует четких границ между этапами, а проектная документация непрерывно уточняется. Приемлемое решение в такой ситуации заключается в разделении проекта на небольшие модули и управлении ходом разработки за счет частого выпуска выполняемых версий этих модулей (некоторые из этих выпусков могут предназначаться для внутреннего применения, а другие — поставляться заказчику).

- В-третьих, объектные решения значительно сложнее прежних структурированных систем. *Сложность* возникает в основном из-за интенсивного обмена сообщениями между объектами и компонентами. Эта проблема усугубляется плохой архитектурой, допускающей неограниченные связи между взаимодействующими объектами. Системы, имеющие такую архитектуру, трудно эксплуатировать и расширять.

Сложности, связанные с объектным подходом, не отменяют тот факт, что, по словам Артура Кларка, "время нельзя повернуть вспять". Возврата к вызывающему ностальгию процедурному стилю пакетных приложений на языке COBOL нет. Все участники проектов по созданию информационных систем хорошо знакомы с Интернетом, электронной коммерцией, компьютерными играми и другими интерактивными приложениями.

Разработка новых программных приложений отличается значительно большей сложностью, и структурный подход совершенно не соответствует этой задаче. На сегодняшний день объектно-ориентированный подход — единственный известный метод, позволяющий совладать с разработкой нового программного обеспечения, управляемого событиями и отличающегося высоким уровнем интерактивности.

1.4.2. Этапы жизненного цикла

Жизненный цикл разработки программного обеспечения представляет собой точно определенную хронологическую последовательность действий. На *укрупненном уровне* жизненный цикл может включать пять этапов.

1. Бизнес-анализ.
2. Проектирование системы.
3. Реализация.
4. Компоновка и развертывание.
5. Эксплуатация и сопровождение.

На *этапе анализа* (analysis phase) определяются и конкретизируются (моделируются) требования, а также осуществляется разработка и компоновка функциональных и информационных моделей системы. Одновременно формулируются нефункциональные требования и другие системные ограничения.

Этап проектирования (design phase) разделяется на две основные стадии: архитектурное и детализированное проектирование. В частности, на этом этапе уточняется конструкция программы для архитектуры клиент/сервер, которая интегрирует объекты пользовательского интерфейса и базы данных, а также поднимаются и фиксируются вопросы проектирования, влияющие на понятность, приспособленность к сопровождению и масштабируемость системы.

Этап реализации (implementation phase) подразумевает написание программ для клиентских приложений и серверов баз данных. Акцент делается на итеративных процессах реализации с наращиванием возможностей системы. Успех поставки программного продукта не в последнюю очередь определяется *циклической разработкой* (round-trip engineering), характеризующейся периодическим возвратом от реализации клиентских приложений и серверов баз данных к проектным моделям и обратно.

В настоящее время было бы непрактично разрабатывать программное обеспечение "одним махом", объединяя этапы анализа, проектирования и реализации в одно целое. Программное обеспечение разрабатывается в виде небольших модулей (компонентов), которые необходимо согласовывать между собой и другими, уже работающими модулями. Эта процедура составляет сущность этапа *компоновки* и *развертывания* (integration and deployment)

Этап *эксплуатации и сопровождения* (operation and maintenance) начинается в тот момент, когда ранее существовавшая система выводится из работы, а на ее место приходит новая. Как это ни парадоксально, этап эксплуатации знаменует собой лишь начало этапа сопровождения, в течение которого осуществляется исправление и расширение программного обеспечения. В большой степени необходимость сопровождения не связана с качеством разработанной системы. Скорее, этот этап является следствием того непреложного факта, что бизнес постоянно изменяется и эти изменения должны учитываться в программном обеспечении.

1.4.2.1. Бизнес-анализ

Бизнес-анализ (или *анализ требований*) — это деятельность, направленная на выявление и уточнение запросов заказчика. Определение и спецификация требований связаны между собой, но представляют разные виды работы и иногда осуществляются разными людьми. Соответственно, иногда различают бизнес-аналитика и системного аналитика. Бизнес-аналитик определяет требования, а системный аналитик конкретизирует (или моделирует) их.

Бизнес-анализ, даже если он проводится в контексте небольшой предметной области, требует внимательного изучения бизнес-процессов. В этом смысле бизнес-анализ тесно связан с *модернизацией бизнес-процессов* (BPR), описанной в разделе 1.2.3. Цель модернизации — предложить новые способы ведения бизнеса и достижения конкурентного преимущества. Эти "новые способы" должны быть свободны от влияния существующих решений, включая существующие информационные системы.

В результате сочетания модернизации бизнес-процессов и нормальной работы по усовершенствованию проекта бизнес-анализ все чаще становится актом *проектирования требований* (requirements engineering). Действительно, проектирование требований становится все более важной частью проектирования программного обеспечения, тесно связанной с другими дисциплинами.

1.4.2.1.1. Этап установления требований

Котонья и Соммервилль (Kotonya and Sommerville, 1998) определяют **требование** (requirement) как "формулировку системной услуги или ограничения". *Формулировка услуги* (service statement) характеризует поведение системы по отношению к отдельным пользователям или ко всему контингенту пользователей. В последнем случае описание сервиса фактически служит определением *бизнес-правила* (business rule), которое должно выполняться всегда (например, "двухнедельная зарплата выплачивается в среду"). Формулировка услуги может быть связана с некоторым вычислением, которое должна произвести система (например, "вычислить комиссионные продавца на основе объема продаж на прошлую среду с использованием конкретной формулы"). *Формулировка ограничения* (service statement) выражает ограничивающее условие на поведение или разработку системы. Примером первого ограничения может быть ограничение на безопасность — "только непосредственные руководители могут обращаться к информации о зарплате их персонала". Примером последнего типа может быть формулировка "мы должны использовать средства разработки компании Sybase". Обратите внимание на то, что иногда различие между формулировкой ограничения на поведение системы и формулировкой услуги на основе бизнес-правила размыто. Это не составляет проблемы в том случае, если все требования идентифицированы и дублирование исключено.

Задачей этапа определения требований является определение, анализ и обсуждение требований с заказчиками. На этом этапе применяются различные методы сбора информации от заказчиков. Это и исследование концепции с помощью структурированных и неструктурированных интервью пользователей, анкеты, изучение документов и форм, видеозаписи и т.д. Последним методом, применяемым на этапе определения требований, является *быстрая разработка прототипа* (rapid prototyping) решения, так что требования, вызывающие затруднения, могут быть прояснены, что позволяет избежать недоразумений.

Анализ требований включает переговоры между разработчиками и заказчиками. Этот шаг необходим для исключения противоречивых и дублирующихся требований, а также согласования проектного бюджета и сроков.

Результатом этапа установления требований является *техническое задание* (requirements document). Это большей частью текстовый документ с некоторыми неформальными диаграммами и таблицами. В этот документ, как правило, не включаются формальные модели, за исключением, может быть, некоторых простых и широко известных видов нотации, которые легко могут быть восприняты заказчиками и могут облегчить взаимопонимание разработчиков и заказчиков.

1.4.2.1.2. Этап спецификации требований

Этап спецификации требований начинается с того момента, когда разработчики приступают к моделированию требований с использованием определенного метода (например, такого как UML). Для ввода, анализа и документирования модели используются CASE-средства. В результате техническое задание дополняется графическими моделями и отчетами, сгенерированными с помощью CASE-средств. По существу, *спецификация* (specifications document) заменяет собой техническое задание.

В рамках объектно-ориентированного анализа в качестве основных методов спецификации требований используются два типа диаграмм: *диаграммы классов* (class diagrams) и *диаграммы прецедентов* (use case diagrams). Эти методы позволяют разрабатывать спецификации данных и функций. Обычно документ, содержащий спецификацию требований, включает также описание других видов требований. Среди атрибутов системы, требования к которым могут быть приведены в спецификации, относятся, например, производительность, впечатления и ощущения от использования программы, практичность, легкость сопровождения, безопасность, а также политические и юридические требования.

Модели спецификации могут и должны перекрываться. Это позволяет рассмотреть предлагаемое решение с разных точек зрения, выделяя и анализируя различные аспекты решения. Кроме того, это дает возможность проверить непротиворечивость и полноту требований.

В идеале модели спецификаций должны быть независимы от программной и аппаратной платформ, на которых должна разворачиваться система. Учет особенностей программной и аппаратной платформ накладывает жесткие ограничения на словарь (а следовательно, и на выразительность) языка моделирования. Более того, словарь может быть труден для понимания заказчиками и, таким образом, препятствовать общению разработчиков и заказчиков.

Тем не менее некоторые ограничения могут фактически навязывать разработчикам необходимость рассмотрения особенностей программного и аппаратного обеспечения. Более того, сами заказчики могут быть выразителями требований относительно определенной технологии или даже требовать применения определенной технологии. Отсюда следует вывод: по возможности необходимо избегать рассмотрения особенностей программного и аппаратного обеспечения на этапе составления спецификации системы.

1.4.2.2. Проектирование систем

Проектирование систем (system design) намного шире, чем *проектирование программного обеспечения* (software design), хотя, несомненно, проектирование программного обеспечения занимает центральное место. Проектирование систем подразумевает декомпозицию структуры системы и детальную проработку ее компонентов. В соответствии с этим проектирование систем иногда разделяют на проектирование архитектуры и детализированное проектирование.

Проектирование вытекает из анализа. Несмотря на то что это замечание вполне справедливо, архитектурное проектирование можно рассматривать как относительно самостоятельный вид деятельности, нацеленный на достижение архитектурного совершенства с помощью обоснованных и проверенных на практике методов проектирования. В противоположность этому, детализированное проектирование является непосредственным следствием моделей анализа.

Спецификация, полученная в результате анализа, представляет собой разновидность *контракта* между разработчиком и заказчиками на поставку программного продукта. В нем перечисляются все требования, которым должно удовлетворять программное обеспечение. Спецификации являются руководством к действию для системных архитекторов, архитекторов программного обеспечения, проектировщиков и инженеров, разрабатывающих низкоуровневые модели системной архитектуры и ее внутренних компонентов. Проектирование осуществляется с учетом аппаратной и программной платформы, на которой должна быть реализована система.

1.4.2.2.1. Этап архитектурного проектирования

Описание системы в терминах составляющих ее модулей называется *архитектурным проектированием* (architectural design). Оно подразумевает выбор стратегических решений, касающихся клиентской и серверной частей системы. Кроме того, архитектурное проектирование связано с выбором стратегии решения и разбивкой системы на модули. *Стратегия решения* (solution strategy) требует решения вопросов, касающихся клиентской (пользовательский интерфейс) и серверной (база данных) частей системы, а также *промежуточного программного обеспечения* (middleware), необходимого для связывания клиента и сервера. Выбор основных строительных блоков (модулей) относительно слабо связан со стратегией решения, однако детализированный проект модулей должен соответствовать выбранному решению по архитектуре клиент/сервер.

Зачастую модели архитектуры клиент/сервер расширяются до так называемой *трехъярусной архитектуры* (three-tier architecture), в которой логика приложения составляет отдельный слой. Среднее звено представляет собой логический ярус и в этом качестве может поддерживаться или не поддерживаться отдельным аппаратным обеспечением. Логика приложения — это процесс, который может выполняться на клиентской машине или на сервере, т.е. он скомпилирован в виде клиентского или серверного процесса и реализован как библиотека DLL (Dynamic Link Library — динамически подключаемая библиотека), интерфейс API (Application Programming Interface — интерфейс прикладного программирования), RPC-вызовы (Remote Procedure Calls — удаленный вызов процедуры) и т.д.

Качество архитектурного проектирования чрезвычайно важно для долговременного успеха системы. Хорошее архитектурное проектирование позволяет создавать **адаптивные (поддерживаемые) системы**, т.е. системы, поддающиеся по-

ниманию, сопровождению и масштабированию (расширению). Без этих качеств внутренняя сложность программного обеспечения выходит из-под контроля. Следовательно, крайне важно, чтобы в результате архитектурного проектирования возникала адаптивная структура системы, которая сохранялась бы на этапе программирования и тщательно поддерживалась после поставки системы.

1.4.2.2.2. Этап детализированного проектирования

Описание внутренней работы каждого компонента программного обеспечения называется *детализированным проектированием* (detailed design). Его результатом являются подробные алгоритмы и структуры данных для каждого компонента. В конце концов, компоненты разворачиваются на клиентском, серверном или промежуточном узлах базовой платформы реализации. Соответственно, алгоритмы и структуры данных должны учитывать ограничения (как способствующие, так и препятствующие работе системы), накладываемые на базовую платформу.

Детализированный проект *клиента* (пользовательского интерфейса) должен соответствовать принципам проектирования графического пользовательского интерфейса, поддерживаемым Интернет-браузерами, Web-приложениями или требованиями, установленными разработчиками конкретного интерфейса (Windows, Motif, Macintosh). Подобные принципы обычно доступны в интерактивном режиме как часть электронной документации, сопровождающей графические пользовательские интерфейсы (например, Windows 2000).

Основной принцип объектно-ориентированного проектирования интерфейса GUI состоит в том, что управление приложением *является прерогативой пользователя*, а не программы. Программа реагирует на случайные события, источником которых является пользователь, и предоставляет необходимый программный сервис. Остальные принципы проектирования интерфейса GUI являются следствием этого факта. Разумеется, принцип "контроль является прерогативой пользователя" не следует принимать буквально — программа по-прежнему может проверять права пользователя и запретить некоторые действия.

Детализированный проект *сервера* определяет объекты сервера базы данных — скорее всего, реляционной (или, возможно, объектно-реляционной). Часть этих объектов представляет собой контейнеры данных (таблицы, взгляды и т.д.). Другие объекты являются процедурами (например, хранимые процедуры и триггеры).

Детализированный проект *промежуточного программного обеспечения* тесно связан с логикой приложения и бизнес-правилами. Этот слой обеспечивает разделение и связь между пользовательским интерфейсом и базой данных. Этот аспект является очень важным, если необходимо обеспечить приемлемый уровень независимой эволюции прикладного программного обеспечения, обрабатывающего команды пользовательского интерфейса, и базы данных, обеспечивающей доступ к источникам данных.

1.4.2.3. Этап реализации

Реализация (implementation) информационной системы подразумевает *инсталляцию* (installation) приобретенного программного обеспечения и *программирование* программного обеспечения, разрабатываемого под заказ. Кроме того, реализация подразумевает осуществление некоторых других важных действий, таких как загрузка тестовых и производственных баз данных, тестирование, обучение пользователей, а также решение вопросов, связанных с работой аппаратного обеспечения.

Как правило, команда разработчиков, занимающаяся реализацией программного проекта, предполагает разделение программистов на две группы: одну, ответственную за программирование клиентских приложений, и другую, ответственную за программирование сервера баз данных. Клиентские программы реализуют оконный интерфейс и логику приложений (даже если прикладная логика развернута на отдельном сервере, всегда существуют аспекты, связывающие его с клиентом). Клиентские программы также инициируют выполнение деловых транзакций, активизирующих программы серверных баз данных (хранимые процедуры). Ответственность за непротиворечивость баз данных и корректность транзакций лежит на серверных программах.

В духе итеративной и поступательной разработки проект *пользовательских интерфейсов* часто подвергается значительным изменениям на этапе реализации. В частности, прикладные программисты могут изменять вид диалоговых окон, следуя принципам, установленным разработчиками графических пользовательских интерфейсов, облегчать программирование или повышать продуктивность работы пользователей.

Аналогично реализация сервера *баз данных* может вызвать изменения в проектных документах. Непредвиденные проблемы, связанные с базами данных, трудности при программировании хранимых процедур и триггеров, вопросы параллелизма, интеграция с клиентскими процессами, настройка производительности и т.д. — вот перечень только некоторых причин, которые могут повлечь за собой необходимость модификации проекта.

1.4.2.4. Этап интеграции и развертывания

Поступательная разработка предполагает *поэлементную компоновку* (incremental integration) программных модулей и *развертывание* (deployment) программного обеспечения (подсистем). Эта задача не так проста, как может показаться на первый взгляд. Для больших систем компоновка отдельных модулей может потребовать больше времени и усилий, чем любой из более ранних этапов жизненного цикла, включая реализацию. Еще Аристотель заметил: “Целое больше суммы частей”.

Компоновка модулей должна быть тщательно спланирована в самом начале жизненного цикла программного обеспечения. Программные компоненты, подлежащие отдельной реализации, должны быть идентифицированы на ранних стадиях анализа системы. К этому вопросу необходимо постоянно возвращаться,

уточняя детали во время архитектурного проектирования. Порядок реализации должен позволять как можно более плавную поэлементную компоновку.

Основная трудность, связанная с поэлементной компоновкой, заключается в существовании циклических взаимозависимостей между модулями. Хороший проект системы отличается минимальной *циклической связностью* (circular coupling) модулей или ее полным отсутствием. С другой стороны, для обеспечения требуемых функциональных свойств системы может потребоваться линейная связность моделей.

Что делать, если необходимо поставить один из модулей еще до того, как другой готов к применению? Ответ состоит в написании специальной программы для временного "заполнения бреши" так, чтобы все модули оказались интегрированными. Программная процедура, предназначенная для имитации работы отсутствующего модуля, называется *заглушкой* (stubs).

Объектно-ориентированные системы предназначены для компоновки и развертывания. Каждый модуль должен быть как можно более независимым. Зависимости между модулями необходимо идентифицировать и минимизировать на этапах анализа и проектирования. В идеальном случае каждый модуль должен образовывать один поток обработки, который запускается в ответ на определенное требование клиента. Использования заглушек как операций замещения следует, по возможности, избегать. Если система спроектирована недостаточно качественно, этап интеграции приведет к хаосу и поставит под угрозу весь проект по разработке системы.

1.4.2.5. Этап эксплуатации и сопровождения

Этап *эксплуатации* (operation) и *сопровождения* (maintenance) наступает после успешной передачи заказчику каждого последующего программного модуля и в конечном счете всего программного продукта. Сопровождение — не только неотъемлемая часть жизненного цикла программного обеспечения; оно составляет его бо́льшую часть, если речь идет о времени и усилиях персонала подразделений, которое приходится на сопровождение. По оценкам Шаха (Schach, 2005), 75% жизненного цикла приходится на сопровождение программного обеспечения.

Эксплуатация подразумевает *переключение* (changeover) со старого бизнес-решения, основанного или не основанного на программном обеспечении, на новое. Это переключение, как правило, представляет собой постепенный процесс. По возможности, старые и новые системы должны работать параллельно, чтобы гарантировать возврат к прежнему решению, если новое решение окажется неудачным.

Сопровождение состоит из трех различных стадий (Ghezzi et al., 2003; Maciaszek, 1990).

1. Эксплуатация.
2. Адаптивное сопровождение.
3. Совершенствующее сопровождение.

Эксплуатация (housekeeping) связана с рутинными задачами сопровождения, необходимыми для поддержания системы в состоянии готовности к применению пользователями и эксплуатационным персоналом. *Адаптивное сопровождение* (adaptive maintenance) связано с отслеживанием и анализом работы системы, настройкой ее функциональных возможностей применительно к изменениям внешней среды и адаптацией системы для достижения заданной производительности и пропускной способности. Под *совершенствующим сопровождением* (perfective maintenance) понимают перепроектирование и модификацию системы для удовлетворения новых или существенно изменившихся требований.

В конечном итоге непрерывное сопровождение системы становится нецелесообразным, и ее следует свернуть. *Сворачивание* (phasing out) обычно осуществляется не потому, что программное обеспечение теряет свою *полезность* (usefullness). Вполне возможно, что оно по-прежнему остается вполне пригодным для использования, однако становится непригодным для сопровождения. Шах (Schach, 2005) приводит четыре причины сворачивания программного обеспечения.

- Предлагаемые изменения выходят далеко за рамки ближайших возможностей улучшающего сопровождения.
- Система выходит из-под контроля служб сопровождения, и последствия изменений невозможно предвидеть.
- Расширение программного обеспечения в будущем невозможно из-за отсутствия надлежащей документации.
- Аппаратная и/или программная платформы, на которых реализована система, подлежат замене, а видимых путей для миграции нет.

1.4.3. Действия, выполняемые на протяжении всего жизненного цикла

Некоторые эксперты и авторы книг включают в проект этапы *планирования* (planning) и *тестирования* (testing). Однако эти действия нельзя связать с *отдельными* этапами, поскольку они охватывают *весь* жизненный цикл.

План управления проектом по разработке программного обеспечения создается в самом начале процесса, значительно уточняется после этапа спецификации и совершенствуется на протяжении всего оставшегося жизненного цикла. Аналогично, тестирование становится наиболее интенсивным после реализации, но применяется и на ранних этапах для проверки артефактов программного обеспечения.

Прогресс проекта отслеживается по плану. Этот процесс связан с другими действиями, выполняемыми на протяжении всего жизненного цикла проекта, — *сбором показателей*.

1.4.3.1. Планирование проекта

Известное изречение гласит: "То, что нельзя спланировать, нельзя и осуществить". Планирование охватывает весь жизненный цикл программного проекта. Оно начинается после того, как в результате работ по *системному планированию* определена бизнес-стратегия организации и обозначен программный проект. **Планирование проекта** (project planning) — это деятельность, направленная на оценку комплекта поставки, затрат, рисков, этапов и требуемых ресурсов. Оно также включает выбор методов разработки, процессов, средств, стандартов, бригадной организации и т.д.

Проектные планы подобны ускользающей цели. Они меньше всего напоминают что-то, раз и навсегда заданное и неизменное. Проектные планы подвержены изменениям на протяжении всего жизненного цикла. При этом данные изменения не выходят за рамки, устанавливаемые несколькими *фиксированными ограничениями* (fixed requirements).

В качестве типичных ограничений выступают *время* и *деньги* — каждый проект имеет четкий конечный срок и строго ограниченный бюджет. Одна из первых задач проектного планирования состоит в оценке осуществимости проекта в условиях временны*х*, бюджетных и прочих ограничений. Если проект осуществим, то ограничения фиксируются документально и могут быть изменены только в рамках формальной процедуры утверждения.

Осуществимость проекта оценивается с учетом нескольких факторов (Hoffer et al., 2002; Whitten and Bertley, 1998).

- *Практическая осуществимость* (operational feasibility) повторно поднимает вопросы, впервые изученные при *планировании системы*, когда были очерчены контуры проекта. Она связана с изучением того, как предлагаемая система повлияет на организационные структуры, процедуры и людей.
- *Экономическая осуществимость* (economical feasibility) связана с оценкой затрат на проект и приносимых им выгод. (Эта процедура известна также как анализ затрат и результатов (cost-benefit analysis).)
- *Техническая осуществимость* (technical feasibility) связана с оценкой практичности предлагаемых технических решений и наличия необходимых навыков, опыта и ресурсов.
- *Календарная осуществимость* (schedule feasibility) связана с оценкой обоснованности расписания проекта.

Не все ограничения известны или могут быть оценены во время открытия проекта. На этапе выработки требований выявляются дополнительные ограничения, которые должны подвергаться изучению с точки зрения их влияния на осуществимость проекта. К подобным ограничениям можно отнести юридические, договорные, политические ограничения и ограничения, связанные с безопасностью.

С учетом оценки осуществимости составляется *план проекта* и устанавливаются правила управления проектом и процессом. В плане проекта находят отражение следующие вопросы (Whitten and Bentley, 1998).

- Рамки проекта.
- Проектные задания.
- Управление и контроль проекта.
- Управление качеством.
- Показатели и измерения.
- Расписание проекта.
- Распределение ресурсов (людских, материальных, инструментальных).
- Руководство людьми.

1.4.3.2. Показатели

Измерение времени и усилий, затраченных на проект, а также принятие на вооружение других *показателей* (metrics), характеризующих артефакты проекта, в действительности являются важной частью *управления проектом и процессом*. Несмотря на важность этой части, в организациях с низким уровнем технологической зрелости ею часто пренебрегают. Цена этого вопроса высока. Не "измеряя" прошлого, организация не в состоянии точно планировать будущее.

Показатели обычно рассматриваются в контексте *качества* и *сложности* программного обеспечения — они применяются в отношении качества и сложности *программного продукта* (Fenton and Pfleeger, 1997; Henderson-Sellers, 1996; Pressman, 2005).

Показатели используются для измерения таких характеристик качества, как корректность, надежность, продуктивность, целостность, практичность, пригодность к сопровождению, гибкость и тестируемость. Например, надежность программного обеспечения можно оценить с помощью измерения частоты и серьезности отказов, среднего времени между отказами, точности выходных результатов, восстанавливаемости после отказов и т.д.

Другим важным применением показателей является оценка моделей разработки (*продуктов разработки*) на различных этапах жизненного цикла программного обеспечения. Кроме того, показатели используются для оценки эффективности процесса и повышения качества работы на различных этапах жизненного цикла.

Типичные показатели, которые применяются к *процессу создания* программного обеспечения и могут быть приняты к использованию на различных этапах жизненного цикла, приведены ниже (Schach, 2005).

- Изменчивость требований. Процент требований, претерпевающих изменения до завершения этапа спецификации требований. Этот показатель может отражать трудность получения требований от заказчиков.

- Изменчивость требований после этапа спецификации требований. Этот показатель может указывать на низкое качество документального оформления требований.
- Прогнозирование проблемных мест системы. Частота, с которой пользователи пытаются выполнить различные функции с помощью прототипа программного продукта.
- Объем документов, содержащих спецификации, которые генерируются CASE-средствами, и другие более детализированные показатели, взятые из репозитория CASE-системы, например количество классов в модели классов. Если их применить к нескольким предыдущим проектам, затраты и время выполнения которых известны, то эти показатели способны обеспечить идеальную плановую "базу данных" для прогнозирования времени и усилий, необходимых для будущих проектов.
- Фиксирование статистики отказов, времени их появления, обнаружения и устранения. Подобная статистика может отражать доскональности системы обеспечения качества, процессов пересмотра и деятельности по тестированию программного обеспечения.
- Среднее число тестов, после проведения которых тестируемый компонент считается готовым для интеграции поставки заказчику. Этот показатель может отражать уровень процедур отладки, используемых программистами.

1.4.3.3. Тестирование

Подобно планированию или измерению характеристик программного обеспечения, *тестирование* (testing) — это деятельность, охватывающая весь жизненный цикл программного обеспечения. Тестирование — это не просто отдельный этап жизненного цикла, следующий за реализацией. После того как дело зашло уже довольно далеко и программный продукт реализован, приступать к тестированию слишком поздно. Исправление ошибок, допущенных на ранних этапах жизненного цикла, обходится безмерно дорого (Schach, 2005).

Тестирование должно быть тщательно спланированным. Планирование процесса тестирования начинается с установления перечня *тестовых прецедентов* (test cases). Тестовые прецеденты (или планы теста) определяют шаги тестирования, которые необходимо предпринять, чтобы попытаться "сломать" программную модель или продукт.

Тестовые прецеденты должны быть определены для каждого функционального модуля (*прецедента использования*), описанного в виде документально оформленных требований. Соотнесение тестовых прецедентов с прецедентами использования устанавливает траекторию *соподчинения* (traceability) между тестами и требованиями пользователей. Тестируемость программного артефакта определяется, в частности, его соподчиненностью. Для того чтобы программное обеспечение было тестируемым, необходимо, чтобы оно было соподчиненным.

Вполне естественно, что каждый разработчик тестирует результаты своего труда. Однако от авторов программных артефактов трудно ожидать непредвзятого отношения к результатам своей работы. Для более эффективного тестирования требуется привлечение независимых (по крайней мере, относительно) посредников, которые могут провести непредвзятое тестирование. Эту задачу можно поручить подразделению, занимающемуся поддержкой качества программного обеспечения (software quality assurance group — группа гарантии качества программного обеспечения). В это подразделение должны входить лучшие разработчики организации. Их задача заключается в тестировании, а не в разработке. Именно на эту группу (а не на разработчиков) возлагается ответственность за качество программного продукта.

Чем больший объем тестирования выполнен на ранних этапах разработки, тем больше выигрыш. Требования, спецификации и любые другие документы (включая тексты программ) должны быть протестированы с использованием *формальных экспертиз* (formal review) (*сквозного контроля* (walkthrough) и *инспекции* (inspection)).

Формальные экспертизы — это тщательно подготовленные заседания, целью которых является анализ определенной части документации или системы. Специально назначенный эксперт заранее изучает документ и формулирует различные вопросы. Заседание решает, действительно ли проект имеет недостаток, но не делает при этом никаких попыток предложить немедленное решение проблемы. Позднее разработчик попытается исправить выявленный дефект. При условии, что заседание проходит в дружественной атмосфере, а участники избегают "указывать пальцем" на виновников ошибок, совместные усилия приводят к раннему обнаружению и исправлению многих ошибок.

После создания программных прототипов и первых версий программного продукта осуществляется *тестирование, основанное на выполнении программы* (execution-based testing). Существуют два основных вида тестирования, основанного на выполнении программы.

- Тестирование по спецификации, или тестирование методом "черного ящика".
- Тестирование по коду, или тестирование методом "прозрачного ящика".

При *тестировании по спецификации* (testing to specification) сама программа рассматривается как "черный ящик", о котором ничего не известно, за исключением того, что он получает некоторую информацию на входе и вырабатывает некоторую информацию на выходе. На вход программы подаются некоторые входные данные, а результирующие данные анализируются на предмет наличия ошибок. Тестирование по спецификации особенно полезно для выявления некорректных или упущенных требований.

При *тестировании по программному коду* (testing to code) программная логика подвергается "сквозному просмотру" с целью установить, какие данные необходимо подать на вход, чтобы испытать различные выполняемые ветви программы.

Тестирование по коду дополняет тестирование по спецификации — эти два вида тестирования направлены на выявление различных категорий ошибок.

Поступательная разработка предполагает не только поэлементную компоновку программных модулей, но и *поступательное*, или *регрессионное тестирование* (regression testing). Регрессионное тестирование представляет собой повторное выполнение предыдущих тестовых прецедентов на том же *базисном наборе данных* (baseline data set) после расширения возможностей ранее выпущенных программных модулей. Тестирование проводится в предположении, что прежние функциональные возможности должны остаться неизменными и не должны быть нарушены за счет расширения.

Неплохим инструментом поддержки регрессионного тестирования могут служить *средства записи-воспроизведения* (captur-playback tools), позволяющие зафиксировать взаимодействие пользователя с программой, а затем воспроизвести их без дальнейшего вмешательства пользователя. Основная трудность регрессионного тестирования заключается в нестабильности базисного набора данных. Дело в том, что поступательная разработка не только расширяет процедурную логику программы, но и модифицирует основные структуры данных. Программный продукт с расширенными возможностями может заставить изменить базисный набор данных, таким образом лишая смысла сравнение результатов.

Контрольные вопросы 1.4

КВ1. Какой подход к разработке программного обеспечения — структурный или объектно-ориентированный — дает преимущество при функциональной декомпозиции?

КВ2. Как еще называется бизнес-анализ?

КВ3. На каком этапе разработки выполняется основная работа, связанная с производством/поставкой адаптивной системы?

КВ4. С каким этапом разработки связано понятие программы-заглушки?

КВ5. Какие виды деятельности охватывают весь жизненный цикл проекта, а какие ограничены отдельными этапами?

1.5. Модели и методы разработки программного обеспечения

Модели и методы разработки изучают вопросы, связанные с производством программного обеспечения. Аналогично *моделям и процессам жизненного цикла* (раздел 1.1.2.2) модели и методы разработки представляют собой концепцию,

а не стандарт, используемый для аккредитации организаций (такой как CMM, ISO 9000 или ITIL), и не среду (как COBIT).

Отсюда следует, что организации могут самостоятельно изобретать собственные процессы жизненного цикла, смешивая элементы разных моделей и методов. Помимо всего прочего, процесс разработки должен отражать уникальный характер каждой организации. Он является следствием социальных, культурных, организационных факторов, внешней среды и других подобных обстоятельств. Кроме того, процесс может изменяться от проекта к проекту с учетом размера, предметной области, требуемых программных инструментов и т.п.

Современные процессы разработки носят *итеративный и поступательный характер* (см. раздел 1.1.2.2.1). Проект разработки системы состоит из многих итераций. Каждая итерация завершается созданием расширенной (улучшенной) версии продукта. Итерации и расширения не выходят за рамки системы, а это значит, что каждое новое расширение обычно добавляет к существующей версии новые функциональные свойства (например, новую подсистему). Расширение версии улучшает функциональность, легкость использования, производительность и другие качества системы, не изменяя ее масштаба. Добавление нового функционального свойства к существующей системе осуществляется путем *интеграции* программного обеспечения (см. раздел 1.1.3), которая сама по себе является итеративной и поступательной.

Итеративную и поступательную разработку можно описать с помощью различных моделей и методов. Ниже перечислены наиболее распространенные модели и методы (Maciaszek and Liong, 2005).

- Спиральная модель.
- Унифицированный процесс RUP (IBM Rational Unified Process).
- Архитектура, управляемая моделями.
- Ускоренная разработка программного обеспечения.
- Аспектно-ориентированная разработка.

1.5.1. Спиральная модель

Спиральная модель (Boehm, 1988) является базовой для всех итеративных и поступательных процессов разработки программного обеспечения. Эта модель была предложена в эпоху доминирования структурного подхода (см. раздел 1.4.1.1), но, будучи по существу мета-моделью, она точно так же может быть использована в рамках объектно-ориентированного подхода (см. раздел 1.4.1.2). Спиральная модель (spiral model) рассматривает деятельность, связанную с разработкой программного обеспечения, в более широком контексте планирования системы, анализа рисков и пользовательских оценок. Эти четыре вида деятельности изображены в виде квадрантов декартовой системы координат и спирали (рис. 1.4).



Рис. 1.4. Спиральная модель

В соответствии со спиральной моделью разработка системы начинается с этапа *планирования* (planning), на котором оценивается осуществимость проекта и собираются первичные требования. Кроме того, на этом этапе составляется календарный план и смета проекта.

Затем спираль разработки переходит в квадрант *анализа риска* (risk analysis), в котором оценивается величина рисков, которым подвергается разработка. *Рисками* называются любые неблагоприятные обстоятельства и неопределенности, которые могут повлиять на проект. На этапе анализа рисков оцениваются потенциальные результаты проекта, а также сравниваются приемлемые уровни риска и соответствующие вероятности успеха. На этапе анализа рисков принимается решение продолжать или не продолжать разработку. Это решение зависит исключительно от оценки рисков и ориентировано на будущее (т.е. оно не должно зависеть от таких факторов, как стоимость проекта и т.п.).

Квадрант *конструирование* (engineering) связан с конкретным созданием системы. Прогресс проекта измеряется именно по результатам, полученным на этом этапе. Конструирование включает в себя все виды моделирования, программирования, интеграции и развертывания системы.

Прежде чем перейти к следующему квадранту, необходимо собрать *пользовательские оценки* (customer evaluation). Для этого используется формальный процесс, с помощью которого осуществляется обратная связь с потребителями. Эти оценки сравниваются с требованиями, которым должна удовлетворять система. Кроме того, переходу на новый этап планирования должен предшествовать сбор всей информации, поступающей от пользователей.

1.5.2. Унифицированный процесс RUP

Унифицированный процесс RUP (IBM Rational Unified Process) представляет собой платформу для разработки программного обеспечения (RUP, 2003). Эта платформа образует среду, состоящую из справочной и обучающей документации, хороших практичных шаблонов, методов, основанных на использовании сети Web, и т.д. Процесс RUP организует проекты в двумерной системе координат. На горизонтальной оси изображаются последовательные *стадии* каждой итерации. Процесс RUP состоит из четырех стадий — начало, проектирование, конструирование и внедрение. Вертикальная ось соответствует *дисциплинам*, связанным с проектированием программного обеспечения. К ним относятся бизнес-моделирование, требования, анализ и проектирование, реализация, тестирование, развертывание, сопровождение, управление конфигурацией, изменениями, проектом и средой. Эти дисциплины описывают основные части проекта, или потоки работы.

Разделение проекта на стадии и дисциплины подразумевает измерение количественных показателей, одновременно порождая и решая множество проблем. Цель этого процесса — продемонстрировать взаимосвязь двух размерностей: горизонтальной и вертикальной (динамической и статической). Динамическая размерность символизирует прогресс процесса в виде итераций и контрольных отметок. Статическая размерность соответствует видам деятельности и артефактам, создаваемым в ходе процесса.

На практике четкое разделение между горизонтальной и вертикальной размерностями проекта осуществить довольно трудно. Часто возникают вопросы наподобие такого: “В чем заключается разница между конструированием и реализацией, а также между внедрением и развертыванием?” Для того чтобы не возникало недоразумений, на рис. 1.5 дисциплины процесса RUP, соответствующие вертикальной размерности, изображены в виде круговой диаграммы. Горизонтальная размерность на этом рисунке не показана, но подразумевается. Например, начальная стадия доминирует на стадии бизнес-моделирования, но отсутствует на этапе развертывания. С другой стороны, стадия развертывания занимает значительное место в процессе развертывания, но не используется для бизнес-моделирования.

Подобно спиральной модели, процесс RUP делает акцент на итеративной разработке и важности раннего и непрерывного анализа рисков. Для процесса RUP характерен частый выпуск выполняемых версий программного обеспечения. Процесс RUP подразумевает настройку на конкретный проект. Настройка означает, что для каждой организации, задачи, коллектива и даже отдельных проектировщиков можно выбирать разные компоненты процесса RUP. Таким образом, процесс RUP носит универсальный характер, но тем не менее позволяет учесть специфику проекта.



Рис. 1.5. Унифицированный процесс (воспроизводится по справочнику Rational Suite Tutorial 2002, © 2002 International Business Machines Corporation)

1.5.3. Архитектура, управляемая моделями

Архитектура, управляемая моделями (model-driven architecture — MDA) (Kleppe et al., 2003; MDA, 2006), — это довольно старая идея, возрожденная в новой форме. Она восходит к концепции формальных спецификаций и моделей трансформации (Ghezzi et al., 2003). Архитектура MDA представляет собой среду моделирования и генерирования программ на основе спецификаций.

Архитектура MDA использует различные стандарты группы OMG и позволяет создавать как платформно-независимые, так и платформно-зависимые модели системы. Стандарты, допускающие такие спецификации, приведены ниже.

- Язык UML (Universal Modeling Language), предназначенный для моделирования задач.
- Модель MOF (Meta-Object Facility), позволяющая использовать репозиторий мета-моделей с целью согласования генерируемых спецификаций.
- Стандарт XMI (XML Meta-Data Interchange), предназначенный для обмена моделями языков UML и XML.
- Модель CWM (Common Warehouse Meta-model), предназначенная для отображения архитектуры, управляемой моделями, в схемы баз данных и проведения гибкого интеллектуального анализа.

Архитектура, управляемая моделями, нацелена на вывод платформно-независимых моделей, включающих в себя полные спецификации состояния и поведения системы. Это позволяет отделить логику бизнес-приложений от технологических изменений. На следующем этапе архитектура MDA предоставляет в распоряжение разработчиков инструменты и методы для создания платформно-зависимых моделей, предназначенных для реализации в таких средах, как J2EE, .NET и Web Services.

На рис. 1.6 показано, как концепции архитектуры, управляемой моделями, связаны с тремя основными этапами разработки — анализом, проектированием и реализацией. "Мостики", созданные с помощью платформно-зависимых моделей и программных модулей, взаимодействуют друг с другом, позволяя разрабатываемой системе охватывать разные платформы.

В качестве естественного следствия выполняемого моделирования архитектура MDA учитывает *технологию компонентного программного обеспечения* (component technology). Компоненты сначала определяются в рамках платформно-независимых моделей, а затем реализуются в виде платформно-зависимых модулей. Группа OMG использует архитектуру MDA для создания трансформируемых моделей и повторно используемых компонентов, представляющих собой стандартное решение для вертикально организованных операций, выполняемых, например, в системах телекоммуникации или больницах.

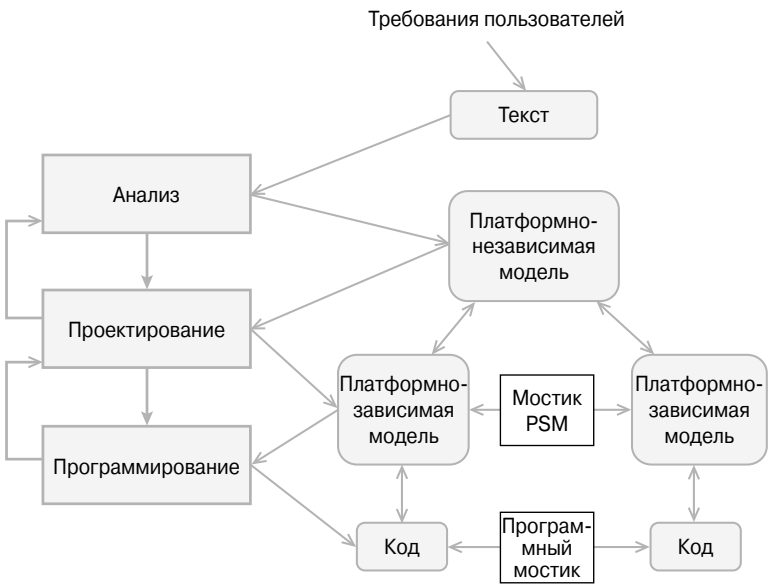


Рис. 1.6. Архитектура, управляемая моделями

1.5.4. Ускоренная разработка программного обеспечения

Ускоренная разработка программного обеспечения (agile software development) представляет собой относительно новое усовершенствование итеративных и поступательных моделей. Эта концепция распространяется некоммерческой организацией Agile Alliance (Agile, 2006). Ускоренная разработка трактует изменения как внутренние аспекты проектирования программного обеспечения и предлагает "облегченные" методы для реагирования на изменения требований, выводя на передний план разработки вопросы программирования.

В документе "Manifesto for agile software development" организация Agile Alliance сформулировала основные принципы ускоренной разработки.

- Люди и их взаимодействие важнее, чем процессы и средства.
- Работающее программное обеспечение важнее, чем исчерпывающая документация.
- Сотрудничество с заказчиком важнее, чем обсуждение условий контракта.
- Реагирование на изменения важнее, чем следование плану.

Ускоренная разработка представляет собой итеративный и поступательный процесс, направленный на замену формальностей частыми поставками заказчикам выполняемых программ. Эта особенность ускоренной разработки отражена в ее терминологии. Названия этапов жизненного цикла — анализ, проектирование, реализация и развертывание — заменяются новыми терминами: *пользовательские истории* (user stories), *приемочные тесты* (acceptance tests), *рефакторинг* (refactoring), *разработка через тестирование* (test-driven development) и *непрерывная интеграция* (continuous integration) (рис. 1.7). Более внимательное изучение показывает, что технологические изменения не отменяют того факта, что ускоренная разработка хорошо сочетается с более основательными итеративными и поступательными процессами.

Пользовательские истории в рамках ускоренной разработки соответствуют анализу требований в других моделях. Этот подход требует, чтобы список историй и описания мнений пользователей о свойствах системы уточнялись на протяжении всего проекта. Эти истории используются для планирования расписания и сметы итераций разработки.

Ускоренная разработка заменяет моделирование и реализацию проекта циклом приемочных тестов, рефакторингом и разработкой через тестирование. *Приемочные тесты* представляют собой спецификации программ, которым должна удовлетворять прикладная программа, подвергающаяся тестированию, чтобы обеспечить выполнение требований пользователей. Вследствие этого реализация этих программ осуществляется через тестирование. Программы пишутся так, чтобы они проходили приемочные тесты. Этот процесс называется *разработкой через тестирование* и порождает так называемое *целенаправленное программирование* (intentional programming) — способность и возможность описывать цель программы, проходящей приемочный тест еще до ее кодирования.

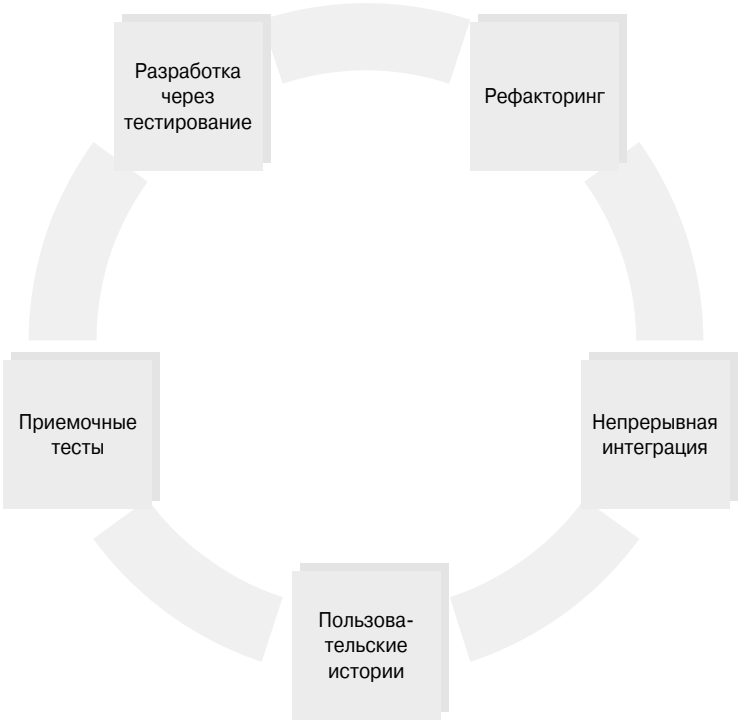


Рис. 1.7. Ускоренная разработка программного обеспечения

Разработка через тестирование проводится парами программистов. Все программирование осуществляется двумя программистами, которые используют для тестирования одну и ту же рабочую станцию, обмениваются идеями и проводят немедленную верификацию концепций с другими людьми. Кроме того, *парное программирование* (pair programming) порождает *коллективное владение* (collective ownership), при котором ни один человек не является единоличным владельцем кода и всегда существует второй человек, понимающий ранее написанный код.

Ускоренная разработка основывается на *рефакторинге*, представляющем собой вид деятельности, направленный на улучшение кода путем реструктуризации (пересмотра архитектуры) без изменения его поведения. Рефакторинг основан на предположении, что исходная архитектура является устойчивой и гибкой. Кроме того, он предполагает, что программирование осуществляется с помощью хорошо зарекомендовавших себя практических методов, проектных схем и шаблонов реализации.

Каждая итерация ускоренной разработки планируется как *короткий цикл*, продолжительность которого не должна превышать двух недель. Короткие циклы означают необходимость *непрерывной интеграции* нового кода с ранее существующими программами. Код, полученный после двухнедельной интеграции,

называется *промежуточной поставкой* (minor delivery), предназначенной для оценки пользователями. *Основная поставка* (major delivery) продукции обычно планируется после трех коротких циклов, т.е. через шесть недель.

Ускоренные методы значительно обогатили итеративный и поступательный процесс разработки. Существует множество вариантов и приемов, которые либо попадают в категорию ускоренной разработки, либо легко сочетаются с методами ускоренной разработки. В частности, к ним относятся перечисленные ниже методы.

Экстремальное программирование (XP) (Beck, 1999; Extreme, 2006).

- Разработка, управляемая свойствами (Feature, 2006).
- Упрощенная разработка (Poppendieck and Poppendieck, 2003).

Существуют сомнения, что методы ускоренной разработки можно применить для создания крупных и очень крупных систем. По-видимому, они больше подходят для организации работы небольших коллективов, состоящих из не более чем пятидесяти человек, тесно сотрудничающих друг с другом, нацеленных на результат и обращающих мало внимания на планы, формальности, требования менеджеров и даже условия контракта. Было бы странно переносить эти характеристики в коллективы, занимающиеся разработкой крупных промышленных систем. Такие проекты обычно выполняются на основе формальных и хорошо документированных стандартов и инструментов (см. раздел 1.1.2.2).

1.5.5. Аспектно-ориентированная разработка программного обеспечения

Аспектно-ориентированное программирование (aspect-oriented programming — AOP) (Kiczales et al., 1997) не является революционной идеей — к нему относится лишь несколько действительно полезных идей. Основные концепции, лежащие в основе аспектно-ориентированного программирования, хорошо известны и широко использовались в прошлом, часто под другими названиями и в сочетании с другими технологиями. Основная цель технологии AOP — создание модульных систем на основе *комплексных задач* (crosscutting concerns) и отдельных программных модулей, реализующих эти свойства. Эти модули называются *аспектами* (aspects). Эти аспекты интегрируются вместе с помощью процесса, называемого *сплетением аспектов* (aspect weaving).

Отправной точкой аспектно-ориентированного программирования является понимание того, что программная система состоит из многих вертикальных модулей. Опорные модули содержат компоненты программного обеспечения, реализующие функциональные требования к системе. Однако каждая система должна также удовлетворять нефункциональным требованиям, определяющим такие качества программного обеспечения, как корректность, надежность, безопасность, производительность, параллельность и др. Этими качествами должны обладать

разные компоненты системы (и даже большинство из них), реализующие ее функциональные свойства. В "традиционном" объектно-ориентированном программировании код, обладающий этими качествами, должен быть дублирован (разбросан) во многих компонентах. Эти нефункциональные качества известны в аспектно-ориентированном программировании под названием *задачи* (concerns) — цели, которые должно достигать приложение. Поскольку объектно-ориентированная реализация этих задач требует охвата многих компонентов в один комплекс, они называются *комплексными задачами* (crosscutting concerns).

Для того чтобы избежать дублирования кода при реализации комплексных задач, аспектно-ориентированное программирование предлагает объединять эти фрагменты в отдельных аспектах. Несмотря на то что аспекты, как правило, реализуют нефункциональные требования, в целом они могут образовывать основу функциональной декомпозиции системы. В частности, они могут реализовывать разнообразные бизнес-правила, согласованные с классами, лежащими в основе логики бизнес-приложения.

Таким образом, аспектно-ориентированное программирование осуществляет декомпозицию систем на аспекты, построенные вокруг основных функциональных компонентов, получивших название *базовый код* (base code). Аспекты содержат отдельный *аспектный код* (aspect code). Для того чтобы такая система работала, аспекты должны образовывать логический поток программы. Этот процесс композиции программного обеспечения называется *сплетением аспектов* (aspect weaving). Одни аспекты могут сплетаться на этапе компиляции (*статическое сплетение*), а другие — на этапе выполнения программы (*динамическое сплетение*).

Сплетение аспектов применяется к *точкам соединения* (joint points) в ходе выполнения программы. Точками соединения являются заранее установленные точки композиции программного обеспечения, например точка вызова метода, точка доступа к атрибуту, точка создания экземпляра объекта, точка генерации исключения и т.д. Конкретное действие, которое должно быть выполнено в точке соединения, называется *уведомлением* (advice). Например, уведомлением является проверка прав пользователя или начало новой транзакции. Уведомление может быть выполнено как до достижения точки соединения (*заблаговременное уведомление* (before advice)), так и после (*последующее уведомление* (after advice)). Кроме того, уведомление может заменять точку соединения (*сквозное уведомление* (around advice)).

Одно и то же уведомление может применяться ко многим точкам соединения программы. Множество точек соединения, связанных с одним и тем же уведомлением, называется *срезом точек соединения* (pointcut). Эти срезы часто определяются программным путем с помощью подстановок или регулярных выражений. Кроме того, возможны (и даже желательны) композиции срезов точек соединения.



Рис. 1.8. Аспектно-ориентированная разработка программного обеспечения

Подобно ускоренной разработке, аспектно-ориентированная разработка программного обеспечения сосредоточивает внимание на задачах программирования и вводит новые термины (рис. 1.8). Как и в рамках ускоренной разработки, более внимательное изучение терминологии показывает, что аспектно-ориентированная разработка является новым способом применения итеративного и поступательного процесса для создания адаптивного программного обеспечения.

Несмотря на все преимущества технологии AOP, позволяющие улучшать модульность (а значит, и адаптивность) программного обеспечения, она может быть источником потенциальной опасности и даже причиной неправильного поведения (см., например, работу Murphy and Schwanninger, 2006). Это объясняется тем, что аспекты неявно модифицируют поведение точек соединения и могут исказить функциональную логику приложения. Более того, аспекты сами по себе не всегда являются независимыми и могут скрыто влиять друг на друга, порождая опасное поведение.

Совершенно очевидно, что практика аспектно-ориентированной разработки должна гарантировать согласованную модификацию кода аспекта и базового кода, точное документирование задач и постоянное информирование разработчиков приложения о внесенных изменениях. Эту дилемму особенно сложно

решить при наличии динамического сплетения (см., например, работу Hirschfeld and Hanenberg, 2005). Необходимым условием успешного решения этой проблемы является осведомленность разработчиков о взаимном влиянии изменений в базовом и аспектном коде.

Контрольные вопросы 1.5

КВ1. К какому действию относится добавление нового функционального свойства — к итерации или интеграции?

КВ2. Какие модели и методы разработки должны в первую очередь анализироваться с точки зрения риска?

КВ3. Какие модели и методы разработки непосредственно связаны с традиционной концепцией формальных спецификаций?

КВ4. Какие модели и методы разработки непосредственно связаны с концепцией целенаправленного программирования?

КВ5. Какие модели и методы разработки непосредственно связаны с концепцией комплексных задач?

1.6. Учебные примеры

Кроме учебного примера ("Электронный магазин"), помещенного в отдельной главе в конце книги, приводятся семь учебных примеров, иллюстрирующих концепции разработки и методы моделирования. На первый взгляд эти примеры могут дать лишь приблизительное представление о процессе разработки программного обеспечения, но читатели должны постоянно возвращаться к этим примерам во время чтения последующих глав.

Способы представления всех учебных примеров примерно одинаковы: сначала формулируется задача, а затем описывается способ его решения. Это позволяет читателю попытаться найти собственное решение и сравнить его с предложенным. В книге рассматриваются следующие учебные примеры.

- "Зачисление в университет".
- "Магазин видеокассет".
- "Управление взаимоотношениями с заказчиками".
- "Прямой маркетинг по телефону ".
- "Затраты на рекламу".
- "Регистрация времени".
- "Конвертация валют".

1.6.1. "Зачисление в университет"

"Зачисление в университет" является классическим учебным примером (Quatrani, 2000; Stevens and Pooley, 2000). Эта задача отличается неожиданно высокой сложностью из-за большого количества часто изменяющихся бизнес-правил и необходимости хранить архивные данные, соответствующие бизнес-правилам, действовавшим в разное время.

Все университеты отличаются друг от друга. Каждый из них обладает интересными особенностями, которые можно использовать в учебных примерах для демонстрации особенностей анализа и проектирования систем. В нашем учебном примере основное внимание уделяется сложностям моделирования состояний и учета временной составляющей (временной информации), а также хранению бизнес-правил в структурах данных.

Пример 1. "Зачисление в университет"

Университет среднего размера имеет много вакансий для зачисления студентов и аспирантов на стационарную и заочную форму обучения. Учебная часть университета состоит из нескольких отделений. Отделения состоят из нескольких факультетов. Каждое отделение проводит обучение по определенной специальности, причем это обучение может включать в себя курсы, которые преподают на других отделениях. Фактически университет предоставляет своим студентам свободу выбора курсов в рамках получаемой специальности.

Гибкость выбора курсов создает большие сложности при создании системы, моделирующей зачисление абитуриентов в университет. Индивидуальные программы не должны противоречить правилам, регламентирующим обучение по конкретным специальностям, например, эти программы должны содержать обязательные курсы. Выбор курсов может ограничиваться расписанием, максимальной вместимостью аудиторий и другими факторами.

Гибкость образования, предлагаемого университетами, является основной причиной их постоянно возрастающей популярности, выражающейся в росте количества абитуриентов. Однако, для того чтобы не снижать качество образования, ныне существующую систему управления зачислением в университет, частично предполагающую ручной труд, необходимо заменить новой автоматизированной системой. Предварительный поиск готового программного обеспечения не привел к успеху. Система управления зачислением в университет носит настолько уникальный характер, что ее разработку следует поручить внутренним подразделениям.

Система должна поддерживать работу, связанную с подготовкой к зачислению, а также выполнять собственно процедуры зачисления в университет. Предварительная работа предполагает рассылку студентам результатов их последних экзаменов, а также соответствующих инструкций. На протяжении периода зачисления система должна принимать программы курсов, предлагаемые студентами, оценивать их соответствие правилам, расписанию, вместимости аудиторий, особым ограничениям и т.д. Для решения этих проблем может понадобиться консультация с университетскими менеджерами и профессорами, читающими курсы.

1.6.2. "Магазин видеокассет"

Второй учебный пример посвящен рутинному приложению, типичному для малого бизнеса. В нем описываются операции, осуществляемые в магазине видеокассет. Такой магазин обычно хранит множество видеокассет развлекательного характера. Основные операции в этом магазине относятся к выдаче кассет напрокат.

Обычную компьютерную систему, предназначенную для ведения операции в небольшом магазине видеокассет, можно создать с помощью настройки готового программного обеспечения или другой существующей системы. Эта система должна быть основана на одной из распространенных систем управления базами данных, которые можно развернуть на персональных компьютерах.

Систему управления базой данных можно развернуть на одном компьютере, снабдив ее графическим пользовательским интерфейсом на основе простого языка программирования четвертого поколения (4GL), обеспечивающим вывод информации на экран, генерирование кода и доступ к простой базе данных.

Отличительной особенностью магазина видеокассет является цепочка интенсивных операций — от упорядочения фильмов, хранящихся на складе, до учета кассет, выданных напрокат или проданных покупателям. Эти операции описываются в рамках модели *цепочек начисления стоимости* (value chain model) (см. раздел 1.2.2).

Пример 2. "Магазин видеокассет"

Новый магазин видеокассет предлагает прокат и продажу развлекательных материалов. Для поддержки этих операций менеджеры решили создать компьютерную систему. На рынке существует большое количество готовых программ, допускающих настройку и расширение. Для того чтобы правильно выбрать пакет программ, магазин нанимает бизнес-аналитика, в обязанности которого входит определение и формулировка требований.

Магазин будет хранить запас видеокассет, а также компакт-дисков (игры и музыка) и DVD. На складе уже хранится упорядоченный запас кассет и дисков, поступивших от одного поставщика, но в будущем количество поставщиков будет увеличиваться. Все видеокассеты и диски снабжены штрих-кодами, поэтому при их продаже и выдаче напрокат можно использовать сканер. Карточки членов видеоклуба также имеют штрих-коды.

Покупатели смогут оставлять заказы на развлекательные материалы. Эти заказы будут выполняться к заданной дате. Система должна иметь гибкую поисковую машину, способную обрабатывать заказы, в том числе на видеоматериалы, которых нет на складе.

1.6.3. Управление взаимоотношениями с заказчиками

Задача управления взаимоотношениями с заказчиками относится к наиболее актуальным предметным областям. Часто эту задачу обозначают аббревиатурой *CRM* (contact or customer relationship management). Управление взаимоотношениями с заказчиками является важным компонентом систем *планирования ресурсов предприятий* (enterprise resource planning — EPR). Системы EPR автоматизируют *вспомогательные операции* предприятий. Типичными компонентами систем EPR являются бухгалтерский учет, производство и управление персоналом. Система CRM относится к области управления персоналом.

Системы EPR представляют собой очень крупные масштабируемые решения. Некоторые люди называют их мегапакетами. Вполне естественно, что компонент CRM системы EPR может оказаться очень сложным. Учебный пример описывает лишь небольшую часть проблем, связанных с управлением взаимоотношениями с заказчиками.

Приложения, автоматизирующие управление взаимоотношениями с заказчиками, характеризуются интересными графическими пользовательскими интерфейсами, с помощью которых сотрудники соответствующих отделов или подразделений могут планировать свою деятельность. По существу, такая система представляет собой дневник, в котором регистрируют ежедневные задания и события, а также вносят отметки об их выполнении.

Дневник должен быть ориентирован на базу данных, чтобы допускать динамическое управление расписанием, а также отслеживание задач и событий, связанных со многими сотрудниками. Как и большинство систем управления персоналом, система управления взаимоотношениями с заказчиками требует сложной авторизации, чтобы защитить конфиденциальную информацию от несанкционированного доступа.

Пример 3. "Управление взаимоотношениями с заказчиками"

Компания, которая проводит маркетинговые исследования, имеет базу заказчиков, покупающих у нее аналитические отчеты. Кроме того, некоторые крупные заказчики покупают у компании специализированное программное обеспечение. Впоследствии эти заказчики получают необработанную информацию, на основе которой можно генерировать новый отчет.

Компания постоянно ищет новых заказчиков, даже если новые клиенты интересуются лишь узкоспециализированными маркетинговыми отчетами. Поскольку потенциальные заказчики еще не являются полноценными клиентами, компания предпочитает называть их *контактами*, а систему — системой управления взаимоотношениями с заказчиками (будущими, настоящими и прошлыми).

Новая система управления взаимоотношениями с заказчиками должна разрабатываться внутренним подразделением и быть доступной всем сотрудникам компании, правда, с разными уровнями доступа. Владельцами системы будут сотрудники отдела обслуживания клиентов. Система обеспечит создание и переработку гибких расписаний действий, связанных с контактами. Это позволит находить новых клиентов и укреплять существующие отношения.

1.6.4. Прямой маркетинг по телефону

Многие организации распространяют свои товары и услуги с помощью телемаркетинга, т.е. с помощью непосредственного контакта с клиентами по телефону. Системы телемаркетинга должны поддерживать расписание телефонных звонков, чтобы обеспечивать автоматическое соединение, облегчая общение и запись результатов.

К особенностям систем прямого маркетинга по телефону относится использование базы данных, позволяющей активно работать с расписаниями и динамично изменять их, обеспечивая параллельное общение с несколькими абонентами. Другим интересным аспектом этих систем является способность обеспечивать автоматическое соединение с абонентами.

Пример 4. "Прямой маркетинг по телефону"

Для сбора пожертвований благотворительное общество продает лотерейные билеты. Сбор средств проходит в форме *кампаний*, посвященных определенным целям. Общество хранит список прошлых жертвователей (*спонсоров*). Для каждой новой кампании прямого телефонного маркетинга или рассылки почтовых сообщений адресаты выбираются в первую очередь из этого списка.

Чтобы привлечь новых спонсоров, общество решило применить новые изобретательные приемы. В частности, решено было провести специальную *бонусную кампанию*, предусматривающую награждение спонсоров за оптовые покупки лотерейных билетов, привлечение новых жертвователей и т.д. Общество не планирует случайным образом выбирать потенциальных спонсоров с помощью телефонных справочников или других источников.

Для того чтобы обеспечить эту возможность, общество решило заказать новую систему управления прямым маркетингом по телефону, которая должна поддерживать большое количество одновременных контактов. Эта система должна иметь способность создавать расписание телефонных звонков в соответствии с заранее заданными приоритетами и другими ограничениями.

Система должна устанавливать автоматическое соединение с указанными телефонными номерами. Неудачные попытки соединения должны регистрироваться для последующего повтора. Кроме того, необходимо учитывать обратные звонки спонсоров. Система должна также записывать результаты разговора, включая заказ билетов и любые изменения, касающиеся спонсоров.;

1.6.5. "Затраты на рекламу"

В эпоху глобализации и географической отдаленности покупателей и продавцов продажа товаров и услуг невозможна без крупных расходов на рекламу. Не удивительно, что компании хотят знать, как оптимально использовать рекламный бюджет и сравнить свои расходы с расходами конкурентов. Такую информацию могут продать маркетинговые компании, которые занимаются сбором и анализом данных.

Особенностью систем, анализирующих расходы на рекламу, является необходимость сочетания сбора и хранения данных в оперативной базе данных с пополнением коллекции в хранилище данных. Именно хранилище данных играет центральную роль в анализе собранной информации, на основе которого производятся и продаются отчеты.

Пример 5. "Затраты на рекламу"

Организации, проводящие маркетинговые исследования, собирают данные об эффективности рекламы из разных источников массовой информации (телеканалы, радиостанции, газеты и журналы), а также у рекламных агентов, работающих в кинотеатрах, на улице и в Интернете. Собранные данные можно проанализировать разными способами. Цель таких исследований — оценить эффективность расходов на рекламу. Таким образом, организации необходимо прикладное программное обеспечение, оценивающее эффективность рекламного бюджета.

Эта система позволит создавать две разновидности отчетов, представляемых клиентам организации. Во-первых, клиент может заказать отчет, содержащий оценки ожидаемой отдачи от рекламы (мониторинг рекламной кампании). Во-вторых, клиент может потребовать отчет о конкурентной позиции компании на конкретном рекламном рынке (отчет о расходах). Такой отчет содержит информацию о затратах, которые должен понести рекламодатель, с учетом специфических условий: времени, региона, вида средства массовой информации и т.д.

Создание отчетов о расходах является основной деятельностью маркетинговой организации. Фактически любой клиент, который дает рекламу своей продукции или услуг, может купить такой отчет в форме специального программного обеспечения или в виде бумажного издания. База данных организации содержит информацию об отдельных рекламодателях, рекламных агентствах, средствах массовой информации, консультантах, а также о директорах по продажам и маркетингу, начальниках отделов планирования рекламных компаний и покупателях отчетов.

Организация имеет контрактные соглашения со многими средствами массовой информации на получение регулярных электронных файлов регистрации рекламных объявлений и клипов. Эта информация заносится в базу данных системы и тщательно проверяется — как в автоматическом, так и в ручном режиме. Задача верификации — подтвердить, что все данные, касающиеся рекламы, корректны и не противоречат другим данным. Основной работой на этапе верификации является ручной ввод (мониторинг) в базу данных информации о рекламе, которая не регистрируется в электронных файлах.

После ввода и верификации реклама проходит *валоризацию* (valorization), представляющую собой процесс оценки расходов.

1.6.6. "Регистрация времени"

Существует разница между *прикладным* и *системным* программным обеспечением. Она заключается в том, что системное программное обеспечение занимает специальный рыночный сегмент и открыто продается желающим в большом количестве. Примерами таких систем являются текстовые процессоры, электронные таблицы и системы управления базами данных. Многие из этих систем представляют собой обобщенное решение, полученное для точно определенной прикладной области. Система регистрации времени относится именно к такому виду приложений. Компании могут покупать такие системы, для того чтобы регистрировать время, затрачиваемое их сотрудниками на выполнение различных проектов и задач.

Для регистрации времени необходимо специальное программное обеспечение. Этот инструмент должен привлекать покупателей, быть очень надежным в ис-

пользовании и допускать модернизацию. Это накладывает особые ограничения на графический пользовательский интерфейс, подразумевает строгое тестирование и масштабируемость архитектуры программного обеспечения.

Пример 6. "Регистрация времени"

Перед компанией, производящей программное обеспечение, поставлена задача разработать систему регистрации времени для продажи разным организациям, желающим контролировать производительность труда своих сотрудников. Компания рассчитывает, что это программное обеспечение превзойдет лидера рынка, программу Time Logger компании Responsive Software (Responsive, 2003).

Система регистрации времени позволит сотрудникам делать записи о времени, проведенном в работе над разными проектами и задачами, а также о продолжительности простоев (пауз, обеденных перерывов, выходных и т.д.). Записи можно ввести либо вручную (указывая начало и конец работы), либо с помощью специального регистрирующего устройства. Это устройство связано с компьютерными часами и позволяет сотрудникам регистрировать начало и конец работы простым щелчком на кнопке.

Система позволит идентифицировать клиентов, для которых выполняется работа. Она будет поддерживать функции выставления счета клиенту, выписывания фактуры и отслеживания платежей. Рабочие затраты можно измерить либо с помощью времени, либо с помощью другого фиксированного показателя. Некоторые виды работы, зарегистрированные системой, не подлежат оплате клиентами.

Система регистрации времени позволит формировать отчеты о затратах рабочего времени, в которых указываются различные детали по выбору заказчика.

Система обеспечит легкое изменение сделанных записей, кроме того, будет поддерживать функции сортировки, поиска и фильтрации данных.

1.6.7. "Конвертация валют"

Банки, другие финансовые организации и даже универсальные Web-порталы поддерживают функции конвертации валют для пользователей Интернета. Текущие средства конвертации валют представляют собой Web-приложения, функционирующие как интерактивные калькуляторы, позволяющие перевести суммы, выраженные в одной валюте (и дорожных чеках), в другую. Калькулятор использует текущие валютные курсы, а некоторые из них могут учитывать даже ретроспективную информацию.

Валютный калькулятор представляет собой небольшую утилиту, обладающую широкими функциональными возможностями. Поскольку она невелика, то позволяет подробно проиллюстрировать аспекты ее проектирования и даже показать некоторые фрагменты кода. Так как она является Web-приложением, то позволяет объяснить архитектурные аспекты, используемые клиентским браузером, и способ доступа к базе данных, находящейся на сервере и предназначенной для заполнения полей формы браузера (поддерживаемые валюты и текущие курсы).

Пример 7. "Конвертация валют"

Банку необходимо разместить на своем Web-портале специальный валютный калькулятор, позволяющий его клиентам вычислить, какую сумму они могут получить, конвертировав ее в ту или иную валюту. Обычно такие калькуляторы используют большинство текущих валютных курсов, но приложение должно иметь возможность делать расчеты "задним числом", привязанные к любой дате.

Приложение можно реализовать с помощью одной или двух Web-страниц. Если приложение реализуется на двух страницах, то первая страница должна позволить клиенту ввести сумму конвертируемых денег, выбрать в окнах прокрутки пункты "из" и "в", а затем щелкнуть на кнопке "Вычислить". Вторая страница должна выводить результаты и давать возможность при необходимости вернуться на первую страницу (с помощью кнопки "Повторить").

Если приложение реализуется на одной странице, то форма должна содержать поле для вывода результатов, не допускающее редактирования и даже вначале невидимое. Это поле должно появляться только после того, как пользователь щелкнет на кнопке "Вычислить".

Резюме

В данной главе рассмотрены стратегические вопросы, касающиеся процесса разработки программного обеспечения. Для некоторых читателей материал этой главы значит не больше родительской нотации. Читателям, обладающим некоторым опытом в области разработки программного обеспечения, эта глава могла дать дополнительный интеллектуальный заряд. По отношению ко всем читателям данная глава была задумана как введение к предстоящему более всестороннему обсуждению.

По своей *природе* разработка программного обеспечения ближе к *ремеслу* или даже *искусству*. Результаты программного проекта нельзя полностью предвидеть в самом начале. Основная *непредсказуемая трудность* при разработке программного обеспечения связана с участниками проекта — программный продукт должен

дать им ощутимые выгоды, в противном случае его ждет провал. В "треугольник" факторов, обеспечивающий успех проекта, помимо человеческого фактора входят надежный *процесс*, а также *поддержка языка и средств моделирования*. Процесс усовершенствования моделей и среды разработки включает в себя модель технологической зрелости (Capability Maturity Model), а также стандарты ISO 9000, ITIL и COBIT. UML — это стандартный язык моделирования.

Цель разработки программного обеспечения — предоставить заказчику *действующую* систему. Разработке программного обеспечения предшествует *планирование системы*, в ходе которого определяется, какие продукты будут наиболее *эффективными* для организации. Существуют разные способы планирования системы. В данной главе были рассмотрены четыре известных подхода: SWOT, VCM, BPR и ISA.

Информационные системы создаются для трех *уровней управления*: оперативного, тактического и стратегического. Соответственно программное обеспечение классифицируется как системы обработки транзакций, системы аналитической обработки и системы обработки знаний. Системы, поддерживающие стратегический уровень принятия решений, обеспечивают наибольшую эффективность и являются наиболее сложными для разработки.

В прошлом программные продукты были *процедурными* — запрограммированная процедура выполняла свою задачу более-менее последовательным и предсказуемым образом, после чего завершалась. Для производства подобных систем успешно использовался *структурный подход к разработке*.

Современные программные продукты являются *объектно-ориентированными*, т.е. программа состоит из программных объектов, которые выполняются случайным, непредсказуемым образом, и программа не завершается до тех пор, пока пользователь не прекратит ее выполнение. Объекты "бездействуют", ожидая наступления инициированного пользователем события, чтобы начать вычисление; для выполнения задачи они могут попросить другие объекты предоставить им услуги, после чего снова впадают в "спячку", но сразу "поднимаются по тревоге", стоит только пользователю инициировать другое событие. Современные клиент-серверные приложения информационных систем, разработанные с использованием графического пользовательского интерфейса, являются объектно-ориентированными, и объектно-ориентированный подход к разработке наилучшим образом пригоден для производства таких приложений. Оставшаяся часть книги посвящена объектно-ориентированному подходу.

Разработка программного обеспечения проходит определенный *жизненный цикл*, к основным этапам которого относятся анализ, проектирование, реализация, компоновка и развертывание, а также эксплуатация и сопровождение. Некоторые важные виды деятельности охватывают весь жизненный цикл. К ним относятся планирование, измерения и тестирование.

В этой книге наибольшее внимание уделяется двум этапам жизненного цикла: *анализу* и *проектированию*. Другие этапы включают реализацию, компоновку и сопровождение. Разработка программного обеспечения подразумевает и некоторые другие виды деятельности, такие как планирование проекта, сбор данных измерений, тестирование и управление изменениями. Эти виды деятельности не относятся к отдельным этапам, поскольку они регулярно повторяются на протяжении всего жизненного цикла.

Современные процессы разработки являются *итеративными* и *поступательными*. Основой всех итеративных и поступательных процессов разработки является спиральная модель. Это относится также к стандартам CMM и ISO 9000, используемым при аккредитации организаций. К другим важным популярным моделям относятся модель Rational Unified Process® (RUP®), архитектура, управляемая моделями (MDA), а также методы ускоренной и аспектно-ориентированной разработки программного обеспечения.

Данная глава содержит семь постановок задач для семи учебных примеров, используемых в последующих главах (наряду с приложением), для иллюстрации нетривиальных концепций и методов моделирования. Эти примеры охватывают разные предметные области — "Зачисление в университет", "Магазин видеокассет", "Управление взаимоотношениями с заказчиками", "Прямой маркетинг по телефону", "Затраты на рекламу", "Регистрация времени" и "Конвертация валют".

Ключевые термины

BPR (Business Process Re-engineering). Модернизация бизнес-процессов.

CASE (Computer-Aided Software Engineering). Автоматизированное проектирование и создание программ.

CMM (Capability Maturity Model). Модель технологической зрелости.

COBIT (Control Objectives for Information and related Technology). Стандарт управления информационными технологиями и их аудит.

COTS (Commercial Off-The Shelf software package). Коммерческий пакет программ.

DFD (Data Flow Diagram). Диаграмма потоков данных.

ERD (Entity Relationship Diagrams). Диаграммы отношений логических объектов.

ERP. Система планирования корпоративных ресурсов (Enterprise Resource Planning System).

ISA (Information System Architecture). Архитектура информационных систем.

ISO (International Organization for Standardization). Международная организация по стандартизации.

ITIL (IT Infrastructure Library). Библиотека инфраструктуры информационных технологий.

MDA (Model-Driven Architecture). Архитектура, управляемая моделями.

OLAP (OnLine Analytical Processing). Система аналитической обработки данных в реальном времени.

OLTP (OnLine Transaction Processing). Система оперативной обработки транзакций.

OMG (Object Management Group). Группа управления объектами.

RUP (Rational Unified Process). Унифицированный процесс.

SOA (Service-Oriented Architecture). Архитектура, ориентированная на сервисы.

Web-хранилище данных (data webhouse). Распределенное *хранилище данных*, реализованное с помощью сети Web и не имеющее центрального узла.

Адаптивность (adaptiveness). Качество программного обеспечения, определенное тремя свойствами системы: понятностью, легкостью сопровождения и масштабируемостью (расширяемостью).

Архитектура (architecture). Описание системы с помощью ее модулей (*компонентов*). Она определяет, каким образом проектируется система и как компоненты связаны между собой.

Границы проекта (project scope). Набор функциональных требований к системе, предъявляемых пользователем и оговоренных в контракте на выполнение проекта.

Данные (data). Исходные факты, представляющие собой числа, количественные показатели, концепции и события, относящиеся к бизнесу.

Жизненный цикл (lifecicle). *Процесс*, используемый для создания и поддержки (на протяжении срока действия) программного обеспечения (поставляемых продуктов и услуг), разработанного в рамках проекта.

Знание (knowledge). Понимание *информации*, полученное опытным путем или в ходе исследования, результатом которого является способность действовать эффективно и полезно.

Интеллектуальный анализ данных (datamining). Область управления знаниями, связанная с исследованиями данных для выявления зависимостей и шаблонов (ранее неизвестных или забытых).

Информационная лавка (data mart). Уменьшенный вариант *хранилища данных*, хранящий подмножество производственных данных, относящихся к конкретному подразделению или функции.

Информация (information). Факты, имеющие дополнительную ценность; *данные*, прошедшие обработку, выявившую новые свойства и тенденции.

Итерация (iteration). Один цикл разработки программного обеспечения или проекта, связанного с интеграцией системы, результатом которого является *конструкция*.

Компонент (component). Выполняемый модуль программного обеспечения с точно определенными функциональными свойствами (сервисами) и протоколами связи (интерфейсами) с другими компонентами.

Конструкция (build). Выполняемый код, поставляемый пользователю для *расширения* системы.

Модель (model). Абстракция реальности; представление определенных аспектов внешней реальности в программном обеспечении.

Объект (object). Модуль программного обеспечения, способный реагировать на внешние события/сообщения и выполнять задачи, поставленные перед ним. Модель состоит из данных и операций, связанных с этими данными.

Планирование проекта (project planning). Деятельность, связанная с оценкой сроков поставки продукции, затрат, времени, рисков, контрольных отметок и ресурсов.

Поддерживаемость (supportability). См. *Адаптируемость*.

Показатели (metrics). Результаты измерения атрибутов программного обеспечения, например корректности, надежности, эффективности, целостности. полезности, легкости сопровождения, гибкости и тестируемости.

Процесс (process). Виды деятельности и организационные процедуры, используемые при разработке и сопровождении программного обеспечения.

Расширение (increment). Следующая улучшенная версия программного обеспечения, полученная в результате *итерации* разработки системы или интеграции проекта; расширение не нарушает пределов проекта.

Сервис (service). Функционирующий экземпляр программного обеспечения, допускающий локализацию и использование в других программных системах с помощью XML-сообщений, передаваемых с помощью Интернет-протоколов.

Требование (requirement). Формулировка системной услуги или ограничения.

Участник проекта (stakeholder). Любой человек, испытывающий влияние системы или участвующий в ее разработке.

Хранилище данных (data warehouse). База данных, содержащая архивную информацию и связанная с процессом принятия решения в рамках модели OLAP.

Многовариантные тесты

МТ1. Следуя Бруксу, укажите, какая из перечисленных трудностей не представляет существенную сложность при разработке?

а. Согласованность.

б. Невидимость.

в. Корректность.

г. Изменяемость.

МТ2. Адаптивность программного обеспечения связана с одним из следующих аспектов.

а. Надежность.

б. Легкость использования.

в. Легкость сопровождения.

г. Все вместе.

МТ3. Второй уровень технологической зрелости означает следующее.

а. Формализация и строгое следование процессам управления и разработки.

б. Для управления процессом используются количественные показатели.

в. Происходит непрерывное совершенствование процесса.

г. Ни одно из вышеперечисленных.

МТ4. Программа непрерывного улучшения сервиса (Continuous Service Improvement Programme — CSIP) является частью одного из следующих стандартов.

а. CMM.

б. ISO 9000.

в. ITIL.

г. COBIT.

МТ5. Модель UML включает в себя следующие модели.

а. Модели состояний.

б. Модели изменения состояний.

в. Модели поведения.

г. Все вышеперечисленное.

МТ6. Компоновка, ориентированная на процессы, является интеграцией элементов, обладающей следующими свойствами.

а. Она происходит на уровне баз данных или интерфейсов прикладного программирования (API), который поставляет информацию другим приложениям.

б. В ней информация поступает из многих программных систем в общий пользовательский интерфейс.

в. Она связывает прикладные интерфейсы (т.е. *сервисы*, определенные с помощью абстракции интерфейса).

г. Ни одно из вышеперечисленных.

МТ7. Что из перечисленного ниже *не* является методом планирования систем?

а. EPR.

б. SWOT.

в. ISA.

г. Все.

МТ8. Что из перечисленного ниже является основным видом деятельности в подходе VCM?

а. Управление кадрами.

б. Услуги.

в. Администрирование и обеспечение инфраструктуры.

г. Ни одно из вышеперечисленного.

МТ9. Что из перечисленного ниже не относится к подходу ISA?

а. Субподрядчик.

б. Владелец.

в. Управление.

г. Все вышеперечисленное.

МТ10. Что из перечисленного ниже относится к технологии OLAP?

а. Web-хранилище данных.

б. Хранилище данных.

в. Лавка данных.

г. Все вышеперечисленное.

МТ11. Что из перечисленного ниже не относится к задачам интеллектуального анализа?

а. Классификация.

б. Кластеризация.

в. Упаковка.

г. Ни одно из перечисленных выше.

МТ12. Что из перечисленного ниже *не* относится к методам структурного моделирования?

а. UML.

б. ERD.

в. DFD.

г. Все вышеперечисленное.

MT13. Что из перечисленного ниже *не* относится к методу итеративной и поступательной разработки?

а. Спиральная модель.

б. Функциональная декомпозиция.

в. Архитектура, ориентированная на модели.

г. Ничего из вышеперечисленного.

MT14. Какой из перечисленных ниже методов и моделей связан с программированием?

а. Аспектно-ориентированная разработка.

б. Ускоренная разработка программного обеспечения.

в. Генетическое программирование.

г. Ничего из вышеперечисленного.

Вопросы

В1. Исходя из своего опыта в отношении программных продуктов, как бы вы могли интерпретировать замечание Фреда Брукса (Fred Brooks) о том, что сущность разработки программного обеспечения объясняется такими его свойствами, как сложность, податливость, изменчивость и неосязаемость? Какое объяснение вы могли бы дать этим четырем факторам? Чем разработка программного обеспечения отличается от традиционных инженерных дисциплин, таких как строительство или машиностроение?

В2. *Компонент* (component) — это исполняемый модуль, реализующий четко определенные функции (сервисы) и коммуникационные протоколы (интерфейсы) взаимодействия с другими компонентами. Найдите самую новую спецификацию языка UML (UML, 2005, см. библиографию) и дайте определение понятий *пакет*, *подсистема* и *модуль*. В чем проявляются сходства и различия между понятием *компонент*, *пакет* и *модуль*?

В3. В главе утверждается, что разработка программного обеспечения — это искусство или ремесло. В качестве подтверждения этого тезиса можно привести высказывание Андре Жида (Andre Gide):"Искусство — это сотрудничество между Богом и художником, и чем меньше делает художник, тем лучше". Какой урок могут вынести из этого высказывания разработчики программного обеспечения? Согласны ли вы с ним?

B4. Напомните определение понятия *участник проекта*? Является ли поставщик программного обеспечения или технический сотрудник участником проекта? Обоснуйте свой ответ.

B5. Какой *уровень технологической зрелости* требуется от организации, чтобы она была способной успешно справляться с кризисными ситуациями? Обоснуйте свой ответ.

B6. Прочитайте статью "ITIL Overview" (ITIL, 2004, см. библиографию). Как стандарт ITIL обеспечивает согласованность информационных и деловых процессов, гарантируя своевременное и правильное решение поставленных задач? Перечислите семь основных модулей стандарта ITIL. Какие принципы разработки, указанные в этих модулях, непосредственно относятся к проблеме согласованности информационных и деловых процессов?

B7. Прочитайте статью "Aligning COBIT, ITIL" (COBIT, 2005, см. библиографию). Как стандарт COBIT обеспечивает согласованность информационных и деловых процессов, гарантируя своевременное и правильное решение поставленных задач? Перечислите контрольные цели стандарта COBIT, обеспечивающие согласованность информационных и деловых процессов.

B8. В ходе объяснения SWOT-подхода к планированию систем утверждалось, что "верно сформулированная миссия отводит главное место потребностям клиентов, а не товарам или услугам, которые предоставляет организация". Пожалуйста, объясните и проиллюстрируйте, каким образом нацеливание формулировки миссии на определенные товары или услуги может привести к утрате основной цели системного планирования — достижения *эффективности*.

B9. Найдите в Интернете статьи, описывающие метод Захмана ISA. Если вы не знаете, как это сделать, проанализируйте древнеримское изречение: "Divida et empera" ("Разделяй и властвуй"). Объясните, почему подход Захмана удобен для применения метода "разделяй и властвуй" при реализации крупных и сложных систем.

B10. Как *модернизация бизнес-процессов* (BPR) связана с *потоком работы*?

B11. Назовите три уровня управления организацией. Рассмотрите банковское приложение, которое предназначено для отслеживания стереотипов поведения владельцев кредитных карточек, чтобы автоматически блокировать карточку, если банк заподозрит злоупотребление (кража, подделка и т.д.). Какой уровень управления поддерживает подобное приложение? Обоснуйте свой ответ.

B12. Перечислите основные методы структурной разработки.

B13. Назовите основные причины, стимулирующие переход от структурной разработки к объектно-ориентированной разработке программного обеспечения.

B14. Объектно-ориентированная система должна *проектироваться с учетом компоновки*. Что это значит?

B15. Системное планирование и измерения характеристик программного обеспечения тесно связаны друг с другом. Объясните почему.

B16. Объясните взаимосвязь между соподчиненностью и тестируемостью.

B17. Прочитайте статьи о методе RUP и спиральной модели (воспользуйтесь гиперссылкой *References* на Web-сайте, посвященном книге, см. библиографию). Как метод RUP связан со спиральной моделью?

B18. Объясните, как архитектура MDA использует или объединяет стандарты OMG — UML, MOF, XML и CWM. Найдите в Интернете самую свежую информацию об этих стандартах. Воспользуйтесь адресами `www.omg.org/mda` и `www.omg.org/technology/documents/_modeling_spec_catalog.htm`.

B19. Кратко объясните следующие методы и модели ускоренной разработки программного обеспечения: экстремальное программирование (XP), аспектно-ориентированная разработка и разработка, ориентированная на свойства. Сравните второй и третий методы с методом экстремального программирования с точки зрения цикла анализ–проектирование–реализация.

B20. Перечислите схожие черты и отличия между методами RUP® и XP. Для того чтобы облегчить решение этой задачи, прочитайте статью "RUP vs XP" (Smith, 2003, см. библиографию).

Ответы на контрольные вопросы

Контрольные вопросы 1.1

KB1. Нет. "Существенные трудности" образуют инвариант.

KB2. Заказчики и разработчики.

KB3. Нет. *Расширение* улучшает нефункциональные свойства программного обеспечения, такие как корректность, надежность, устойчивость, легкость использования и т.д.

KB4. Стандарт COBIT является *стандартом продукции*, а стандарты ITIL, CMM и ISO 9000 являются *стандартами процессов*.

КВ5. Нет. *Компоновка, ориентированная на портал*, может рассматриваться как особая разновидность информационно-ориентированной компоновки, которая относится к понятию интерфейсов программирования (т.е. к объявлению сервисов/операций), а не пользовательских интерфейсов (например, графических пользовательских интерфейсов, предоставляемых браузерами).

Контрольные вопросы 1.2

КВ1. Эффективность.

КВ2. Наоборот — цели выводятся из объектов.

КВ3. К основной деятельности.

КВ4. В существовании владельца процесса.

КВ5. Планировщик, владелец, проектировщик, конструктор и субподрядчик.

Контрольные вопросы 1.3

КВ1. Тактический уровень принятия решений.

КВ2. Управление параллельной работой и восстановление после сбоев.

КВ3. Лавки данных.

КВ4. Интеллектуальный анализ.

Контрольные вопросы 1.4

КВ1. Структурный подход.

КВ2. Анализ требований.

КВ3. Архитектурное проектирование.

КВ4. С этапами компоновки и развертывания.

КВ5. Планирование, тестирование и измерение.

Контрольные вопросы 1.5

КВ1. Компоновка.

КВ2. Спиральная модель.

КВ3. Архитектура, ориентированная на модели.

КВ4. Ускоренная разработка программного обеспечения.

КВ5. Аспектно-ориентированная разработка программного обеспечения.

Ответы к многовариантным тестам

МТ1. в

МТ2. в

МТ3. г

МТ4. в

МТ5. г

МТ6. г

МТ7. а

МТ8. б

МТ9. в

МТ10. г

МТ11. в

МТ12. а

МТ13. б

МТ14. б

Ответы на вопросы с нечетными номерами

В1

В статье Брукса (Brooks, 1987) рассматриваются вопросы, связанные с идентификацией *причин сбоев программного обеспечения*. Одни из этих причин являются следствием "существенных" трудностей и являются инвариантами, другие — "второстепенные трудности"— можно контролировать и устранять.

К четырем существенным трудностям относятся: 1) сложность, 2) согласованность, 3) изменчивость и 4) невидимость. Первая трудность — *сложность* — представляет собой наиболее серьезное препятствие. Экспоненциальный рост сложности при увеличении размера задач является причиной многих проблем. "Размер" задачи может означать как потенциальное количество состояний программы, так и количество связей между объектами. Любое расширение программного обеспечения приводит к его резкому усложнению. Крупномасштабная разработка очень отличается от локальной. Сложность является внутренним свойством программного обеспечения. Его можно ограничить лишь за счет применения продуманных методов проектирования на основе подхода "разделяй и властвуй" и иерархического представления программных модулей (Maciaszek and Liong, 2005).

В отличие от природных систем (изучаемых биологией или физикой) сложность программного обеспечения усугубляется другими существенными факторами: согласованностью, изменчивостью и невидимостью. Системы программного обеспечения должны быть *согласованными* с рукотворной, запутанной и случайной средой. Будучи частью среды, окружающей человека, новая система программного обеспечения должна быть согласованной с уже существующими интерфейсами, независимо от того, насколько эти интерфейсы разумны.

Программное обеспечение является моделью реальности, окружающей человека. При изменении этой среды работающее (до поры до времени) программное обеспечение также должно изменяться. Часто эти изменения выходят за рамки осуществимого.

В заключение отметим, что программное обеспечение является нематериальным объектом. Если программное обеспечение является *невидимым*, его невозможно *визуализировать*, не приложив значительных усилий. Проблема заключается не в отсутствии средств графического представления (эта книга в основном посвящена именно этим вопросам!). Дело в том, что не существует *одной* графической модели, позволяющей полностью представить структуру и поведение программного обеспечения. Для того чтобы создать программное обеспечение, разработчик должен работать с большим количеством графических абстракций, накладывающихся друг на друга. Это действительно сложная задача, особенно если модели не позволяют представить системы в иерархическом виде.

Разница между разработкой программного обеспечения и обычным техническим конструированием выявлена Бруксом, который указал, что для любого программного обеспечения характерны четыре указанные выше *трудности*, а в обычном техническом конструировании существуют только второстепенные трудности.

Любая попытка описать программное обеспечение без учета его сложности, согласованности, изменчивости и невидимости часто приводит к потере абстрактности вообще. Разработчик программного обеспечения часто находится в незавидной позиции, поскольку он не может абстрагироваться от определенных аспектов проблемы, так как все они образуют тесно сплетенную сущность решения. Рано или поздно все проигнорированные свойства придется восстанавливать в виде последовательных моделей, и эти модели должны быть настолько согласованными друг с другом, чтобы образовывать полную интерпретацию системы. Мораль ясна — *хороший разработчик программного обеспечения в первую очередь должен уметь абстрагироваться.*

В3

Этот вопрос связан с первым вопросом. Если мы признаем существование четырех существенных трудностей, связанных с разработкой программного обеспечения, то обязательно должны устранить эти трудности, предложив *простые решения*.

Простое решение не означает *упрощенное*. В данном контексте простое решение означает, что оно является работоспособным с точки зрения пользователей и достаточно легким для разработчиков — не слишком изощренным, не слишком амбициозным, без лишнего "барабанного боя". Анализ завершенных программных проектов показывает, что избыточная сложность лишь отталкивает пользователей. Некоторые проекты закончились провалом именно по причине их излишней сложности.

Вероятно, наиболее важный совет, который можно было бы дать новичкам, содержится в аббревиатуре *KISS* (keep it simple, stupid — будь проще, дурачок). В связи с этим часто упоминают закон Мерфи (Murphy): "*Создайте систему, которой может пользоваться даже дурак и только дурак*".

B5

Для того чтобы разрешить кризисную ситуацию, организация должна находиться как минимум на *третьем уровне* технологической зрелости. Организация, находящаяся на первом уровне, опирается в своей работе на ключевых сотрудников. Процесс разработки в такой организации не документируется. Даже если разработка проводится в соответствии с установленными процедурами, в такой организации нет понимания того, как должны измениться эти процедуры, чтобы разрешить кризисную ситуацию.

Организация, находящаяся на *втором уровне* технологической зрелости, используя прошлый опыт, вырабатывает интуитивный процесс разработки. Кризисная ситуация порождает неизвестный доселе аспект этого процесса и, возможно, может привести этот процесс к неудаче. Если организация способна извлечь уроки из этого кризиса, то она может улучшить процесс разработки и сделать его более устойчивым к будущим неблагоприятным условиям.

Организация, пребывающая на *третьем уровне* технологической зрелости, формализует процесс разработки и следует этим принципам при выполнении всех проектов. Столкнувшись с кризисной ситуацией, такая организация не впадает в панику и продолжает выполнять установленные процедуры. Принцип "пусть все идет, как идет" ("steady as it goes") помогает восстановить порядок и спокойствие и, в конце концов, может преодолеть кризис. Однако если кризис приобрел большой размах, то третий уровень технологической зрелости может оказаться совершенно недостаточным (принцип "пусть все идет, как идет" может превратиться в принцип "пусть все гибнет, как гибнет").

B7

Стандарт COBIT регламентирует выполнение технических условий, касающихся контроля над организационными процессами и направленных на улучшение качества управления информационными технологиями. Он разделят вопросы управления информационными технологиями на четыре категории: 1) планиро-

вание и организация, 2) приобретение и реализация, 3) поставка и поддержка, 4) мониторинг.

Первая группа регламентирует условия согласованности между бизнес-процессами и информационными технологиями — планирование и организацию. Перечислим некоторые задачи, относящиеся к первой категории (plsn and organize — PO).

- PO1.1 — информационная технология как часть долгосрочных и краткосрочных планов.
- PO1.2 — долгосрочный план развития информационных технологий
- PO1.3 — долгосрочное планирование развития информационных технологий — подход и структура.
- PO1.4 — изменения в долгосрочном плане развития информационных технологий.
- PO1.8 — оценка существующих систем.
- PO3.3 — непредвиденная технологическая структура.
- PO3.4 — планы по приобретению программного и аппаратного обеспечения.
- PO4.3 — ревизия организационных достижений.
- PO4.5 — ответственность за качество.
- PO6.1 — благоприятная среда для управления информацией.
- PO9.1 — оценка деловых рисков.

B9

Древнеримский принцип "разделяй и властвуй" утверждает, что для победы над врагами их сначала необходимо изолировать, а затем использовать разногласия между ними. При решении задач этот принцип часто используется в несколько ином смысле. Он требует, чтобы крупная задача была разделена на небольшие подзадачи, затем была решена каждая из подзадач и на основе полученных решений найдено решение исходной задачи.

Принцип "разделяй и властвуй" приводит к иерархической структуре проблемной области. При разработке систем он приводит к разделению системы на подсистемы и пакеты. Разделение системы должно быть тщательно спланированным, чтобы уменьшить зависимость между подсистемами и пакетами, входящими в иерархию.

Архитектура информационных систем ISA обеспечивает среду, в которой можно провести управляемую разработку системных модулей с помощью *ракурсов* и *описаний*. В этом случае ресурсы, выделенные на разработку системы, можно распределить по отдельным *ячейкам* (находящимся на пересечении ракурсов и описаний). Это облегчает процесс разработки и контроля.

Преимущества подхода ISA приведены ниже.

- Улучшенная связь между участниками проекта.
- Идентификация инструментов, наилучшим образом подходящих для поддержки процессов, соответствующих ячейкам.
- Идентификация областей, в которых процесс разработки обнаруживает свои преимущества и слабости.
- Интеграция методов и инструментов разработки.
- Создание основы для оценки риска.

B11

В организации существуют три уровня управления: стратегический, тактический и оперативный. Применение информационных систем на самом низком уровне принятия решений (т.е. оперативном) сводится к переработке данных в *информацию*. Применение информационных систем на самом высоком уровне принятия решений (т.е. стратегическом) сводится к переработке данных в *знания*. Переход от низших уровней принятия решений к высшим совпадает с желанием превращать информацию в знания. Владение знаниями дает организации конкурентное преимущество.

Система управления кредитными карточками в банке находится как минимум на тактическом уровне принятия решений. Она больше связана с *анализом*, чем с обработкой *транзакций*. Анализ транзакций по кредитным карточкам сводится к следующим задачам.

- Выявление типичных шаблонов поведения владельцев карточек.
- Определение вероятности мошенничества в зависимости от места (например, страны), в котором используется карточка.
- Выяснение, используется ли карточка для снятия денег (для этого необходимо ввести PIN-код).
- Выяснение, проверяет ли пользователь состояние счета по телефону или с помощью Интернета (для этого необходимо ввести идентификационный код и пароль).
- Выявление проблем, существовавших с карточкой в прошлом.
- Выяснение, можно ли позвонить пользователю по телефону (перед тем как банк заблокирует карточку, служащий банка должен попытаться дозвониться до ее владельца по телефону).

B13

Объектно-ориентированный подход вовсе не нов. Его история восходит к языку программирования Simula, разработанному в конце 1960-х годов. Доминирующее положение, занимаемое объектной технологией, объясняется следующими причинами. Наиболее важные факторы связаны с достижениями в области аппаратного

обеспечения, в частности средств, реализующих графические пользовательские интерфейсы. Эти факторы *способствуют* широкому распространению объектных систем. Современные интерфейсы GUI требуют *программирования, ориентированного на события*, которому лучше всего соответствует объектная технология.

Кроме того, переход к объектно-ориентированной технологии стимулируется потребностями *новых приложений*, которые можно реализовать на современных платформах. Эти приложения образуют две основные категории: *групповые вычисления* (workgroup computing) и *мультимедийные системы* (multimedia systems).

К тому же объектно-ориентированная технология позволяет снизить постоянно увеличивающийся *дефицит ресурсов* (application backlog) для крупных систем, которые трудно поддерживать и еще труднее модернизировать. Решить эти проблемы можно с помощью *объектной упаковки* (object wrapping).

B15

Сбор показателей, характеризующих процесс разработки и качество продукции, поначалу кажется бессмысленным. Их полезность начинает проявляться, когда проект завершается и становится ясной продолжительность и стоимость его разработки. Выяснив корреляцию между показателями качества разработки и *запланированными целями* завершенного проекта, следующий проект можно спланировать точнее.

Показатели нового проекта можно сравнить с показателями предыдущего проекта. Если они близки к старым, то можно предположить, что плановые ограничения (время, деньги и т.д.) будут примерно такими же. Это тем более вероятно, если коллектив разработчиков не изменился, приложение относится к той же предметной области (например, система бухгалтерского учета), а база клиентов остается прежней.

Если организация постоянно измеряет показатели многих проектов, то выгода от планирования становится еще больше. Большее количество показателей означает более точное планирование и возможность найти полезные параметры для планирования нетипичных проектов и модификации проектов в изменившихся обстоятельствах.

B17

Процесс RUP® представляет собой коммерческую среду разработки, а спиральная модель носит скорее теоретический характер. Поскольку процесс RUP® является настраиваемой средой, его можно уточнять на каждой итерации спирального планирования по схеме "планирование–риск–анализ–конструирование–оценка пользователей".

Настраиваемый процесс RUP® определяет унифицированные действия и методы разработчиков и пользователей. Кроме того, он определяет шаблон, рабочие потоки, шаблоны документов, инструментальные панели и руководства разработчиков. Спиральная модель не решает проблемы менеджмента и не предлагает новых методов разработки.

Для того чтобы оставаться в рамках спиральной модели, процесс RUP® должен на каждой итерации учитывать ее основные аспекты. К этим аспектам относятся циклические изменения календарного плана и бюджета с учетом целей и ограничений, установленных пользователем, идентификация и устранение риска, готовность аннулировать проект исключительно на основе оценок риска и результатов конструирования в виде прототипов и артефактов, которые пользователи могут непосредственно оценить.

B19

Экстремальное программирование (XP) — это оригинальный, наиболее популярный и всесторонний способ ускоренной разработки. Несмотря на наличие слова "программирование" в его названии, на самом деле этот способ представляет собой подход к разработке программного обеспечения.

Экстремальное программирование охватывает методы объектной технологии и наилучшие приемы объектно-ориентированной разработки. Эти приемы включают в себя пользовательские истории, разработку через тестирование, парное программирование, уточнение кода и улучшение проекта путем рефакторинга, а также частый выпуск небольших версий, предназначенных для пользовательской оценки.

В основе аспектно-ориентированной разработки программного обеспечения лежит понятие архитектуры. Она делает акцент на модульности программного обеспечения в соответствии с комплексными задачами. Эти задачи могут регламентировать уровень модульности, а также функциональные и нефункциональные свойства. К примерам таких задач относятся безопасность системы, требуемый уровень параллельности и стратегия выявления объектов. Аргументация этого метода заключается в том, что разработка в соответствии с задачами (аспектами) позволяет найти масштабируемое решение с устойчивой архитектурой.

Задачи программируются отдельно и реализуются в виде прикладного кода. Поскольку существующие языки программирования слабо поддерживают возможность создания аспектного кода и его воссоединение с основным прикладным кодом, аспектно-ориентированное программирование требует отдельных инструментов (например, AspectJ), обеспечивающих разделение прикладной логики и комплексных задач. В системе AspectJ точкой воссоединения может быть любой момент времени на протяжении выполнения программы. Он может быть точкой вызова метода или возврата управления, а также объектной конструкцией.

В отличие от экстремального программирования аспектно-ориентированная технология действительно относится к программированию и может использоваться в качестве альтернативы частного рефакторинга в экстремальном программировании.

Разработка, ориентированная на функции, состоит из пяти процессов: 1) разработка общей объектной модели бизнеса; 2) создание списка функций; 3) пла-

нирование функций; 4) проектирование функций; 5) конструирование функций. Первый процесс выполняется экспертами в предметной области, а его результатом является высокоуровневая диаграмма классов. Второй процесс обеспечивает функциональную декомпозицию, отражаемую в диаграммах потоков данных. Декомпозиция проводится по предметным областям, а также по видам и этапам деловой активности. Разработка функций, лежащих в основе функциональной декомпозиции, не должна занимать больше двух недель. Разработка этих функций разбивается на произвольные контрольные отметки. Остальные три процесса образуют цикл "план–проектирование–конструирование". Функции обладают приоритетами, проектируются по одной и конструируются с помощью интенсивного тестирования.

По сравнению с экстремальным программированием разработка, ориентированная на функции, охватывает те же части жизненного цикла, т.е. анализ–проектирование–реализация. Однако итерации экстремального программирования в большей степени относятся к реализации, а разработка, ориентированная на функции, следует классическому жизненному циклу, зависящему от анализа и проектирования, предшествующего любому виду программирования.

ГЛАВА 2

Определение требований

Цели

2.1. Переход от бизнес-процессов к концепции решения
2.2. Определение требований
2.3. Согласование и оценка требований
2.4. Управление требованиями
2.5. Бизнес-модель требований
2.6. Техническое задание
Резюме
Ключевые термины
Многовариантные тесты
Вопросы
Упражнения. Затраты на рекламу
Упражнения. Регистрация времени
Ответы на контрольные вопросы
Ответы к многовариантным тестам
Ответы на вопросы с нечетными номерами
Объяснение упражнений. Затраты на рекламу

Цели

Определение требований требует от разработчиков социального, коммуникативного и управленческого опыта. По сравнению с другими этапами разработки систем определение требований в наименьшей степени касается технических сторон проекта. Однако недостаточно тщательное выполнение этого этапа может привести к более серьезным последствиям. Лавинообразный рост расходов, вызванных неучтенными, пропущенными или неверно понятыми требованиями заказчиков, может впоследствии губительно повлиять на процесс разработки.

Данная глава охватывает широкий спектр вопросов, касающихся определения требований.

Прочитав эту главу, читатели будут

- понимать необходимость согласования бизнес-процессов и информационных технологий при поиске решений;
- знать методы моделирования иерархических и бизнес-процессов;
- понимать отношения между стратегиями реализации и задачами анализа требований и проектирования системы;
- понимать различия между функциональными и нефункциональными требованиями;
- знать способы определения требований, приемы ведения переговоров о требованиях и методов их оценки, а также понимать принципы управления требованиями;
- уметь конструировать модели бизнес-требований, включая модели прецедентов использования и классов;
- знать желательную структуру типичного технического задания.

2.1. Переход от бизнес-процессов к концепции решения

Для создания современных гибких бизнес-систем необходимо исследовать бизнес-возможности и находить способы удовлетворения изменяющихся требований. Бизнес-процессы требуют реализации информационных процессов и систем. В одних ситуациях информационные технологии просто решают бизнес-проблемы, в других — способствуют внедрению новшеств и порождают новые бизнес-идеи. Однако во всех ситуациях информационная система представляет собой инфраструктурный сервис, сначала дающий возможность получить конкурентное преимущество, но затем становящийся *товаром*, которым могут владеть и конкуренты.

Бизнес требует инициативы. В то же время из-за динамичного развития новых технологий и появления готовых, легко настраиваемых решений специалисты по информационным технологиям все чаще рассматриваются как *поставщики бизнес-решений*, а не обычные разработчики систем. Целью планирования системы является согласование бизнес-стратегии и информационных технологий. Эта деятельность является частью процессов разработки информационных систем и стандартов. Как указывается в работе (Polikoff et al., 2006): "В настоящее время почти все лучшие методы учитывают необходимость *выработки бизнес-стратегии*, а также *моделирования* и *определения концепции системы* как обязательный элемент анализа требований и разработки программного обеспечения. Однако для выполнения этих фундаментальных задач используются готовые и испытанные средства".

Основная трудность согласования бизнес-идеи с соответствующей информационной системой объясняется отсутствием каналов обратной связи между бизнесменами и экспертами в области информационных технологий. Эти каналы должны иметь социальное и организационное измерение, но изначальная трудность (и необходимое условие для общения) связана с существованием (или отсутствием) общего языка моделирования, использующего систему графических обозначений для изображения бизнес-процессов.

Несмотря на то что для ликвидации разрыва между проектированием и реализацией бизнес-процессов были предложены многие языки и системы обозначений, основным стал язык **BPMN** (Business Process Modeling Notation) (White, 2004; BPMN, 2006). Изначально этот язык был разработан консорциумом Business Process Management Initiative, которым в настоящее время управляет группа OMG в рамках работы комиссии Business Modeling & Integration Domain Task Force (BMI DTF).

2.1.1. Моделирование иерархии процессов

Язык BPMN предназначен для моделирования **бизнес-процессов** (business processes), представляющих собой деятельность, направленную на производство ценностей, необходимых для организации или внешних участников проекта. Понятие проекта имеет иерархический характер — процесс может содержать другой процесс (*подпроцесс*). Атомарное действие процесса называется **задачей** (task).

Модель бизнес-процесса (business process model) можно разрабатывать для процессов разного масштаба — от производственного процесса всего предприятия до процесса, выполняемого отдельным человеком. С формальной точки зрения язык BPMN не имеет структур для моделирования процессов, но на практике он оказался довольно полезным для функциональной декомпозиции процессов. В прошлом для функциональной декомпозиции процессов использовались **диаграммы** потоков данных, представляющие собой достаточно сложный инструмент. В настоящее время все бо́льшую популярность приобретают более простые диаграммы иерархических процессов.

Диаграммы иерархии процессов (process hierarchy diagram) определяют статическую структуру модели бизнес-процесса. Они демонстрируют иерархическую структуру процессов путем их декомпозиции на подпроцессы. Как правило, продолжать декомпозицию процессов вплоть до атомарных задач не обязательно.

2.1.1.1. Процессы и декомпозиция процессов

Бизнес-процесс может выполняться вручную или автоматически. Каждый процесс имеет хотя бы один входной и один выходной поток. Если процесс является управляемым, то он преобразовывает входной поток в выходной.

Процесс может быть атомарным или составным. Атомарный процесс (задача) не содержит ни одного подпроцесса. Составной процесс состоит из подпроцессов. Для определения составных процессов используются связи декомпозиции (decomposition links). Соответствующие обозначения показаны на рис. 2.1.

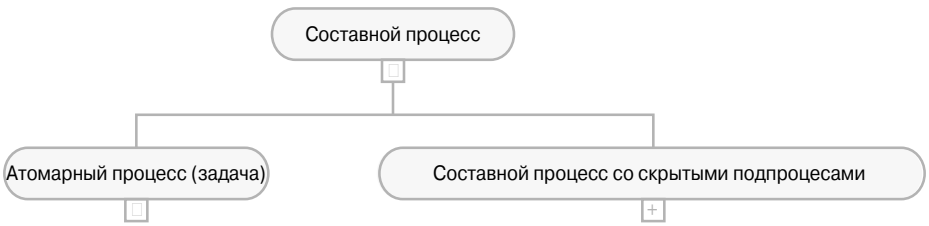


Рис. 2.1. Диаграмма иерархии процессов — обозначения

2.1.1.2. Диаграмма иерархии процессов

Диаграмма — это графическое представление модели. Часто диаграмма отражает конкретную точку зрения на модель, а полное представление модели может состоять из многих диаграмм. Более того, одну и ту же диаграмму можно строить с разным уровнем детализации — одни и те же графические символы могут сопровождаться разным количеством информации, чтобы отобразить некоторые аспекты модели и улучшить ее понимание.

Пример 2.1. Затраты на рекламу

Рассмотрите задачу 5, в которой описывается система оценки затрат на рекламу (см. раздел 1.6.5), и постройте для нее диаграмму иерархии процессов. Не стоит стараться провести полную декомпозицию высокоуровневых процессов. Для достижения иллюстративных целей достаточно провести декомпозицию только одного высокоуровневого процесса.

Учтите следующие обстоятельства.

- В описании задачи описана основная последовательность выполняемых действий, связанных со сбором подробной информации о рекламных объявлениях и оценкой рекламных расходов. Однако процессы управления

в ней не указаны. В модель иерархии процессов входят по крайней мере два важных процесса управления. Они связаны с поддержанием контактов, а также дебиторской задолженностью.

- Поддержание контактов связано с привлечением новых заказчиков, текущим контролем существующих заказчиков, отслеживанием контактов и поддержанием связи с поставщиками рекламной информации.
- Управление контактами и дебиторской задолженностью подразумевает проведение переговоров, выписывание счетов, мониторинг контрактов, обработка счетов и оплаты, а также анализ продаж.

На рис. 2.2 показана диаграмма иерархии процессов из примера 2.1. На ней показаны шесть высокоуровневых составных процессов в рамках общего бизнес-процесса `Измерение рекламных затрат`. Для иллюстрации показана декомпозиция процесса `Сбор измерений` на три задачи.

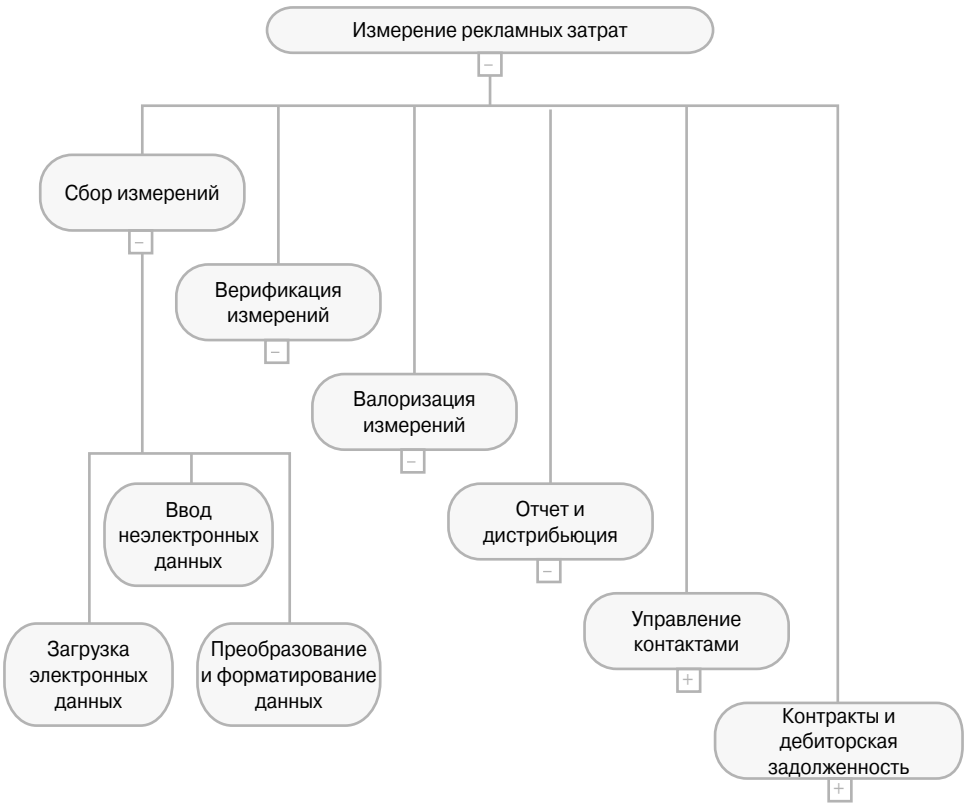


Рис. 2.2. Модель иерархии процессов в системе оценки затрат на рекламу

2.1.2. Моделирование бизнес-процессов

Язык BPMN позволяет создавать разнообразные диаграммы для моделирования сквозных бизнес-процессов внутри отдельной организации и их взаимодействия под управлением бизнес-сущностей. Диаграммы первого типа называются *внутренними бизнес-процессами* (internal business processes), а диаграммы второго типа — *процессами сотрудничества B2B* (collaborative business-to-business processes — B2B).

В языке BPMN нет обозначений для конкретной методологии моделирования бизнес-процесса. Скорее он является универсальным языком общения бизнесменов и специалистов по информационным технологиям. Для этой цели предназначены другие системы обозначений, такие как диаграммы языка UML. Поскольку языки BPMN и UML разрабатываются одной и той же группой OMG, ожидается, что в будущем произойдет консолидация диаграмм бизнес-процессов и диаграмм деятельности. В связи с этим возникает необходимость отобразить эти системы обозначений в исполняемые языки, в частности в язык **BPEL** (Business Process Execution Language) — стандарт исполнения процессов в системах, основанных на сервисно-ориентированной архитектуре (service-oriented architecture — SOA).

2.1.2.1. Потоковые и связующие объекты, дорожки и артефакты

Язык BPMN предлагает четыре основные категории (White, 2004).

- Потоковые объекты (flow objects).
- Связующие объекты (connecting objects).
- Пулы (pools), или дорожки (swiplanes).
- Артефакты (artifacts).

Потоковые объекты представляют собой основную часть языка BPMN. Существуют три категории потоковых объектов — события (events), действия (activities) и шлюзы (gateways) (рис. 2.3).

Событие — это нечто иногда происходящее. Обычно у него есть причина (триггер) или эффект (результат). Начальное событие (начало) означает старт определенного процесса. Конечное событие (конец) означает завершение процесса. Промежуточные события происходят между началом и концом. Существуют разные типы событий, например таймер, ошибка или отмена.

Действие — это некоторая работа, подлежащая выполнению. Она может представлять собой задачу или подпроцесс. Если на нижней границе прямоугольника с закругленными углами, символизирующего подпроцесс, указан знак "плюс", то действие является составным. Составное действие можно разбить на множество компонентов (subactivities) и раскрыть визуально. Дополнительные графические обозначения на нижней границе прямоугольника с закругленными углами символизируют дополнительные свойства, такие как цикл или наличие многих экземпляров.

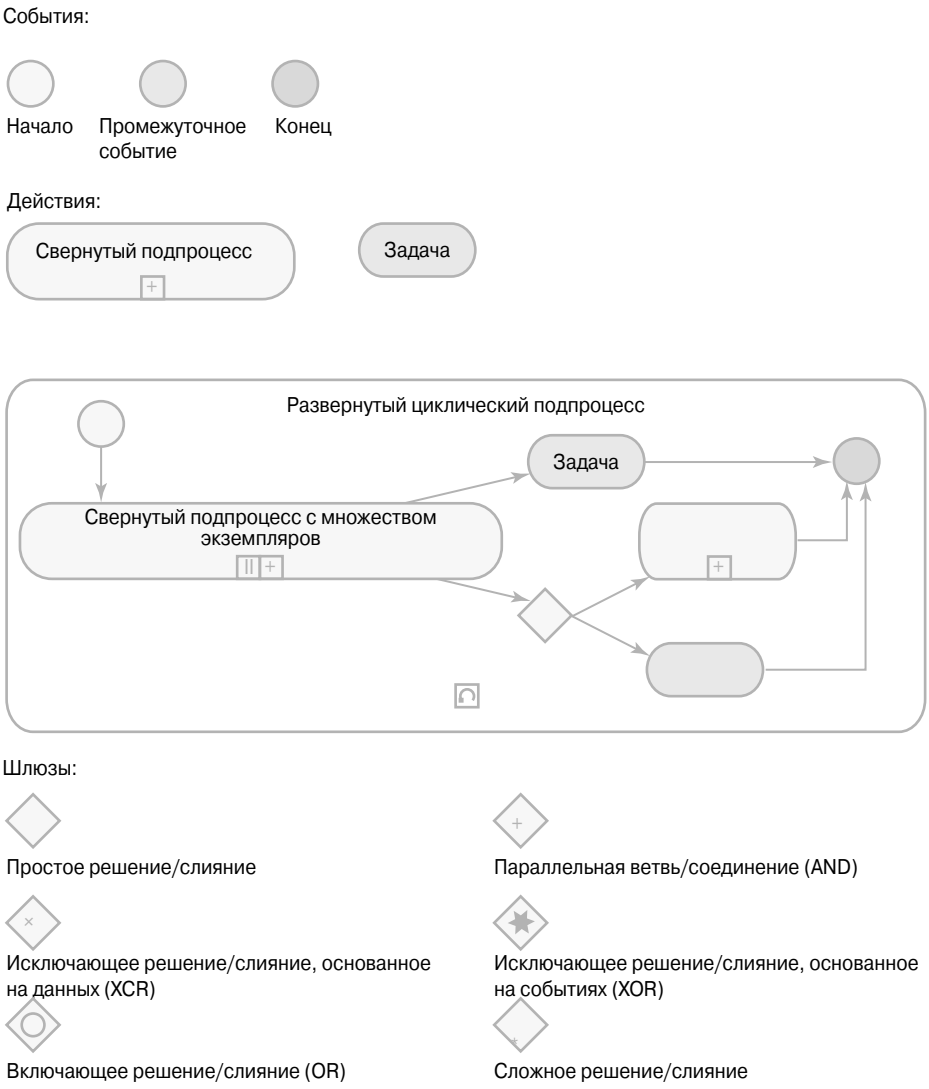


Рис. 2.3. Язык BPNM — потоковые объекты

Шлюз используется для управления разветвлением или слиянием нескольких потоков действий. Существует шесть разновидностей шлюзов: от простого разветвления до разделения и слияния путей.

Связующие объекты (или *коннекторы*) используются для связывания потоковых объектов, чтобы определить структуру бизнес-процесса. Существуют три категории связующих объектов — потоки действий (sequence flows), потоки сообщений (message flows) и ассоциации (associations) (рис. 2.4).

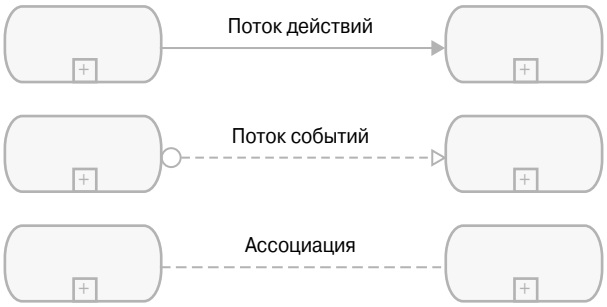


Рис. 2.4. Язык BPMN — связующие объекты

Поток действий используется, чтобы показать порядок, в котором выполняются действия, образующие процесс. *Поток сообщений* позволяет продемонстрировать передачу сообщений (данных) между двумя бизнес-сущностями (двумя участниками процесса), способными посылать и принимать сообщения. *Ассоциация* используется для связывания потоковых объектов или артефактов с другими потоковыми объектами.

Пул представляет собой бизнес-сущность (участника) процесса. Он действует как механизм "дорожки" (swimlane mechanism), организующий действия, относящиеся к разным категориям, чтобы проиллюстрировать разные функциональные свойства или обязанности. Пулы представляют собой множества автономных процессов. Соответственно, потоки действий не могут пересекать границы пула. Участники разных пулов могут обмениваться информацией с помощью потоков сообщений или через ассоциации с артефактами. Существуют два вида пулов — горизонтальный и вертикальный (см. рис. 2.5). Их выбор диктуется соображениями удобства.



Рис. 2.5. Язык BPMN — пулы (дорожки)

Артефакты обеспечивают дополнительную гибкость, позволяя расширять основную систему обозначений в особых ситуациях, например, для изображения процессов, протекающих на так называемых вертикальных рынках (например, в сфере телекоммуникаций, здравоохранении или в банковском сегменте). В языке BPMN предусмотрены три вида артефактов — объекты данных, группы и аннотации (рис. 2.6).



Рис. 2.6. Язык BPMN — артефакты

Объекты данных символизируют данные, требуемые или созданные процессом. Они не влияют непосредственно на потоки действий или сообщений, а лишь обеспечивают дополнительную информацию об этих действиях. *Группы* используются для объединения действий, не влияющих на поток действий в процессе. Группировку можно использовать для создания документации или для анализа (например, чтобы идентифицировать действия в ходе распределенной транзакции внутри пула). *Аннотации* обеспечивают дополнительную текстуальную информацию для читателей диаграмм бизнес-процессов.

2.1.2.2. Диаграмма бизнес-процессов

Используя описанные выше элементы обозначений, можно построить модель бизнес-процессов с выбранным уровнем точности и создать диаграммы бизнес-процессов и многоуровневые диаграммы.

На рис. 2.7 показана диаграмма бизнес-процессов, описанных в примере 2.2.

Пример 2.2. Затраты на рекламу

Рассмотрите задачу 5, в которой описывается система оценки затрат на рекламу (см. раздел 1.6.5), и диаграмму иерархии процессов, построенную при анализе примера 2.1. Постройте диаграмму бизнес-процесса "Управление сбором данных".

Учтите следующие обстоятельства.

- Основным методом сбора данных является электронная пересылка от поставщиков. Этот процесс происходит ежедневно и в большинстве случаев выполняется автоматически глубокой ночью. Данные поступают от телеканалов, радиостанций и компаний, распространяющих рекламные объявления в кинотеатрах. Для передачи данных используются специальные коммуникационные пакеты. Поставщик данных обязан записать информацию из регистрационного журнала за предыдущий день в специальный файл на сервере еще до начала процесса загрузки. Как правило, это происходит в два часа ночи.
- Данные, которые не могут пересылаться в электронном виде, извлекаются из документов. Как правило, это относится к рекламным объявлениям в газетах и журналах, а также к рекламным каталогам.

- Каждый поставщик информации использует собственный уникальный формат данных. Таким образом, первым процессом, который выполняется над этими данными, является их преобразование в стандартный формат, допускающий унифицированную обработку. Если источниками данных являются телевизионные каналы, некоторые сети формируют один файл, который сначала необходимо разделить на отдельные компоненты, соответствующие разным телеканалам.

Диаграмма бизнес-процесса из примера 2.2 показана на рис. 2.7. На ней показан как внешний процесс `Поставщик`, так и два процесса, которыми управляют внутренние бизнес-сущности, — `Отдел информационных технологий` и `Отдел ввода данных`. Диаграмма использует возможности счетчиков промежуточных событий, которыми обладают пулы `Отдел информационных технологий` и `Отдел ввода данных`. Кроме того, на ней показаны подпроцессы `Передача данных` и `Загрузка электронных данных`, имеющие несколько экземпляров. В процессе также участвуют три объекта данных.



Рис. 2.7. Модель бизнес-процесса для системы оценки затрат на рекламу

2.1.3. Выработка концепции решения

Выработка концепции решения (solution envisioning) (Polikoff et al., 2006) — это ориентированный на ценности подход к информационным услугам (т.е. не являющийся обычной системой программного обеспечения), предназначенный для решения бизнес-проблемы или стимулирующий внедрение бизнес-новаций. Концепция решения тесно связывает бизнес и участников информационной системы, а также объединяет бизнес-стратегии и методы разработки программного обеспечения.

Концепцию решения можно считать результатом применения интернет-технологий, подходов к планированию систем, описанных в разделе 1.2, а концепций трех уровней системы, изложенных в разделе 1.3. В основе этого подхода лежит концепция 3E (efficiency, effectiveness, edge) — эффективность, производительность и конкурентное преимущество. Результатом этого подхода является требование изменения бизнеса. Уникальность и новизна решения в основном зависят от понимания того факта, что современные проекты создания программного обеспечения редко носят автономный характер и разрабатываются с нуля, но образуют пакеты и состоят из компонентов (см. раздел 1.1.1), требующих интеграции (см. раздел 1.1.3).

2.1.3.1. Процесс выработки концепции системы

Процесс выработки концепции системы заключается в установлении эффективных связей между бизнес-процессами и информационными процессами. Поликоф и его соавторы (Polikoff et al., 2006) разделяют этот процесс на три стадии, каждая из которых состоит из определенных групп действий и действий, протекающих внутри этих групп. Процесс выработки концепции решения продемонстрирован на рис. 2.8. Три стадии этого процесса изображены в виде трех вертикальных пулов, группы действий представлены в виде расширенных подпроцессов, а действия — в виде задач.

На первой стадии процесса выработки концепции решения проводится **исследование бизнес-возможностей** (business capability exploration), т.е. способов, которым информационные технологии позволяют достичь конкретных результатов. Эта стадия посвящена описанию **прецедентов возможностей** (capability cases) — идей, на основе которых разрабатываются бизнес-прецеденты для каждой возможности. С точки зрения информационных технологий каждый прецедент возможностей можно интерпретировать как компонент программного обеспечения, предназначенный для достижения бизнес-целей.

"Первая стадия считается завершенной, если достигнуто понимание природы бизнес-проблем и способов их решения. Это понимание оформляется в виде предварительной концепции предлагаемого решения, которая распространяется среди участников проекта для последующей оценки и принятия решения в ходе специального семинара" (Polikoff et al., 2006).

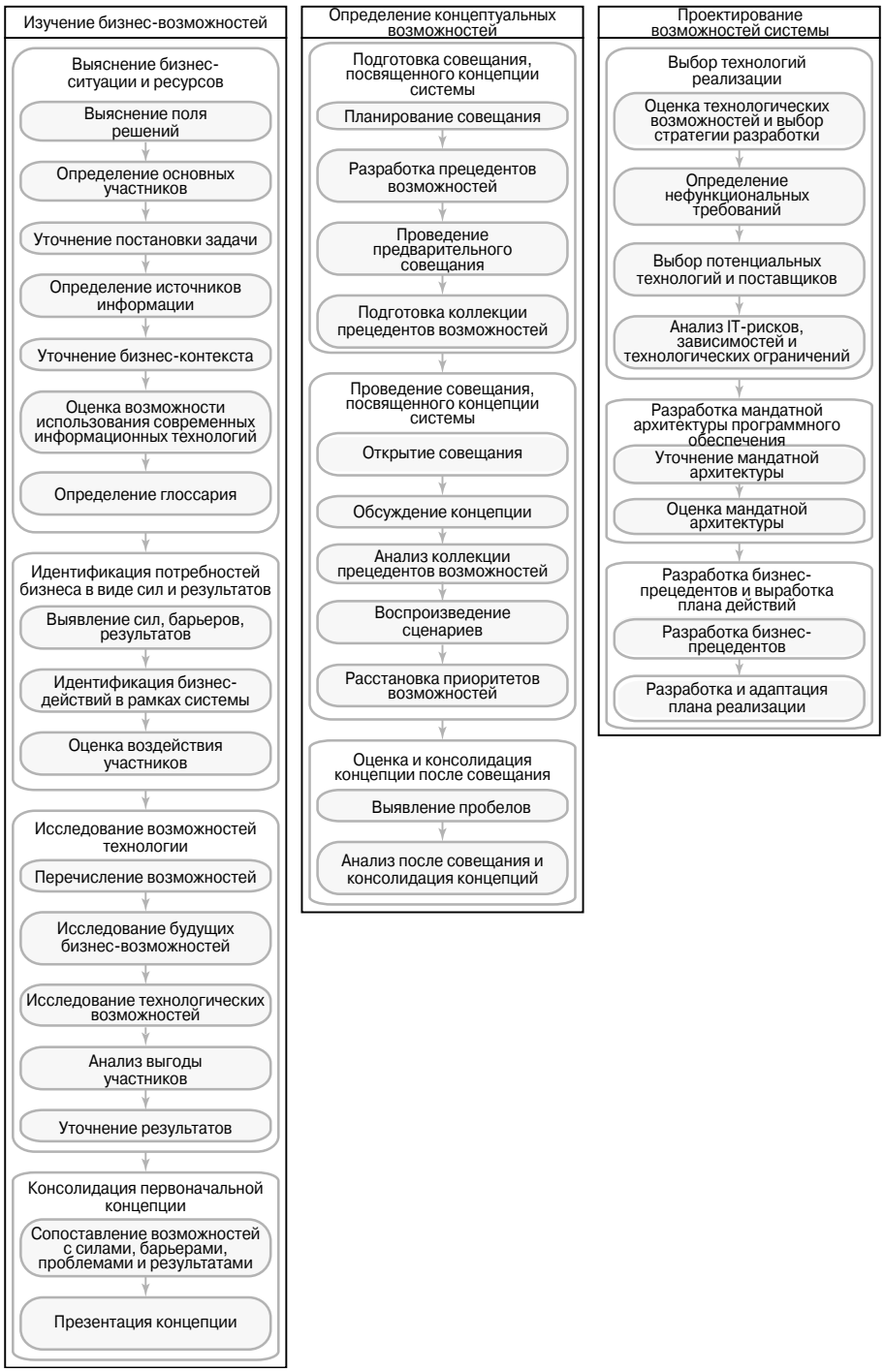


Рис. 2.8. Процесс выработки концепции решения

Вторая стадия подразумевает **определение концептуальных возможностей** (solution capability envisioning). Ее цель — разработка прецедентов возможностей, образующих часть **концепции решения** (solution concept) и гарантирующих, что это решение согласовано с участниками проекта. В качестве входной информации при выработке концепции решения используется бизнес-контекст, а результатом являются сценарии будущей работы системы. Концепция решения воплощается в окончательной архитектуре системы и вырабатывается на специальных совещаниях.

Третья стадия посвящена **проектированию возможностей системы** (software capability design). На этом этапе осуществляется выбор технологий реализации, разработка **мандатной архитектуры системы** (capability architecture), уточнение бизнес-прецедентов, а также планирование проекта и анализ рисков. Проектирование возможностей системы представляет собой моделирование работы программного обеспечения. Оно подразумевает создание высокоуровневых моделей и планирование системы. Предметом моделирования являются функциональные возможности (функциональные требования), атрибуты качества (нефункциональные требования), а также мандатная архитектура, демонстрирующая взаимодействие высокоуровневых компонентов программного обеспечения.

2.1.3.2. Стратегии реализации и мандатная архитектура

Для выявления возможностей, подлежащих исследованию, на первом этапе выработки концепции решения используются прецеденты возможностей, представляющие собой многочисленные *схемы решения* (solution sketches). Позднее некоторые из этих прецедентов возможностей становятся техническими *проектами* (blueprints). Наличие обратной связи между бизнес-прецедентами и информационными технологиями вынуждает достаточно рано выбирать стратегию реализации системы.

Как указывалось ранее, есть три основных *стратегии реализации* (implementation strategies) (Polikoff et al., 2006).

- Разработка на заказ (custom development). Индивидуальное проектирование программного обеспечения с нуля, охватывающее все этапы процесса разработки программного обеспечения (жизненный цикл) и выполняемое внутренними подразделениями и/или по контракту консалтинговыми и проектными фирмами.
- Пакетная разработка (package-based development). Процесс разработки, в ходе которого решение создается на основе готовых программных пакетов, таких как COTS, ERP и CRM.
- Покомпонентная разработка (component-based development). Процесс разработки, в ходе которого решение создается путем интеграции компонентов программного обеспечения, поступающих от многочисленных поставщиков и бизнес-партнеров; чаще всего основывается на архитектурных моделях SOA и/или MDA.

Несмотря на очевидные различия между стратегиями реализации, выбор конкретной стратегии слабо влияет на моделирование, выполняемое на этапе анализа требований и проектирования системы. Прежде всего, следует подчеркнуть, что *анализ требований* (requirements analysis) не зависит от реализации (по крайней мере, теоретически). Он должен представлять собой анализ деловых, а не информационных процессов. Правда, пакетная и покомпонентная разработка подразумевает использование готового программного обеспечения для реализации некоторых бизнес-процессов (однако это не освобождает разработчиков от необходимости моделирования бизнес-процессов, протекающих на конкретном предприятии!)

В то же время *проектирование систем* (system design) зависит от выбора стратегии реализации. Это влияние проявляется не столько в самом проектировании (методы проектирования одинаковы при любых стратегиях реализации), сколько в необходимости явной визуализации проектных моделей, для которых необходимо создавать компоненты или настраивать готовые пакеты. Очевидно, что элементы, разработанные на заказ, представляют собой "прозрачный ящик". Это значит, что проектирование этих элементов не должно скрывать внутренних деталей. При пакетной и компонентой разработке проектирование может использовать принцип "черного ящика". Иначе говоря, внешние функциональные свойства этих элементов должны быть инкапсулированы и интегрированы в моделях проектирования, а их внутреннее устройство может быть скрыто.

Одно из противоречий, связанных с современными подходами к выработке концепций, связано с ролью и местом *системной архитектуры* (system architecture), а значит, с ролью и местом *архитектурного проектирования* (architectural design). Как указывалось в предыдущем разделе, проектирование возможностей программного обеспечения приводит к мандатной архитектуре, т.е. модели, идентифицирующей высокоуровневые функциональные компоненты системы и взаимодействия между ними. Подход, предполагающий выработку концепции системы, подразумевает разработку мандатной архитектуры на ранних стадиях одновременно с проведением бизнес-анализа, а также идентификацией прецедентов возможностей и требований пользователей. Более того, мандатная архитектура обусловливает выбор стратегии реализации и сильно влияет на реализацию системы.

Перенос архитектурного проектирования на самые ранние этапы процесса разработки систем объясняется различиями между стратегиями реализации. Автор этой книги разделяет мнение Поликофа и его соавторов (Polikoff et al., 2006) в том, что "отсутствие такой архитектурной модели или ее недостаточная проработка часто является сигналом тревоги для руководителя проекта". Читатели должны помнить об этом, приступая к разработке проекта, и простить нас за то, что более полное обсуждение архитектурного проектирования из-за сугубо методических соображений перенесено в последующие главы (в частности, в главы 4 и 6).

Контрольные вопросы 2.1

КВ1. Как называется наиболее популярный язык визуального моделирования бизнес-процессов, заполняющий разрыв между бизнесменами и специалистами в области информационных технологий?

КВ2. Перечислите четыре категории элементов моделирования в языке BPMN?

КВ3. Может ли поток действий соединять два пула?

КВ4. Как называется ориентированный на ценности подход к информационным услугам, предназначенный для решения бизнес-проблемы или стимулирующий внедрение бизнес-новаций.

КВ5. Что является основным результатом проектирования возможностей программного обеспечения?

КВ6. Перечислите три стратегии реализации, рассматриваемые в процессе выработке концепции системы.

2.2. Определение требований

Бизнес-аналитик выявляет требования к системе в ходе консультаций, в которых участвуют заказчики и эксперты в предметной области. В некоторых случаях бизнес-аналитик обладает достаточным опытом в предметной области, и помощь эксперта может не потребоваться. В этом случае бизнес-аналитик сам является экспертом в предметной области. Этот факт отражен на рис. 2.9 с помощью отношения обобщения. (На этом рисунке использована система обозначений языка UML, подробно описанного в главе 3. В данном случае обозначения прецедентов использования выбраны из соображений удобства.)

Требования, выявленные с помощью экспертов проблемной области, составляют основу знаний о ней. Они фиксируют широко признанные, не зависящие от времени бизнес-правила, применимые к большинству организаций и систем. Требования, выявленные в ходе консультаций с заказчиками, выражаются в сценариях прецедентов. Они выходят за рамки базовых знаний о предметной области и фиксируют уникальные черты организации — способ ведения бизнеса "здесь и сейчас" либо пожелания в отношении того, как следует вести бизнес.

Задача бизнес-аналитика состоит в том, чтобы объединить два набора требований в бизнес-модели. Отношение агрегирования, использованное на рис. 2.9, показывает, что бизнес-модель состоит из **модели бизнес-классов** (business class model) и **модели бизнес-прецедентов использования** (business use case model).

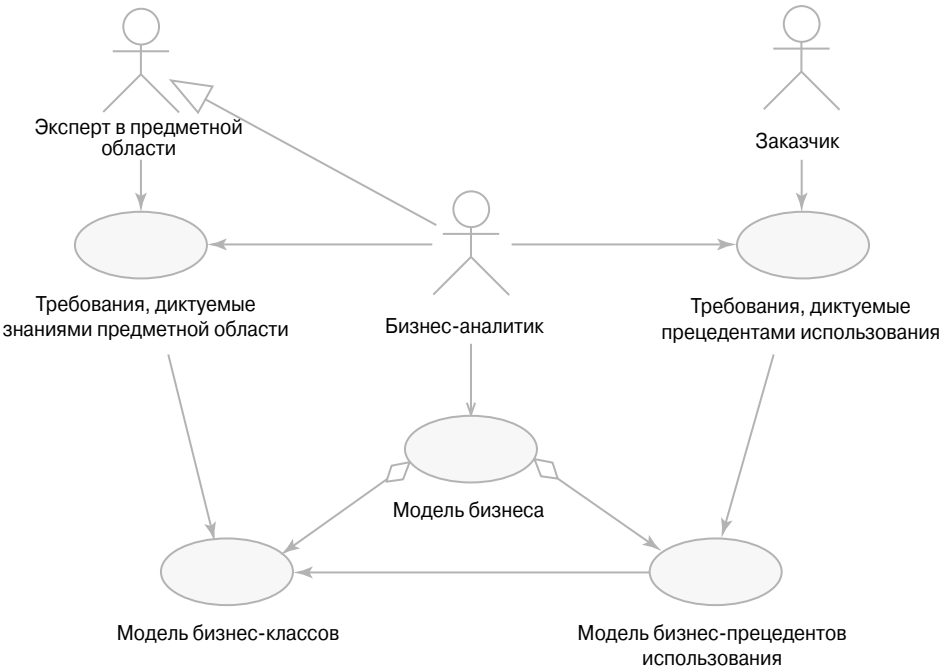


Рис. 2.9. Взаимные влияния сущностей, участвующих в процессе определения требований

Модель бизнес-классов представляет собой диаграмму классов верхнего уровня, идентифицирующую и связывающую между собой бизнес-объекты. **Класс** (class) — это абстракция, описывающая набор объектов с общими атрибутами, операциями, отношениями и семантическими ограничениями. Модель классов идентифицирует и связывает между собой **бизнес-объекты** (business objects) — фундаментальные классы в предметной области.

Модель бизнес-прецедентов — это диаграмма прецедентов верхнего уровня, идентифицирующая основные функциональные конструкции системы. Эта модель описывает **бизнес-прецеденты использования** (business use cases), отношения между ними и способы взаимодействия с **действующими лицами** (business actors). Бизнес-прецедент использования может быть представлен такой же функциональной конструкцией, как и **прецедент возможностей** (capability case) (см. раздел 2.1.3.1). Различия между ними заключаются в акцентах, а также роли и месте этих двух концепций на протяжении жизненного цикла проекта. Основное внимание в *прецедентах возможностей* уделяется деловой ценности функционального свойства. В то же время *бизнес-прецедент использования* подразумевает наличие деловой ценности и описывает взаимодействие пользователя (действующее лицо) и системы, приводящее к достижению конкретной деловой ценности.

В общем случае, классы предметной области (бизнес-объекты) не должны выводиться из прецедентов (Rumbaugh, 1994). Однако на практике правильность мо-

дели бизнес-классов должна подтверждаться посредством сравнения с моделью бизнес-прецедентов использования. Это сравнение, как правило, приводит к некоторой настройке или расширению модели бизнес-классов.

Следуя Хоффреру (Hoffer) и его соавторам (Hoffer et al., 2002), мы различаем традиционные и современные методы выявления фактов и сбора информации (разделы 2.2.2 и 2.2.3).

2.2.1. Системные требования

Планирование системы (см. раздел 2.1) определяет стратегические направление развития организации. *Моделирование бизнес-процессов* основывается на планировании системы и описывает как рутинные бизнес-процессы, так и новые идеи, дающие организации конкурентные преимущества. Планирование системы и моделирование бизнес-процессов вместе определяют, какие именно информационные системы должны быть разработаны.

Определение требований (requirements determination) — это первый этап жизненного цикла разработки и реализации системы. Цель этого этапа — дать описание функциональных и других требований, удовлетворения которых участники проекта ожидают от реализованной и внедренной системы. Границы системы определяются моделью бизнес-процессов. Соответственно, модель бизнес-процессов может служить точкой отсчета для идентификации высокоуровневых системных требований. Кроме того, желательно разделить первоначальные требования на те, которые можно реализовать с помощью программного обеспечения, и те, которые требуют вмешательства человека.

Как указывалось в разделе 1.4.2.1.1, требования определяют ожидаемые услуги системы (**формулировки сервиса**) и ограничения, которым должны удовлетворять система (**формулировка ограничений**). Формулировки сервиса задают **функциональные требования**, предъявляемые к системе. Функциональные требования можно разделить на требования, описывающие границы системы, необходимые бизнес-функции и требуемые структуры данных.

Формулировки ограничений можно классифицировать в соответствии с условиями, накладываемыми на систему, например, на ее внешний вид, производительность и безопасность. Формулировки ограничений образуют **нефункциональные требования**, предъявляемые к системе. Нефункциональные требования иногда называют *второстепенными требованиями*.

2.2.1.1. Функциональные требования

Используя элементы обозначений, описанные выше, можно визуализировать модели бизнес-процессов при заданном уровне точности и создать многие диаграммы бизнес-процессов и многоуровневые иерархические диаграммы.

Сначала необходимо выяснить у заказчиков (пользователей и владельцев системы) их функциональные требования. *Выявлением требований* занимается биз-

нес-аналитик (или системный аналитик). Для этого можно применять разные методы — чаще других используются традиционные интервью с заказчиками, которые завершаются (при необходимости) созданием прототипов программного обеспечения, удовлетворяющих большинству сформулированных требований.

Собранные функциональные требования должны стать предметом тщательного изучения, чтобы выявить дубликаты и противоречия. Это приводит к *пересмотру требований и повторным переговорам* с заказчиками. Согласованные функциональные требования моделируются с помощью графических обозначений, а затем уточняются в тексте.

2.2.1.2. Нефункциональные требования

Нефункциональные требования по своей природе не относятся к поведению системы. Они регламентируют проектирование и реализацию системы. Степень соблюдения этих требований определяет качество системы. Нефункциональные требования можно разделить на категории, ориентируясь на следующие понятия (Ghezzi et al., 2003; Lethbridge and Laganiére, 2001; Maciaszek and Liong, 2005).

- Удобство.
- Возможность повторного использования.
- Надежность.
- Производительность.
- Эффективность.
- Адаптивность (легкость сопровождения).
- Другие ограничения.

Удобство (usability) определяет легкость использования системы. Система считается удобной, если она не вызывает затруднений при ее использовании. Удобство определяется такими факторами, как наличие документации и справочной системы, возможность обучения, эстетический вид, непротиворечивость пользовательского интерфейса, обработка ошибок и т.д. Удобство — понятие относительное. То, что удобно для опытного пользователя, может оказаться неудобным для новичка.

Возможность повторного использования (resuability) определяет легкость использования ранее реализованных компонентов программного обеспечения в ходе новой разработки. Под "компонентом программного обеспечения" в этом случае подразумевается практически любая часть реализованной системы и даже идея (шаблон). Понятие повторного использования относится к интерфейсам, классам, пакетам, интегрированным средам и т.д. Возможность повторного использования отражает зрелость коллектива разработчиков, а также промышленной отрасли.

Надежность (reliability) характеризует частоту и серьезность сбоев системы, а также легкость возобновления ее работы. Надежность определяется работоспособностью системы в течение определенного периода времени, интервалами

времени между сбоями, точностью получаемых результатов и т.д. Надежная система — это система, заслуживающая доверие пользователя.

Производительность (performance) определяется ожиданиями, касающимися времени реакции системы, ее пропускной способности, потребления ресурсов, возможного количества одновременно обслуживаемых пользователей и т.д. Требования к производительности системы зависят от разных бизнес-функций, пиков загрузки и пользователей.

Эффективность (efficiency) регламентирует стоимость и время получения результатов и достижения целей, включая ожидаемый уровень производительности. В нее входят стоимость аппаратного обеспечения, уровень оплаты работы программистов и другие ресурсы. Более эффективные системы для выполнения задачи используют меньше ресурсов.

Адаптивность (*легкость сопровождения*) сводится к трем ограничениям — понятности, легкости эксплуатации и масштабируемости. Адаптивность определяет легкость, с которой можно понять, исправить, улучшить и расширить систему. Она зависит от ясности и простоты архитектурного проекта и точности реализации проекта.

Другие ограничения охватывают все остальные нефункциональные требования к системе. Вопросы, принадлежащие к этой категории, включают стратегические решения, касающиеся инфраструктуры системы, юридические аспекты проекта, требуемый уровень мобильности системы, требования к взаимодействию сетей и своевременность поставки.

Требования, согласованные с заказчиками, определяются, классифицируются, нумеруются и получают приоритеты в *техническом задании*. Этот документ формируется по *шаблону* (template), установленному организацией.

Несмотря на то что техническое задание представляет собой описательный документ, в него часто включают высокоуровневые диаграммы, иллюстрирующие *бизнес-модель*, которая обычно состоит из **модели границ системы**, *модели бизнес-прецедентов использования* и *модели бизнес-классов*.

Требования заказчиков изменчивы. Для того чтобы учесть изменчивые требования, необходимо уметь управлять ими. *Управление требованиями* подразумевает влияние изменений на другие ограничения и остальную часть системы.

2.2.2. Традиционные методы выявления требований

К традиционным методам выявления требований относятся использование интервью и анкет, наблюдение и изучение деловых документов. Это простые и экономичные методы. Эффективность традиционных методов обратно пропорциональна риску проекта. Высокий риск означает, что систему трудно реализовать — не вполне ясны даже обобщенные требования. Для таких проектов традиционных методов вряд ли будет достаточно.

2.2.2.1. Интервьюирование заказчиков и экспертов в проблемной области

Использование *интервью* (interviews) представляет собой основной метод выявления фактов и сбора информации. Большинство интервью проводится с заказчиками. Интервью с заказчиками позволяют выявить по большей части "прецедентные" требования, т.е. требования, вытекающие из "прецедентов" (рис. 2.9). Если бизнес-аналитик не обладает достаточным опытом в проблемной области, можно также проинтервьюировать соответствующих экспертов. Интервью с экспертами в прикладной области зачастую представляет собой простой процесс передачи знаний — занятие по обучению бизнес-аналитика.

Интервью с заказчиками отличает бо́льшая сложность (Kotonya and Sommerville, 1998; Sommerville and Sawyer, 1997). Заказчики могут иметь весьма смутное представление о своих требованиях. Они могут не желать сотрудничать или не умеют выражать свои требования в понятной форме. Заказчики также могут запрашивать реализацию требований, которые превосходят бюджет проекта или нереализуемы. И наконец, весьма вероятно, что требования, исходящие от различных групп пользователей, могут оказаться противоречивыми.

Существуют два основных типа интервью: структурированное (формальное) и неструктурированное (неформальное). *Структурированное интервью* (structured interview) готовится заранее, отличается ясностью постановки вопросов, а многие вопросы могут быть известны априори. Заранее сформулированные вопросы можно разделить на две категории: *вопросы открытого типа* (open-ended question) (ответы на эту категорию вопросов заранее неизвестны) и *вопросы закрытого типа* (closed-ended question) (ответы на эту категорию вопросов можно выбрать из списка предлагаемых ответов).

Структурированному интервью должно сопутствовать *неструктурированное интервью* (unstructured interview). Неструктурированное интервью больше напоминает неформальную встречу, которой не свойственны заготовленные вопросы или заранее поставленные цели. Цель неструктурированного интервью — подтолкнуть заказчика к тому, чтобы он поделился своими мыслями и в процессе беседы подошел к требованиям, которых бизнес-аналитик мог и не ожидать и, следовательно, не мог задать нужные вопросы.

Как структурированное, так и неструктурированное интервью нуждается в некоторой отправной точке и контексте для обсуждения. Это может быть небольшой документ или записка, отправленная по электронной почте интервьюируемому перед встречей, цель которых — объяснить цели интервьюера или задать некоторые вопросы.

Существуют три категории вопросов, которых, в общем случае, необходимо избегать (Whitten and Bentley, 1998).

- *Некорректные вопросы* (opinionated questions), в которых интервьюер выражает (прямо или косвенно) свое мнение по вопросу ("Должны ли мы работать так, как мы работаем?").
- *Предвзятые вопросы* (biased questions) аналогичны небеспристрастным, но отличаются от последних тем, что мнение интервьюера является явно тенденциозным ("Вы ведь не станете этого делать, не так ли?").
- *Наводящие вопросы* (imposing questions), которые предполагают ответ в самом вопросе ("Вы ведь сделаете именно так, не правда ли?").

Летбридж и Лажаньер (Lethbridge and Laganiére, 2001) рекомендуют использовать в интервью следующие категории вопросов.

- *Вопросы о конкретных деталях* любой темы, поднятой в ходе интервью. — Что, где, когда, кто и почему?
- *Вопросы о перспективе*. Интервьюер может находиться в счастливом неведении о различных ограничениях системы и может лелеять новаторские, но совершенно неосуществимые идеи.
- *Вопросы об альтернативных идеях*. Вопросы и соображения, предлагаемые интервьюируемому для оценки.
- *Вопрос о минимально допустимом решении*. Хорошие системы часто оказываются очень простыми, поэтому выделение минимальных требований играет важную роль при определении масштаба системы.
- *Вопросы о других источниках информации*. Выявление важных документов и других источников знаний, до сих пор не известных интервьюеру.
- *Вопросы о диаграмме запросов* (soliciting diagram). Простые графические модели, созданные интервьюером для объяснения бизнес-процессов, могут оказать неоценимую помощь при выявлении требований.

Успех интервью зависит от многих факторов, но едва ли не главными среди них являются навыки интервьюера в области *коммуникации* и *межличностного* общения. Хотя основная задача интервьюера — задавать вопросы и владеть ситуацией, не менее важно в ходе беседы внимательно слушать и проявлять терпение к интервьюируемому, чтобы он чувствовал себя непринужденно.

Для сохранения хороших личных отношений и в расчете на получение дополнительных сведений от интервьюируемого необходимо отправить ему в течение одного-двух дней после интервью записку, содержащую краткие итоги интервью, для комментария.

Интервью имеет много преимуществ и недостатков (Bennett et al., 2002). К *преимуществам* относятся следующие факторы.

- Гибкость и своевременность собранной информации, обеспеченные возможностью динамически реагировать на ответы интервьюируемого эксперта.

- Возможность более глубокого понимания требований, обеспечиваемая последовательно задаваемыми и постепенно уточняющимися вопросами, а также сбором соответствующих документов.
- Возможность взять интервью даже у далеко живущих участников проекта с помощью видеоконференций.

Интервью имеет следующие *недостатки*.

- Большие затраты времени и денег, поскольку интервью проводится в ходе личной встречи и требует определенной работы, например, прослушивания записей, стенографирования, а также уточнения ответов с помощью обратной связи с интервьюируемым.
- Результаты интервью могут искажаться или неверно интерпретироваться (правда, этот недостаток присущ и многим другим методам определения требований).
- Результаты интервью могут противоречить информации, полученной из других источников (ситуацию можно облегчить, организовав работу группы интервьюеров или применив современные методы выявления требований, например мозговой штурм (brainstorming)).

2.2.2.2. Анкетирование

Использование анкет или анкетирование (questionnaires) — эффективный способ сбора информации от многих заказчиков. Обычно анкеты используются в дополнение к интервью, а не вместо них. Исключение могут составлять проекты с низким риском, цели которых ясно очерчены. Для таких проектов обычно достаточно использовать анкеты с вопросами, которые носят пассивный характер и не отличаются большой глубиной.

В общем случае, анкетирование менее продуктивно, чем интервью, поскольку в вопросы или возможные ответы нельзя внести дополнительную ясность. Анкетирование пассивно — это можно расценивать и как достоинство, и как недостаток. С одной стороны, это достоинство, поскольку у респондента есть время подумать над ответом, а анкета может остаться анонимной. С другой стороны, это недостаток, поскольку респонденту нелегко прояснить для себя вопросы.

Анкета (или вопросник) должна быть разработана таким образом, чтобы, по возможности, облегчить ответы на вопросы. В частности, следует избегать вопросов открытого типа — большинство вопросов должны относиться к вопросам закрытого типа. Подобные вопросы могут принимать три формы (Whitten and Benteley, 1998).

- *Многовариантные вопросы* (multiple-choice questions). При ответе на эти вопросы респондент должен указать один или более ответов, выбрав их из прилагаемого списка (кроме того, иногда допускаются дополнительные комментарии к вопросам со стороны респондентов).

- *Оценочные вопросы* (rating questions). При ответе на этот тип вопросов респондент должен выразить свое мнение в отношении высказанного утверждения. Для этого могут использоваться такие рейтинговые значения, как "абсолютно согласен", "согласен", "отношусь нейтрально", "не согласен", "абсолютно не согласен" и "не знаю".
- *Вопросы с ранжированием* (ranking questions). Этот тип вопросов предусматривает ранжирование ответов с помощью присваивания им последовательных номеров, процентных значений и использования других средств упорядочения.

Тщательно продуманная, понятная анкета поощряет респондента к тому, чтобы сразу вернуть заполненный документ. Однако при оценке результатов анкетирования бизнес-аналитик должен предусмотреть возможность искажения информации из-за того, что те люди, которые не отвечали на вопросы, могли бы ответить на них иначе, чем принимавшие участие в опросе (Hoffer et al., 2002).

Анкетирование особенно полезно и экономично, если необходимо учесть мнения большого количества людей, живущих в разных регионах. Однако, для того чтобы подготовить хорошую анкету и правильно провести статистическую обработку результатов, необходим большой опыт.

2.2.2.3. Наблюдение

В некоторых ситуациях бизнес-аналитику трудно получить полную информацию, используя интервью и анкеты. У заказчика по тем или иным причинам может отсутствовать возможность (или умение) предоставить необходимую информацию, либо он может не иметь полного представления о бизнес-процессе в целом. В подобных случаях в качестве эффективного метода выявления фактов может выступать *наблюдение* (observation). Как говорится, лучший способ научиться завязывать галстук — понаблюдать за процессом.

Наблюдение может осуществляться в трех формах.

- *Пассивное наблюдение* (passive observation). Бизнес-аналитик наблюдает за различными видами деловой деятельности без вмешательства или прямого участия в них. В некоторых случаях, для того чтобы наблюдение было как можно менее навязчивым, можно даже использовать видеокамеры.
- *Активное наблюдение* (active observation). Бизнес-аналитик участвует в деятельности и становится фактически частью команды.
- *Объяснительное наблюдение* (explanatory observation). Пользователь объясняет наблюдателю свои действия в ходе выполнения работы.

Для того чтобы результаты наблюдений были репрезентативными, наблюдения необходимо проводить в течение продолжительного периода времени, в разные временны*е* интервалы и при разной рабочей нагрузке (выборочные периоды). Основная трудность, связанная с наблюдением, состоит в том, что люди, за кото-

рыми наблюдают, склонны вести себя иначе, чем в обычной обстановке. В частности, они стремятся работать в соответствии с формальными правилами и процедурами. Это приводит к искажению реального положения дел за счет утаивания "рациональных" приемов работы — как положительных, так и отрицательных. Следует помнить, что "работа по правилам" — это одна из эффективных форм забастовочного движения.

К положительным свойствам наблюдения относится то, что оно является незаменимым способом оценки времени, требуемого для выполнения работы. Кроме того, наблюдение может хорошо прояснить информацию, полученную с помощью других методов определения требований. В заключение отметим, что наблюдение носит визуальный характер и поэтому позволяет выявить такие приемы работы, которые невозможно обнаружить с помощью других методов.

2.2.2.4. Изучение документов и программных систем

Изучение документов и программных систем является неоценимым методом выявления как требований, связанных с прецедентами использования, так и требований, относящихся к предметной области. Этот метод используется всегда, хотя он может касаться только отдельных сторон системы.

Требования, связанные с прецедентами использования (use case requirements), выявляются посредством изучения существующих организационных документов, а также системных форм и отчетов (если существует компьютерная система, что типично для больших организаций). Одним из наиболее полезных способов постижения сути требований, связанных с прецедентами использования, является фиксация дефектов (естественно, при их наличии) и формирование запросов на изменение существующей системы.

К *организационным документам*, подлежащим изучению, относятся: формы деловых документов (по возможности — заполненные), описания рабочих процедур, должностные обязанности, методические руководства, бизнес-планы, схемы организационных структур, внутренняя корреспонденция, протоколы совещаний, учетные записи, внешняя корреспонденция, жалобы клиентов и т.д.

Системные формы и отчеты, подлежащие изучению, включают копии экранов и отчеты вместе с соответствующей документацией (системные руководства по эксплуатации, пользовательская документация, техническая документация, системные модели анализа и проектирования).

Требования, основанные на знании проблемной области, выясняются посредством изучения журналов и учебников, которые относятся к данной сфере. Изучение патентованных программных продуктов наподобие систем ERP (Enterprise Resource Planning Systems — системы планирования ресурсов предприятия), COTS (Commercial Off-The-Shelf — готовые коммерческие пакеты) и CMR (Customer Relationship Management — системы управления взаимоотношениями с заказчиками) также может стать богатым источником знаний о проблемной области.

Следовательно, частью процесса выявления требований являются посещения библиотек и поставщиков программного обеспечения (конечно, Интернет позволяет осуществить многие такие "визиты", не покидая офиса).

2.2.3. Современные методы выявления требований

К современным методам выявления требований относится использование программных прототипов, а также такие методы, как **JAD** (Joint Application Development — совместная разработка приложений) и **RAD** (Rapid Application Development — быстрая разработка приложений). Эти подходы предлагают более глубокое проникновение в суть требований, но за счет большей цены и усилий. Однако долговременные вложения в эти методы могут окупиться с лихвой.

Применение современных методов обычно связано с высоким проектным риском. Существует множество факторов, обусловливающих высокий *риск проекта*. К таким факторам относятся неясные цели, недокументированные процедуры, нестабильные требования, слабое знание дела пользователями, неопытные разработчики, недостаточная приверженность пользователей разработке и т.д.

2.2.3.1. Создание прототипов

Создание прототипов (prototyping) — это наиболее часто используемый современный метод выявления требований. Программные прототипы конструируются для визуализации системы или ее части для заказчиков с целью получения их отзывов.

Прототип представляет собой демонстрационную систему — "наскоро и грубо" сделанную рабочую модель решения, которая представляет графический пользовательский интерфейс и моделирует поведение системы при инициировании пользователем различных событий. Информационное наполнение экранов чаще жестче запрограммировано в программе прототипа, чем получается автоматически из базы данных.

Сложность (и растущие "аппетиты" заказчиков) современных графических пользовательских интерфейсов делает создание прототипов обязательным элементом разработки программного обеспечения. Прототипы позволяют довольно неплохо оценить реализуемость и полезность системы до начала ее реализации.

В общем случае, прототип — это весьма эффективный способ выявления требований, которые трудно получить от заказчика с помощью других средств. Чаще всего такая ситуация встречается для систем, которые должны предоставить в распоряжение пользователей новые бизнес-функции. Подобная ситуация также характерна для случаев противоречивых требований и наличия проблем в кооперации между заказчиками и разработчиками.

Существуют две основные разновидности прототипов (Kotonya and Sommerville, 1998).

- *"Одноразовый" прототип* ("throw-away" prototype), который отбрасывается после того, как выявление требований завершено. Разработка "одноразового" прототипа нацелена только на этап определения требований. Как правило, этот прототип концентрируется на наименее понятных требованиях.
- *Эволюционный прототип* (evolutionary prototype), который сохраняется после выявления требований и используется для создания конечного программного продукта. Эволюционный прототип нацелен на ускорение поставки продукта. Как правило, он концентрируется на ясно изложенных требованиях, так что первую версию продукта можно предоставить заказчику довольно быстро (хотя ее функциональные возможности, как правило, неполны).

Дополнительным доводом в пользу использования именно "одноразового" прототипа может служить стремление избежать риска "консервации" скорых и грубых и, как следствие, неэффективных решений в конечном продукте. Однако мощь и гибкость современных средств создания программного обеспечения ослабляют этот довод. Если управление проектом осуществляется надлежащим образом, то причины, по которой нельзя было бы избавиться от неэффективных предложенных для прототипа решений, не существует.

2.2.3.2. "Мозговой штурм"

"Мозговой штурм" (brainstorming) — это способ проведения совещаний, направленный на выявление новых идей и поиск решений конкретной проблемы путем сопоставления суждений, социальных запретов и правил (Brainstorming, 2003). В целом "мозговой штурм" не относится к анализу проблемы и методам принятия решений. Он направлен только на поиск новых идей и возможных решений. Анализ и принятие решений происходит после "мозгового штурма", а не во время его проведения.

"Мозговой штурм" полезен при выявлении требований, поскольку в ходе такого совещания трудно найти консенсус между участниками проекта. Более того, участники совещания, как правило, имеют узкие взгляды на требования, — ориентируясь на свой опыт, — и "мозговой штурм" позволяет им расширить свои творческие возможности.

Работу "мозгового штурма" должен координировать *посредник* (facilitator). Перед совещанием посредник должен сформулировать проблему, для решения которой нужны свежие идеи. Эта формулировка называется *провозможностью* (probortunity statement). Слово "провозможность" представляет собой сокращение двух слов — "проблема" и "возможность" (кроме того, этот неологизм устраняет негативный оттенок слова *проблема*).

Формулировка *провозможности* определяет повестку дня конкретного "мозгового штурма" и может принимать форму вопроса, проблемы, задачи, трудности, загадки или головоломки. При выявлении требований *провозможность*, как правило, состоит из *триггерных вопросов* (trogger questions).

Летбридж и Лажаньер (Lethbridge and Laganiére, 2001) приводят следующие примеры триггерных вопросов, позволяющих выявить требования.

- Какие функциональные возможности должна иметь система?
- Что является входными и выходными данными системы?
- Какие классы необходимы для бизнес-модели или объектной модели предметной области?
- Какие вопросы следует задать в ходе интервью или в анкете?
- Какие вопросы требуют внимания?
- Каковы основные риски проекта?
- Какие триггерные вопросы необходимо задать в ходе "мозгового штурма"?

В "мозговом штурме" должны принимать участие от двенадцати до двадцати человек, сидящих за круглым столом. Круглый стол лучше всего подходит для этой цели. Идея заключается в том, что посредник представляет собой лишь "человека из толпы" и все участники совещания равноправны. Участники совещания должны иметь блокноты, ручки, большие лекционные планшеты (из расчета один планшет на двух-трех участников), а также проектор для демонстрации формулировки провозможности и триггерных вопросов.

В ходе совещания участники размышляют над поставленными триггерными вопросами и либо высказывают их, либо молча записывают на бумагу, по одной идее на странице. На практике чаще используется последний подход, поскольку он не отпугивает людей. Ответы можно передать по кругу. Это стимулирует людей предлагать собственные идеи.

Этот процесс продолжается до тех пор, пока не родится новая идея или не пройдет фиксированный период времени (например, пятнадцать минут) (Lethbridge and Laganiére, 2001). По истечении указанного времени участников совещания просят зачитать вслух то, что записано на листке бумаги, лежащем перед ними. Скорее всего, на этом листке будут записаны идеи других участников (обычно анонимных). Эти идеи отображаются на лекционных планшетах. После этого начинается краткая дискуссия.

На последнем этапе совещания участники рабочей группы голосуют за предложенные идеи. Для этого достаточно раздать каждому участнику определенное количество клейких листочков и попросить приклеить их рядом с идеями, перечисленными на планшете. Подсчитав количество голосов, поданных за каждую идею, можно определить самую привлекательную из них.

2.2.3.3. Совместная разработка приложений (метод JAD)

Метод JAD полностью соответствует своему названию — это совместная разработка приложений (Joint Application Development), осуществляемая в ходе одного или нескольких совещаний с привлечением всех участников проекта (заказчиков

и разработчиков) (Wood and Silver, 1995). Хотя мы относим подход JAD к современным способам выявления требований, этот метод был впервые введен в конце 1970-х годов компанией IBM.

Существует много разновидностей метода JAD и много фирм, предлагающих услуги по организации и проведению JAD-совещаний. Проведение JAD-совещаний может занимать несколько часов, несколько дней или даже одну-две недели. Количество участников не должно превышать двадцати пяти–тридцати человек. В совещании принимает участие следующий круг лиц (Hoffer et al., 2002; Whitten and Bentley, 1998).

- *Ведущий* (leader). Человек, который председательствует на совещании. Он должен иметь исключительные способности в области коммуникации, не относиться к числу участников проекта (помимо того, что он ведет совещание) и обладать основательным знанием проблемной области (однако не обязательно хорошо владеть методами разработки программного обеспечения).
- *Секретарь* (scribe). Человек, фиксирующий ход JAD-совещания с использованием компьютера. Этот человек должен уметь быстро вводить текст в компьютер и хорошо владеть вопросами разработки программного обеспечения. Для документальной фиксации хода совещания и разработки первоначальных моделей решений секретарь может использовать CASE-средства.
- *Заказчики* (*пользователи* или *руководители*). Основные участники, излагающие и обсуждающие требования, принимающие решения, утверждающие проектные цели и т.д.
- *Разработчики*. Бизнес-аналитики и другие члены проектной бригады. Эти участники больше слушают, чем говорят, — они присутствуют на совещании для того, чтобы уяснить фактическую сторону дела и собрать информацию, а не занимать всецело внимание других участников в процессе совещания.

Метод JAD использует преимущества групповой работы. Групповые усилия более перспективны с точки зрения получения наилучших решений проблем. Группы способствуют повышению продуктивности, быстрее обучаются, склонны к более квалифицированным заключениям, позволяют исключить многие ошибки, принимают рискованные решения (иногда это может носить негативный характер!), концентрируют внимание участников на наиболее важных вопросах, объединяют людей и т.д.

Если совещание проводится в соответствии с правилами, можно добиться удивительно хороших результатов. Однако не следует забывать о предупреждении: "...компания Ford Motor в 1950-х годах потерпела неудачу, пытаясь вывести на рынок модель Edsel — автомобиль, разработанный комитетом" (Wood and Silver, 1995).

2.2.3.4. Быстрая разработка приложений (метод RAD)

Метод быстрой разработки приложений (rapid application development — RAD) представляет собой нечто большее, чем метод выявления требований, — это целостный подход к разработке программного обеспечения (Hoffer et al., 2002). Как ясно из названия метода, он предполагает быструю выработку системных решений. Техническое превосходство отступает на второе место в сравнении со скоростью поставки программного обеспечения.

Согласно Вуду и Сильверу (Wood and Silver, 1995), технология RAD сочетает в себе пять методов, приведенных ниже.

- Эволюционное создание прототипов (см. раздел 2.2.3.1).
- Применение CASE-средств с возможностями генерации программ и циклической разработкой с переходом от проектных моделей к программе и обратно.
- Метод SWAT (Specialists with Advanced Tools), подразумевающий работу коллектива, в который входят лучшие аналитики, проектировщики и программисты, которых только может привлечь организация, применяющая метод RAD. Команда работает в рамках строгого временного режима и размещается вместе с пользователями.
- Интерактивный метод JAD, предполагающий проведение совещаний (см. раздел 2.2.3.3), во время которого секретарь заменяется бригадой SWAT, оснащенной CASE-средствами.
- Метод жестких временных рамок (timeboxing), подразумевающий управление проектом, при котором коллективу проектировщиков, использующих метод SWAT, отводится фиксированный период времени (timebox) для завершения проекта. Этот метод препятствует "расползанию рамок проекта": если проект затягивается, то рамки решения сужаются, чтобы дать возможность завершить проект своевременно.

Использование подхода RAD может оказаться привлекательным вариантом для многих проектов, особенно небольших, которые не затрагивают сферу ключевых бизнес-процессов организации и которые, таким образом, не задают план решения для других проектов по разработке ПО. Маловероятно, чтобы быстрые решения были оптимальными или долговременными для ключевых сфер деятельности организации. С использованием метода RAD связан ряд проблем; некоторые из которых перечислены ниже.

- Противоречивый графический пользовательский интерфейс.
- Вместо общих решений, способствующих многократному использованию программного обеспечения, вырабатываются специализированные решения.
- Неполная документация.
- Трудное для поддержки и масштабирования программное обеспечение.

Контрольные вопросы 2.2

КВ1. Как называется профессия человека, выявляющего и документирующего требования, связанные с предметной областью и прецедентами использования?

КВ2. Назовите два вида требований.

КВ3. Перечислите три вида вопросов закрытого типа.

КВ4. Кто принимает участие в совещаниях JAD?

КВ5. Как называется коллектив разработчиков, использующих метод RAD?

2.3. Согласование и оценка требований

Требования, полученные от пользователей, могут дублироваться или противоречить друг другу. Одни требования могут быть неясными или нереальными, другие могут остаться невыясненными. По этой причине, прежде чем требования попадут в документ описания требований, их необходимо согласовать.

В действительности *согласование* и *проверка обоснованности требований* (requirements negotiation and validation) осуществляются параллельно с *выявлением требований* (requirements elicitation). После того как требования выявлены, они подвергаются проверке. Для всех современных методов выявления требований, которые связаны с так называемой *групповой динамикой* (group dynamics), это вполне естественно. Тем не менее после выявления требований они в любом случае должны быть подвергнуты тщательному обсуждению и проверке.

Согласование и проверка требований не могут быть отделены от процесса подготовки технического задания. Обычно *согласование требований* основано на черновом варианте этого документа. Требования, перечисленные в черновом варианте технического задания, обсуждаются и при необходимости корректируются. При этом удаляются неверные требования и добавляются вновь выявленные.

Для *проверки обоснованности требований* (requirements validation) необходим более полный вариант технического задания, в котором все требования четко идентифицированы и классифицированы. Участники проекта изучают документ и проводят совещания по их формальному пересмотру. *Пересмотры* (reviews) часто структурированы в виде так называемых процедур *сквозного контроля* (walkthroughs) или *инспекций* (inspections). Пересмотры являются разновидностью *тестирования* (testing).

2.3.1. Требования, выходящие за рамки проекта

Выбор проекта в сфере информационных технологий и, следовательно, создаваемой системы (и, в более общем смысле, ее масштаба) определяется в рамках системного планирования и бизнес-моделирования (см. разделы 1.2 и 2.1). Однако детализированные взаимозависимости между системами могут быть раскрыты только на этапе *анализа требований* (requirements analysis). *Границы системы* (ее *масштаб*) следует определять на этапе анализа требований, чтобы решить проблему "раздувания системы" уже на ранних этапах процесса разработки.

Для того чтобы выяснить, не выходит ли конкретное требование за пределы разумного, необходима эталонная модель, по отношению к которой должно приниматься решение. Исторически роль подобной эталонной модели играла *контекстная диаграмма* (context diagram) — высокоуровневая диаграмма популярного метода структурного моделирования на основе диаграмм потоков данных (data flow diagrams — DFD). Несмотря на то что в языке UML место этой диаграммы заняла диаграмма прецедентов, контекстная диаграмма по-прежнему остается превосходным методом установления границ системы.

Однако существуют и другие причины, по которым требование может быть расценено как выходящее за рамки проекта (Sommerville and Sawyer, 1997). Например, требование может представлять большую трудность для реализации в компьютеризованной системе и должно остаться прерогативой человека, участвующего в процессе. Кроме того, требование может иметь низкий приоритет и должно быть исключено из первой версии системы. Некоторые требования могут быть также реализованы с помощью аппаратного обеспечения или внешних устройств и, таким образом, оказаться вне контроля со стороны программного обеспечения.

2.3.2. Матрица зависимости требований

Когда все требования четко идентифицированы и пронумерованы (разд. 2.4.1), можно сконструировать *матрицу зависимостей требований* (requirements dependency matrix) (или *матрицу взаимодействия* (interaction matrix)) (Kotonya and Sommerville, 1998; Sommerville and Sawyer, 1997). В этой матрице перечисляются упорядоченные идентификаторы требований, как показано в табл. 2.1

Таблица 2.1. Матрица зависимостей требований

Требование	R1	R2	R3	R4
R1				
R2	Конфликт			
R3				
R4		Перекрытие	Перекрытие	

Верхняя правая часть матрицы (над диагональю включительно) не используется. Оставшиеся ячейки указывают на то, перекрываются ли два любых требования, противоречат друг другу или независимы друг от друга (пустые ячейки).

Противоречивые требования (conflicting requirements) необходимо обсудить с заказчиками и при возможности переформулировать, чтобы смягчить конфликт (запись о конфликте следует сохранить, чтобы учитывать эту возможность в ходе последующей разработки). *Перекрывающиеся требования* (overlapping requirements) также должны быть сформулированы заново.

Матрица зависимости требований представляет собой простой, но эффективный метод обнаружения противоречий и перекрытий, когда количество требований относительно невелико. В противном случае описанный метод можно применить лишь после группировки требований по категориям и сравнения в пределах каждой категории по отдельности.

2.3.3. Требования — риски и приоритеты

Устранив противоречия и повторы требований, необходимо провести анализ рисков и назначить приоритеты. *Анализ рисков* (risk analysis) позволяет идентифицировать требования, являющиеся потенциальными источниками трудностей. *Назначение приоритетов* (prioritization) необходимо для того, чтобы обеспечить возможность без труда изменить рамки проекта в случае возникновения непредвиденных задержек.

Осуществимость (feasibility) проекта зависит от его рискованности. *Риск* (risk) — это угроза, мешающая осуществить проект (недостаток финансирования, времени, ресурсов и т.д.). Идентифицируя риски, менеджер получает возможность управлять ими. Требования могут быть "рискованными" по разным причинам. Эти причины разделяются на следующие категории (Summerville and Sawyer, 1997).

- *Технический риск* (technical risk) означает, что требование технически трудно реализовать.
- *Риск снижения производительности* (performance risk) означает, что реализация требования может замедлить реакцию системы.
- *Риск, связанный с нарушением норм безопасности* (security risk), означает, что реализация требований может создать бреши в защите системы.
- *Риск, связанный с нарушением целостности баз данных* (database integrity risk), означает, что требование трудно проверить и может возникнуть противоречивость данных.
- *Риск, связанный с процессом разработки* (development process risk), означает, что для реализации требования необходимо использовать необычные методы, незнакомые разработчикам (например, методы формальной спецификации).

- *Политический риск* (political risk) означает, что требование может оказаться неосуществимым по внутриполитическим причинам.
- *Юридический риск* (legal risk) означает, что требование может привести к нарушению действующих законов или противоречить ожидаемым изменениям закона.
- *Риск, связанный с изменчивостью* (volatility risk), означает, что требование может потенциально изменяться или эволюционировать в течение процесса разработки.

В идеале *приоритеты* требований назначаются индивидуальными заказчиками в процессе их выявления. Затем их согласовывают на совещаниях и снова изменяют после приложения к ним факторов риска.

Для того чтобы исключить неопределенность и облегчить назначение приоритетов, количество приоритетов должно быть небольшим. Обычно достаточно от трех до пяти приоритетов. Их можно обозначить как "высокий", "средний", "низкий" и "неопределенный". Альтернативный перечень приоритетов может выглядеть, например, так: "существенное", "полезное", "трудно определимое" и "отложенное".

Контрольные вопросы 2.3

КВ1. Какой метод визуального моделирования лучше других позволяет определить границы системы?

КВ2. Какие виды зависимости между требованиями явно выявляются с помощью матрицы зависимости требований?

КВ3. Какой риск связан со сценарием, в котором требование постоянно изменяется или уточняется?

2.4. Управление требованиями

Требованиями необходимо управлять. *Управление требованиями* (requirements management) в действительности представляет собой часть общего управления проектом. Оно связано с тремя основными вопросами.

- Идентификация, классификация, организация и документирование требований.
- Изменение требований, т.е. формулировка, согласование, проверка достоверности и документирование неизбежных уточнений.
- Трассировка требований, т.е. установление зависимости между требованиями, с одной стороны, и остальными системными артефактами — с другой, а также между самими требованиями.

2.4.1. Требования — идентификация и классификация

Требования описываются на *естественном языке* (natural language statements), например, следующим образом.

- "Система должна запланировать следующий телефонный звонок клиенту по запросу оператора".
- "Система должна автоматически набирать запланированный телефонный номер и одновременно отображать на экране оператора информацию, включающую телефонный номер, номер клиента, имя абонента".
- "В случае успешного соединения система должна отобразить вводный текст, с которым оператор может обратиться к абоненту для того, чтобы завязать разговор".

Типичная система может состоять из сотен или тысяч формулировок требований, подобных тем, которые приведены выше. Для надлежащего управления таким огромным количеством требований их необходимо пронумеровать с помощью некоторой *схемы идентификации* (identification scheme). Эта схема может подразумевать *классификацию* (classification) требований по группам, которые легче поддаются управлению.

Существует несколько методов идентификации и классификации требований (Kotonya and Sommerville, 1998).

- *Уникальный идентификатор* (unique identifier) — это, как правило, порядковый номер, присвоенный вручную или сгенерированный с использованием базы данных CASE-инструмента, т.е. базы данных (или *репозитория*), в которой CASE-инструмент хранит артефакты, созданные в результате анализа или проектирования.
- *Порядковый номер в иерархии документов* (sequential number within document hierarchy) — номер, присвоенный с учетом места требования в техническом задании (например, седьмое требование в третьем разделе второй главы может быть пронумеровано как 2.3.7).
- *Порядковый номер в категории требований* (sequential number within requirements category) — номер, присвоенный в дополнение к мнемоническому имени, обозначающему категорию требований (где под подкатегорией требований подразумевают функциональное требование, требование к данным, требования к производительности, требования к безопасности и т.д.).

Каждый из методов идентификации обладает своими преимуществами и недостатками. Наибольшей гибкостью и защищенностью от ошибок отличаются *уникальные идентификаторы, генерируемые с помощью базы данных* (database-integrated unique database). Базы данных обладают встроенными возможностями генерации уникальных идентификаторов для каждой новой записи данных в условиях одновременного доступа к данным со стороны многих пользователей.

Некоторые базы данных способны дополнительно поддерживать сопровождение нескольких версий одной и той же записи за счет расширения значения уникального идентификатора с помощью номера версии. Наконец, базы данных могут обладать возможностями поддержки целостности на уровне ссылок между артефактами моделирования, включая требования, и могут, таким образом, обеспечить необходимую поддержку изменений и трассировки требований.

2.4.2. Иерархии требований

Требования можно упорядочить в виде иерархически упорядоченной структуры, представляющей *отношения родитель–потомок* (parent-child relationship). Родительское требование составлено из дочерних требований. Дочернее требование, по существу, представляет собой часть родительского требования.

Иерархические отношения (hierarchical relationships) позволяют ввести дополнительный уровень классификации требований. Иногда этот факт отражается в идентификационном номере (с помощью использования точки в записи составного имени). Например, требование, пронумерованное как 4.9, может быть девятым "потомком" "родителя" с идентификационным номером, равным 4.

Приведенный ниже набор формулировок представляет собой иерархически упорядоченные требования.

1. Система должна запланировать следующий телефонный звонок клиенту по запросу оператора.
   - **1.1.** Система должна активизировать клавишу `Next Call` (Следующий звонок) при открытии формы `Telemarketing Control` (Управление телемаркетингом) или по завершении предыдущего вызова.
   - **1.2.** Система должна удалить звонок из начала очереди запланированных звонков и придать ему статус текущего звонка.
   - **1.3.** И так далее.

Иерархии требований позволяют определять их на различных *уровнях абстракции* (levels of abstractions). Это согласуется с общим принципом моделирования, в соответствии с которым при переходе к более низкому уровню абстракции модели систематически обогащаются все новыми деталями. В результате для родительских требований можно сконструировать обобщенные требования, а детализированные модели можно связать с дочерними требованиями.

2.4.3. Управление изменениями

Требования изменяются. Они могут уточняться, удаляться или добавлять вновь на любом этапе разработки. Изменения нельзя рассматривать как "удар", а вот неуправляемые изменения — это настоящий удар по проекту.

Чем дальше продвинулась разработка, тем дороже обходится внесение изменений. Фактически *минимальные затраты* (downstream cost), необходимые для исправления проекта после внесения изменений, всегда увеличиваются и зачастую растут экспоненциально. Изменение только что сформулированного требования, не связанного с другими требованиями, представляет собой простое упражнение в редактировании. Изменение того же самого требования после его реализации может обойтись чрезвычайно дорого.

Изменения могут быть связаны с субъективными ошибками, но зачастую они вызваны изменениями внутренней политики или внешними факторами, например конкурентными силами, глобальными рынками или технологическими достижениями. Независимо от причин, для документирования *запросов на изменение* (change request), оценки *влияния, оказываемого изменениями* (change impact), и осуществления самих изменений необходима продуманная стратегия управления.

Поскольку изменение требований обходится дорого, для каждого запроса на изменение необходимо создавать формальный *бизнес-прецедент* (business case). Обоснованное изменение, ранее не встречавшееся, оценивается с точки зрения его технической осуществимости, влияния на остальной проект и затрат. После согласования требование включается в соответствующие модели и реализуется в программном обеспечении.

Управление изменениями, помимо прочего, включает в себя отслеживание больших объемов взаимосвязанной информации в течение длительного периода времени. Без инструментов поддержки управление изменениями обречено. В идеале изменения требований должны храниться и отслеживаться с помощью средств управления конфигурациями.

2.4.4. Трассировка требований

Трассировка требований (requirements traceability) — всего лишь часть управления изменениями (change management), хотя и крайне важная. Она позволяет отслеживать изменения, исходящие от заказчика или вносимые в требования на протяжении жизненного цикла проекта.

Рассмотрим следующее требование: “Система должна запланировать следующий телефонный звонок клиенту по запросу оператора”. Это требование можно впоследствии смоделировать с помощью диаграммы последовательностей, которая активизируется из графического пользовательского интерфейса с помощью кнопки запуска действия, помеченной как `Next Call` (Следующий звонок), и триггера, запрограммированного в базе данных. Если между всеми этими элементами существует *отношение трассировки* (traceability relationship), то изменение любого элемента вновь открывает возможность для дискуссий, поскольку траектория системы становится *подозрительной* (suspect).

Отношение трассировки может охватывать многие модели последовательных этапов жизненного цикла. Непосредственной модификации подлежат только

смежные связи. Например, если элемент *A* связан с элементом *B*, а элемент *B* — с элементом *C*, то изменение, внесенное в любой из конечных точек отношения, должно осуществляться в два этапа: сначала посредством модификации связи *A*–*B*, а затем *B*–*C*. (Более подробно трассировка и управление требованиями рассматриваются в главе 9.)

Контрольные вопросы 2.4

КВ1. Перечислите методы идентификации требований.

КВ2. Назовите инструмент, предназначенный для управления изменениями требований.

КВ3. Какая траектория системы называется подозрительной?

2.5. Бизнес-модель требований

На *этапе определения требований* (requirements determination phase) осуществляется их фиксация и определение, преимущественно в виде формулировок на естественном языке. Формальное моделирование требований с использованием языка UML проводится позже, на *этапе спецификации требований* (requirements specification phase). Тем не менее во время определения требований постоянно производится обобщенное визуальное представление собранных требований — *бизнес-моделирование* (requirements business modeling).

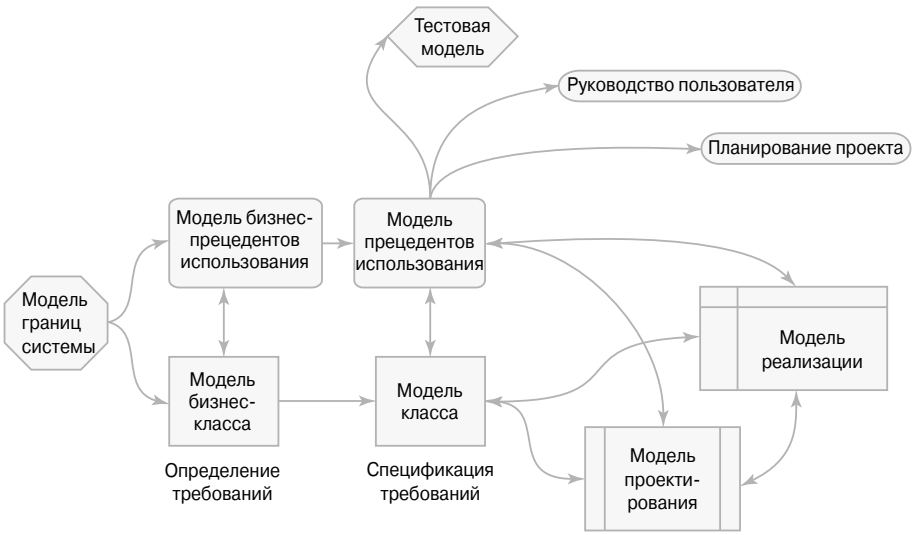


Рис. 2.10. Моделирование в ходе жизненного цикла

Обобщенная визуальная модель позволяет обозначить основные прецеденты использования и установить наиболее существенные бизнес-классы. На рис. 2.10 показаны зависимости между этими тремя моделями этапа определения требований и моделями остальных этапов жизненного цикла разработки.

На рис. 2.10 демонстрируется ведущая роль диаграмм прецедентов использования в жизненном цикле разработки. Из него ясно, что все тестовые прецеденты, пользовательская документация и проектные планы являются следствиями модели прецедентов использования. Более того, диаграммы прецедентов использования и модели классов используются параллельно и поочередно играют роль "лидера" в последовательных итерациях разработки. Проектирование и реализация также тесно переплетены и могут поддерживать обратную связь с моделями спецификации требований.

2.5.1. Модель границ системы

Из-за постоянного изменения требований наибольшее беспокойство при разработке обычно доставляет "размывание границ" системы. Несмотря на то что некоторые изменения требований неизбежны, необходимо строго следить за тем, чтобы заявленные изменения не выходили за пределы принятых границ проекта.

Ответ на вопрос: "Каким образом мы определяем границы системы?" совершенно не очевиден, поскольку любая система является всего лишь частью большей по своим масштабам среды — частью совокупности систем, которые вместе образуют эту среду. Системы взаимодействуют, обмениваясь информацией и обращаясь к услугам друг друга. Следовательно, поставленный выше вопрос можно переформулировать следующим образом: "Должны ли мы реализовать требования, или же выполнение требуемых функций возлагается на другую систему?"

Чтобы ответить на вопрос о рамках системы, необходимо знать, в каком *контексте* функционирует наша система. Необходимо знать, какие *внешние сущности* (external entities), т.е. другие системы, организации, люди, машины и т.д., рассчитывают на получение услуг от нас или готовы предоставить услуги нам. В бизнес-системах подобные услуги приобретают форму информации — *потоков данных* (data flow).

Таким образом, границы системы можно определить, обозначив внешние сущности, а также входные/выходные потоки данных между системой и этими сущностями. Система получает входную информацию и выполняет ее *обработку* с целью получения выходной информации. Всякое требование, которое не может быть поддержано за счет внутрисистемных возможностей обработки, выходит за границы системы.

Язык UML не обладает средствами построения визуальной модели, позволяющей определить границы системы. Поэтому зачастую для решения этой задачи прибегают к помощи давно известных *контекстных диаграмм*. Обозначения контекстной диаграммы приведены на рис. 2.11.

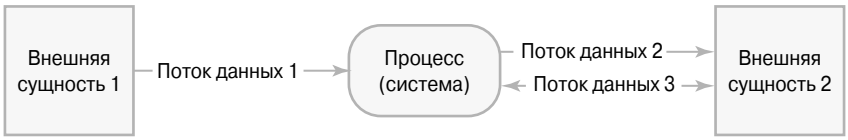


Рис. 2.11. Обозначения контекстной диаграммы

Пример 2.3. Прямой маркетинг по телефону

Рассмотрите постановку задачи для приложения "Прямой маркетинг по телефону" (см. раздел 1.4.6) и постройте для нее контекстную диаграмму. Кроме того, примите к сведению следующие замечания.

- Кампании планируются на основе рекомендации доверенных лиц из местной общины, решающих, насколько своевременными и подобающими являются те или иные благотворительные мероприятия. Кампании должны быть согласованы с местной администрацией. Проектирование и планирование кампании осуществляется на основе отдельной базы данных.
- Существует также отдельная база данных `Supporter Database` (База данных благотворителей), предназначенная для хранения и ведения информации обо всех благотворителях — прошлых и настоящих. Эта база данных используется для отбора благотворителей, с которыми необходимо установить контакт в ходе конкретной кампании. Выбранный *сегмент* благотворителей используется для прямого маркетинга по телефону.
- Заявки на лотерейные билеты регистрируются во время сеансов телефонного маркетинга и анализируются системой обработки заказов. Система обработки заказов отслеживает состояние заявок в базе данных благотворителей.

Контекстная диаграмма для данного примера показана на рис. 2.12. Прямоугольник с закругленными углами в центре диаграммы символизирует систему. Прямоугольники вокруг него обозначают внешние сущности. Стрелки изображают потоки данных. Подробное содержание информации, передаваемой потоками данных, на рисунке не показано — содержимое потоков данных определяется отдельно и хранится в CASE-репозитории.

Система `Telemarketing` (Телемаркетинг) получает информацию о текущей кампании от внешней сущности `Campaign Database` (База данных кампании). Эта информация включает в себя количество билетов и цены на них, перечень призов для победителей лотереи, продолжительность кампании и т.д.

Аналогично, сущность `Telemarketing` получает подробную информацию о спонсорах от сущности `Supporter Database`. Во время звонка может по-

явиться новая информация о спонсоре (например, о том, что спонсор собирается сменить свой телефонный номер). База `Supporter Database` (База данных о спонсорах) должна быть соответствующим образом актуализирована, поэтому поток данных `Supporter Details` (Подробная информация о спонсорах) является двунаправленным.

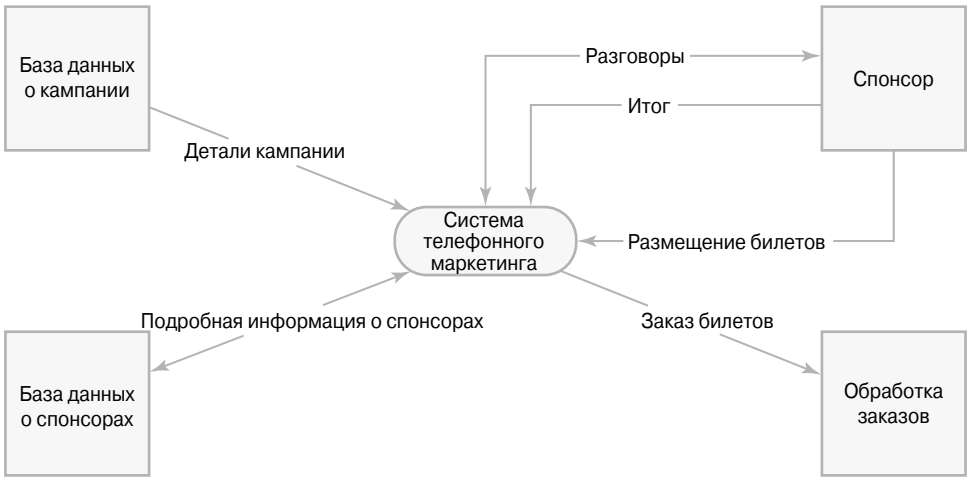


Рис. 2.12. Контекстная диаграмма для системы прямого маркетинга по телефону

Основная деятельность разворачивается между приложениями-сущностями `Telemarketing` и `Supporter` (Спонсор). Поток данных `Conversation` (Разговор) содержит информацию, которой обмениваются собеседники во время телефонного разговора. Ответ спонсора на предложение оператора приобрести лотерейные билеты передается с потоком данных `Outcome` (Результат). Для регистрации деталей, касающихся билетов, заказанных спонсором, используется отдельный поток данных `Ticket Placement` (Размещение билетов).

Дальнейшая обработка заказов на билеты выходит за рамки нашей системы. Поток данных `Ticket Order` (Заказ билетов) перенаправляется внешней сущности `Order Processing` (Обработка заказов). Можно предположить, что после ввода заказов другие сущности могут обработать платежи от спонсоров, отправку билетов по почте, розыгрыш призов и т.д. Это нас не интересует, поскольку текущее состояние заказов, платежей и т.д. доступно нам через внешние сущности `Campaign Database` и `Supporter Database`.

2.5.2. Модель бизнес-прецедентов использования

Модель бизнес-прецедентов использования (business use case model) (Kruchten, 2003) относится к верхнему уровню абстракции (см. разд. 2.2.2). Она определяет обобщенные бизнес-процессы, называемые *бизнес-прецедентами использования* (business use case).

Эти прецеденты могут определить процесс с помощью метода, полностью абстрагируемого из его реализации. Затем *бизнес-процесс* (business process) отделяется от *информационного процесса* (information system process), а модель в целом представляет деловую точку зрения с акцентом на организационные виды деятельности и рабочие задания (не всегда поддерживаемые компьютерными системами).

Однако на практике модели бизнес-прецедентов использования разрабатываются именно для того, чтобы бизнес-процессы поддерживались информационной системой. В таких случаях бизнес-процесс становится разновидностью информационного процесса. Бизнес-прецедент использования соответствует *свойству системы* (system feature). (Свойства системы идентифицируются в *концепции* (vision document). Концепцию можно использовать вместо модели бизнес-прецедентов использования.)

Диаграмма бизнес-прецедентов концентрируется на *архитектуре* бизнес-процессов. Эта диаграмма дает возможность взглянуть на предполагаемое поведение системы "с высоты птичьего полета". Неформальное описание каждого из бизнес-прецедентов должно быть кратким, ориентированным на деловую сторону системы, и концентрироваться на основных потоках видов деятельности. Модель бизнес-прецедентов не вполне подходит для того, чтобы объяснить разработчику, что же должна делать система.

На этапе спецификации требований *бизнес-прецеденты использования* превращаются в обычные *прецеденты использования*. Именно на этом этапе идентифицируются детальные прецеденты использования, а неформальные описания дополняются подпроцессами и альтернативными процессами, некоторыми копиями экранов, демонстрирующих интерфейс GUI, а также устанавливаются взаимосвязи между прецедентами.

Действующие лица (actors), являющиеся частью диаграммы бизнес-прецедентов использования, иногда могут представлять собой *внешние сущности* на контекстной диаграмме. Такие действующие лица называются *второстепенными* (secondary). Для того чтобы вступить во взаимодействие с прецедентом использования, они должны связаться с *первичным действующим лицом* (primary actor), играющим главную роль в системе и способным активно работать с прецедентом использования. Первичные действующие лица стимулируют работу второстепенных, посылая им сообщения о *событиях* (events).

Прецеденты управляются событиями. Линии, соединяющие действующих лиц и прецеденты использования, не являются потоками данных. Эти линии связи представляют поток *сообщений о событиях*, исходящих от субъектов, и поток *откликов* (responses), исходящих от прецедентов. В языке UML *отношения связи* (communication relationships) между действующими лицами и прецедентами использования представляются линиями, которые могут сопровождаться меткой «communicate».

Бизнес-прецеденты использования могут быть связаны разными отношениями. Отношение значимости в модели бизнес-прецедента использования является *отношением ассоциации* (association relationship). *Ассоциация* представляется в виде линии, которая может иметь размерную стрелку. Линия с размерной стрелкой символизирует *ассоциацию клиент–спонсор* (client-supporter), т.е. факт, что клиент знает нечто о спонсоре. Это также значит, что клиент каким-то образом зависит от спонсора. Между бизнес-прецедентами использования, как правило, не бывает других отношений, кроме ассоциации. Обозначения, используемые на диаграммах бизнес-прецедентов использования, показаны на рис. 2.13.

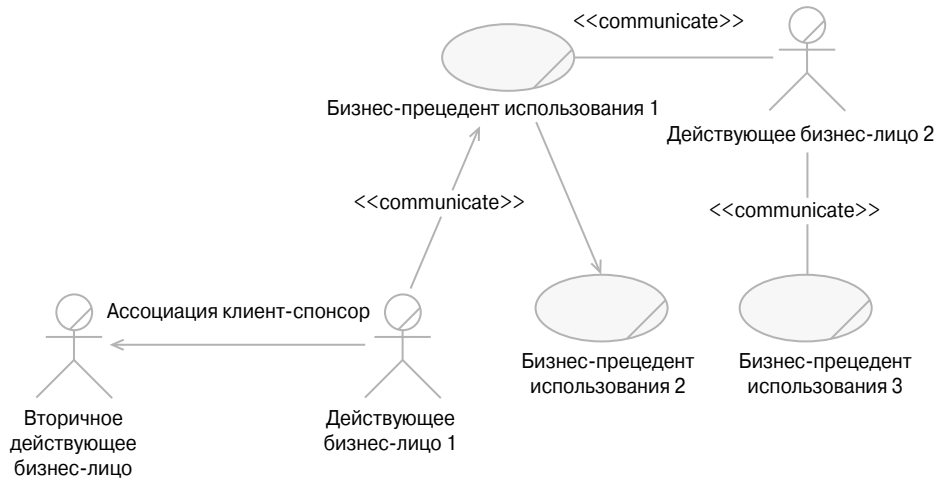


Рис. 2.13. Обозначения, используемые на диаграммах бизнес-прецедентов использования

Действующие лица обладают любопытной двойственной природой. По отношению к системе действующие лица могут быть как внешними, так и внутренними сущностями. Они являются внешними сущностями, поскольку взаимодействуют с системой, оставаясь вне ее. В то же время они остаются внутренними сущностями, так как система может поддерживать информацию о действующих лицах и может "осознанно" взаимодействовать с внешними действующими лицами. Системная спецификация — как модель системы — должна описывать систему и ее среду. Среда состоит из действующих лиц. Кроме того, система сама может хранить информацию о действующих лицах. Следовательно, спецификация заключает в себе две модели, связанные с действующими лицами, — модель действующего лица и модель данных о действующих лицах, подлежащих регистрации.

Один из возможных вариантов диаграммы бизнес-прецедентов использования показан на рис. 2.14. На диаграмме представлены два действующих лица: `Telemarketer` (Оператор) и `Supporter` (Спонсор). Оператор является первичным действующим лицом, а спонсор — вторичным.

Пример 2.4. Прямой маркетинг по телефону

Рассмотрите постановку задачи для приложения "Прямой маркетинг по телефону" (см. разделы 1.6.4 и 2.5.1) и постройте для нее диаграмму бизнес-прецедентов использования.

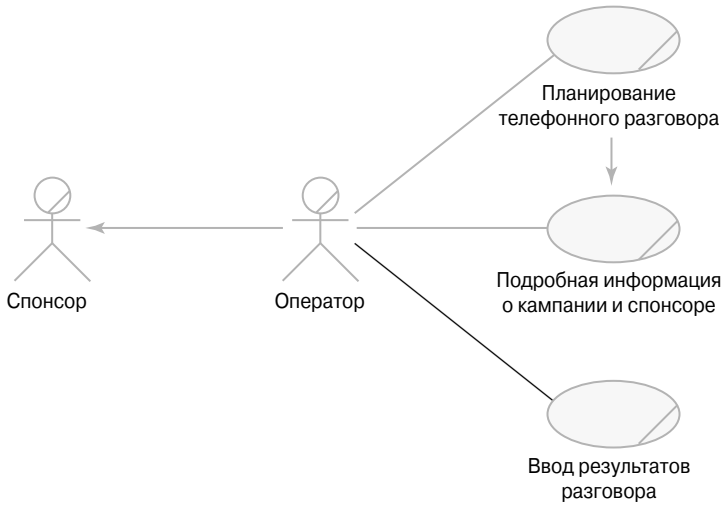


Рис. 2.14. Диаграмма бизнес-прецедента использования системы прямого маркетинга по телефону

Оператор просит систему запланировать телефонный звонок спонсору и осуществить соединение. При успешном соединении спонсор играет роль вторичного действующего лица. Бизнес-прецедент `Schedule Phone Conversation` (Планирование телефонного разговора), который в данном случае включает установление соединения, становится частью функции, видимой извне обоими действующими лицами.

Во время разговора оператору может потребоваться получить доступ и внести изменения в подробную информацию о кампании и спонсоре. Эти функции фиксируются в виде бизнес-прецедента `CRUD Campaign and Supporter Details` (Подробная информация о кампании и спонсоре). CRUD — популярная аббревиатура, обозначающая четыре основных операции над данными: Create, Read, Update, Delete ("Создать", "Читать", "Обновить" "Удалить").

Наконец, бизнес-прецедент `Enter Conversation Outcome` (Ввод результатов разговора) предназначен для ввода успешных или неуспешных результатов прямого маркетинга по телефону. Этот прецедент имеет очевидное значение для обоих действующих лиц.

Указание отношений между прецедентами `CRUD Campaign and Supporter Details` и `Enter Conversation Outcome` не обязательно. В общем случае все отношения между прецедентами можно опустить, чтобы избежать беспорядка и загромождения диаграммы. Как правило, прецеденты некоторым образом связаны с большинством остальных прецедентов, и включение в диаграмму всех отношений противоречит ее предназначению.

2.5.3. Бизнес-глоссарий

Одним из незаметных, но важных аспектов разработки программного обеспечения является ясность деловой и системной терминологии. Без этого общение между участниками проекта становится неточным и может привести к неверному решению. Для того чтобы улучшить качество общения и избежать недоразумений, необходимо создать *глоссарий* (glossary) и распространить его среди участников проектов.

На практике каждая компания имеет частичный глоссарий, возможно, оставшийся после разработки подобных систем. Если такого глоссария нет, то именно с его создания следует начинать этап определения требований. Если глоссарий есть, то его следует расширить и дополнить.

Глоссарий можно оформить в виде таблицы. Он должен содержать термины и их описания, перечисленные в алфавитном порядке и связанные перекрестными ссылками.

Пример 2.5. Прямой маркетинг по телефону

Рассмотрите постановку задачи для приложения "Прямой маркетинг по телефону" (см. раздел 1.6.4 и 2.5.1) и создайте для нее контекстную диаграмму и глоссарий.

В табл. 2.2 представлен глоссарий бизнес-терминов, связанных с прямым маркетингом по телефону. Слова, напечатанные курсивом в столбце "Определение", являются ссылками на другие термины, включенные в глоссарий.

Таблица 2.2. Деловой глоссарий

Термин	Определение
Билет	*Лотерейный билет*, имеющий определенную стоимость, идентифицирующий *кампанию* и имеющий уникальный *номер*
Бонусная кампания	Специальная последовательность мероприятий, проводимых в рамках *кампании* и направленных на поощрение *спонсоров* покупать благотворительные *билеты*. Типичным примером такой кампании является выдача бесплатных билетов в качестве награды, либо за оптовую

Окончание табл. 2.2

Термин	Определение
	покупку билетов, либо за приобретение билетов в числе первых покупателей, либо за привлечение новых спонсоров. Бонусные кампании могут быть частью других кампаний
Жеребьевка	Акт случайного выбора конкретного выигрышного *лотерейного билета*
Заказ на билеты	Подтвержденная *заявка* на билеты, имеющая конкретный номер и рассматриваемая как заказ, требующий выполнения
Заявка	Приобретение *спонсором* одного или нескольких лотерейных билетов в ходе *прямого маркетинга по телефону*. Заявка оплачивается спонсором с помощью кредитной карточки
Кампания	Одобренная правительством и тщательно спланированная последовательность мероприятий, направленных на достижение целей *лотереи*
Лотерея	Игра, направленная на сбор пожертвований, организованная благотворительным обществом, в ходе которой участники (*спонсоры*) покупают пронумерованные *билеты*, чтобы получить шанс выиграть *приз*, если их номер будет извлечен в результате *жеребьевки*
Оператор	Сотрудник, вступающий в контакт с абонентом в ходе *прямого маркетинга по телефону*
Приз	Награда, выдаваемая *спонсору*, купившему выигрышный лотерейный *билет*
Прямой маркетинг по телефону	Рекламные мероприятия, направленные на продажу лотерейных билетов по телефону
Сегмент	Группа спонсоров из базы данных, являющихся целью *конкретной* кампании *прямого маркетинга по телефону*
Спонсор	Лицо или организация, указанная в базе данных в качестве бывшего или потенциального покупателя лотерейных *билетов*

2.5.4. Модель бизнес-классов

Модели бизнес-классов (business class model) — это разновидность *моделей классов* в языке UML. Как и модели бизнес-прецедентов использования, они относятся к более высокому уровню абстракции, чем модели классов. Модель бизнес-классов определяет основные категории бизнес-объектов системы.

Бизнес-объекты системы постоянно хранятся в базах данных предприятий. Они должны отличаться от экземпляров других классов, например, реализующих пользовательский интерфейс или прикладную логику системы.

Как правило, бизнес-классы являются структурами данных для бизнес-процессов, составляющих основу системы и определяющих ее деятельность. Более

того, модель бизнес-классов часто не имеет явных атрибутов, так как имени и краткого описания часто вполне достаточно.

Бизнес-классы могут быть связаны другом с другом тремя отношениями UML: ассоциации, обобщения и агрегации. Ассоциация и агрегация выражают семантические отношения между экземплярами классов (т.е. объектами). Обобщение — это отношение между классами, т.е. между типами объектов.

Модель бизнес-классов представляется на высоком уровне абстракции. На этом уровне нас меньше интересует содержание атрибутов классов — достаточно имен классов и их краткого описания.

Ассоциация (association) выражает знания, которые объекты одного класса имеют относительно объектов другого класса (или других объектов того же самого класса). Эти знания касаются семантических связей между объектом и другими объектами. Существование таких связей позволяет осуществлять навигацию между объектами (программа может перемещать фокус управления между связанными объектами).

Ассоциации имеют важные свойства кратности и участия. *Кратность* (multiplicity) определяет количество возможных экземпляров класса, который может быть связан с отдельным экземпляром другого класса. Кратность определяется на обоих концах линии ассоциации, соединяющей классы. Этот показатель может принимать значения 0, 1 или *n*, где *n* — количество объектов, с которыми может быть связан конкретный экземпляр данного класса.

Для того чтобы показать, что некоторые объекты могут не иметь ассоциаций ни с одним объектом, кратность можно задавать парой чисел, т.е. 0...1 или 1...*n*. Число 0 означает, что *участие* (participation) объекта в ассоциации с другими объектами является потенциальным (т.е. объект может иметь, но может и не иметь ассоциативные связи с другими объектами.)

Агрегация (aggregation) является более строгой разновидностью ассоциации. Например, экземпляр одного класса "содержит" экземпляры другого класса. Говорят, что класс супермножества "содержит" класс подмножества или класс подмножества "является частью" класса супермножества. Например, экземпляр класса `Книга` содержит экземпляры класса `Главы`.

Обобщение (generalization) — это отношение между классами, в котором один класс "может быть" другим классом. Говорят, что суперкласс "может быть" подклассом и что подкласс "является разновидностью" суперкласса. Например, экземпляр класса `Сотрудник` может быть экземпляром класса `Менеджер`.

Довольно интересно, что часто *действующие бизнес-лица* (business actors) в модели бизнес-прецедентов использования представляют собой *бизнес-классы* (business classes) в модели бизнес-классов. Это согласуется с наблюдением за тем, что действующие лица часто одновременно бывают как внутренними, так и внешними (раздел 2.5.2). Обозначения, используемые при моделировании бизнес-классов, приведены на рис. 2.15.

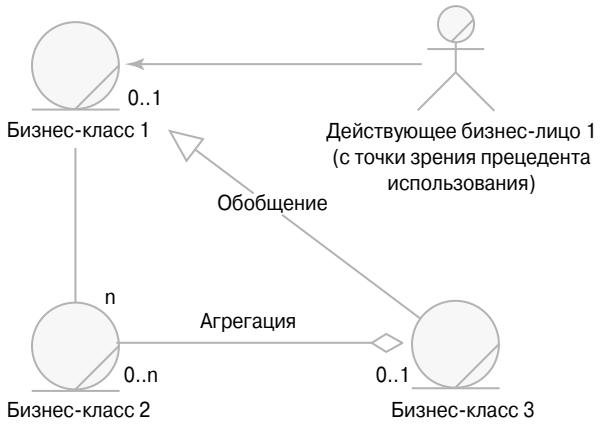


Рис. 2.15. Обозначения, используемые на диаграммах бизнес-классов

Первоначальная диаграмма бизнес-класса показана на рис. 2.16. Она содержит шесть классов, два из которых (`Supporter` и `Telemarketer`) являются производными от действующих лиц в модели бизнес-прецедентов использования. Алгоритм расписания телефонных звонков получает телефонный номер и другую информацию от класса `Supporter` и создает расписание звонков, доступных объекту класса `Telemarketer`. Этот алгоритм будет реализован в базе данных в виде *хранимой процедуры* (stored procedure).

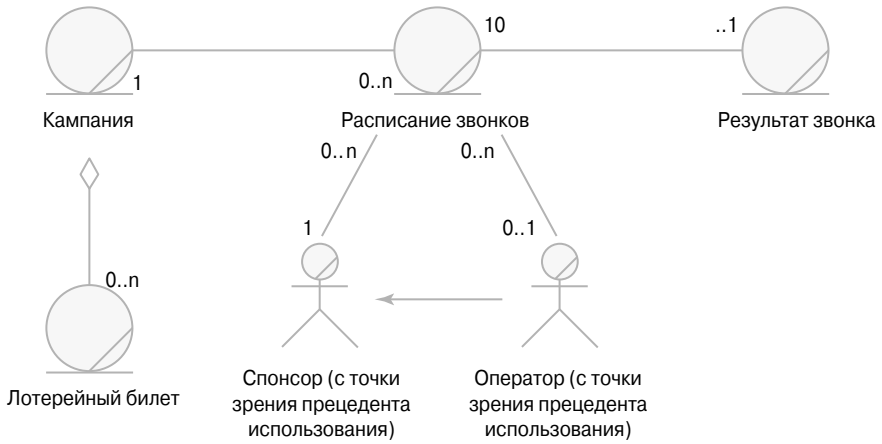


Рис. 2.16. Диаграмма бизнес-класса для системы прямого маркетинга по телефону

Пример 2.6. Прямой маркетинг по телефону

Рассмотрите постановку задачи, диаграмму контекста и диаграмму бизнес-прецедентов и бизнес-глоссарий для приложения "Прямой маркетинг по телефону" (см. разделы 1.6.4, 2.5.1–2.5.3) и постройте для диаграмму бизнес-классов. В качестве подсказок можно использовать следующие соображения.

- Акцент в системе сделан на планирование звонков. Само по себе планирование звонков является вычислительной процедурой, т.е. здесь решение является строго алгоритмическим по природе. Тем не менее очереди запланированных звонков и результаты звонков должны запоминаться в некоторой структуре данных.
- Как обсуждалось выше, может потребоваться хранить информацию о субъектах в классах.

Класс `Call scheduled` (Планируемый звонок) содержит текущую очередь звонков, включая звонки, которые в текущий момент являются активными. Результаты звонков регистрируются в классе `Call Outcome` (Результат звонка), а также распространяются в другие задействованные классы, такие как `Campaign Ticket` (Билет кампании) и `Supporter` (Спонсор).

Класс `Campaign` содержит класс `Campaign tickets` и связан с классом `Scheduled` отношением "один ко многим". Аналогично, объекты классов `Supporter` и `Telemarketer` могут обладать многими объектами класса `Call scheduled`. Ассоциация между классами `Call scheduled` и `Call outcome` имеет кратность "один к одному".

Контрольные вопросы 2.5

КВ1. Назовите синоним бизнес-прецедента использования.

КВ2. Как называется отношение, представляющее поток событий между действующими лицами и прецедентами использования?

КВ3. Перечислите три основные категории отношений между бизнес-классами.

КВ4. Каким образом в языке UML визуализируется возможное участие бизнес-классов в ассоциации?

2.6. Техническое задание

Техническое задание (requirements document) является осязаемым результатом этапа определения требований. Большинство организаций вырабатывают техническое задание в соответствии с заранее определенным шаблоном. Шаблон определяет структуру (содержание) и стиль этого документа.

Ядро технического задания состоит из формулировок (изложения) требований. В разделе 2.2.1 указывалось, что требования могут быть сгруппированы в виде *функциональных* (формулировок сервисов) и *нефункциональных требований* (формулировок ограничений). Функциональные требования можно разделить на *требования к функциям* (function requirements) и *требования к данным* (data requirements). (В литературе термин "функциональные требования" (functional requirements) используется в широком и в узком смысле. При использовании в узком смысле он означает требования к функциям.)

Кроме требований, техническое задание должно описывать *проектные вопросы* (project issues). Обычно проектные вопросы рассматриваются в начале документа, а затем в конце. Во вводной части документа рассматривается бизнес-контекст проекта, включая цель проекта, участников проекта и основные ограничения. Ближе к заключительной части документа поднимаются другие проектные вопросы, включая план-график выполнения проектных работ, бюджет, риски, документацию и т.д.

2.6.1. Шаблоны документа

Шаблоны (templates) для технического задания широко доступны. Их можно найти в учебниках, стандартах, выпускаемых такими организациями, как ISO, IEEE и др., на Web-страницах консалтинговых фирм, в программных средствах разработки и т.д. Со временем каждая организация разрабатывает собственные стандарты, которые соответствуют принятой в организации практике, корпоративной культуре, кругу читателей, типам разрабатываемых систем и т.д.

Шаблон технического задания определяет структуру документа и содержит подробные указания о содержании каждого из разделов документа. Указания могут включать содержание вопросов, мотивацию, примеры и дополнительные соображения (Robertson and Robertson, 2003).

На рис. 2.17 показано типичное оглавление технического задания. Последующие разделы включают объяснение приведенного оглавления.

2.6.2. Предварительные замечания к проекту

Часть технического задания, содержащая предварительные замечания к проекту, предназначена для руководителей, которые, вероятно, не станут подробно изучать документ целиком. В начале документа необходимо ясно обозначить *цели* и *границы* проекта, а затем описать деловой контекст системы.

Техническое задание
Содержание

1 Введение
1.1 Цель и масштаб проекта
1.2 Деловой контекст
1.3 Участники проекта
1.4 Идеи решений
1.5 Обзор документа
2 Сервисы системы
2.1 Границы системы
2.2 Функциональные требования
2.3 Требования к данным
3 Системные ограничения
3.1 Ограничения, касающиеся интерфейса
3.2 Ограничения, относящиеся к производительности
3.3 Ограничения, относящиеся к безопасности
3.4 Операционные ограничения
3.5 Политические и юридические ограничения
3.6 Другие ограничения
4 Проектные вопросы
4.1 Нерешенные проблемы
4.2 Предварительный календарный план
4.3 Предварительная смета
Приложения
Глоссарий
Бизнес-документы и формы
Ссылки

Рис. 2.17. Содержание технического задания

Техническое задание должно создать *бизнес-прецедент* (business case) для системы. В частности, в нем следует упомянуть обо всех усилиях, приложенных для обоснования необходимости системы на этапе планирования системы (см. разделы 1.2 и 2.1). Техническое задание должно прояснить вопрос о том, каким образом предлагаемая система может способствовать достижению целей, стоящих перед организацией.

В техническом задании необходимо указать *участников проекта* (см. раздел 1.1.2.1). Важно, чтобы *заказчик* выступал не в виде безликого подразделения или офиса, поэтому следует перечислить конкретные имена. Ведь именно эти люди будут решать, приемлемо ли разработанное программное обеспечение или нет.

Несмотря на то что техническое задание должно быть как можно более далеким от конкретных технических решений, все же важно обсудить идеи, касающиеся программного обеспечения, на самых ранних этапах жизненного цикла разработки. Действительно, прежде чем приступить к работе, необходимо *представить* себе ее результат (см. раздел 2.1). Особый интерес представляют готовые решения. Всегда выгоднее купить готовый продукт вместо его разработки с нуля.

Техническое задание должно содержать список существующих программных пакетов и компонентов, которые следует рассмотреть в качестве возможных вариантов. Следует заметить, что приобретение готового решения изменяет процесс разработки, однако не избавляет от необходимости проведения анализа требований и проектирования системы!

Наконец, во введении целесообразно привести *обзор оставшейся части* документа. Это может подтолкнуть занятого читателя к тому, чтобы изучить остальные части документа, а также способствует лучшему пониманию содержания документа. Обзор также может содержать пояснения методологии анализа проектирования, выбранной разработчиками.

2.6.3. Системные сервисы

Основная часть документа описания требований посвящена определению *системных сервисов* (см. раздел 2.2.1). Эта часть может занимать более половины всего документа. Это также, пожалуй, единственная часть документа, которая может содержать абстрактные модели — *модели бизнес-требований* (см. раздел 2.5).

Границы системы (scope of the system) можно моделировать с помощью *контекстной диаграммы* (см. раздел 2.5.1). Они точно определяются в пояснениях к контекстной диаграмме. Без подобного определения проект не застрахован от попыток "раздвинуть" его рамки.

Функциональные требования можно моделировать с помощью *диаграммы бизнес-прецедентов использования* (см. раздел 2.5.2). Однако диаграмма охватывает перечень функциональных требований лишь в самом общем виде. Как отмечалось в разделе 2.4, все требования необходимо идентифицировать, классифицировать и определить.

Требования к данным можно моделировать с помощью *диаграммы бизнес-классов* (см. раздел 2.5.4). Так же как и в случае функциональных требований, диаграмма бизнес-классов не дает полного определения структур данных для бизнес-процессов. Каждый бизнес-класс требует дальнейших пояснений. В техническом задании необходимо описать атрибутное наполнение классов и определить их идентифицирующие атрибуты. В противном случае невозможно будет правильно объяснить ассоциации.

2.6.4. Системные ограничения

Системные сервисы определяют, *что именно* должна делать система. Системные ограничения (см. раздел 2.2.1) определяют, *какими* условиями связана система при выполнении своих функций. Системные ограничения связаны со следующими видами требований.

- Требования к интерфейсу.
- Требования к производительности.

- Требования к безопасности.
- Эксплуатационные требования.
- Политические и юридические требования.

Требования к интерфейсу (interface requirements) определяют, как система взаимодействует с пользователями. В техническом задании определяется только "впечатление и ощущение" от графического пользовательского интерфейса. Начальное проектирование (разрисовка) графического интерфейса проводится во время *спецификации требований* и позже, во время *проектирования системы*.

В зависимости от области приложения *требования к производительности* (performance requirements) могут играть довольно значительную роль в успехе проекта. В узком смысле они задают скорость (время отклика системы) выполнения различных заданий. В широком смысле требования к производительности включают другие ограничения — в отношении надежности, готовности, пропускной способности и т.д.

Требования к безопасности (security requirements) описывают пользовательские права доступа к информации, контролируемые системой. Пользователям может быть предоставлен ограниченный доступ к данным или ограниченные права на выполнение определенных операций с данными.

Эксплуатационные требования (operational requirements) определяют программное и аппаратное обеспечение, с помощью которого должна функционировать система. Эти требования могут оказывать влияние на другие аспекты проекта, такие как подготовка пользователей и сопровождение системы

Политические и юридические требования (political and legal requirements) зачастую подразумеваются и не указываются явно в техническом задании. Подобная ошибка может обойтись очень дорого. Пока эти требования не выявлены, программный продукт может быть трудно и даже невозможно развернуть по политическим или юридическим причинам.

Возможны и *другие виды ограничений*. Например, к некоторым системам могут предъявляться повышенные *требования к удобству* (usability requirements) или *легкости сопровождения* (maintainability requirements).

Значение недвусмысленных формулировок системных ограничений трудно переоценить. Существует немало примеров проектов, которые провалились из-за упущенных или неверно понятых ограничений. Эта проблема в равной мере относится как к заказчикам, так и к разработчикам. Недобросовестные или доведенные до отчаяния разработчики могут разыграть "карту системных ограничений", чтобы получить преимущество в своем стремлении уклониться от ответственности.

2.6.5. Проектные вопросы

Заключительная часть технического задания посвящена другим проектным вопросам. Один из важных разделов этой части называется "Нерешенные проблемы" (open issues). Здесь поднимаются все вопросы, которые могут сказаться

на успехе проекта и которые не рассматривались в других разделах документа. К этим проблемам относятся ожидаемое возрастание важности некоторых требований, которые в текущий момент выходят за рамки проекта, а также любые потенциальные проблемы и отклонения в поведении системы, которые могут начаться в связи с развертыванием системы.

Необходимо разработать *предварительный календарный план проекта* (preliminary schedule), включающий в себя первоначальное распределение трудовых и других ресурсов. Для выработки стандартных плановых графиков можно использовать программные средства управления проектами, например систему PERT или диаграммы Гантта (Gantt) (Maciaszek and Liong, 2005).

Прямым результатом составления календарного графика может быть разработка *предварительной сметы* (preliminary budget). Стоимость проекта может быть выражена скорее в виде диапазона затрат, а не их конкретной суммы. При наличии надлежащим образом документированных требований для оценки затрат можно использовать один из подходящих методов (например, *балльную функциональную оценку* (function point analysis)).

2.6.6. Приложения

Приложения (appendices) содержат остальную полезную для понимания требований информацию. Основным приложением является бизнес-глоссарий (см. раздел 2.5.3). В нем определяются термины, сокращения и аббревиатуры, используемые в техническом задании. Значение хорошего глоссария трудно переоценить. Неверное истолкование терминологии таит в себе большую опасность для проекта.

Одна из особенностей, которую часто упускают из виду при составлении технического задания, заключается в том, что в проблемной области, определяемой документом, можно достаточно хорошо разобраться с помощью изучения *документов и форм*, используемых в процессах делопроизводства. При возможности следует включать в документ заполненные формы — "пустые" формы не дают такого же уровня понимания бизнес-процессов.

Перечень *ссылок* (references) содержит список документов, упоминаемых или используемых при подготовке технического задания. К ним могут относиться книги и другие опубликованные источники информации, а также — что еще более важно — протоколы совещаний, служебные записки и внутренние документы.

Контрольные вопросы 2.6

КВ1. Как можно классифицировать функциональные требования?

КВ2. Как учесть в техническом задании ограничения, имеющие значение, но выходящие за рамки системы?

Резюме

В данной главе были полностью рассмотрены вопросы, связанные с определением требований. Этот этап предшествует спецификации требований. Определение требований подразумевает их выявление и оформление в виде описательного документа. В результате спецификации требований, рассматриваемой в следующей главе, возникают более формализованные модели требований.

Разработка информационных систем опирается на результаты анализа *бизнес-процессов*. С другой стороны, *информационные процессы* могут способствовать внедрению новаций и порождать новые деловые идеи. Следовательно, при создании *моделей бизнес-процессов* чрезвычайно важно рассмотреть *стратегии их реализации*. В частности, выбор стратегии реализации определяет *мандатную архитектуру* (capability architecture) системы. Таким образом, наряду с моделированием бизнес-процессов необходимо проводить архитектурное проектирование.

Выявление требований связано с поиском в двух направлениях — в предметной области и среди прецедентов использования. Эти два направления исследования дополняют друг друга и приводят к определению модели деловой деятельности для разрабатываемой системы.

Существуют различные *методы выявления требований*: интервью с заказчиками и экспертами в предметной области, анкетирование, наблюдение, изучение документации и программных систем, создание прототипов, JAD- и RAD-методы.

Полученные от заказчиков требования могут перекрываться и противоречить друг другу. Устранение дублирования и снятие противоречий — задача бизнес-аналитика, который решает ее с помощью *согласования и оценки требований*. Для надлежащего решения этой задачи бизнес-аналитик должен построить *матрицу зависимости требований*, а также приписать требованиям определенные *риски* и *приоритеты*.

Большие проекты связаны с *управлением* большими массивами требований. Для таких проектов весьма существенным моментом является *идентификация* и *классификация* требований. Затем можно определить *иерархию* требований. Осуществление подобных шагов обеспечивает необходимый уровень *трассировки требований* на последующих стадиях проекта, а также надлежащую обработку запросов на *изменение требований*.

Несмотря на то что установление требований не подразумевает формального моделирования систем, уже на этом этапе жизненного цикла разработки программного обеспечения можно построить *бизнес-модель требований*. Бизнес-модель может быть представлена в виде трех общих диаграмм — контекстной диаграммы, диаграммы бизнес-прецедентов использования и диаграммы бизнес-классов. Кроме того, в глоссарии перечисляются определения понятий, использованных в техническом задании.

Документ, который создается в результате выполнения этапа установления требований — *техническое задание*, — начинается с *предварительных замечаний*, предназначенных в основном для руководителей. Основные части документа описывают *системные сервисы* (функциональные требования) и *системные ограничения* (нефункциональные требования). Заключительная часть документа связана с остальными проектными вопросами, включая подробности, касающиеся проектного плана-графика и бюджета.

Ключевые термины

BPEL (Business Process Execution Language). Язык выполнения бизнес-процессов.

BPMN (Business Process Modeling Notation). Система обозначений для моделирования бизнес-процессов.

CRM (Customer Relationship Management). Концепция управления отношениями с заказчиками.

JAD (Joint Application Development). Метод совместной разработки систем.

RAD (Rapid Application Development). Метод быстрой разработки системы.

Артефакт (artifact). Элемент системы BPMN, обеспечивающий дополнительную гибкость моделирования за счет расширения основных обозначений; три типа артефактов определены заранее: объект данных, группа и аннотация.

Бизнес-объект (business object). Фундаментальный класс, принадлежащий предметной области бизнеса; бизнес-сущность.

Бизнес-прецедент использования (business use case). Высокоуровневая бизнес-функция; функциональное свойство системы.

Бизнес-процесс (business process). Деятельность, в ходе которой создается нечто ценное для организации или внешнего участника проекта.

Возможность повторного использования (reusability). Нефункциональное требование, характеризующее легкость или трудность повторного использования ранее реализованных компонентов программного обеспечения во вновь создаваемых системах.

Действующее бизнес-лицо (business actor). Человек, организация, компьютерная система, устройство или другой вид действующего объекта, способные взаимодействовать с системой и извлекать выгоду из этого.

Деловая возможность (business capability). Любая потенциальная возможность, относящаяся к способности информационной системы достигать конкретных результатов.

Диаграмма (diagram). Графическое представление модели.

Диаграмма иерархии процессов (process hierarchy diagram). Диаграмма, демонстрирующая статическую структуру бизнес-процесса.

Задача (task). Атомарное действие в процессе.

Исследование деловой возможности (business capability exploration). Первый этап процесса выработки концепции решения, в ходе которого определяются потенциальные бизнес-возможности системы.

Класс (class). Абстракция, описывающая множество объектов, имеющих общие атрибуты, операции, отношения и семантические ограничения.

Контекстная диаграмма (context diagram). Высокоуровневая диаграмма DFD.

Концепция возможностей системы (solution capability envisioning). Вторая фаза процесса выработки концепции системы, направленная на разработку прецедента возможности в рамках выбранной концепции и согласование решения с участниками проекта.

Концепция решения (solution concept). Артефакт концепции возможностей системы, на вход которого поступает контекст бизнеса, а на выходе — сценарии новых способов будущей работы системы.

Мандатная архитектура (capability architecture). Архитектурный проект системы, идентифицирующий высокоуровневые компоненты системы и взаимодействия между ними.

Модель бизнес-класса (business class model). Высокоуровневая модель бизнес-классов, демонстрирующая бизнес-объекты и отношения.

Модель бизнес-прецедентов использования (business use case model). Высокоуровневая диаграмма прецедента использования, идентифицирующая основные конструкции системы.

Модель бизнес-процесса (business process model). Диаграмма, демонстрирующая динамическую структуру бизнес-процесса.

Модель границ системы (system scope model). Высокоуровневая бизнес-модель, определяющая границы и основные обязанности системы.

"Мозговой штурм" (brainstorming). Метод проведения совещаний, позволяющий найти новые идеи или решения конкретной задачи путем отбрасывания предрассудков, социальных запретов и правил.

Надежность (reliability). Нефункциональное требование, связанное с частотой и серьезностью отказов системы, а также легкостью возобновления ее работы.

Нефункциональное требование (non-functional requirement). См. *Формулировка ограничения*.

Пакетная разработка (package-based development). Процесс разработки, в ходе которого решение создается на основе готовых программных пакетов, таких как COTS, ERP и CRM.

Покомпонентная разработка (component-based development). Процесс разработки, в ходе которого решение создается путем интеграции компонентов

программного обеспечения, поступающих от многочисленных поставщиков и бизнес-партнеров; чаще всего основывается на архитектурных моделях SOA и/или MDA.

Потоковый объект (flow object). Основной элемент системы BPMN; существуют три категории потоковых объектов: события, действия и шлюзы.

Представление решения (solution envisioning). Подход, ориентированный на бизнес-ценности и предназначенный для обеспечения информационной услуги, которая решает насущную бизнес-проблему или стимулирует внедрение бизнес-новации.

Прецедент возможности (capability case). Идея, на основе которой создается бизнес-прецедент для бизнес-возможности.

Провозможность (probortunity). Сочетание слов "проблема" и "возможность"; используется при организации "мозгового штурма".

Проектирование возможностей программного обеспечения (software capability design). Третья фаза процесса выработки концепции решения, на котором выбирается технология реализации системы, разрабатывается ее мандатная архитектура, создаются бизнес-прецеденты, формируются планы и проводится анализ рисков.

Производительность (perfomance). Нефункциональное требование, определяющее время реакции системы, ее пропускную способность, потребление ресурсов, возможное количество одновременно обслуживаемых пользователей и т.д.

Прототип (prototype). Приблизительная рабочая модель решения, представляющая графический пользовательский интерфейс (GUI) и имитирующая поведение системы при разных действиях пользователя.

Процесс (process). См. *Бизнес-процесс*.

Пул (pool). Элемент системы BPMN, представляющий бизнес-сущность, участвующую в процессе; иногда называется *дорожкой* (swimlane).

Разработка на заказ (custom development). Индивидуальное проектирование программного обеспечения с нуля, охватывающее все этапы процесса разработки программного обеспечения (жизненный цикл) и выполняемое внутренними подразделениями и/или по контракту консалтинговыми и проектными фирмами.

Риск (Risk). Угроза плану проекта, связанная со сметой, временем, ресурсами и т.д.

Связующий объект (connecting object). Элемент системы BPNM, использующийся для связывания потоковых объектов, чтобы определить структуру бизнес-процесса; существуют три категории связующих объектов: потоки операций, потоки сообщений и ассоциации.

Удобство (usability). Нефункциональное требование, определяющее легкость использования системы.

Формулировка ограничения (constraint statement). Ограничение, которому должна удовлетворять система.

Формулировка сервиса (service statement). Требование, определяющее услугу, ожидаемую от системы.

Функциональное требование (functional requirement). См. *Формулировка сервиса*.

Многовариантные тесты

MT1. Что называется атомарным процессом в системе BPMN?

а. Действие.

б. Задача.

в. Событие.

г. Задание.

MT2. Что называется потоком сообщений в системе BPMN?

а. Потоковый объект.

б. Дорожка.

в. Артефакт.

г. Коннектор.

MT3. Что является элементом моделирования, определяющим бизнес-ценность функционального свойства в рамках концепции решения?

а. Бизнес-прецедент использования.

б. Прецедент решения.

в. Прецедент возможности.

г. Бизнес-прецедент.

MT4. Какой метод выявления требований работает с понятием провозможности?

а. Анкета.

б. "Мозговой штурм".

в. JAD.

г. Ни один из перечисленных.

MT5. Какой метод выявления требований работает с понятием триггерного вопроса?

а. Анкета.

б. RAD.

в. JAD.

г. Ни один из перечисленных.

МТ6. Какое отношение утверждает, что один класс может быть разновидностью другого?

а. Обобщение.

б. Агрегация.

в. Ассоциация.

г. Ни один из перечисленных.

МТ7. Что из перечисленного ниже является требованием к интерфейсу?

а. Функциональное требование.

б. Системный сервис.

в. Системное ограничение.

г. Ни одно из перечисленных.

Вопросы

В1. В разделе 2.1.2 указано, что важной целью моделирования бизнес-процессов является способность перевести его обозначения BPMN (Business-Process Modeling Notation) на язык BPEL (Business Process Execution Language). Найдите в Википедии (`http://en.wikipedia.org/wiki/Main Page`) описание языка BPEL. Прочитайте работу Михельсона (Michelson, 2005), указанную в списке библиографии. Подумайте о следующем определении языка BPEL: "BPEL — это ...язык, основанный на языке XML и спецификациях основных Web-сервисов и предназначенный для определения и управления долгосрочными сервисами и процессами" (Michelson, 2005). Расшифруйте это определение и объясните входящие в него понятия.

В2. Выработка концепции решения предполагает проведение специальных семинаров. Каждый семинар проводится в течение одного-двух дней и рассматривает вопросы, связанные с главными проблемами проекта: 1) исследование текущего состояния; 2) исследование изменений — анализ причин и трендов; 3) мандатное проектирование, представляющее собой сплав технологического и делового проектирования; 4) определение целей; 5) план действий. Определите перечень вопросов, относящихся к каждой из основных проблем. Какие вопросы вы хотели бы задать на семинаре?

В3. Найдите в Интернете и/или в других источниках информацию, позволяющую расшифровать аббревиатуру FURPS. Кратко опишите смысл этого обозначения. Как модель FURPS связана с функциональными и нефункциональными требованиями?

В4. Найдите в Интернете и/или в других источниках информацию о модели Мак-Колла для оценки качества программного обеспечения (McCall software quality model).

В5. В процессе установления требований согласовываются общие требования, связанные с представлениями о предметной области, и требования, полученные в результате анализа прецедентов использования. Объясните, в чем состоит различие между этими двумя типами требований? Может ли один из них превалировать над другим в процессе установления требований? Объясните свои доводы.

В6. При проведении интервью рекомендуется избегать некорректных, предвзятых и наводящих вопросов. Можете ли вы представить себе ситуацию, при которой подобные вопросы могли бы принести выгоду проекту?

В7. Чем опрос отличается от анкетирования при определении требований?

В8. Какие действия следует выполнить при подготовке и проведении наблюдений в ходе определения требований? Какими принципами следует руководствоваться при этом?

В9. Что представляет собой создание прототипов? Насколько оно полезно для установления требований?

В10. В чем заключаются основные преимущества "мозгового штурма" и метода JAD по сравнению с другими методами определения требований?

В11. Что такое "размывание границ проекта"? Как справиться с этим явлением при определении требований?

В12. Опишите предназначение и структуру технического задания. Попробуйте найти ответ в Интернете.

В13. Следует ли нумеровать требования?

В14. Чем отличаются действующие лица в диаграммах бизнес-прецедентов использования от внешних сущностей в контекстных диаграммах?

Упражнения. Затраты на рекламу

ЗР1. Рассмотрим задачу 5, посвященную измерению затрат на рекламу (см. раздел 1.6.5), и диаграмму иерархии процессов, описанную в примере 2.1 (см. раздел 2.1.1.2). При управлении контрактами и дебиторской задолженностью организации, оценивающие затраты на рекламу, должны учитывать следующие факторы.

- Как правило, контракты заключаются на один год и ежегодно пересматриваются с учетом регистрационных записей о клиентах. Регистрационные записи позволяют определить, насколько часто клиент обращался в компанию и как часто у него возникали проблемы на протяжении года. Информация, содержащаяся в регистрационных записях, позволяет проводить переговоры быстро, в течение одного совещания.
- Счета клиентам выставляются раз в месяц. Содержание счета зависит от контракта и дополнительных услуг, не предусмотренных в контракте.
- Если контракт истек или счет не оплачен, то обслуживание клиентов может быть приостановлено, пока контракт не будет пересмотрен.
- Счета генерируются и рассылаются в конце каждого месяца. Оплата отслеживается. Детали счетов и оплаты записываются в отчеты.

Постройте диаграмму бизнес-процесса "Контракты и дебиторская задолженность".

ЗР2. Рассмотрите задачу 5 из раздела 1.6.5. Сосредоточьте внимание только на бизнес-процессах. В частности, проигнорируйте разделы "Управление контактами" и "Контракты и дебиторская задолженность" (см. рис. 2.2 в разделе 2.1.1.2).

Постройте контекстную диаграмму системы. Опишите модель.

ЗР3. Рассмотрите задачу 5 из раздела 1.6.5. Сосредоточьте внимание только на бизнес-процессах. В частности, проигнорируйте разделы "Управление контактами" и "Контракты и дебиторская задолженность" (см. рис. 2.2 в разделе 2.1.1.2).

Постройте диаграмму бизнес-прецедентов использования системы. Опишите модель.

ЗР4. Рассмотрите задачу 5 из раздела 1.6.5.

Напишите бизнес-глоссарий. Внесите в него только понятия, имеющие отношение к оценке затрат на рекламу.

ЗР5. Рассмотрите задачу 5 из раздела 1.6.5.

Постройте диаграмму бизнес-классов для системы оценки затрат на рекламу. Используйте бизнес-глоссарий, разработанный в упражнении А4, и смоделируйте лишь бизнес-классы, имеющие отношение к оценке затрат на рекламу. Опишите модель.

Упражнения. Регистрация времени

РВ1. Рассмотрите задачу 6 из раздела 1.6.6.

Постройте высокоуровневую диаграмму прецедентов использования для системы регистрации времени. Опишите эту модель.

РВ2. Рассмотрите задачу 6 из раздела 1.6.6.

Напишите глоссарий для системы регистрации времени.

РВ3. Рассмотрите задачу 6 из раздела 1.6.6.

Постройте высокоуровневую диаграмму классов для системы регистрации времени. Опишите эту модель.

Ответы на контрольные вопросы

Контрольные вопросы 2.1

КВ1. Система BPNM.

КВ2. Потоковые объекты, связующие объекты, пулы и артефакты.

КВ3. Нет, не может. Пул может обмениваться сообщениями с помощью потоков или ассоциаций с общими артефактами.

КВ4. Концепция решения.

КВ5. Мандатная архитектура.

КВ6. Разработка на заказ, пакетная разработка и покомпонентная разработка.

Контрольные вопросы 2.2

КВ1. Бизнес-аналитик.

КВ2. Функциональные требования (формулировки сервисов) и нефункциональные требования (формулировки ограничений, дополнительные требования).

КВ3. Многовариантные вопросы, оценочные вопросы и вопросы с ранжированием.

КВ4. Ведущий, секретарь, заказчики и разработчики.

КВ5. Модель SWAT.

Контрольные вопросы 2.3

КВ1. Контекстная диаграмма.

КВ2. Противоречивые и перекрывающиеся требования.

КВ3. Риск изменчивости.

Контрольные вопросы 2.4

КВ1. Уникальный идентификатор, порядковый номер в иерархии документов и порядковый номер в категории требований.

КВ2. Инструмент управления конфигурацией программного обеспечения.

КВ3. Подозрительная траектория — признак в матрице трассировки, свидетельствующий о том, что отношение между двумя связанными требованиями в результате изменения одного из них может быть нарушено.

Контрольные вопросы 2.5

КВ1. Свойство системы.

КВ2. Отношение связи.

КВ3. Ассоциация, обобщение и агрегация.

КВ4. Оно визуализируется в определении кратности с помощью нуля, приписанного возле соответствующего конца ассоциации.

Контрольные вопросы 2.6

КВ1. Функциональные требования можно разделить на требования к функциям и данным.

КВ2. Их можно перечислить в разделе “Нерешенные проблемы”.

Ответы к многовариантным тестам

МТ1. б

МТ2. г

МТ3. в

МТ4. б

МТ5. г (“мозговой штурм”, посвященный этому понятию)

МТ6. а

МТ7. в

Ответы на вопросы с нечетными номерами

B1

BPEL — это язык, предназначенный для формальной спецификации бизнес-процессов и моделей взаимодействия процессов. Этот язык основан на языке XML. Язык XML (Extensible Markup Language) описывает структуру данных в Web-документах, чтобы обеспечить обмен структурированными документами и обмен данными через Интернет.

Язык BPEL расширяет модель взаимодействия Web-сервисов, представляющих собой компоненты программного обеспечения, использующие систему обмена сообщениями XML и доступные в Интернете. Для того чтобы обеспечить доступность Web-сервиса в Интернете, этот компонент описывает свои функциональные свойства с помощью открытого интерфейса и предлагает помощь, чтобы заинтересованные лица могли его найти. Web-сервис представляет собой разновидность сервисов, которые могут быть частью модели SOA.

Язык BPEL ориентирован на высокоуровневое программирование и обеспечивает поддержку спецификаций крупных бизнес-операций между сотрудничающими сервисами. "Оркестровка — это разновидность взаимодействия, в которой первичный сервис непосредственно вызывает другие сервисы. Первичный сервис знает последовательность действий и интерфейсов, ответов и возвратных состояний вызываемых сервисов" (Michelson, 2005).

B3

Модель FURPS, предложенная компанией *Hewlett-Packard*, предназначена для оценки качества программного обеспечения (Grady, 1992). Эта аббревиатура означает пять факторов — функциональность (functionality), удобство (usability), надежность (reliability), производительность (performance) и легкость сопровождения (supportability).

- *Функциональность* определяет набор свойств и возможностей системы программного обеспечения. Это определение включает в себя список предлагаемых функций и аспектов системы, связанных с безопасностью ее работы.
- *Удобство* описывает восприятие человеком легкости использования системы. Это понятие включает в себя такие аспекты системы, как ее эстетический вид, непротиворечивость, доступность документации, наличие справочной системы и возможности обучения, необходимые для эффективного и производительного использования системы.
- *Надежность* описывает частоту и серьезность отказов системы, точности полученных результатов, среднее время работы до очередного сбоя, способность возобновления работы после сбоя и общую предсказуемость системы.

- *Производительность* оценивает время реакции системы (среднее и максимальное), пропускную способность, потребление ресурсов, эффективность при разной нагрузке и т.д.
- *Легкость сопровождения* определяет степень сложности поддержки системы. Она является сочетанием разнообразных свойств, таких как понятность, эксплуатационная надежность (приспособляемость к изменениям и удобство обслуживания), возможность масштабирования (расширения), тестирования и конфигурирования, легкость инсталляции и локализации.

Модель FURPS обычно применяется для оценки качества поставляемого программного обеспечения. Однако эти пять показателей качества могут служить удобным средством классификации пользовательских требований к разрабатываемой системе. В этой классификации функциональный компонент модели FURPS связан с *функциональными требованиями*, а остальные четыре — с *нефункциональными требованиями*.

Пропорция один к четырем, характеризующая соотношение между функциональными и нефункциональными требованиями, никоим образом не отражает относительную важность эти двух категорий требований. При разработке системы затраты сил и денег, направленные на удовлетворение функциональных требований (системных сервисов), часто превышают затраты сил и денег на обеспечение нефункциональных требований (системных ограничений).

B5

Требования, связанные с предметной областью, являются следствием общих представлений о сфере приложения системы. К ним относятся опыт эксперта (или аналитика) в предметной области, публикации, посвященные предметной области, общепринятая практика, а также решения, воплощенные в готовых системах.

Требования, связанные с прецедентами использования, выявляются в результате изучения специфической бизнес-практики и процессов. Они относятся к способам ведения бизнеса в конкретной организации. Некоторые требования, связанные с прецедентами использования, согласуются с требованиями, связанными с предметной областью, а другие — нет (поскольку они отражают конкретные способы ведения бизнеса в организации).

Как правило, оба вида требований определяются более или менее параллельно. Иногда требования, связанные с предметной областью, изучаются в первую очередь, но в итоге все эти требования должны оцениваться заказчиками. Иначе говоря, требования, связанные с предметной областью, либо инкорпорируются в прецеденты использования (и становятся требованиями, связанными с прецедентами использования), либо отбрасываются.

Процесс оценки *требований, связанных с предметной областью,* часто составляет главную часть процесса пересмотра модели бизнес-классов (а на более позд-

них этапах — моделей классов). Этот факт отражен на рис. 2.9 с помощью отношения зависимости между моделью бизнес-прецедентов использования и моделью бизнес-классов.

Несмотря на то что на рис. 2.9 обратная связь между моделью бизнес-классов (и, следовательно, требованиями, связанными с предметной областью) не показана, она вполне возможна. Если обратная связь существует, то это может означать, что "клиент не всегда прав", т.е. заказчик готов к переоценке требований, связанных с прецедентами использования, чтобы добиться их согласованности с общими требованиями, связанными с предметной областью.

На более поздних этапах анализа и проектирования на первый план выходят требования, связанные с прецедентами использования. Все модели и артефакты разработки постоянно сравниваются с требованиями, связанными с прецедентами использования.

В7

Как опрос, так и анкетирование с целью выявления требований опираются на анкеты. Основная разница между ними заключается в том, что *опрос* направлен на проверку некоторых фактов, а *анкетирование* — на их поиск.

Для опросов и анкетирования характерны неточность и предвзятость респондентов. Результаты опросов могут быть искажены из-за неспособности респондентов ответить на вопрос или стремления приукрасить факты. Результаты анкетирования могут страдать такими же недостатками, но они могут быть усугублены вопросами, направленными на выявление оценки или субъективного мнения. Еще больше сложностей порождают вопросы, связанные с необходимостью ранжировать ответы по соответствующей шкале.

В9

Создание прототипов — это способ быстрого моделирования предлагаемой системы. *Прототип* представляет собой неэффективное и грубое решение, предполагающее множество компромиссов. Основное предназначение прототипов — дать пользователям "почувствовать" систему.

Прототип оценивается пользователями и впоследствии используется для оценки и уточнения требований к разрабатываемой системе. Прототип обеспечивает динамическую визуализацию и имитирует взаимодействие с пользователем. Это позволяет пользователям глубже понять требования к программному обеспечению и оценить его поведение.

Если для оценки прототипа выделено достаточно ресурсов, а изменения сделаны вовремя, прототип становится лучшим способом определения требований.

B11

Размывание границ проекта выражается в росте пользовательских ожиданий в ходе разработки и предъявлении новых требований к системе. Как правило, новые требования никак не учтены в смете и календарном плане.

Для того чтобы предотвратить размывание границ, все *запросы на изменение требований* должны документироваться и рассматриваться в контексте высокоуровневой модели границ системы, описанной в техническом задании. Высокоуровневая модель границ часто описывается с помощью контекстной диаграммы. Более подробные требования описываются в диаграммах прецедентов использования.

Любые попытки выдвинуть новые требования должны отражаться на контекстной диаграмме и/или диаграммах прецедентов, а пользователям необходимо объяснить динамику изменения параметров проекта. Эти параметры должны отражать изменение календарного плана и сметы.

Размывание границ возникает не только из-за изменения условий ведения бизнеса, но и из-за недостаточно тщательной работы на этапе определения требований. Снизить риск размывания границ можно с помощью современных методов идентификации требований, например создания прототипов.

В заключение следует заметить, что разработка системы с помощью коротких итераций и частых поставок позволяет снизить необходимость изменения требований в рамках итераций. Запросы на изменения требований, возникающие между итерациями, менее опасны.

B13

Система создается для того, чтобы удовлетворить требования пользователей. Во всех системах, кроме самых простых, требования распределяются по нескольким уровням: от наиболее абстрактных до очень конкретных. Общее количество требований часто может достигать нескольких тысяч.

Для того чтобы обеспечить выполнение требований заказчиков, их необходимо *структурировать и нумеровать*. Лишь после этого их можно систематически проверять и контролировать.

Управление проектом и его изменениями зависит от точности *идентификации* всех требований. Эти требования распределяются среди разработчиков и субподрядчиков, планы структурируются в соответствии с группами требований, дефекты отслеживаются на основе нумерации требований и т.д.

Объяснения упражнений. Затраты на рекламу

Более подробно системы, предназначенные для оценки затрат на рекламу, описаны в работе (Maciaszek and Liong, 2005).

ЗР1

Диаграмма бизнес-процесса показана на рис. 2.18. Процесс и задачи распределены по двум пулам — отделу поддержки отношений с заказчиками и отделу обслуживания дебиторской задолженности. Для уточнения потоков действий используются таймеры и правила обработки событий.

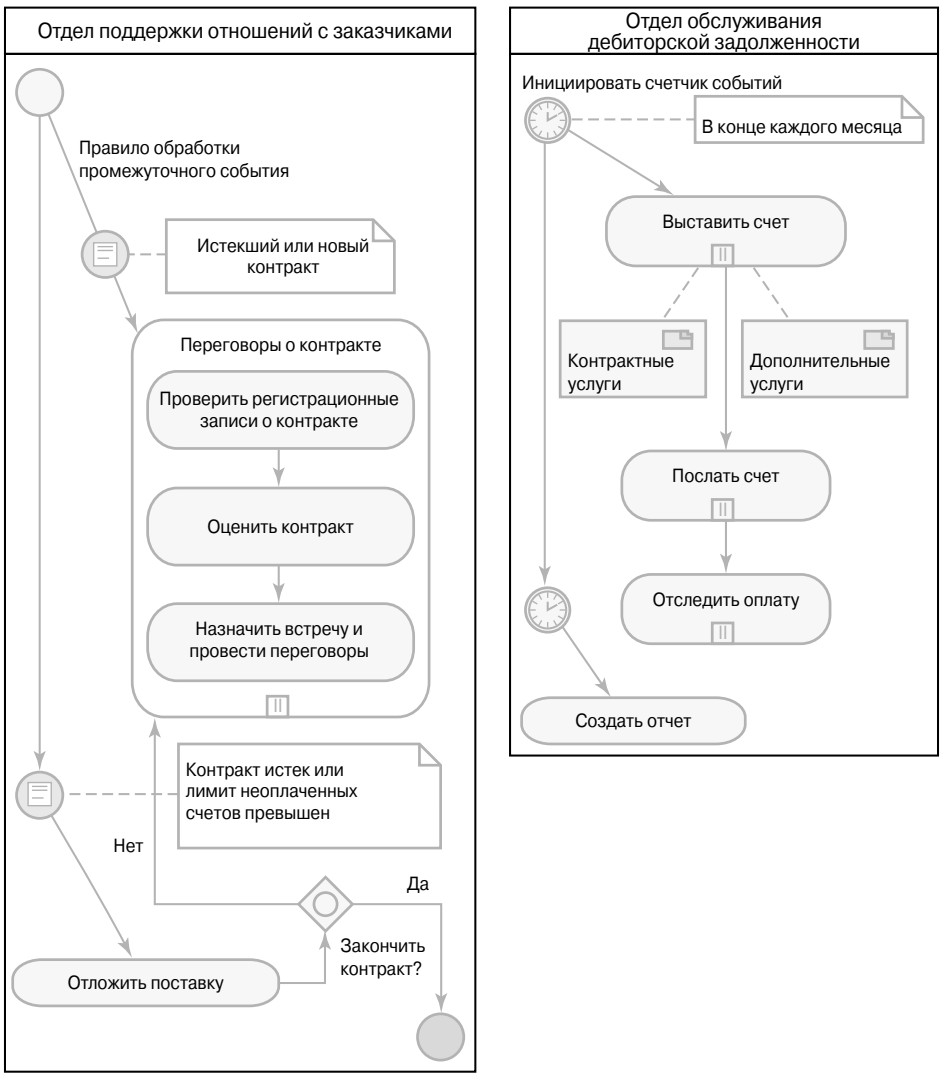


Рис. 2.18. Диаграмма бизнес-процесса для оценки затрат на рекламу

ЗР2

Контекстная диаграмма, приведенная на рис. 2.19, представляет собой высокоуровневую модель, определяющую границы системы АЕ, предназначенной для оценки затрат на рекламу. На этой диаграмме показаны две внешние сущности: `Средство массовой информации` и `Клиент системы АЕ`. Средство массовой информации поставляет данные о рекламе в `Систему АЕ`, а `Клиент системы АЕ` получает от нее отчет о мониторинге кампании и отчет о затратах.

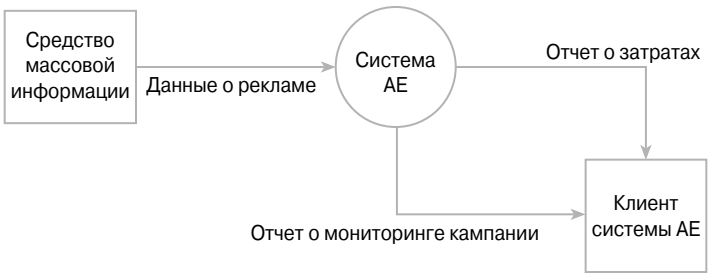


Рис. 2.19. Контекстная диаграмма бизнес-процесса для оценки затрат на рекламу

Преобразование данных о рекламе в отчет является внутренней функцией системы `АЕ`. Верификация рекламы, ручной ввод данных и другие функции системы `АЕ` также подразумевают потоки данных, но эти данные рассматриваются как внутренние потоки системы. Например, поступление газет, журналов и видеокассет, а также ввод данных в систему считаются внутренними функциями (по крайней мере, в настоящее время).

ЗР3

Диаграмма бизнес-прецедентов, представленная на рис. 2.20, организована по принципу "сверху вниз" и демонстрирует последовательность выполнения бизнес-функций.

Модель бизнес-прецедентов использования разделяет систему АЕ на четыре бизнес-прецедента, связанных односторонними ассоциациями. Последовательность ассоциаций определяет основной рабочий процесс системы АЕ — от сбора данных до верификации, валоризации и создания отчета.

Бизнес-процесс `Сбор данных` взаимодействует с действующим лицом `Регистратор данных` и получает от него информацию о рекламе (как правило, в электронном виде). Процесс сбора данных включает в себя сравнение полученной информации с данными, записанными в базе данных. Процесс сравнения данных не выделен в виде отдельного прецедента использования.

Прецедент использования `Сбор данных` подтверждает, что полученные данные являются корректными и не противоречат остальной информации (другим объявлениям, программам и т.п.). После ввода и верификации данные о рекламе

поступают в прецедент использования `Валоризация`. На этом этапе происходит оценка затрат на рекламу.

Прецедент использования `Создание отчета` предназначен для генерации отчетов, которые затем рассылаются заказчикам. Отчеты распространяются в разной форме, например по электронной почте, на компакт-дисках или на бумаге.

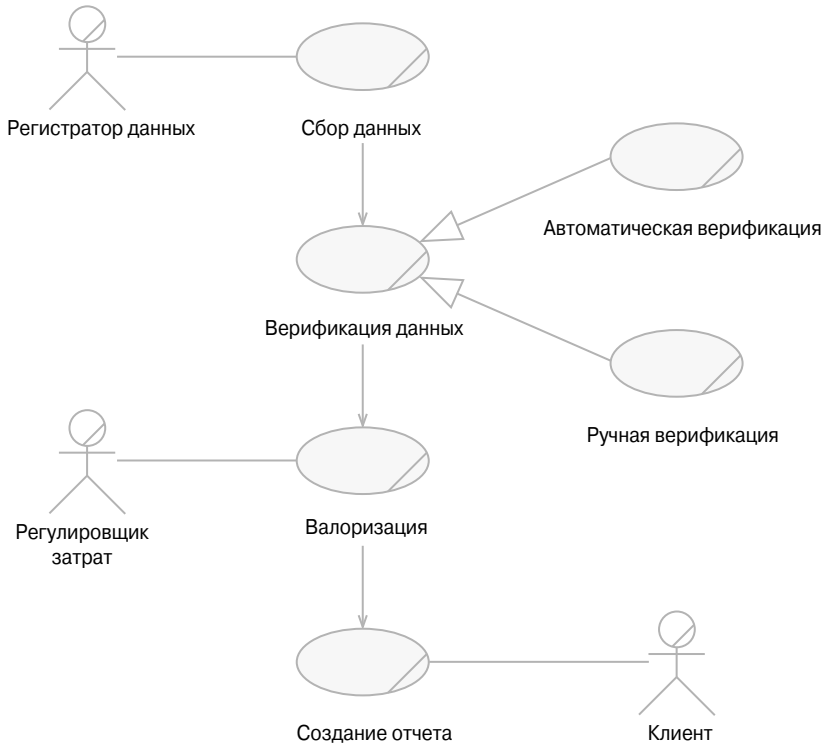


Рис. 2.20. Диаграмма бизнес-прецедентов для системы оценки затрат на рекламу

ЗР4

Глоссарий приведен в табл. 2.3

Таблица 2.3. Деловой глоссарий для оценки затрат на рекламу

Термин	Определение
Экземпляр рекламного объявления	Конкретная форма *рекламы*, т.е. каждое появление рекламы в соответствующем *источнике*
Рекламная связь	Ассоциация между *рекламой*, *продукцией*, которую она описывает, *рекламодателем*, оплачивающим рекламное объявление, и *агентством*, обязанным регистрировать затраты

Окончание табл. 2.3

Термин	Определение
Реклама	Уникальная форма творческой работы, которая может быть многократно озвучена на радио, показана на экране, опубликована в газете или распространена иным способом. Каждое появление рекламы в соответствующем *источнике* регистрируется в системе AE в виде *экземпляра рекламного объявления*
Рекламодатель	Компания, использующая *источник* для распространения *рекламы* с целью продвижения своей *продукции*
Агентство	Организация, занимающаяся планированием рекламной кампании и заключением контрактов со средствами массовой информации в интересах *рекламодателя*. Цель *агентства* — оптимизация затрат на рекламу
Категория	Название в иерархической классификации продукции. Иерархия может содержать неограниченное количество уровней (т.е. категорий, подкатегорий, вплоть до отдельных продуктов). *Продукция* может классифицироваться лишь на самом низком уровне иерархии категорий
Организация	Бизнес-сущность, с которой взаимодействует система AE. Существуют разные виды организации, включая *рекламодателей*, *агентства* и *источники рекламы*
Источник	Организация, распространяющая *рекламу*. Ею может быть телевизионный канал, радиостанция или компания, распространяющая рекламу в кинотеатрах или на улицах
Продукция	Товар или услуга, которые могут быть предметом рекламы. Продукция может разделяться на *категории*

ЗР5

Выделяются шесть бизнес-классов (рис. 2.21). Класс `AdvertismentIstance` (Экземпляр рекламного объявления) учитывает отдельные появления рекламы. Класс `Advertisment` (Реклама) представляет собой уникальную форму творческой работы, которая может быть многократно озвучена на радио, показана на экране, опубликована в газете или распространена иным способом. Класс `AdvertismentProduct` (Рекламируемая продукция) описывает товар или услуги, являющиеся предметом рекламы.

Кроме того, существуют три организационных класса. Класс `Outlet` (Источник) хранит информацию о средствах массовой информации. Класс `Advertiser` (Рекламодатель) описывает организацию, использующую средства массовой информации для рекламы своей продукции. Класс `Agency` (Агентство) описывает организацию, занимающуюся планированием рекламной кампании и заключением контрактов со средствами массовой информации в интересах рекламодателя.

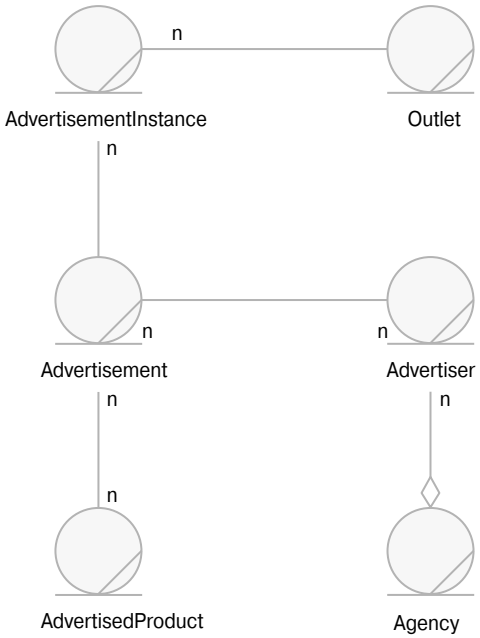


Рис. 2.21. Диаграмма бизнес-классов для системы оценки затрат на рекламу

На диаграмме также показаны основные отношения между бизнес-классами. Отношение между классами `Agency` и `Advertiser` является агрегацией; другие отношения являются ассоциациями.

ГЛАВА 3

Основы визуального моделирования

Цели

3.1. Ракурс прецедентов использования
3.2. Ракурс деятельности
3.3. Ракурс структуры
3.4. Ракурс взаимодействий
3.5. Ракурс конечных автоматов
3.6. Ракурс реализации
Резюме
Ключевые термины
Многовариантные тесты
Вопросы
Упражнения
Упражнения. Магазин видеокассет
Ответы на контрольные вопросы
Ответы к многовариантным тестам
Ответы на вопросы с нечетными номерами
Объяснение упражнений с нечетными номерами
Объяснение упражнений. Магазин видеокассет

Цели

В данной главе изложены основы визуального объектного моделирования с помощью диаграмм UML. Каждая диаграмма UML акцентирует внимание на определенном *ракурсе* разрабатываемой системы. Для того чтобы понять систему в целом, необходимо создать и объединить много диаграмм UML, представляющих разные ракурсы системы. Чтобы проиллюстрировать принципы интеграции диаграмм, в главе используется одна и та же предметная область — учебный пример "Магазин видеокассет" (см. раздел 1.6.2 главы 1).

Предполагается, что читатели знают основы объектно-ориентированной технологии, в противном случае мы рекомендуем им сначала изучить приложение "Основы объектной терминологии". Читатели, владеющие объектной терминологией, могут использовать это приложение в качестве справочника.

Прочитав эту главу, читатели будут

- владеть методами описания разных ракурсов разрабатываемой системы с помощью модели UML;
- понимать, что реальная ценность каждой модели заключается не в графическом представлении, а в текстуальном описании и других формах спецификации;
- осознавать, что моделирование поведения системы с помощью прецедентов использования и прецедентов деятельности требует применения структурных моделей, включая диаграммы классов;
- понимать, что диаграммы классов представляют собой наиболее полное определение разрабатываемой системы, а итоговая реализация является реализацией классов и других объектов, определенных в модели классов;
- признавать важность моделирования взаимодействий при определении и анализе функционирования системы в реальном времени;
- осознавать необходимость конструирования моделей конечных автоматов для классов и других элементов системы, демонстрирующих динамические изменения состояний и соответствующих строгим бизнес-правилам;
- владеть знаниями о моделях компонентов и развертывания, позволяющих описать ракурс реализации системы.

3.1. Ракурс прецедентов использования

Модель прецедента использования (use case model) — это основной инструмент языка UML, предназначенный для моделирования поведения системы. Модель функционирования описывает деловые транзакции, операции и алгоритмы, выполняемые над данными. Существует несколько методов визуализации, использующихся для моделирования поведения системы, — диаграмма прецедентов использования, диаграмма последовательностей, диаграмма коммуникации и диаграмма деятельности.

На практике важность **прецедентов использования** (use cases) еще выше, чем в теории. Прецеденты использования направляют процесс разработки программного обеспечения на всем протяжении его жизненного цикла, от анализа требований до тестирования и сопровождения. Они являются главным инструментом в большинстве видов деятельности, связанных с проектированием (см. рис. 2.10 и раздел 2.5 главы 2).

Функционирование системы — это ее реакция в ответ на внешние события. В языке UML внешне наблюдаемое и допускающее тестирование поведение системы фиксируется в виде прецедентов использования. Поскольку модели можно применять на любом уровне абстракции, прецедент использования может описывать как функционирование системы в целом, так и работу любой части системы, например отдельных подсистем, компонентов или классов.

Прецедент использования выполняет бизнес-функцию, которая является *видимой извне* (outwardly visible) для действующего лица и допускает последующее изолированное *тестирование* в процессе разработки. *Действующее лицо* (actor) — это некто или нечто (человек, машина и т.д.), взаимодействующее с прецедентом и ожидающее получить некий полезный результат.

Диаграмма прецедентов использования (use case diagram) — это наглядное представление действующих лиц и прецедентов использования вместе с любыми дополнительными определениями и спецификациями. Диаграмма прецедентов представляет собой не просто некую схему, а является полностью документированной частичной моделью предполагаемого поведения системы. Такое же понимание применимо в отношении других диаграмм языка UML. Такая модель называется частичной, поскольку обычно *модель UML* (UML model) состоит из многих диаграмм (и связанной с ними документации), представляющих разные ракурсы разрабатываемой системы.

Следует еще раз подчеркнуть (см. раздел 2.5.2 главы 2), что модель прецедентов использования можно рассматривать как обобщенный способ описания всех бизнес-процессов, а не только процессов, протекающих в информационных системах. В модель прецедентов использования необходимо включить все неавтоматизированные бизнес-процессы, а затем решить, какие из них следует автоматизировать (и тем самым превратить в процессы информационной системы). Несмотря на привлекательность такого подхода, он не стал стандартным в системном моделировании, и, как правило, в модель прецедентов использования включаются лишь автоматизированные процессы.

3.1.1. Действующие лица

Действующие лица (actors) и прецеденты использования определяются в результате анализа функциональных требований. Функциональные требования воплощаются в прецедентах использования. Прецеденты использования удовлетворяют функциональные требования за счет предоставления действующему лицу полезного результата. При этом не имеет значения, в какой последовательности бизнес-аналитик решает свои задачи: сначала обозначает действующих лиц, а затем прецеденты использования, или наоборот.

Действующее лицо — это **роль** (role), которую некая сущность, внешняя по отношению к *субъекту*, играет в прецеденте использования. Действующее лицо не является конкретным экземпляром какой-либо сущности, поэтому нельзя говорить, что некий Джо является действующим лицом. Этот Джо может играть роль заказчика и представляться в модели прецедентов использования как действующее лицо Заказчик. В целом действующее лицо Заказчик может быть даже не человеком, а организацией или машиной.

Субъект (subject) — это любая группа прецедентов использования, для которой создана модель прецедентов (например, подсистема, компонент или класс). Действующее лицо *связывается* (comminicates) с субъектом, например, обмениваясь с ним сигналами и данными.

Типичным графическим изображением действующего лица является фигурка человечка (рис. 3.1). В общем случае субъект может быть показан в виде прямоугольного символа **класса**. Подобно обычному классу, действующее лицо может обладать атрибутами и операциями (связанными с событиями, сообщения о которых оно отправляет и получает). Три способа графического представления действующих лиц показаны на рис. 3.1.

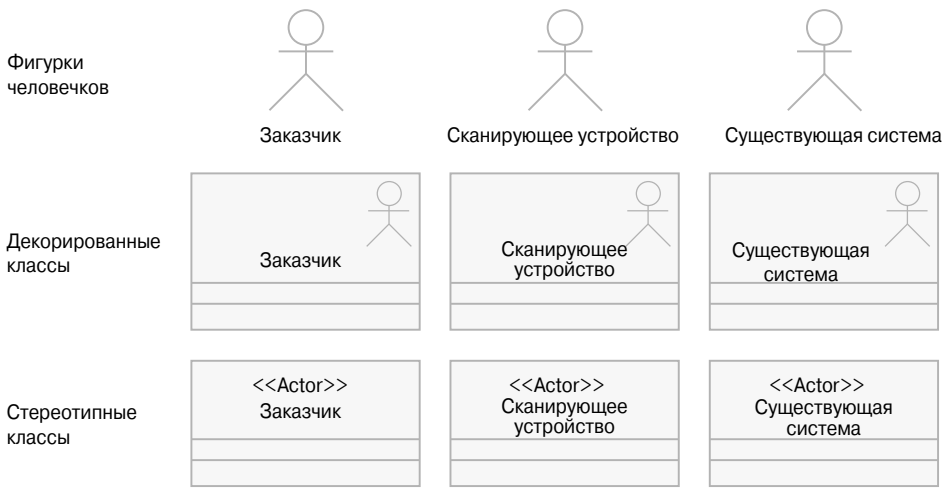


Рис. 3.1. Три способа графического представления действующих лиц

3.1.2. Прецеденты использования

Прецедент использования представляет собой некий целостный набор функций, имеющих определенную ценность для действующего лица. Однако не все прецеденты использования должны быть непосредственно связаны с действующим лицом. Такие прецеденты могут быть связаны с другими прецедентами, в свою очередь связанными с действующим лицом.

Прецеденты использования можно группировать по отношению к субъекту. "Каждый прецедент использования определяет некоторое функциональное свойство, возможно, с вариантами, которое субъект может реализовать в сотрудничестве с одним или несколькими действующими лицами. Прецеденты использования определяют предлагаемое функциональное свойство субъекта без ссылок на его внутреннюю структуру" (UML, 2005).

Прецеденты можно вывести на основе идентификации задач для субъекта. Для этого следует поставить вопрос: "Каковы обязанности действующего лица по отношению к субъекту и чего он ожидает от субъекта?" Кроме того, прецеденты использования можно определить в результате непосредственного анализа функциональных требований. Во многих случаях *функциональное требование* непосредственно отражается в виде *прецедента использования*.

В табл. 3.1 показано, как можно с помощью функциональных требований к системе управления магазином видеокассет идентифицировать действующих лиц и прецеденты. Несмотря на то что два действующих лица упоминаются во всех четырех требованиях, совершенно очевидно, что уровень их вовлеченности в процесс непостоянен. Как видим, `Сканирующее устройство` не рассматривается как действующее лицо — оно относится к внутреннему устройству системы.

Таблица 3.1. Распределение требования среди действующих лиц и прецедентов использования в магазине видеокассет

Номер требования	Требование	Действующие лица	Прецеденты использования
1	Перед тем как выдать кассету напрокат, система подтверждает личность клиента, сканируя его карточку	`Клиент` `Сотрудник`	`Сканирование карточки`
2	В ходе обработки заявки видеокассету или диск можно сканировать, чтобы выяснить их описание или цену	`Клиент` `Сотрудник`	`Сканирование видеокассеты`
3	Прежде чем взять видеокассету или диск напрокат, клиент выплачивает ее номинальную стоимость либо наличными, либо по кредитной карточке	`Клиент` `Сотрудник`	`Принятие платежа` `Списание денег со счета`

Окончание табл. 3.1

Номер требования	Требование	Действующие лица	Прецеденты использования
4	Система проверяет выполнение условий проката, удостоверяется, что транзакция может быть продолжена, и выдает клиенту квитанцию	`Клиент` `Сотрудник`	`Распечатка квитанции`

Прецеденты использования можно называть, ориентируясь на субъект или действующее лицо. В табл. 3.1 отражается точка зрения действующего лица. Это выражается, в частности, в том, что один из прецедентов называется `Принятие платежа`, а не `Платеж`.

Название прецедентов использования с точки зрения действующих лиц не всегда рекомендуется. Легко понять, что их следует называть, занимая позицию внешних действующих лиц. Однако в этом случае труднее установить связь между прецедентами и моделями/артефактами, разработанными позднее, поскольку эти модели и артефакты тесно связаны с внутренним устройством субъектов и подсистем.

Прецеденты использования, приведенные в табл. 3.1, показаны на рис. 3.2. В языке UML прецедент использования изображается в виде эллипса с названием внутри.

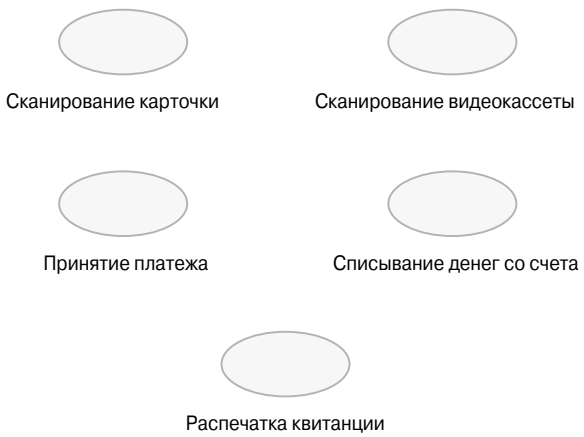


Рис. 3.2. Прецеденты использования системы управления магазином видеокассет

3.1.3. Диаграммы прецедентов использования

Диаграмма прецедентов использования приписывает прецеденты использования к действующим лицам. Она также позволяет пользователю установить отношения между прецедентами использования, конечно, если такие отношения существуют. Эти отношения рассматриваются в главе 4.

На рис. 3.3 показана диаграмма, демонстрирующая связи между прецедентами использования, показанных на рис. 3.2, и действующими лицами. На рис. 3.4 показана та же самая модель, содержащая прецеденты использования, связанные с субъектом. Несмотря на то что расположение действующих лиц и прецедентов использования на диаграмме произвольно, при моделировании субъектов действующих лиц необходимо заключить в прямоугольную рамку. Обратите внимание на то, что одно и то же действующее лицо может быть изображено несколько раз.

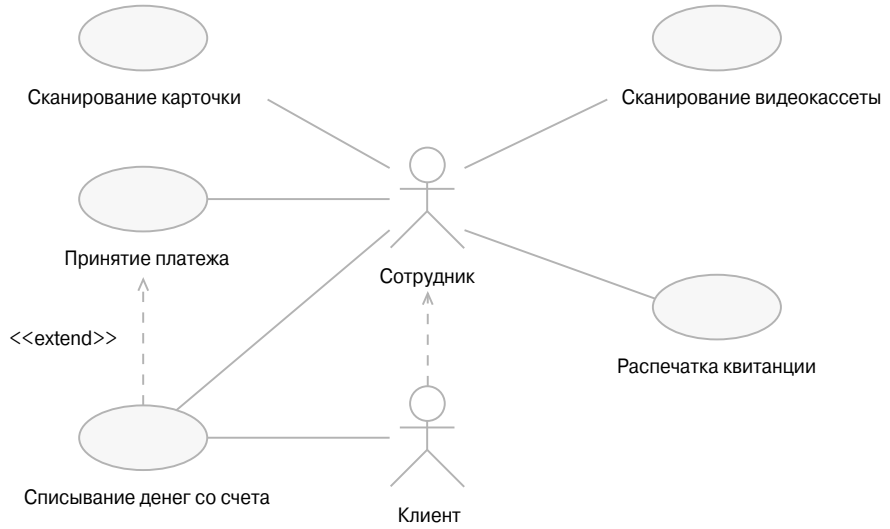


Рис. 3.3. Диаграмма прецедентов использования системы управления магазином видеокассет

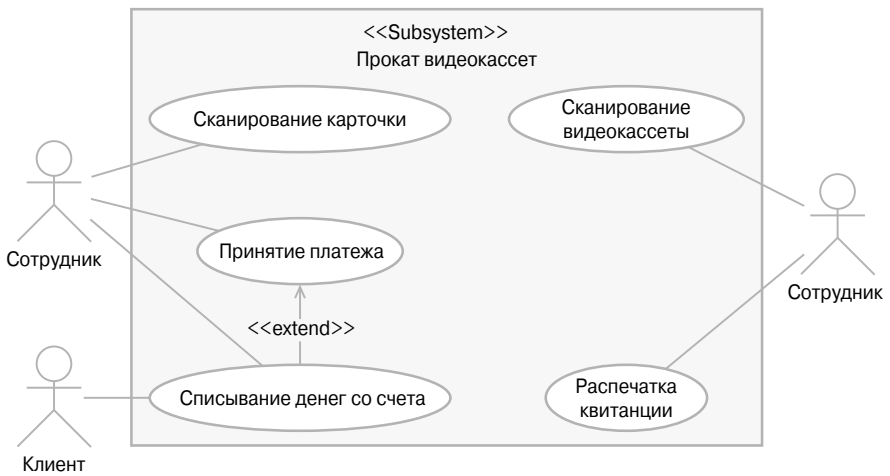


Рис. 3.4. Диаграмма прецедентов использования, описывающая субъект системы управления магазином видеокассет

Модели, показанные на рис. 3.3 и 3.4, демонстрируют, что действующее лицо `Сотрудник` непосредственно связано со всеми прецедентами использования. Действующее лицо `Клиент` может достичь своих целей лишь с помощью действующего лица `Сотрудник`, поэтому между ними существует *зависимость* (dependency). Прямая связь между действующим лицом `Клиент` и прецедентом использования `Списание денег со счета` означает, что клиент должен ввести PIN-код и подтвердить оплату с помощью сканирующего устройства.

В целом диаграммы прецедентов использования допускают использование нескольких отношений между элементами. Эти отношения рассматриваются в главе 4. Отношение «`extend`» на рис. 3.3 и 3.4 означает, что функциональное свойство `Принятие платежа` иногда может быть расширено (поддержано) прецедентом использования `Списание денег со счета`.

3.1.4. Документирование прецедентов использования

Динамика каждого прецедента использования должна быть описана с помощью других моделей UML и/или с помощью *спецификации потока событий* (flow of events document), т.е. текстового документа, определяющего, что должна делать система, когда действующее лицо инициирует прецедент использования. Структура *спецификации прецедента использования* (use case document) может варьироваться, однако типичное описание должно содержать следующую информацию (Quatrani, 2000).

- Краткое описание.
- Участвующие действующие лица.
- Предусловия, необходимые для инициирования прецедента использования.
- Детализированное описание потока событий, включающее в себя:
  - основной поток событий, который можно разбить на несколько частей, чтобы показать подчиненные потоки событий (подчиненные потоки могут быть разделены затем на еще более мелкие потоки, с целью улучшить читабельность документа);
  - альтернативные потоки для определения исключительных ситуаций.
- Постусловия, определяющие состояние системы, по достижении которого прецедент использования завершается.

Спецификация прецедента использования развивается по ходу разработки. На ранней стадии определения требований составляется только краткое описание. Остальные части документа создаются постепенно и итеративно. Полный документ возникает в конце этапа спецификации требований. На этой стадии документ может быть дополнен прототипами экранов графического пользовательского интерфейса. Позднее спецификация прецедентов используется для создания пользовательской документации реализованной системы.

В табл. 3.2 приведено описание прецедента использования Принятие платежа, показанного на рис. 3.3 и 3.4. Оно содержит спецификацию прецедента использования Списание денег со счета, расширяющего прецедент Принятие платежа. Табличный способ документирования прецедентов использования нельзя считать стандартным. Спецификации прецедентов использования могут быть многостраничными (в среднем около десятка страниц) и обладать стандартной для документов структурой, дополненной оглавлением. Пример более реалистичной спецификации прецедентов использования приведен в разделе 6.5.3 главы 6.

Таблица 3.2. Описание прецедента использования системы управления магазином видеокассет

	Прецедент использования: Принятие платежа
Краткое описание	Этот прецедент использования позволяет действующему лицу Сотрудник принимать платежи у действующего лица Клиент за прокат видеокассет. Платеж может быть осуществлен в наличной форме или по кредитной карточке
Действующие лица	Сотрудник, Клиент
Предусловия	Действующее лицо Клиент выражает желание взять видеокассету напрокат, имеет членскую карточку, а требуемая видеокассета есть на складе
Основной поток	Прецедент использования начинается с того, что Клиент решает оплатить прокат видеокассет и предлагает оплату наличными или по кредитной карточке. Действующее лицо Сотрудник запрашивает у системы информацию о плате за прокат, а также о клиенте и кассете. Если действующее лицо Клиент предлагает оплату наличными, то действующее лицо Сотрудник принимает деньги, регистрирует в системе платеж и просит систему сделать соответствующую запись. Если действующее лицо Клиент предлагает оплату с помощью кредитной карточки, то действующее лицо Сотрудник считывает информацию с карточки, просит действующее лицо Клиент ввести PIN-код, указать номер депозитного или кредитного счета и осуществить платеж. После электронного подтверждения платежа система регистрирует этот факт. Прецедент использования завершается
Альтернативные потоки	Действующее лицо Клиент не имеет наличных денег и не предлагает оплату с помощью кредитной карточки. Действующее лицо Сотрудник просит систему проверить рейтинг действующего лица Клиент (на основе истории его платежей). Действующее лицо Сотрудник решает, выдать ли кассету без полной или частичной оплаты.

Окончание табл. 3.2

	Прецедент использования: `Принятие платежа`
	В зависимости от решения, действующее лицо `Сотрудник` отменяет транзакцию (и прекращает прецедент использования) или принимает частичную оплату (и продолжает прецедент использования).
	Кредитная карточка действующего лица `Клиент` не принимается сканером. После трех безуспешных попыток действующее лицо `Сотрудник` вводит номер карточки вручную. Прецедент использования продолжается
Постусловия	Если прецедент использования завершился успешно, то платеж регистрируется в базе данных. В противном случае состояние системы остается неизменным

Контрольные вопросы 3.1

КВ1. Перечислите наиболее важные методы моделирования.

КВ2. Совпадает ли диаграмма прецедентов использования со спецификацией прецедентов использования?

3.2. Ракурс деятельности

Модель деятельности (activity model) описывает поведение, в которое вовлечено несколько элементов системы. Это поведение может быть спецификацией прецедента использования. Кроме того, она может быть частью функциональных свойств, которые можно многократно использовать в разных частях системы. Модель деятельности заполняет пробел между высокоуровневым поведением системы в *моделях прецедентов использования* (use case models) и низкоуровневым представлением поведения *моделей взаимодействия* (interaction models), т.е. диаграмм последовательности и коммуникации.

Диаграмма деятельности показывает шаги вычисления. Каждый шаг **деятельности** (activity) называется **действием** (action). Действия нельзя разбить на более мелкие части. Диаграмма деятельности описывает, какие шаги выполняются последовательно, а какие параллельно. Передача управления от одного состояния вида деятельности к другому называется **потоком управления** (control flow). “Под потоком мы подразумеваем, что выполнение действия в одном узле влияет на выполнение действий в других узлах и, в свою очередь, испытывает их влияние; эти зависимости представляются *ребрами* диаграммы деятельности” (UML, 2005).

После завершения спецификации прецедентов использования деятельность и действия можно установить по описанию основного и альтернативных потоков. Однако модели деятельности можно использовать не только для спецификации прецедентов использования (Fowler, 2004). Их можно также использовать для анализа бизнес-процессов на высоком уровне абстракции еще до создания прецедентов. Кроме того, их можно использовать на гораздо более высоком уровне абстракций для разработки сложных последовательных или параллельных алгоритмов.

3.2.1. Действия

Если моделирование видов деятельности используется для визуализации последовательности видов деятельности, связанных с прецедентом использования, то действия можно установить на основе его спецификации. В табл. 3.3 приведены формулировки основного и альтернативного потоков событий в спецификации прецедентов использования, а также идентифицированные действия.

Таблица 3.3. Выявление действий в основном и альтернативном потоках

Номер	Содержание прецедента использования	Действие
1	Действующее лицо `Сотрудник` запрашивает у системы информацию о плате за прокат, а также о клиенте и кассете	`Вывести на экран информацию о транзакции`
2	Если действующее лицо `Клиент` предлагает оплату наличными, то действующее лицо `Сотрудник` принимает деньги, регистрирует в системе платеж и просит систему сделать соответствующую запись	`Ввести с клавиатуры сумму платежа` `Подтвердить транзакцию`

3	Если действующее лицо `Клиент` предлагает оплату с помощью кредитной карточки, то действующее лицо `Сотрудник` считывает информацию с карточки, просит действующее лицо `Клиент` ввести PIN-код, указать номер депозитного или кредитного счета и осуществить платеж. После электронного подтверждения платежа система регистрирует этот факт	`Сканировать кредитную карточку`
		`Принять номер кредитной карточки`
		`Выбрать номер кредитной карточки`
		`Подтвердить транзакцию`

4	Действующее лицо `Клиент` не имеет наличных денег и не предлагает оплату с помощью кредитной карточки. Действующее лицо `Сотрудник` просит систему проверить рейтинг действующего лица `Клиент` (на основе истории	`Проверить рейтинг клиента`
		`Отменить транзакцию`
		`Разрешить прокат без оплаты`
		`Разрешить прокат с частичной оплатой`

Окончание табл. 3.3

Номер	Содержание прецедента использования	Действие
	его платежей). Действующее лицо `Сотрудник` решает, выдать ли кассету без полной или частичной оплаты. В зависимости от решения действующее лицо `Сотрудник` отменяет транзакцию (и прекращает прецедент использования) или принимает частичную оплату (и продолжает прецедент использования)	
5	Кредитная карточка действующего лица `Клиент` не принимается сканером. После трех безуспешных попыток действующее лицо `Сотрудник` вводит номер карточки вручную	`Ввести номер кредитной карточки вручную`

В языке UML действия представляются в виде прямоугольника с закругленными углами. Действия, перечисленные в табл. 3.3, показаны на рис. 3.5.



Рис. 3.5. Действия в прецедентах использования системы управления магазином видеокассет

3.2.2. Диаграммы деятельности

Диаграмма деятельности (activity diagram) показывает потоки между действиями и **узлами** (nodes), например решениями, разветвлениями, соединениями, слияниями и объектами. Обычно между видами деятельности и диаграммами

деятельности существует взаимно однозначное соответствие, т.е. деятельность изображается с помощью диаграмм деятельности.

Если вид деятельности не представляет собой замкнутого цикла, то диаграмма содержит начальное состояние вида деятельности и одно или несколько конечных состояний вида деятельности. Начальное состояние обозначается полностью закрашенной окружностью. Конечное состояние изображается в виде окружности с закрашенной центральной частью ("бычий глаз").

Потоки могут *разветвляться* (branch) в зависимости от условий и *сливаться* (merge). В результате возникают *альтернативные потоки вычислений* (alternative computation thread). Условие ветвления обозначается ромбом. Выход из условия ветвления управляется событием, например Да или Нет, или сторожевым условием (guard condition), например [светло-зеленый], [высокий рейтинг].

Кроме того, потоки могут *разделяться* (fork) и *воссоединяться* (rejoin). В результате возникают *параллельные потоки вычислений* (concurrent computation threads). Разделение и воссоединение потоков представляется в виде жирной линии. Диаграмма деятельности, в которой отсутствуют параллельные процессы, похожа на обычную *блок-схему* (в этом разделе параллельное функционирование системы не рассматривается).

Для того чтобы нарисовать диаграмму, демонстрирующую работу магазина видеокассет, достаточно соединить действия, идентифицированные на рис. 3.5, соответствующими потоками, как показано на рис. 3.6. Начальным является действие `Вывести на экран информацию о транзакции`. *Рекурсивный поток* (recursive flow), связанный с этим действием, отражает тот факт, что изображение дисплея постоянно обновляется, пока вычисления не перейдут в следующий узел.

На протяжении действия `Вывести на экран информацию о транзакции` заказчик может предложить оплату наличными или с помощью кредитной карточки. Это приводит к выполнению одного из двух возможных потоков вычислений. В узле `Обработать платеж по карточке` (см. рис. 3.6) объединено несколько действий, связанных с принятием платежа по кредитной карточке. Это вложение удобно, когда источником потока действий может быть любое из вложенных действий. В таком случае поток может исходить из узла действий. Примером может служить поток, входящий в узел ветвления `Есть ли проблемы с оплатой?`

Необходимость проверки условия `Есть ли проблемы с оплатой?` объясняется возможной нехваткой денег у заказчика. Если проблем нет, то транзакция подтверждается и процесс переходит к заключительному действию. В противном случае следует проверить рейтинг заказчика. В зависимости от рейтинга транзакция отменяется (если выполняется сторожевое условие `[Плохой]`), допускается частичная оплата (если выполняется сторожевое условие `[Хороший]`) или допускается прокат без оплаты (если выполняется сторожевое условие `[Отличный]`).

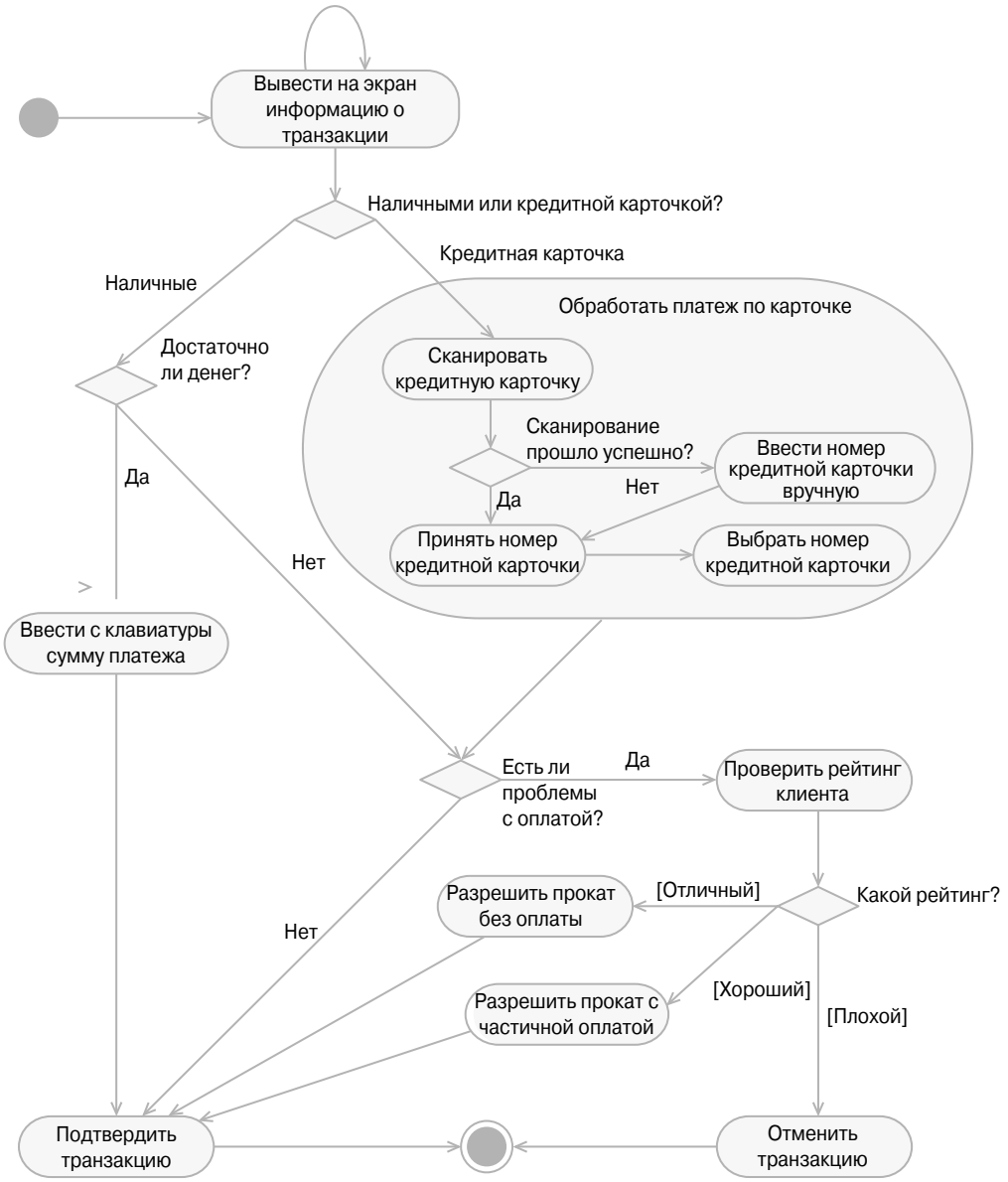


Рис. 3.6. Диаграмма деятельности магазина видеокассет

Контрольные вопросы 3.2

КВ1. Можно ли использовать модель деятельности в качестве спецификации прецедента использования?

КВ2. Потоки на диаграмме действия соединяют действия и другие узлы. Что это за узлы?

3.3. Ракурс структуры

Ракурс структуры (structure view) отражает *статический аспект* системы (static view) — он представляет структуры данных и отношения между ними, а также идентифицирует операции над этими данными. Основным способом визуализации статической модели является *диаграмма классов* (class diagram), представляющая собой основную разновидность *структурных диаграмм* (structure diagrams). К остальным структурным диаграммам относятся диаграммы компонентов и развертывания.

Моделирование классов (class modeling) интегрирует и включает в себя все другие виды моделирования. Модели классов определяют структуры, отражающие внутреннее состояние системы. Они идентифицируют классы и их атрибуты, включая отношения. Кроме того, они определяют операции, необходимые для реализации требований динамического поведения системы, зафиксированных в прецедентах использования. Классы, реализованные на конкретном языке программирования, отражают статическую структуру и динамическое поведение приложения.

Соответственно, модели классов связаны практически со всеми фундаментальными концепциями объектной технологии (см. приложение А). Понимание этих концепций является необходимым, но недостаточным условием успешной работы с моделями классов. Они необходимы для того, чтобы понимать модели классов, но недостаточны для того, чтобы их разрабатывать. Разработка моделей классов требует опыта работы с абстракциями и способности итеративно интегрировать множество компонентов в единое связное решение.

Результатом моделирования классов является диаграмма классов и связанная с ней текстовая документация. В данной главе моделирование классов обсуждается после моделирования прецедентов использования, но на практике эти виды деятельности, как правило, выполняются параллельно. Эти две модели дополняют друг друга, обмениваясь дополнительной информацией. Прецеденты использования облегчают выявление классов, а модели классов, наоборот, приводят к идентификации и пересмотру прецедентов использования.

3.3.1. Классы

До сих пор мы использовали классы для определения *бизнес-объектов* (business objects). Примеры классов, приведенные в книге, относились к долгоживущим (персистентным) бизнес-сущностям, таким как заказ, доставка, клиент, студент и т.д. Эти классы определяют *модель базы данных* (database model) для прикладной области. По этой причине такие классы часто называют *классами сущностей* (entity class) или *модельными классами* (model classes).

Классы сущностей определяют основу любой информационной системы. Анализ требований направлен преимущественно на выявление классов сущностей. Однако для функционирования системы требуются также классы другого типа. Пользователям системы необходимы классы, определяющие объекты графического пользовательского интерфейса (например, экранных форм), называемые *классами представления, границ или вида* (presentation, boundary, or view classes)). Системе также необходимы классы, управляющие программной логикой и обработкой событий, генерируемых пользователем, — *управляющие классы* (control classes). Кроме того, существуют другие категории классов, обеспечивающих работу системы, например классы, ответственные за связь с внешними источниками данных, иногда называемые *классами ресурсов* (resource classes). Управление сущностями в оперативной памяти компьютера, обеспечивающего выполнение деловых транзакций, порождает еще одну категорию — *классы-посредники* (mediator classes).

В зависимости от конкретного подхода к моделированию классы, не представляющие собой классы сущностей, могут не рассматриваться в ходе анализа требований. Это же относится и к определению операций в ранних моделях классов. Начальное моделирование классов, не являющихся классами сущностей, и определение операций можно отложить до этапа, на котором определяется вид взаимодействий (см. раздел 3.4), а более подробное моделирование можно отложить до этапа проектирования системы.

Следуя подходу, принятому нами для поиска действующих лиц и прецедентов использования (см. табл. 3.1), можно создать таблицу, помогающую идентифицировать классы на основе анализа функциональных требований. Распределение функциональных требований среди классов сущностей показано в табл. 3.4.

Таблица 3.4. Распределение функциональных требований среди классов сущностей в системе управления магазином видеокассет

Номер	Требование	Класс сущностей
1	Перед тем как выдать кассету напрокат, система подтверждает личность клиента, сканируя его карточку	`Video` (Видео) `Customer` (Заказчик) `MembershipCard` (Членская карточка клиента)

Окончание табл. 3.4

Номер	Требование	Класс сущностей
2	В ходе обработки заявки видеокассету или диск можно сканировать, чтобы выяснить их описание или цену	`Videotape` (Видеокассета) `Videodisk` (Видеодиск) `Customer` (Клиент) `Rental` (Прокат)
3	Прежде чем взять видеокассету или диск напрокат, клиент выплачивает ее номинальную стоимость либо наличными, либо по кредитной карточке	`Customer` (Клиент) `Video` (Видео) `Rental` (Прокат) `Payment` (Платеж)
4	Система проверяет выполнение условий проката, удостоверяется, что транзакция может быть продолжена, и выдает клиенту квитанцию	`Rental` (Прокат) `Receipt` (Квитанция)

Выделение классов представляет собой итеративную задачу, и первоначальный перечень предполагаемых классов, как правило, претерпевает изменения. При определении того, являются ли понятия, присутствующие в требованиях, искомыми классами, могут помочь ответы на следующие вопросы.

1. Является ли понятие контейнером данных?

2. Обладает ли понятие отдельными атрибутами, способными принимать разные значения?

3. Можно ли создать для понятия множество объектов-экземпляров?

4. Принадлежит ли понятие конкретной прикладной области?

Перечень классов, приведенный в табл. 3.4, кроме того, вызывает много вопросов.

- В чем разница между классами `Video` и `Videotape/Videodisk`? Не является ли понятие `Video` обобщенным названием классов `Videotape/Videodisk`? Если да, то не следует ли ввести класс `Movie` (Фильм) и другие классы, описывающие содержание носителей видеоинформации? Необходим ли класс `Movie`?
- Одинаков ли смысл класса `Rental` в требованиях 2–4? Относится ли он к одной и той же транзакции?
- Не является ли класс `MembershipCard` частью класса `Customer`?
- Следует ли разделять классы `CashPayment` (Оплата наличными) и `CardPayment` (Оплата карточкой)?
- Несмотря на то что сотрудник магазина как действующее лицо не упоминается в требованиях 3 и 4, совершенно очевидно, что система должна знать, какие именно сотрудники выполняют транзакции, связанные с выдачей кассет и дисков напрокат и их оплатой. Таким образом, необходим класс `Customer`.

Дать ответы на эти и аналогичные вопросы не легко, для этого требуется глубокое знание требований прикладной области. Все классы, перечисленные в табл. 3.4, показаны на рис. 3.7. Обратите внимание на то, что классы `Customer` и `Employee` уже появлялись в качестве действующих лиц на диаграмме прецедентов использования (поэтому они сопровождаются комментарием "из диаграммы прецедентов использования"). Эта двойственная природа действующих лиц, одновременно являющихся внешними сущностями, взаимодействующими с системой, и внутренними сущностями, о которых система должна иметь определенную информацию, характерна для моделирования систем.

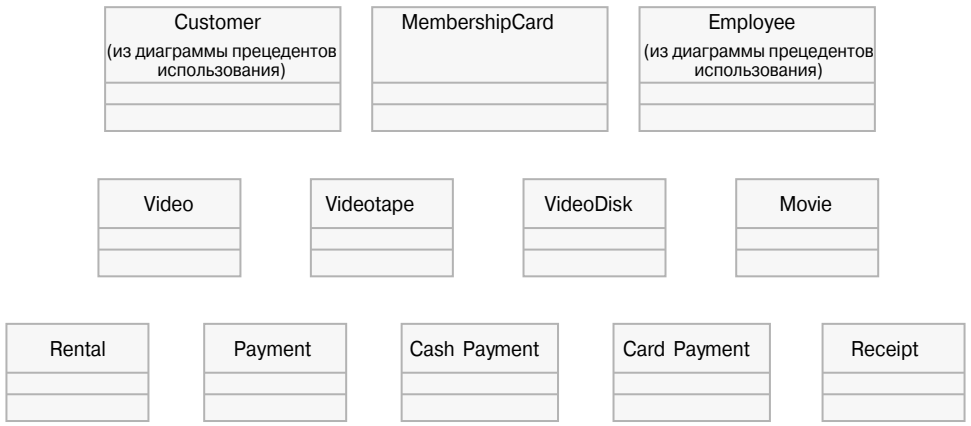


Рис. 3.7. Классы системы управления магазином видеокассет

3.3.2. Атрибуты

Структура класса определяется его **атрибутами** (см. раздел А.3.1 приложения А). Объявляя класс, аналитик должен иметь некоторое представление о структуре атрибутов. На практике основные атрибуты обычно назначаются классу сразу после его добавления к модели.

Атрибуты выявляются из требований пользователя и знаний о предметной области. Сначала разработчик концентрируется на выделении *идентифицирующих атрибутов* каждого класса, т.е. одного или нескольких атрибутов класса, принимающего уникальные значения во всех экземплярах класса. Эти атрибуты часто называют *ключами* (keys). В идеале ключ должен состоять из одного атрибута. В некоторых случаях ключ состоит из нескольких атрибутов.

Выяснив идентифицирующие атрибуты, разработчик должен определить основные *описательные атрибуты* (descriptive attributes) каждого класса. Эти атрибуты описывают информационное содержание класса. На этом этапе еще нет необходимости определять *неэлементарные типы* (non-primitive types) атрибутов (см. раздел А.3.1 приложения А). Большинство атрибутов, требующих неэлемен-

тарных типов, может быть описано в виде строки символов. На более поздних этапах моделирования эти строки можно преобразовать в неэлементарные типы.

На рис. 3.8 показаны два класса, входящие в систему управления магазином видеокассет, с элементарными атрибутами. Оба класса имеют идентифицирующие ключи (`membershipId`). Это является ответом на вопрос, поднятый в разделе 3.3.1, и свидетельствует о том, что класс `MembershipCard` имеет интересные отношения с классом `Customer`. Очевидно, что к этому вопросу следует вернуться на более поздних этапах моделирования.

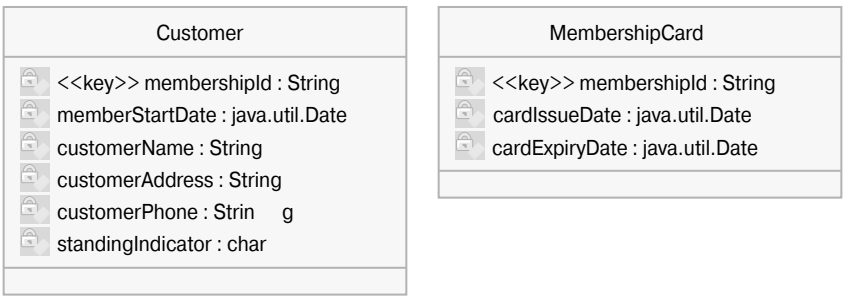


Рис. 3.8. Классы системы управления магазином видеокассет

Некоторые атрибуты на рис. 3.8 показаны как `java.util.Date`. Они имеют тип `Data`, описанный в стандартной библиотеке языка Java. Несмотря на то что этот тип не является элементарным, он *не определяется пользователем* и поэтому не противоречит предположению, что атрибуты могут иметь только элементарные типы.

С точки зрения языка Java тип данных `String` также не является элементарным. Вероятно, некоторые атрибуты, такие как `customerAddress`, могут в будущем получить неэлементарный тип, т.е. возможно, понадобится создать некий класс `Address`. Однако пока эти атрибуты имеют тип `String`.

Атрибут `standingIndicator` имеет тип `char`. Этот атрибут описывает рейтинг, присвоенный каждому клиенту на основе его истории платежей, своевременного возвращения кассет и дисков и т.д. Этот рейтинг может изменяться в определенном диапазоне, например от A до E, где A означает отличный рейтинг, а E — наихудший, т.е. рейтинг клиента, которого следовало бы исключить из клуба.

Общеизвестно и естественно, что существует множество разных решений, касающихся определения атрибутов, указанных на рис. 3.8. Например, необходимость атрибута `memberStartDate` в классе `Customer` может быть поставлена под вопрос. Аналогично, некоторых читателей может удивить, что в классе `MembershipCard` пропущены атрибуты `customerName` и `customerAddress`.

3.3.3. Ассоциации

Ассоциации (associations), связывающие классы, устанавливают пути, облегчающие взаимодействие объектов (см. раздел А.3.1 приложения А). В реализованной системе ассоциации представляются типами атрибутов, обозначающими ассоциированные классы (см. раздел А.3.1.1 приложения А). В модели анализа ассоциации представлены соединительными линиями.

На рис. 3.9 показаны две ассоциации между тремя классами — `Customer`, `Rental` и `Payment`. Обе ассоциации имеют кратность "один ко многим" (см. раздел А.5.2 приложения А). На рис. 3.9 показаны также названия ролей ассоциаций. В реализованной системе названия ролей будут преобразованы в атрибуты, обозначающие классы (см. раздел А.3.1.1 приложения А).



Рис. 3.9. Ассоциации в системе управления магазином видеокассет

Класс `Customer` может быть ассоциирован со многими транзакциями класса `Rental`. Каждый экземпляр класса `Rental` связан с одним экземпляром класса `Customer`. В модели нет никакой информации о том, что одна кассета или диск могут быть взяты напрокат в рамках одной транзакции. Даже если такая возможность существует, все кассеты или диски должны быть арендованы на один и тот же период времени (т.е. возможно только одно значение атрибута `rentalDuration`).

Оплата аренды может быть осуществлена с помощью нескольких платежей, т.е. класс `Rental` может быть связан с несколькими экземплярами класса `Payment`, а атрибут `paymentAmount` может быть меньше атрибута `rentalCharge`. Кроме того, возможен прокат видеодисков и видеокассет без немедленной оплаты (объект класса `Rental` может быть связан с нулевым объектом класса `Payment`).

Это открывает возможность для альтернативного потока действий в спецификации прецедентов использования (см. табл. 3.2).

Модель на рис. 3.9 не содержит явной ассоциации между классами `Payment` и `Customer`. С семантической точки зрения такая ассоциации не нужна. Клиент, осуществляющий платеж, может быть идентифицирован путем навигации по транзакции. Это возможно благодаря тому, что каждый объект класса `Payment` связан с единственным объектом класса `Rental`, а каждый объект класса `Rental`, в свою очередь, связан с единственным объектом класса `Customer`. Однако вполне вероятно, что ассоциация между классами `Payment` и `Customer` будет включена в модель на этапе проектирования (из соображений эффективности).

3.3.4. Агрегация

Агрегация (aggregation) и **композиция** (composition) являются более сильной формой ассоциативной связи, которой присуща семантика владения (см. раздел А.6 приложения А). В типичной коммерческой программной среде агрегация и композиция, скорее всего, должны быть реализованы подобно ассоциации — с помощью типов атрибутов, обозначающих ассоциированные классы. Визуально агрегация изображается в виде белого ромба на конце линии ассоциации, соединяющей ее с агрегированным классом (супермножеством), а композиция — в виде закрашенного ромба.

На рис. 3.10 иллюстрируется отношение агрегации между классами `Customer` и `MembershipCard`. Класс `Customer` может не содержать ни одного класса или содержать один класс `MembershipCard`. Система позволяет хранить информацию о потенциальных клиентах, например людях, пока не имеющих членской карточки клуба. Объекты класса `Customer`, описывающие таких клиентов, не содержат класс `MembershipCard`, поэтому атрибут `memberStartDate` устанавливается равным нулю.

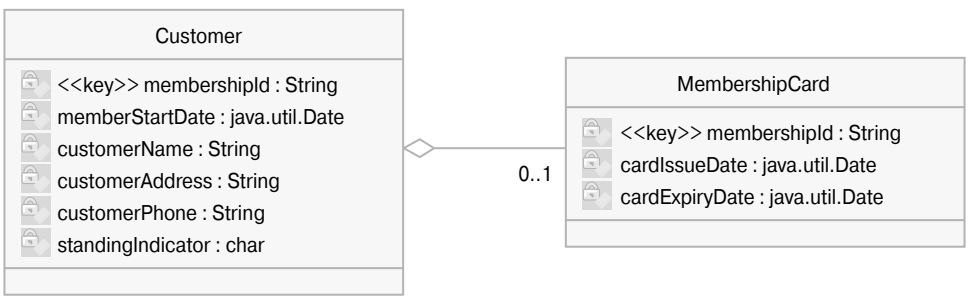


Рис. 3.10. Агрегация в системе управления магазином видеокассет

Белый ромб на конце линии агрегации не всегда означает агрегацию по ссылке (см. раздел А.6 приложения А). Он может также означать, что разработчик еще не решил, как именно реализовать агрегацию. Если диаграмма представляет собой

модель анализа, то реализация агрегации остается неопределенной. Если диаграмма представляет собой проектную модель, то белый ромб действительно означает агрегацию по ссылке.

3.3.5. Обобщение

Обобщение (generalization) представляет собой таксономическое отношение между классами, показывающее, что подклассы *конкретизируют* (specialize) суперкласс (см. раздел А.7 приложения А). Это значит, что экземпляр любого подкласса также является косвенным экземпляром суперкласса и что он наследует характеристики суперкласса. Обобщение визуализируется с помощью сплошной линии, заканчивающейся крупным белым треугольником, касающимся суперкласса.

Обобщение представляет собой мощный метод повторного использования программного обеспечения, существенно упрощающий семантическое и графическое представление моделей. Упрощение достигается двумя разными способами, в зависимости от обстоятельств.

Учитывая то, что подкласс является разновидностью суперкласса, можно провести линию ассоциации от любого класса к суперклассу и предположить, что в реальности любые объекты в иерархии обобщения могут быть связаны этой ассоциацией. С другой стороны, можно нарисовать линию ассоциации от суперкласса к более конкретному классу, находящемуся внизу иерархии, чтобы отразить тот факт, что только объекты этого конкретного подкласса могут быть связаны этой ассоциацией.

На рис. 3.11 показан пример иерархии обобщения, корнем которой является класс *Payment*. Поскольку в магазине существует только два вида платежей (наличными или кредитной карточкой), класс *Payment* может стать абстрактным и поэтому его имя написано курсивом (см. раздел А.8 приложения А). Класс `Receipt` связан с классом *Payment*. В реальности объекты конкретных подклассов класса *Payment* будут связаны с объектами класса `Receipt`.

В диаграмму, показанную на рис. 3.11, включены два новых класса. Новые классы — `DebitCardPayment` и `CreditCardPayment` — являются подклассами класса `CardPayment`. В результате класс `CardPayment` становится абстрактным.

3.3.6. Диаграммы классов

Диаграмма классов (class diagram) — это сердце и душа объектно-ориентированной системы. До сих пор система управления магазином видеокассет демонстрировала лишь возможности *статического моделирования* классов, поскольку классы содержали только атрибуты и не содержали ни одной **операции**. По существу, операции больше относятся к проектированию, а не к анализу. После их включения в класс модель классов начинает описывать *поведение* системы.

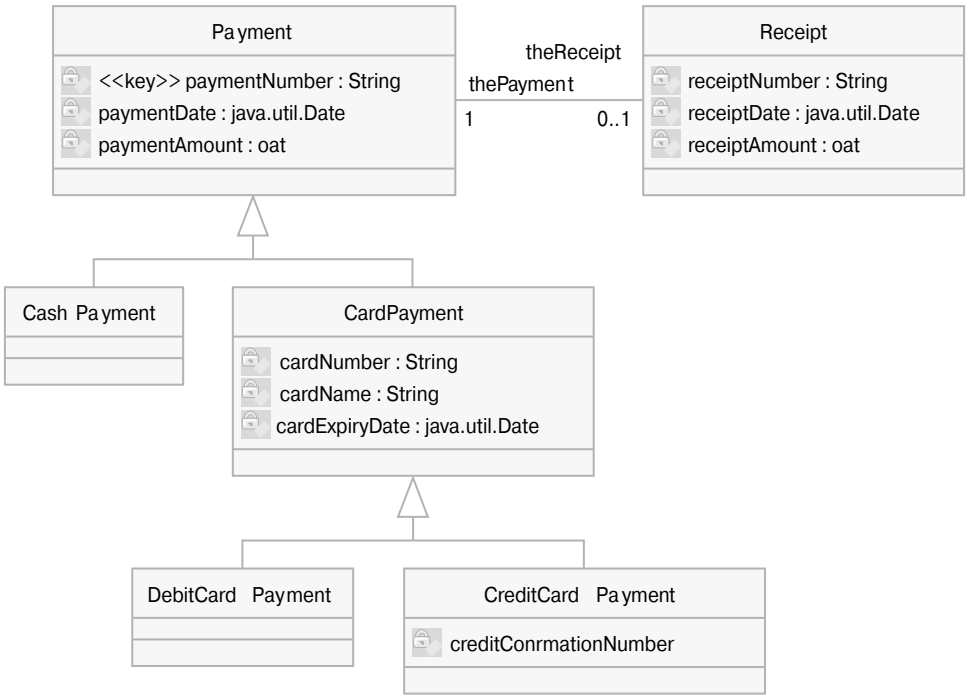


Рис. 3.11. Агрегация в системе управления магазином видеокассет

На рис. 3.12 продемонстрирована диаграмма классов системы управления магазином видеокассет. Эта модель демонстрирует лишь классы, идентифицированные в предыдущих примерах. Другие потенциальные классы, такие как `Sale`, `CD` и `TVProgram`, на диаграмме не показаны. В отличие от классов `Payment` и `CardPayment` класс `Video` оказывается абстрактным. Остальные классы являются конкретными.

Тщательный анализ кратности ассоциаций (см. рис. 3.12) показывает, что класс `Rental` ссылается только на *текущую* аренду. Каждая видеокассета, взятая напрокат, связана с одной и только одной транзакцией аренды. *Прошлые* транзакции той же самой видеокассеты в классе `Rental` не запоминаются.

Класс `Video` (т.е. видеокассета или диск) содержит один или несколько классов `Movie`. С другой стороны, класс `Movie` может быть недоступным, а также быть доступным на одной или нескольких видеокассетах или дисках.

Каждая транзакция аренды связана с соответствующим классом `Employee`. Аналогично, каждый платеж связан с определенным сотрудником. Информацию о классе `Customer`, осуществляющем платеж, можно получить с помощью навигации от объекта `payment` к объекту `customer` через транзакцию `rental` или `receipt`.

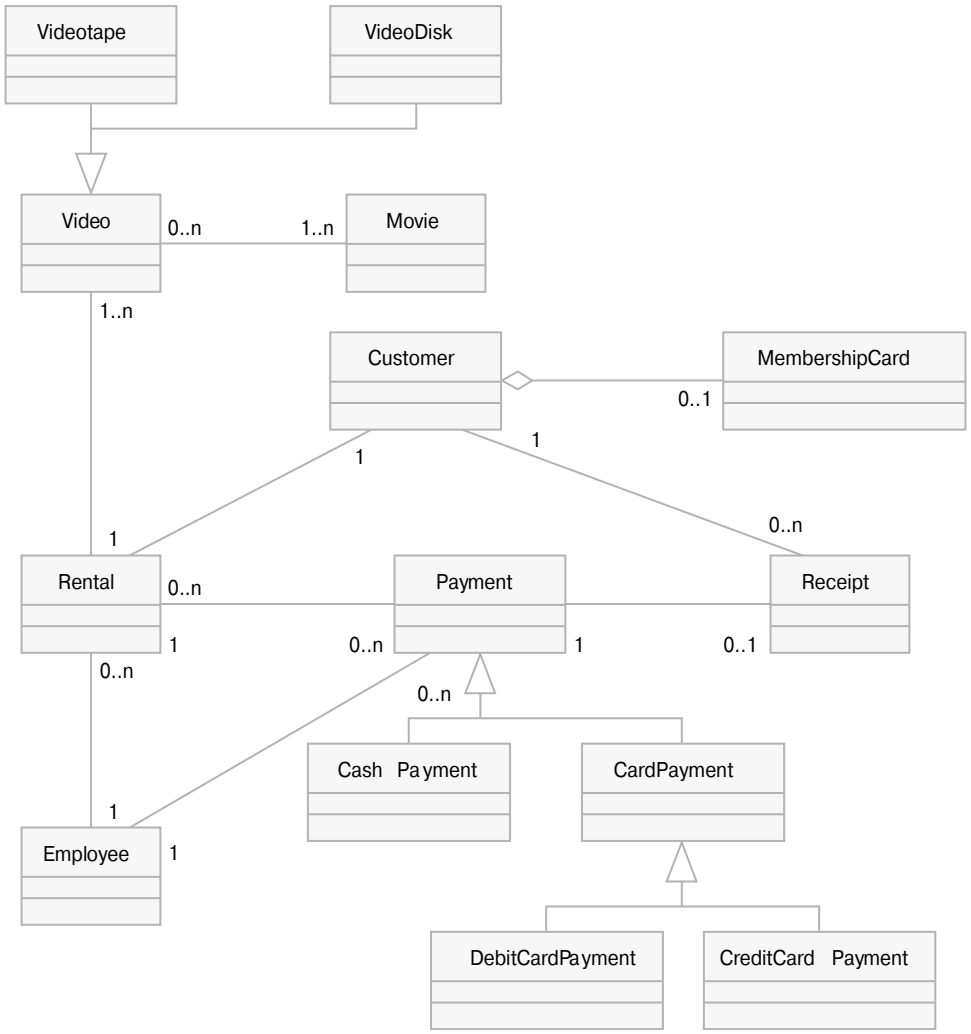


Рис. 3.12. Диаграмма классов системы управления магазином видеокассет

Контрольные вопросы 3.3

КВ1. Являются ли синонимами понятия класса сущностей и бизнес-объекта?

КВ2. Применима ли концепция кратности к агрегации?

3.4. Ракурс взаимодействий

Моделирование взаимодействий (interaction modeling) охватывает вопросы **взаимодействия** объектов, которым для выполнения прецедента использования необходимо обмениваться сообщениями. Модели взаимодействия используются на более поздних стадиях анализа требований, когда становится известной модель классов, так что ссылки на объекты опираются на модель классов.

Приведенное выше наблюдение позволяет установить основное различие между моделированием деятельности (см. раздел 3.2) и моделированием взаимодействий. Модели обоих типов описывают поведение одного прецедента (как правило). *Моделирование деятельности* часто осуществляется на более высоком уровне абстракции — оно отражает последовательность событий вне связи событий с объектами. В то же время *моделирование взаимодействий* отображает последовательность **событий** (**сообщений**), происходящих между взаимодействующими объектами.

Моделирование деятельности и взаимодействий представляет собой реализацию прецедентов использования. Диаграммы деятельности носят более абстрактный характер и часто отражают поведение всего прецедента использования. Диаграммы взаимодействий более подробны и моделируют часть прецедента использования. Иногда диаграммы взаимодействий моделируют отдельный вид деятельности в диаграмме деятельности.

Диаграммы взаимодействий разделяются на два вида — *диаграммы последовательностей* и *диаграммы коммуникации* (до появления версии UML 2.0 они назывались диаграммами *кооперации*). Они могут использоваться как взаимозаменяемые, и, конечно, многие CASE-средства поддерживают автоматическое преобразование одной модели в другую. Разница между моделями заключается в акцентах. Модели последовательностей концентрируются на временны*х* последовательностях событий, а в моделях коммуникации основное внимание уделяется отношениям между объектами (Rumbaugh et al., 2005).

3.4.1. Диаграммы последовательностей

Взаимодействие (interaction) представляет собой набор *сообщений* (messages), которыми обмениваются *роли* (roles), используя связи между ними (см. раздел А.2.3 приложения А). Диаграмма последовательностей представляет собой двумерный граф. *Роли* (объекты) располагаются по горизонтали. Последовательности сообщений располагаются сверху вниз по вертикали. Каждая вертикальная линия называется **линией жизни** (lifeline) объекта. Метод, вызванный в определенной точке линии жизни, называется **активацией** (activation), или *спецификацией выполнения* (execution specification), и изображается в виде узкого вертикального прямоугольника.

На рис. 3.13 показана простая диаграмма последовательностей, представляющая собой последовательность сообщений, необходимых для выполнения действия "Проверить рейтинг клиента" в диаграмме деятельности, показанной на рис. 3.6. Диаграмма содержит *объекты* четырех классов: `Employee`, `RentalWindow`, `CustomerVerifier` и `Customer`. Класс `Employee` описывает действующее лицо `Клиент`, класс `RentalVideo` является классом представления, класс `CustomerVerifier` относится к управляющим классам, класс `Customer` — к классам сущностей. *Линии жизни* объектов показаны пунктирными вертикальными линиями. *Активации* изображены в виде узких прямоугольников, расположенных вдоль линий жизни объектов.

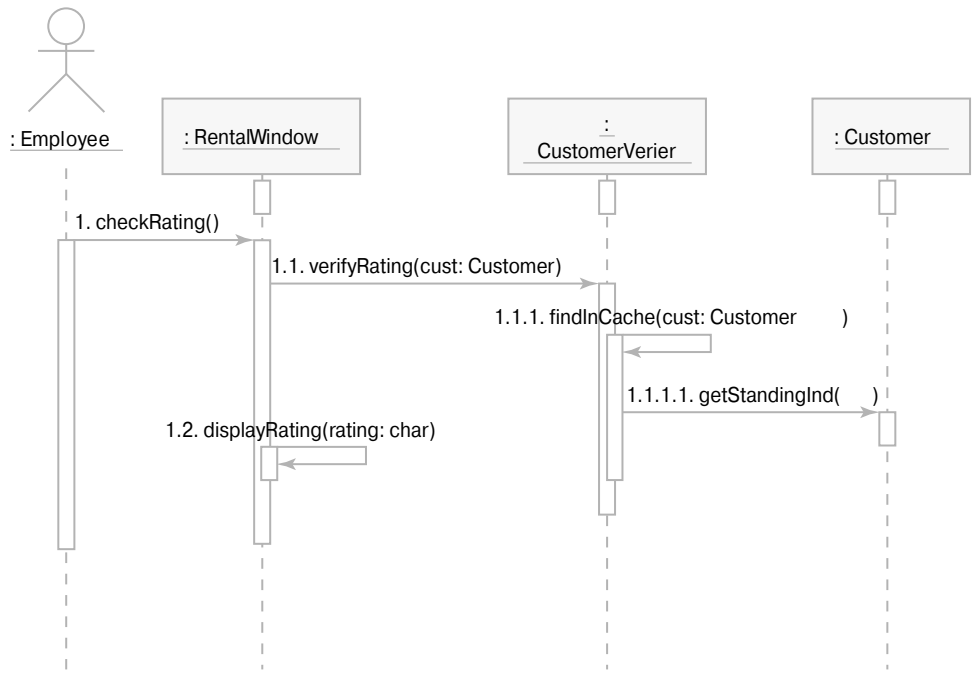


Рис. 3.13. Диаграмма последовательностей для действия "Проверить рейтинг клиента" системы управления магазином видеокассет

Обработка начинается с того, что объект класса `Employee` просит объект класса `RentalWindow` вызвать метод `checkRating()`. Получив сообщение, объект класса `RentalWindow` выводит информацию о транзакции аренды, которую необходимо выполнить для конкретного клиента. Это значит, что объект класса `RentalWindow` имеет ссылку на соответствующий объект класса `Customer`. Соответственно объект класса `RentalWindow` передает объекту класса `CustomerVerifier` объект `Customer` в виде аргумента сообщения `verifyRating()`. Объект `CustomerVerifier` является управляющим и отвечает

за логику программы и управление кэш-памятью объектов сущности. Поскольку текущая транзакция аренды относится к конкретному объекту класса `Customer` и обрабатывается объектом класса `RentalWindow`, можно предположить, что объект класса `Customer` находится в кэш-памяти (т.е. не извлекается из базы данных). Следовательно, объект класса `CustomerVerifier` посылает сообщение *сам себе* (self-message), т.е. сообщение собственному методу, чтобы найти идентификационный номер объекта `Customer`. Для этого используется метод `findCache()`.

Определив идентификационный номер объекта класса `Customer`, объект класса `CustomerVerifier` просит объект класса `Customer` сообщить свой рейтинг, вызвав метод `getStandingInd()`. Объекты, возвращаемые вызывающему объекту вызываемыми методами, на диаграмме последовательностей не показаны. *Возврат* (return) из вызова сообщений к вызывающему объекту происходит неявно (т.е. когда поток управления возвращается в вызывающий объект). Следовательно, значения атрибута `standingInd` класса `Customer` неявно возвращаются объекту класса `RentalWindow`. В этот момент объект класса `RentalWindow` посылает сообщение своему методу `displayRating`, предназначенное для сотрудника.

На рис. 3.13 использована иерархическая нумерация сообщений, демонстрирующая зависимости активации между сообщениями и соответствующими методами. Обратите внимание на то, что сообщение, передаваемое внутри класса, приводит к новой активации. В диаграмме последовательности существуют и другие важные возможности моделирования, которые будут рассмотрены позднее. Ниже перечислены лишь основные свойства диаграммы последовательностей.

Стрелка представляет собой *сообщение*, направляемое от вызывающего объекта (*отправителя*) к операции (методу) вызываемого объекта (*адресата*). Для каждого сообщения как минимум указывается его имя. Кроме того, могут быть указаны *фактические аргументы* (actual arguments) сообщения и другая информация, необходимая для управления. Фактические аргументы соответствуют *формальным аргументам* (formal arguments) метода объекта получателя.

Фактический аргумент может быть *входным аргументом* (передается от отправителя к адресату) или *выходным аргументом* (передается от адресата назад к отправителю). Входные аргументы могут быть обозначены ключевым словом `in` (если ключевое слово отсутствует, то предполагается, что аргумент входной). Выходные аргументы обозначаются ключевым словом `out`. Допускаются также аргументы типа `inout`, но для объектно-ориентированного подхода они не характерны.

Показывать на диаграмме возврат управления от объекта-адресата объекту-отправителю не обязательно. Стрелка, указывающая на объект-адресат, предполагает автоматический возврат управления отправителю. Адресат знает уникальный идентификатор объекта отправителя.

Сообщение может быть отправлено *коллекции* (collection) объектов (коллекция может быть набором, списком, массивом объектов и т.д.). Довольно частой является ситуация, когда вызывающий объект связан с несколькими объектами-адресатами (поскольку кратность ассоциации указана как "один ко многим" или "многие ко многим"). *Итеративный маркер* (iteration marker) — звездочка перед обозначением сообщения — указывает на процесс итерации сообщения по всей коллекции.

3.4.2. Диаграммы коммуникации

Диаграмма коммуникации (communication diagram) является альтернативным способом представления диаграммы последовательностей. Тем не менее между ними существуют важные различия. В диаграммах коммуникации нет линий жизни и ни одной активации. Как и на диаграммах последовательности, на диаграммах коммуникации используется иерархическая нумерация сообщений, однако эта нумерация не всегда представляет собой способ документирования последовательности вызовов методов. Некоторые модели остаются корректными, даже если сообщения на них не нумеруются.

На рис. 3.14 показана диаграмма коммуникации, соответствующая диаграмме последовательностей, приведенной на рис. 3.13. Многие CASE-инструменты способны автоматически преобразовать любую диаграмму последовательностей в диаграмму коммуникации (и наоборот).

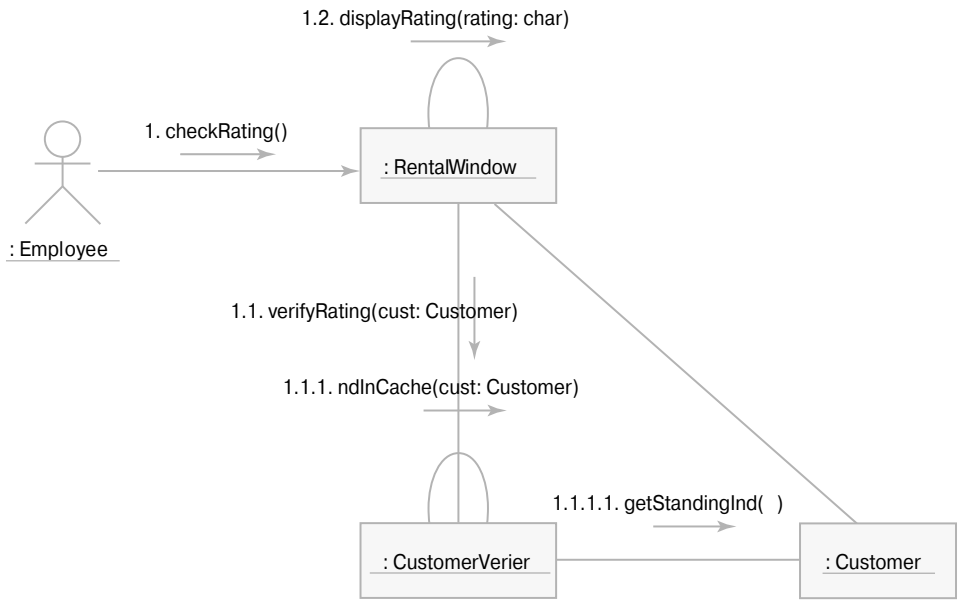


Рис. 3.14. Диаграмма коммуникации для действия "Проверить рейтинг клиента" системы управления магазином видеокассет

В целом, если модели содержат много объектов, диаграммы коммуникации представляют собой более полезный графический инструмент, чем диаграммы последовательностей. Кроме того, в отличие от диаграмм последовательностей сплошные линии между объектами могут (и должны) свидетельствовать о необходимости ассоциаций между классами этих объектов. Создание таких ассоциаций подтверждает тот факт, что объекты этих классов обмениваются сообщениями.

3.4.3. Методы классов

Изучение взаимодействий позволяет выявить **методы** (операции) классов. Зависимости между взаимодействиями и операциями носят очевидный характер. Каждое сообщение вызывает метод вызываемого объекта. Операция называется так же, как и сообщение.

Взаимно однозначное соответствие между *сообщениями* в моделях взаимодействий и *методами* в реализованных классах имеет свои пределы и определяется тем, насколько глубоко проработана модель взаимодействий, — некоторые вещи невозможно или нежелательно делать на этапе анализа. Таким образом, на этапах детального проектирования и реализации могут быть определены дополнительные методы.

Кстати, отметим, что аналогичное однозначное соответствие существует между *сообщениями* и *ассоциациями*, в частности между сообщениями, которыми обмениваются *персистентные* объекты (*сущности*), и ассоциациями. Такие сообщения должны поддерживаться персистентными связями (см. раздел А.2.3.1 приложения А). Аналогичные рассуждения справедливы и для временных объектов, существующих в оперативной памяти, к которым относятся и сущности, загружаемые в память (см. раздел А.2.3.2 приложения А). Следовательно, существование сообщения в диаграмме последовательности обусловливает необходимость ассоциаций в диаграмме классов.

На рис. 3.15 иллюстрируется использование взаимодействий для добавления операций в классы. Сообщения, получаемые объектами на диаграмме последовательностей, преобразуются в методы (операции) классов, представляющих эти объекты. Диаграмма классов также позволяет выявить типы возвращаемых значений и видимость методов. Эти две характеристики методов скрыты на диаграммах последовательностей.

Объект класса `RentalWindow` получает запрос на выполнение метода `checkrating()` и передает его методу `verifyRating()` класса `CustomerVerifier`. Поскольку объект класса `RentalWindow` обрабатывает объекты класса `Customer`, отображаемые в данный момент на экране дисплея, он передает этот объект в качестве аргумента методу `verifyRating()`. Эта передача порождает ассоциативную связь с классом `CustomerVerify`. Ассоциация осуществляется на основе роли `theCustVerifier`, которую можно реализовать как закрытый атрибут класса `RentalWindow` (закрытость этого атрибута обозначается знаком "минус" перед названием роли).

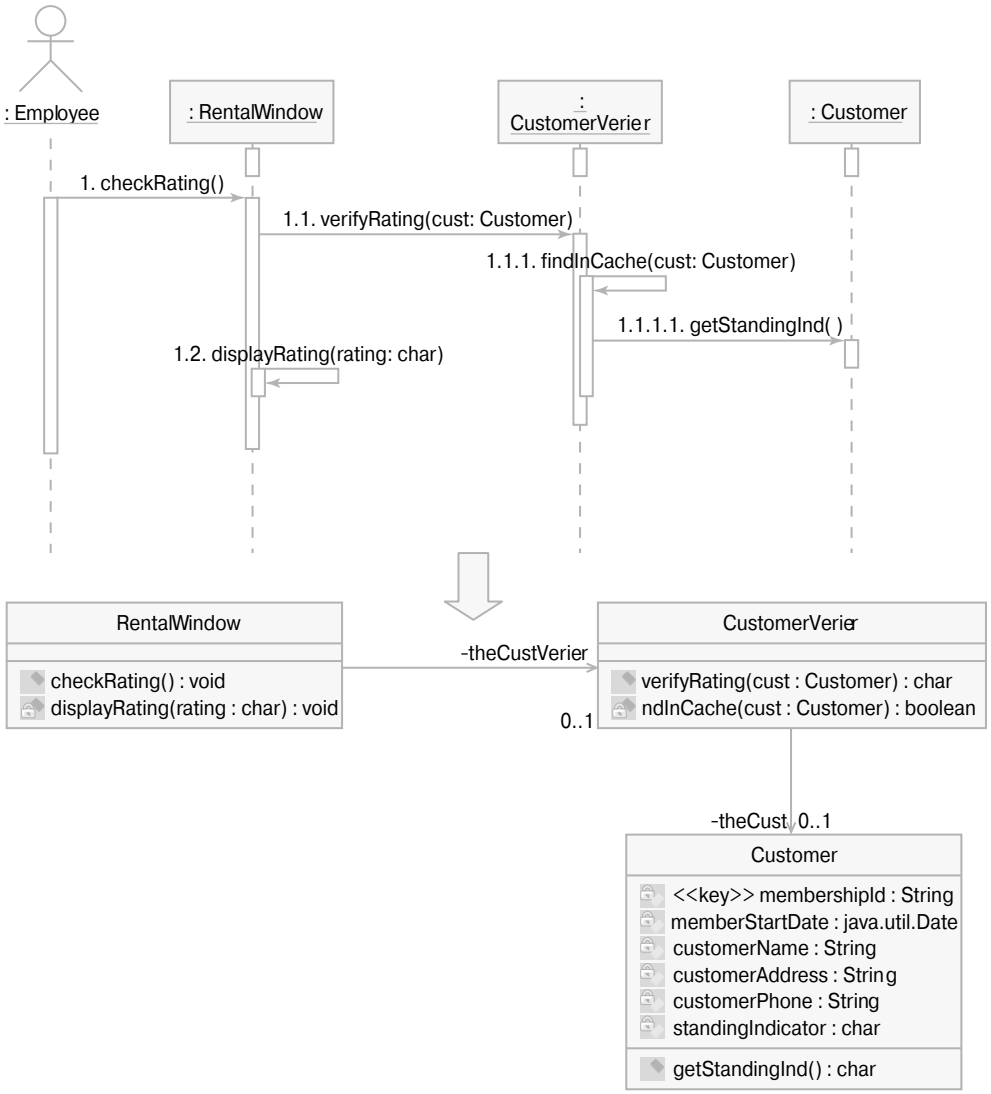


Рис. 3.15. Использование взаимодействий для добавления операций в классы системы управления магазином видеокассет

Для того чтобы убедиться, что объект класса `Customer` находится в памяти, метод `verifyRating()` использует закрытый метод `findInCache()` и присваивает атрибуту `theCust` ссылку на объект класса `Customer` (если этот атрибут не был установлен ранее). Затем объект класса `CustomerVerifier` просит объект класса `Customer` выполнить метод `getStandingInd()` и прочитать его атрибут `standingIndicator`. Значение типа `char`, присвоенное этому атрибуту, возвращается методу `checkRating()` класса `RentalWindow`.

Для того чтобы вывести на экран рейтинг клиента, находящегося под контролем объекта класса `RentalWindow`, метод `checkRating()` посылает сообщение методу `displayRating()`, передавая ему значение рейтинга. Метод `displayRating()` является закрытым, поскольку вызывается в классе `RentalWindow`.

Контрольные вопросы 3.4

КВ1. На каких диаграммах изображаются линии жизни — на диаграммах последовательностей или диаграммах коммуникации?

КВ2. Совпадают ли понятия сообщения и метода?

3.5. Ракурс конечных автоматов

Модель взаимодействий (interaction model) является источником подробной спецификации прецедента использования. *Модель конечного автомата* (state machine model) определяет динамические изменения класса. Эти динамические изменения обычно описывают поведение объекта в рамках нескольких прецедентов использования.

Состояние (state) объекта определяется текущими значениями его атрибутов (как элементарных атрибутов, так и атрибутов, обозначающих другие классы). Модель конечного автомата охватывает возможные состояния, в которых может находиться класс. Индивидуальность объекта остается одной и той же на протяжении всего жизненного цикла — она никогда не изменяется (см. раздел А.2.3 приложения А). Однако состояние объекта изменяется. Диаграмма конечного автомата представляет собой двудольный граф *состояний* и **переходов** (transitions), вызванных *событиями*.

3.5.1. Состояния и переходы

Значения атрибутов объекта изменяются, однако не все изменения приводят к переходу между состояниями. Рассмотрим объект `Bank account` (Банковский счет) и связанное с ним бизнес-правило, по которому банк отказывается от взимания платы за услуги по ведению счета, когда остаток на счету превышает 100 тыс. долл. Можно сказать, что объект `Bank account` переходит в привилегированное состояние. В противном случае объект пребывает в обычном состоянии. Остаток на счету изменяется после каждого снятия/вклада денег, однако состояние изменяется только тогда, когда баланс превышает или становится меньше 100 тыс. долл.

Приведенный выше пример схватывает сущность моделирования состояний. Модели конечных автоматов строятся для классов, которые характеризуются не просто изменениями состояний, а изменениями состояний, представляющими определенный *интерес* с точки зрения предметной области. Решение о том, что представляет интерес, а что нет, является прерогативой моделирования бизнес-процессов. Диаграмма конечного автомата представляет собой модель бизнес-правил. В течение некоторого времени бизнес-правила остаются неизменными. Они относительно независимы от конкретных прецедентов. В действительности прецеденты должны соответствовать бизнес-правилам.

В качестве примера рассмотрим класс `Rental` в системе управления магазином видеокассет. Класс `Rental` имеет атрибут `thePayment`, ассоциированный с классом `Payment` (см. рис. 3.9). В зависимости от природы этой ассоциации объект класса `Rental` может принимать разные состояния.

На рис. 3.16 показана модель конечного автомата для класса `Rental`. Состояния изображаются в виде прямоугольников с закругленными углами. События символизируются стрелками. Начальное состояние класса `Rental` (закрашенный кружок, из которого выходит стрелка) называется `Неполностью оплаченный счет`. Из состояния `Неполностью оплаченный счет` есть два возможных перехода. Если происходит событие "Частичная оплата", объект `Rental` переходит в состояние `Частично оплаченный счет`. В модели предусмотрена только одна возможность частичной оплаты. Если происходит событие "Окончательная оплата", когда класс `Rental` пребывает в состояниях `Неоплаченный счет` или `Частично оплаченный счет`, происходит переход в состояние `Полностью оплаченный счет`.

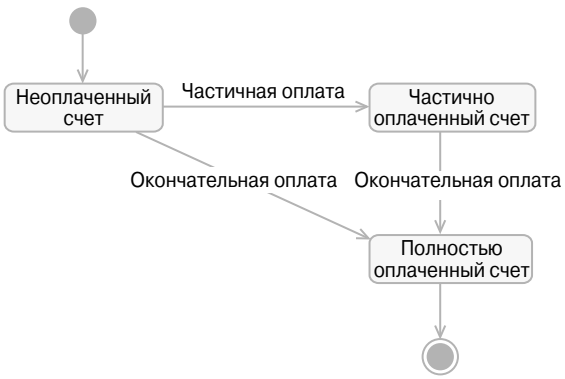


Рис. 3.16. Состояния и события, связанные с классом `Rental` *системы управления магазином видеокассет*

3.5.2. Диаграммы конечных автоматов

Диаграмма конечного автомата, или диаграмма состояний обычно связана с классом, но в общем случае она может ассоциироваться с другими концепциями моделирования, например прецедентами использования. Диаграмма состояний, присоединенная к классу, определяет способ реагирования объектов класса на события. Более точно, диаграмма определяет — для каждого состояния объекта — *действие* (action), выполняемое объектом при получении им сигнала о событии. Один и тот же объект может выполнять различные действия в ответ на одно и то же событие в зависимости от состояния объекта. Выполнение действия, как правило, вызывает изменение состояния.

Полное описание *перехода* (transition) состоит из трех частей:

```
событие (параметры) [сторожевое условие] / действие.
```

Каждая из частей является необязательной. Если строка перехода самоочевидна, допустимо опустить все компоненты. Событие — это кратковременное явление, которое воздействует на объект. Оно может обладать параметрами, например `нажата кнопка мыши (правая)`. Событие может быть защищено **сторожевым условием** (guard), например `нажата кнопка мыши (правая) [внутри окна]`. Событие происходит и воздействует на объект только в том случае, если условие принимает значение "истина".

Различие между событием и **сторожевым условием** не всегда очевидно. Это различие состоит в том, что событие *происходит* и может быть даже сохранено до тех пор, пока объект не приготовится к его обработке. С этих позиций сторожевое условие проверяется на истинность, чтобы определить, должен ли сработать переход.

Действие (action) представляет собой короткое атомарное вычисление, которое выполняется при срабатывании перехода. Действие также может быть связано с состоянием. В общем случае действие — это отклик объекта на обнаруженное событие. Состояние может содержать более продолжительное вычисление — *деятельность* (activity).

Состояние может состоять из других состояний, которые называются *вложенными* (nested state). *Составное состояние* (composite state) является абстрактным — это всего лишь родовое имя для всех вложенных состояний. Переход, который берет начало из-за границы составного состояния, означает, что он может сработать от любого из вложенных состояний. Это делает диаграмму более ясной и лаконичной. Конечно, переход, который берет начало вне границы составного состояния, может сработать от вложенного состояния.

Вернемся к классу `Rental` в примере, связанном с моделированием работы магазина видеокассет, и подумаем о состояниях, в которых могут находиться объекты этого класса (не в связи с оплатой, как на рис. 3.16, а в связи с атрибутами класса `Rental`). Диаграмма конечного автомата для класса `Rental` представлена на рис. 3.17. Эта диаграмма оттеняет преимущества моделирования состояний.

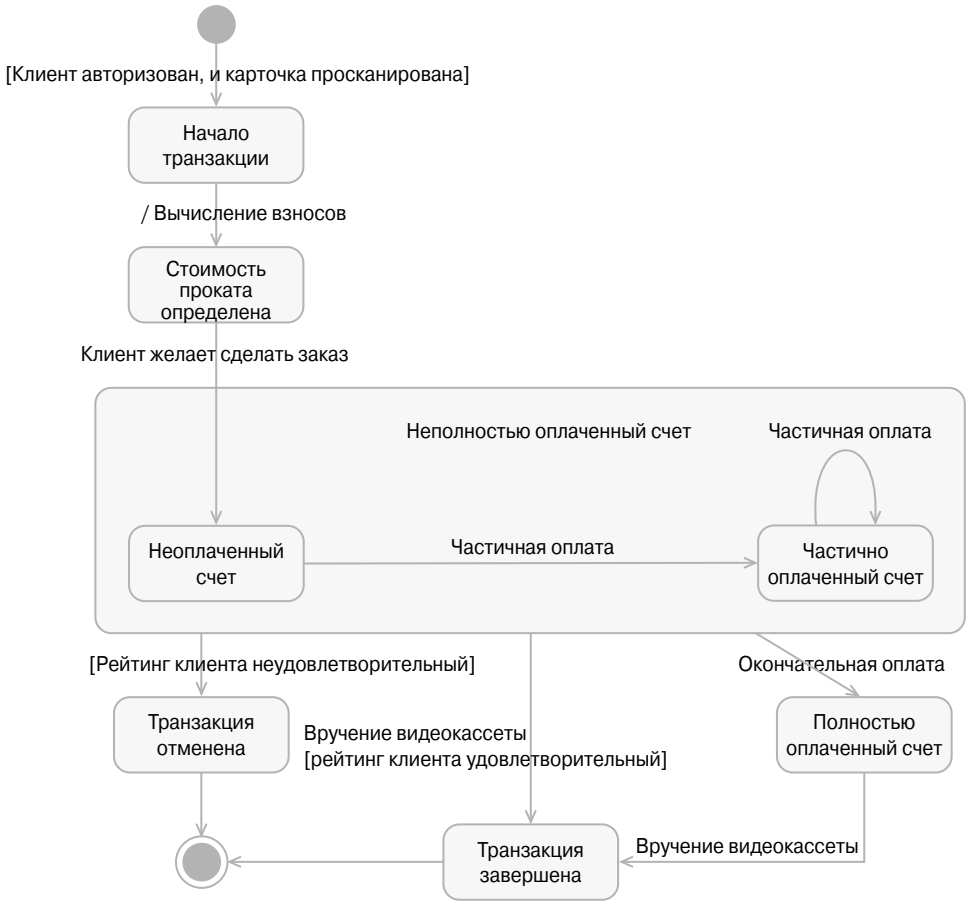


Рис. 3.17. Диаграмма конечного автомата для класса `Rental` *системы управления магазином видеокассет*

Модель конечного автомата, приведенная на рис. 3.17, переходит в состояние `Начало транзакции`, если выполняется сторожевое условие `[Клиент авторизован, и карточка просканирована]`. Переход в следующее состояние (`Стоимость проката определена`) требует выполнения действия `/ Вычисление взносов`. Если происходит *событие* `Клиент желает сделать заказ`, то объект класса `Rental` переходит в состояние `Неоплаченный счет`.

Состояние `Неоплаченный счет` является одним из двух вложенных состояний внутри составного состояния `Неполностью оплаченный счет`. Другим вложенным состоянием является состояние `Частично оплаченный счет`. Когда происходит событие `Частичная оплата`, модель делает *рекурсивный переход* (transition in self) в состояние `Частично оплаченный счет`. Это позволяет принимать несколько частичных платежей до полной оплаты транзакции.

Если модель находится в состоянии `Неоплаченный счет` и выполняется сторожевое условие `[рейтинг клиента неудовлетворительный]`, то происходит переход в состояние `Транзакция отменена`, которое является одним из двух заключительных состояний.

Второе заключительное состояние — `Транзакция завершена`. В это состояние можно перейти двумя путями. Во-первых, может произойти событие `Окончательная оплата`, вызывающее переход из состояния `Неполностью оплаченный счет` в состояние `Полностью оплаченный счет`. Если в этот момент происходит событие `Вручение видеокассеты`, то объект класса `Rental` переходит в состояние `Транзакция завершена`. Во-вторых, в заключительное состояние `Транзакция завершена` можно перейти из состояния `Неполностью оплаченный счет`, если выполняется сторожевое условие `[рейтинг клиента удовлетворительный]`.

Контрольные вопросы 3.5

КВ1. Может ли состояние объекта зависеть от его ассоциативных связей?

КВ2. Чем сторожевое условие отличается от события?

3.6. Ракурс реализации

Язык UML предоставляет инструменты для архитектурного/структурного моделирования физической реализации системы — *диаграммы компонентов* (component diagram) и *диаграммы развертывания* (deployment diagram) (Alhir, 2003; Maciaszek and Liong, 2005). Обе эти диаграммы принадлежат к более широкой категории *структурных диаграмм* (structure diagram), наиболее известными среди которых являются *диаграммы классов* (см. раздел 3.3.6).

Несмотря на то что модели реализации относятся к области физического моделирования, они должны быть тесно связанными с логической структурой системы. Основной логической конструкцией является *класс*, а основной логической структурной моделью — *диаграмма классов*. Существуют также другие понятия, связанные с логической структурой системы, — *подсистема* и **пакет** (package).

3.6.1. Подсистемы и пакеты

Согласно древнеримскому изречению “Разделяй и властвуй!”, рекомендуется занимать доминирующую позицию, изолируя врагов и используя разногласия между ними. При решении задач этот принцип используется в несколько ином смысле. Он требует, чтобы крупная проблема была разделена на несколько задач меньшего размера.

Принцип "Разделяй и властвуй" порождает иерархическую структуру предметной области. При разработке систем он приводит к разделению системы на подсистемы и пакеты. Это разделение должно быть продуманным и нацеленным на сокращение зависимостей между иерархией подсистем и пакетов.

Понятие *подсистемы* конкретизирует (наследует) понятие **компонента** (Ferm, 2003) и моделируется как ее стереотип. Подсистема инкапсулирует определенную часть поведения системы. Сервисы, предоставляемые системой, обеспечиваются за счет ее внутренних частей — классов. Это также означает, что подсистема не предусматривает реализацию как самостоятельная сущность (Selic, 2003).

Сервисы *подсистемы* могут и должны определяться с помощью **интерфейсов** (см. раздел A.9 приложения A). Выгоды инкапсуляции поведения и обеспечения сервисов через интерфейсы многочисленны и включают изоляцию от изменений, возможность замены сервисов, расширяемость и возможность повторного использования.

Подсистемы можно структурировать как архитектурные слои, так что зависимости между слоями являются ациклическими и минимальными. Внутри каждого слоя подсистемы могут быть вложенными друг в друга. Это значит, что подсистема может содержать другие подсистемы.

Пакет — это именованная группа элементов моделирования. Как и подсистема, сервисы, обеспечиваемые пакетом, являются результатом сервисов, обеспечиваемых внутренними частями, т.е. классов. В отличие от подсистемы пакет не раскрывает поведение с помощью интерфейсов. Как указано в работе Ferm (2003): "Разница между подсистемой и пакетом заключается в том, что, работая с пакетом, клиент просит *элемент, находящийся внутри*, выполнить определенную функцию, а работая с подсистемой, он обращается к *ней самой*".

Аналогично подсистеме, пакет может содержать другие пакеты. В отличие от подсистемы пакет можно непосредственно реализовать в виде конструкции языка программирования, например в виде пакета языка Java или пространства имен в языке .NET. Как и подсистема, пакет владеет своими членами. Член (класс, интерфейс) может принадлежать только одной подсистеме или пакету.

Подсистема представляет собой более богатое понятие, обладающее как структурными свойствами пакетов, так и поведенческими аспектами классов. Функционирование подсистемы обеспечивается с помощью одного или нескольких интерфейсов.

На рис. 3.18 показана диаграмма UML, демонстрирующая различия между пакетом и подсистемой. Подсистема — это компонент, стереотипизированный как «`subsystem`». Тот факт, что подсистема инкапсулирует свое поведение, представлен в виде *предоставляемых интерфейсов* (см. следующий раздел). Интерфейс `FinanceInterface` реализуется с помощью подсистемы `Financial transaction` и реализуется классом `RentalWindow` в пакете GUI.

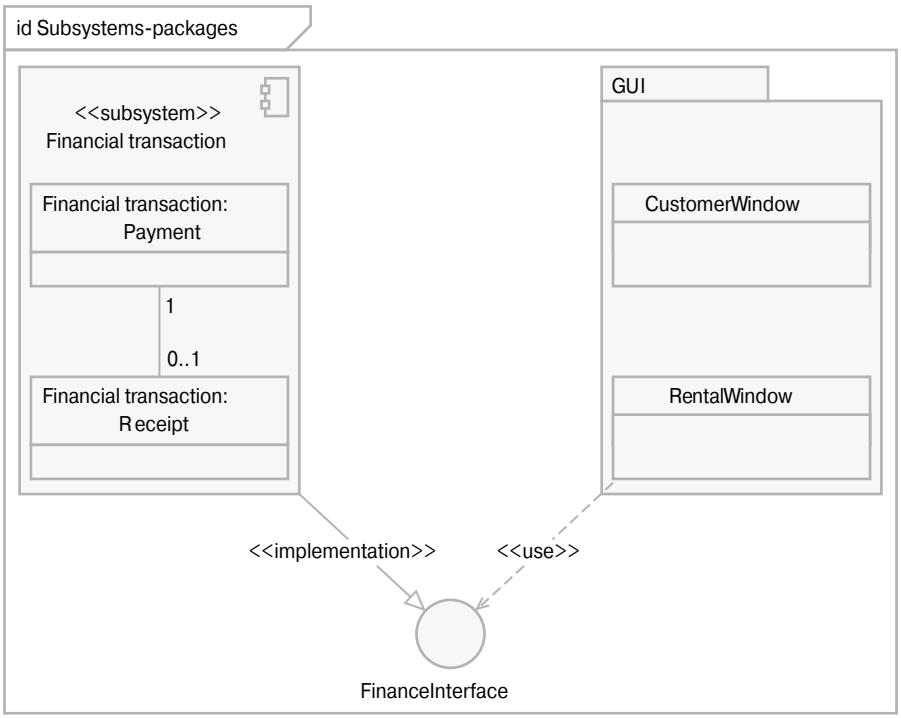


Рис. 3.18. Подсистема и пакет

В целом, для того чтобы минимизировать зависимости в системе, интерфейсы следует разместить вне подсистемы, которая ее реализует (Maziaszek and Liong, 2005). Несмотря на то что этот факт на рис. 3.18 не отражен, возможно размещение всех или большинства интерфейсов в отдельном пакете. И наоборот, если бы на рис. 3.18 классами GUI были только классы, использующие интерфейс `FinanceInterface`, то этот интерфейс можно было бы разместить в пакете GUI.

3.6.2. Компоненты и диаграммы компонентов

"*Компонент* представляет собой модульную часть системы, инкапсулирующую свое содержание и допускающую замену внутри своего окружения. Компонент определяет свое поведение в терминах предоставляемых и требуемых интерфейсов" (UML, 2005).

Предлагаемый интерфейс (provided interface) реализуется компонентом (или другим **классификатором** (classifier), например, классом). Он обеспечивает сервисы, которые его экземпляры предоставляют своим клиентам. *Требуемый интерфейс* (required interface) может использоваться компонентом (или другим классификатором, например, классом). Он обеспечивает сервисы, в которых нуждается компонент, чтобы выполнять свои функции и, в свою очередь, предоставлять услуги своим клиентам.

Диаграмма компонентов (component diagram) относится к моделированию структуры и зависимостей компонентов в реализуемых системах. Несмотря на то что подсистему можно рассматривать как конкретизацию понятия компонента, язык UML использует для визуализации компонентов отдельный графический элемент. Этот элемент отличается от стереотипного пакета (см. рис. 3.18). Компонент изображается как прямоугольник с ключевым словом «`component`» и, возможно, пиктограммой в правом верхнем углу прямоугольника (рис. 3.19).

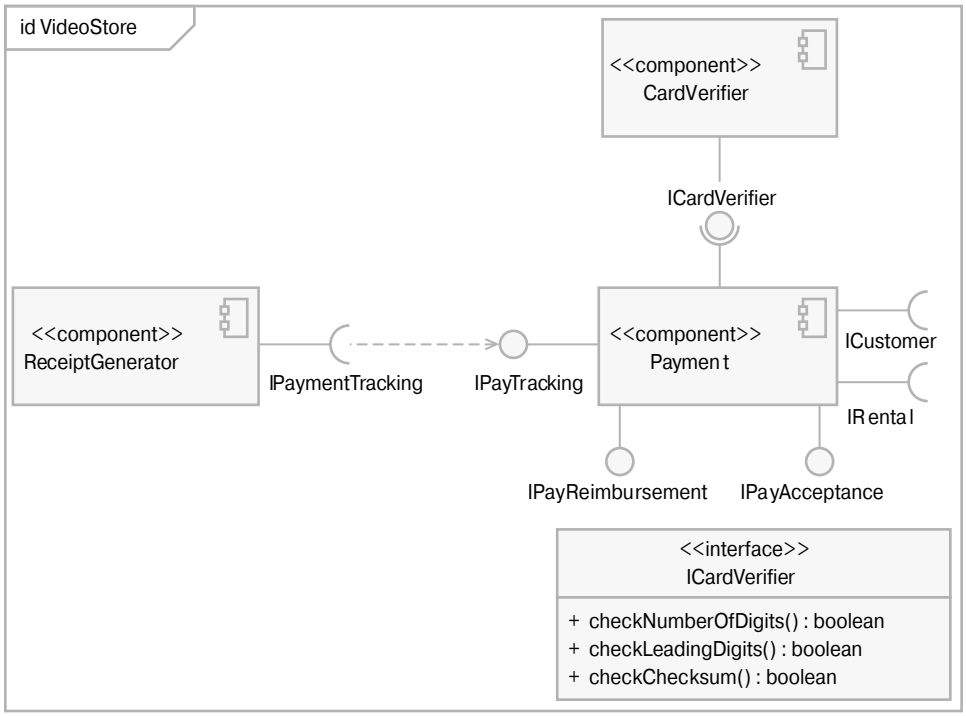


Рис. 3.19. Диаграмма компонентов для системы управления магазином видеокассет

Концепция компонента относится к области покомпонентной разработки (см. раздел 2.1.3.2 главы 2). Обозначение компонентов подчеркивает тот факт, что они состоят из предоставляемых и требуемых интерфейсов. *Предоставляемый интерфейс* изображается в виде маленького кружочка, присоединенного к компоненту. *Требуемый интерфейс* изображается в виде маленького полукруга, присоединенного к компоненту.

На рис. 3.19 показаны три компонента — `ReceiptGenerator`, `CardVerifier` и `Payment`. Компонент `Payment` имеет три предлагаемых и три требуемых интерфейса. Один из требуемых интерфейсов (`ICardVerifier`) предоставляется классом `CardVerifier`. Один из требуемых интерфейсов (`IPayTracking`) запрашивается классом `ReceiptGenerator`. Диаграмма показывает два разных

способа объединения предлагаемых и требуемых интерфейсов. Пунктирная линия зависимостей, связывающая интерфейсы `IPayTracking`, необходима лишь потому, что эти два интерфейса имеют разные имена.

На рис. 3.19 также показана спецификация интерфейса `ICardVerifier`. Этот интерфейс определяет три операции для верификации карточки для оплаты. Первый метод, `checkNumberDigits()`, проверяет, содержит ли кредитная карточка правильное количество цифр на магнитной ленте, соответствующее типу карточки (например, 13, 15 и 16 цифр). Следующий метод, `checkLeadingDigits()`, проверяет корректность начальных цифр для каждого типа кредитных карточек. И наконец, метод `checkChecksum()` вычисляет контрольную сумму для номера кредитной карточки и сравнивает ее с номером, чтобы убедиться, что номер карточки имеет правильную структуру.

3.6.3. Узлы и диаграммы развертывания

"*Узел* — это вычислительный ресурс, с помощью которого артефакт может развертываться для дальнейшего выполнения. Узлы могут быть связаны друг с другом с помощью путей коммуникации, определяющих структуру сети" (UML, 2005). "**Артефакт** (artefact) — это спецификация физической порции информации, т.е. информации, используемой или создаваемой в процессе разработки программного обеспечения или в ходе развертывания и эксплуатации системы. Примерами артефактов служат файлы модели, исходные файлы, сценарии и бинарные выполняемые файлы, таблицы в базах данных, комплектующие узлы программного обеспечения, текстовые процессоры или сообщения электронной почты" (UML, 2005).

Диаграмма развертывания относится к моделированию структуры и зависимостей между узлами, определяющими среду реализации системы. Узлы соединяются с помощью ассоциаций, образующих пути коммуникации. Показатели кратности ассоциаций означают число экземпляров каждого узла, участвующего в ассоциации.

Диаграмма развертывания определяет распределение компонентов и других выполняемых процессорных элементов по узлам сети (Alhir, 2003). Узел может быть сервером (например, принтером, адресом электронной почты, Web-страницей или сервером базы данных), а также любым другим вычислительным или трудовым ресурсом, доступным компонентам и другим процессорным элементам.

Узлы изображаются в виде куба. Зависимость «`deploy`» между артефактом и узлом означает развертывание. Альтернативно символы артефактов можно разместить внутри символов узлов, чтобы указать, где они будут размещены.

На рис. 3.20 продемонстрирован пример диаграммы развертывания, содержащей три узла — `Клиентская машина`, `Сервер приложения` и `Сервер базы данных`. Сервер приложения может быть связан со многими клиентскими машинами. С отдельным сервером базы данных может быть связан только один сервер

приложения. Узел Сервер приложения состоит из трех компонентов, приведенных на рис. 3.20. Артефакт VideoStore.exe развернут на узле Клиентская машина, а артефакт schema.xml — на узле Сервер базы данных.

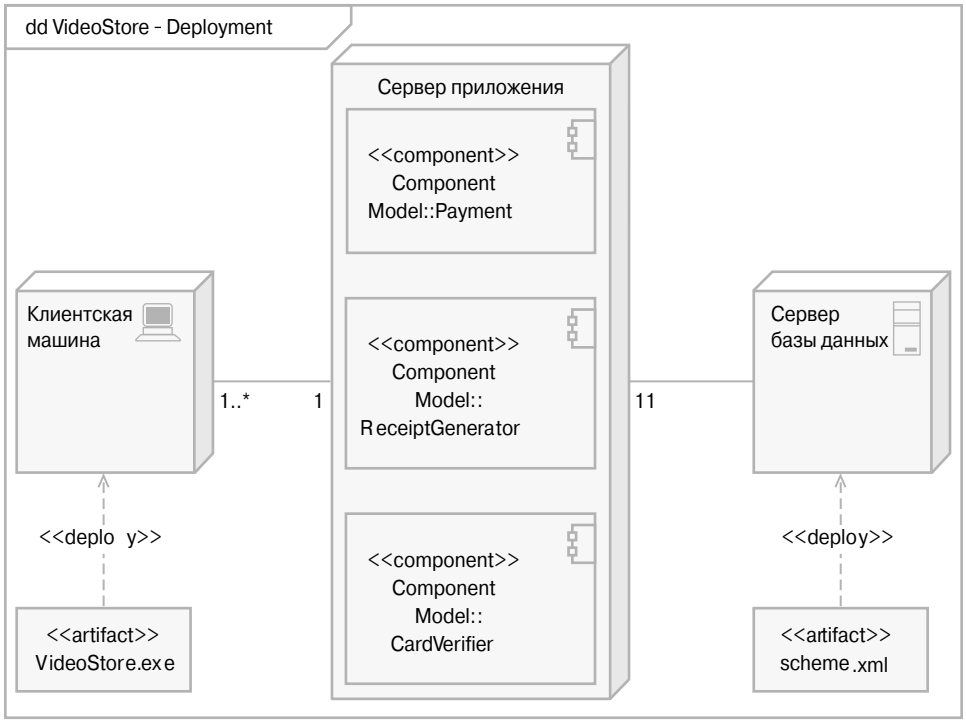


Рис. 3.20. Диаграмма развертывания системы управления магазином видеокассет

Контрольные вопросы 3.6

КВ1. Как моделируется подсистема: как стереотип пакета или стереотип компонента?

КВ2. Приведите несколько примеров артефактов, развернутых в узле.

Резюме

В данной главе нам удалось охватить значительную часть вопросов, связанных с основаниями анализа требований. Были введены основополагающие термины и понятия, касающиеся объектной технологии. При изложении принципов объектного моделирования использовался отдельный учебный пример — система

управления магазином видеокассет. Новичков задания должны были повергнуть в уныние. Награда за настойчивость не заставит себя ждать — вы получите ее в следующих главах.

Стандарт языка UML определяет большое количество моделей и диаграмм, облегчающих подробное и многоцелевое моделирование программного обеспечения. Эти модели и диаграммы можно классифицировать по *ракурсам* системы — от поведения до ее структуры.

Модель прецедентов использования — это основной инструмент языка UML, с помощью которого моделируется поведение систем. Эта модель определяет прецеденты использования, действующих лиц и отношения между ними. Каждый прецедент использования описывается в отдельном текстовом документе.

Модель деятельности позволяет графически представить поток событий в прецеденте использования. Модели активности заполняют пробел между высокоуровневым представлением поведения системы в *моделях прецедентов использования* и более низкоуровневых представлениях поведения *моделей взаимодействия* (диаграмм последовательностей и коммуникации).

Моделирование классов интегрирует и охватывает все виды моделирования. Модели классов идентифицируют классы и их атрибуты, включая отношения. Классы принадлежат разным архитектурным слоям. Типичными группами классов являются классы представления, сущности, посредники и базовые классы.

Моделирование взаимодействий охватывает взаимодействия объектов, необходимое для выполнения прецедентов использования или их частей. Существуют два вида диаграмм взаимодействия: *диаграммы последовательностей* и *диаграммы коммуникаций*. Модели последовательностей описывают временны*е* последовательности, а модели коммуникации — отношения между объектами. Между сообщениями в модели взаимодействия и методами в реализованных классах существует взаимно однозначное соответствие.

Модель конечного автомата определяет динамические изменения классов. Она иллюстрирует разные состояния, в которые могут переходить объекты класса. Эти динамические изменения описывают поведение объекта на протяжении выполнения всех прецедентов использования, включающих класс, которому принадлежит этот объект. Диаграмма конечного автомата представляет собой двудольный граф состояний и переходов, вызванных событиями.

Язык UML содержит *диаграммы компонентов* и *диаграммы развертывания* — два инструмента, предназначенных для архитектурного/структурного моделирования физической реализации системы. Основными элементами этих диаграмм являются компоненты, интерфейсы, узлы и артефакты. Модели реализации используют еще две архитектурные концепции — *подсистему* и *пакет*.

Ключевые термины

Агрегация (aggregation). "Вид ассоциации, определяющей отношение "часть–целое" между агрегатом (целым) и составной частью" (Rumbaugh et al., 2005).

Активация (activation). Называется также спецификацией выполнения и "моделирует выполнение функции или операции, включая период, в течение которого операция вызывает другие субординированные действия" (Rumbaugh et al., 2005).

Артефакт (artefact). "Спецификация физической порции информации, т.е. информации, используемой или создаваемой в процессе разработки программного обеспечения или в ходе развертывания и эксплуатации системы" (UML, 2005).

Ассоциация (association). "Семантическое отношение между двумя или несколькими классификаторами, использующее связи между их экземплярами" (Rumbaugh et al., 2005).

Атрибут (attribute). "Описание именованного элемента конкретного типа в классе; каждый объект класса хранит значение этого типа отдельно" (Rumbaugh et al., 2005).

Взаимодействие (interaction). "Спецификация способом, с помощью которого роли обмениваются сообщениями в контексте выполнения задачи. Этот контекст определяется классификатором или сотрудничеством" (Rumbaugh et al., 2005).

Действие (action). "Основная единица спецификации поведения. Действие принимает набор входных данных и преобразует его в набор выходных данных, причем одно или оба множества данных могут быть пустыми" (UML, 2005).

Деятельность (activity). "Спецификация параметризованной функции в виде координированного упорядочения субординированных единиц, элементами которых являются действия" (UML, 2005).

Интерфейс (interface). "Вид классификатора, представляющего собой объявление связных открытых сущностей и обязанностей. Он определяет некий контракт, которой должен выполнять любой экземпляр классификатора, реализующий данный интерфейс" (UML, 2005).

Класс (class). "Вид спецификатора, свойствами которого являются атрибуты и операции" (UML, 2005).

Классификатор (classifier). "Классификация экземпляров, описывающая множество экземпляров, обладающих общими свойствами. Классификатор — это абстрактный элемент модели и не имеет обозначения" (UML, 2005).

Композиция (composition). Составная агрегация; "сильная форма агрегации, предполагающая, что часть экземпляра в любой момент времени должна входить в состав хотя бы одной композиции" (UML, 2005).

Компонент (component). "Представляет собой модуль системы, инкапсулирующий свое содержание и допускающий замену в своем окружении" (UML, 2005). См. также пункт *Компонент* в разделе *Ключевые термины* главы 1.

Линия жизни (lifeline). "Понятие, представляющее отдельного участника взаимодействия" (UML, 2005).

Метод (method). "Реализация операции, определяющая алгоритм или процедуру, создающую результат операции" (Rumbaugh et al., 2005).

Обобщение (generalization). "Таксономическое отношение между общим и конкретным классификаторами. Каждый экземпляр конкретного классификатора является косвенным экземпляром общего классификатора. Таким образом, конкретный классификатор наследует характеристики более общего классификатора".

Операция (operation). "Функциональное свойство классификатора, определяющее имя, тип, параметры и ограничения при вызове соответствующей функции" (UML, 2005).

Пакет (package). "Используется для группирования элементов и создания пространства имен" (UML, 2005).

Переход (transition). "Отношение между двумя состояниями конечного автомата, означающее, что объект, находящийся в первом состоянии при наступлении определенного события или при выполнении сторожевого условия, осуществляет некое действие или действия и переходит во второе состояние. Если происходит изменение состояния, то говорят, что запускается переход" (UML, 2005).

Поток управления (control flow). "Ребро, начинающееся в узле действия, после того как завершится предыдущее" (UML, 2005).

Прецедент использования (use case). "Спецификация набора действий, выполняемых системой и приводящих к наблюдаемому результату, который обычно имеет ценность для одного или нескольких участников проекта" (UML, 2005).

Роль (role). "Составляющий элемент структурированного классификатора, определяющий внешний вид его экземпляра (или, возможно, набора экземпляров) в определяемом им контексте" (UML, 2005).

Событие (event). "Спецификация случая, который может оказать влияние на объект" (UML, 2005).

Сообщение (message). "Понятие, определяющее конкретный вид коммуникации во взаимодействии. Коммуникацией может быть, например, сигнал, вызывающий операцию, а также создание или уничтожение экземпляра" (UML, 2005).

Состояние (state). "Моделирует ситуацию, в которой (обычно неявно) выполняется инвариантное условие. Инвариант может описывать статичную ситуацию, например объект, ожидающий наступления внешнего события. Однако он может также моделировать динамичные условия, например выполняемый процесс" (UML, 2005).

Сторожевое условие (guard). "Ограничение, обеспечивающее строгий контроль над запуском транзакции. Сторожевое условие проверяется, когда конечный автомат ре-

гистрирует событие. Если сторожевое условие выполняется в этот момент, то транзакция становится возможной, в противном случае она отменяется" (UML, 2005).

Узел (node). "Вычислительный ресурс, в котором разворачивается артефакт для выполнения задачи" (UML, 2005).

Многовариантные тесты

МТ1. Действующее лицо — это ...

а. Роль.

б. Сущность, внешняя по отношению к субъекту.

в. Сущность, действующая в прецеденте использования.

г. Все перечисленное выше.

МТ2. Выполнением этапа деятельности называется...

а. Прецедент.

б. Действие.

в. Работа

г. Поток.

МТ3. Параллельные потоки вычислений в диаграмме деятельности определяются с помощью...

а. Ветвления и воссоединения.

б. Разделения и слияния.

в. Сторожевых условий и слияния.

г. Все вышеприведенное не верно.

МТ4. Классы, ответственные за взаимодействие с внешними источниками данных, называются...

а. Внешними классами.

б. Классами коммуникации.

в. Классами ресурсов.

г. Ни один вариант не подходит.

МТ5. Черный ромб на линии агрегации означает, что...

а. Агрегация осуществляется по значению.

б. Агрегация осуществляется в композиции.

в. Объект подкласса может быть частью только одного суперкласса.

г. Все перечисленное выше.

МТ6. Формальный аргумент может входить в определение...

а. Метода.

б. Сообщения.

в. Вызова.

г. Всего перечисленного выше.

МТ7. Диаграмма конечного автомата — это двудольный граф состояний и переходов, вызванных...

а. Деятельностью.

б. Триггерами.

в. Событиями.

г. Всем перечисленным выше.

МТ8. Вычислительный ресурс, на котором может быть развернут артефакт для последующего выполнения, называется...

а. Компонентом.

б. Узлом.

в. Пакетом.

г. Все вышеприведенное не верно.

Вопросы

В1. Назовите основные характеристики и взаимно дополняющие свойства статической модели, модели поведения и модели конечного автомата.

В2. Может ли действующее лицо иметь атрибуты и операции? Поясните свой ответ.

В3. Объясните, какова роль и место диаграммы деятельности в моделировании системы?

В4. Что такое класс сущностей? Какие другие категории и классы можно выделить в моделировании классов? Обоснуйте свой ответ.

В5. Что такое фактический и формальный аргумент?

В6. В чем заключается разница между действием и деятельностью в диаграммах конечных автоматов? Приведите примеры.

В7. Объясните, почему подсистемы реализуют интерфейсы, а пакеты нет? Какие будут последствия для моделей реализации, если подсистемы, к которым они относятся, не реализуют интерфейсы?

В8. Как артефакты связаны с узлами? Как это отношение моделируется визуально?

Упражнения

У1. Обратитесь к рис. А.2 приложения А (раздел А.2.2). Проанализируйте последствия следующих изменений логики сотрудничества между объектами при перевозке продукции и пополнении запасов.

Перевозка и пополнение представляют собой разные потоки обработки. Когда объект класса `Order` (Заказ) создает новый объект класса `Shipment` (Перевозка) и запрашивает поставку, объект класса `Shipment` получает объекты класса `Product` (Продукция), как показано на рис. А.2. Предположим, что изменением количества товаров управляет не объект класса `Stock`, а объект класса `Shipment`, использующий свои собственные методы обработки объектов класса `Product` и вызывающий метод `getProdQuantity()` непосредственно.

В этом сценарии объект класса `Product` знает количество своих объектов и момент повторного заказа. Следовательно, когда потребуется пополнение запасов, будет создан новый объект класса `Purchase`, чтобы обеспечить сервис `reorder()`.

Модифицируйте диаграмму на рис. А.2, чтобы учесть описанные выше изменения. Сообщения, порождаемые объектами классов `Shipment` и `Purchase`, показывать не обязательно, поскольку вопросы, связанные с созданием объектов, пока объяснены не полностью.

У2. Обратитесь к рис. А.2 приложения А (раздел А.2.2) и рис. 3.23 (т.е. к упражнению У.1). Определите тип возвращаемого значения во всех сообщениях на обеих диаграммах. Объясните, как возвращаемые значения используются при последовательных вызовах.

У3. Обратитесь к рис. А.14 приложения А (раздел А.5.2). Предположим, что курс состоит из лекций и семинаров, причем один преподаватель может читать лекции, а другой вести семинары. Модифицируйте диаграмму на рис. А.14, чтобы отразить этот факт.

У4. Обратитесь к рис. А.16 приложения А (раздел А.5.4). Предложите альтернативную модель, которая не использует ни ассоциативный класс, ни тернарную ассоциацию (что вообще не рекомендуется в этой книге). Опишите семантические различия между моделью, показанной на рис. А.16 и новой моделью, если они существуют.

У5. Обратитесь к рис. 3.21, на котором показано, что объекты класса `Book` (Книга) содержат объекты класса `Chapter` (Глава), а каждый объект класса `Chapter` состоит из объектов класса `Paragraph` (Раздел). Управляющим является класс `Finder` (Поисковая машина). Предположим, что класс `Finder` должен выводить на экран все атрибуты `chptNumber`

и `paraNumber`, описывающие главу и абзац, содержащие строку запроса. Включите в класс соответствующие операции. Нарисуйте диаграмму коммуникации между объектами и объясните, как выполняется обработка запроса, в том числе укажите типы значений, возвращаемых операциями.

У6. Обратитесь к рис. 3.21 и решению упражнения У.5, показанному на рис. 3.25. Дополните диаграмму классов операциями, необходимыми для обработки запроса. Приведите псевдокод или код на языке Java для классов `Finder`, `Book` и `Chapter`.

У7. Обратитесь к рис. А.22 приложения А (раздел А.7.3). Дополните пример, включив атрибуты в классы `Teacher` (Преподаватель), `Student` (Студент), `Postgraduate` (Аспирант) и `Tutor` (Наставник).

У8. Обратитесь к рис. А.24 приложения А (раздел А.9.1) и А.25 (раздел А.9.2). Дополните рис. А.24 и А.25 (см. раздел "Решения упражнений с нечетными номерами"), считая, что класс `Videomedium` (Носитель видеоинформации) является разновидностью класса `EntertainmentMedium` (Средства развлечения), как и класс `SoundMedium` (Носитель аудиоинформации), например компакт-диски. Разумеется, класс `VideoMedium` также является разновидностью класса `SoundMedium`. Аналогичную классификацию можно применить к оборудованию (средствам воспроизведения).

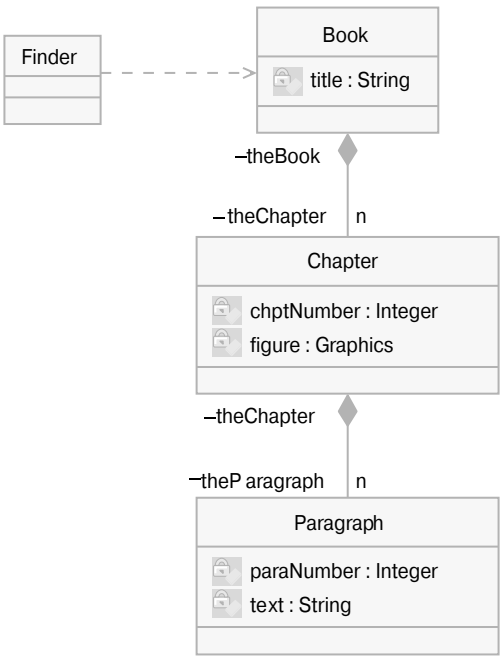


Рис. 3.21. Агрегация класса `Book`

Упражнения. Магазин видеокассет

MB1. На диаграммах прецедентов использования, приведенных на рис. 3.3 и 3.4 (см. раздел 3.1.3), не показано явно, что модель относится к прокату видеокассет (речь идет лишь о сканировании кредитных карточек и видеокассет, принятии платежей и выписывании квитанций). На практике прецеденты использования требуют выполнения транзакции, связанной с выдачей видеокассет напрокат и занесением информации в базу данных. Более того, диаграмма прецедентов использования не предусматривает проверку возраста клиента (для того чтобы брать напрокат видеокассеты с фильмами категории R или X, клиент должен быть старше 18 лет).

Дополните диаграмму прецедентов использования с учетом изложенных выше рассуждений. Кроме того, учтите, что процесс выдачи видеокассеты напрокат начинается после сканирования членской карточки клиента, а также видеокассеты или компакт-диска, и примите во внимание следующие условия: 1) видеокассету можно взять напрокат без внесения оплаты (в некоторых случаях), 2) магазин производит проверку возраста клиента и отказывает лицам младше 18 лет, если фильм относится к категории R или X, и 3) клиент может взять видеокассету напрокат как с квитанцией, так и без нее (по желанию).

MB2. Обратитесь к решению упражнения MB1 (рис. 3.27) и диаграмме деятельности, приведенной на рис. 3.6 (см. раздел 3.2.2). Разработайте дополнительную диаграмму активности для прецедента использования `Прокат видеокассет` и вспомогательный прецедент использования, расширяющий прецедент `Прокат видеокассет` (повторять спецификации прецедента `Принятие платежа`, также показанного на рис. 3.6, не обязательно).

MB3. Обратитесь к рис. 3.9 (см. раздел 3.3.3). Предположим, что не все объекты класса `Payment` связаны с классом `Rental`. Некоторые видеокассеты предназначены для продажи, поэтому объекты класса `Payment` могут быть связаны с классом `Sale`. Кроме того, будем считать, что отдельный платеж может относиться к нескольким заказам сразу. Как эти изменения влияют на модель? Постройте модифицированную и расширенную модель.

MB4. Обратитесь к рис. 3.6 (см. раздел 3.2.2). Постройте диаграмму последовательностей для деятельности `Обработка платежа по кредитной карточке`.

MB5. Обратитесь к диаграмме класса, показанной на рис. 3.12 (см. раздел 3.3.6), и рассмотрите класс `Video`. Кроме информации, которую можно извлечь из модели класса, представьте, что магазин выставляет видеокассеты на

продажу, если количество заказов на соответствующий фильм падает ниже установленного уровня (например, если фильм не заказывают в течение недели). Кроме того, представьте, что в магазине регулярно проверяют качество видеокассет и отбраковывают дефектные.

Постройте диаграмму конечного автомата для класса `Video`. Разработайте альтернативную версию диаграммы, в которой используется абстрактное составное состояние `На складе`, означающее, что видеокассета есть на складе, повреждена или может быть продана.

Вернитесь к обсуждению модели реализации в разделе 3.6. Рассмотрите модель реализации, показанную на рис. 3.22, где скрыты отношения между объектами.

Предположим, что компоненты `Поставка` и `Прокат видеокассет` реализуются классами в пакете `Управление запасами`. Компонент `Поставка` требует доступа к компоненту `Прокат видеокассет`. Работу компонента `Прокат видеокассет` поддерживает компонент `Сервер склада`, обеспечивающий функционирование компонентов `Поставка` и `Сервер магазина видеокассет`. Дополните диаграмму, приведенную на рис. 3.22, показав зависимости и другие отношения.

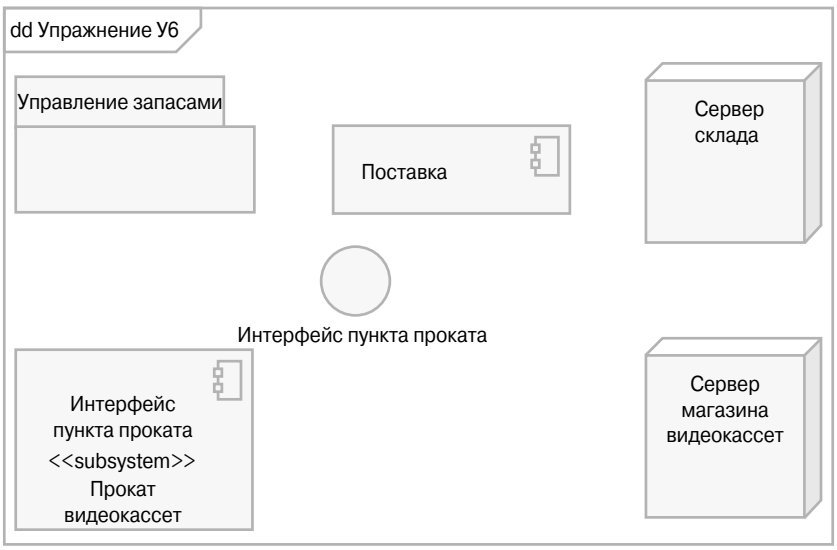


Рис. 3.22. Модель реализации (отношения между объектами не показаны)

Ответы на контрольные вопросы

Контрольные вопросы 3.1

КВ1. Диаграмма прецедентов использования, диаграмма последовательностей, диаграмма коммуникации и диаграмма деятельности.

КВ2. Нет, спецификация прецедента использования содержит диаграмму и полное текстуальное описание прецедента использования с указанным поведением.

Контрольные вопросы 3.2

КВ1. Да. Моделирование деятельности можно выполнять на разных уровнях абстракции, включая внутренние спецификации конкретных этапов прецедента.

КВ2. Решения, разделения, соединения, слияния и узлы объектов.

Контрольные вопросы 3.3

КВ1. Да, это синонимы.

КВ2. Да. Агрегация представляет собой разновидность ассоциации.

Контрольные вопросы 3.4

КВ1. На диаграмме последовательности.

КВ2. Нет. Сообщение — это вызов метода.

Контрольные вопросы 3.5

КВ1. Да. На состояние объекта может влиять любой атрибут, включая атрибуты, означающие другие классы (например, ассоциации).

КВ2. Сторожевое условие оценивается в точке обработки события, чтобы определить, следует ли выполнять переход.

Контрольные вопросы 3.6

КВ1. В языке UML 2.0 подсистема моделируется в качестве стереотипа компонента. Однако в более ранних версиях языка UML она моделировалась как стереотип пакета.

КВ2. Файлы модели, исходные файлы, сценарии и бинарные выполняемые файлы, таблицы базы данных, комплектующие узлы программного обеспечения, текстовые процессоры или сообщения электронной почты.

Ответы к многовариантным тестам

МТ1. г

МТ2. б

МТ3. г (разделение и воссоединение)

МТ4. в

МТ5. г

МТ6. а

МТ7. в

МТ8. б

Ответы на вопросы с нечетными номерами

В1

Статическая модель описывает статичную структуру системы — классы, их внутреннюю структуру, а также отношения между ними. Основным способом визуализации статической модели является диаграмма классов. *Модель функционирования* описывает действия объектов в системе, связанные с выполнением бизнес-функций, — взаимодействие, операции и алгоритмы над данными. Моделирование поведения включает в себя диаграммы прецедентов использования, диаграммы последовательностей, диаграммы коммуникации и диаграммы деятельности. *Модель конечного автомата* описывает динамические изменения состояний объектов на протяжении их линии жизни. Основным способом визуализации конечных автоматов является диаграмма конечного автомата.

Три модели описывают взаимодополняющие ракурсы системы, но часто состоят из одних и тех же элементов моделирования. Статический ракурс демонстрирует виды элементов, существующих в системе. Поведенческий ракурс гарантирует, что элементы способны обеспечить требуемые функциональные свойства системы. Хорошая статическая модель должна допускать добавление новых или расширение старых функциональных свойств системы. Ракурс конечных автоматов определяет среду, в которой происходит эволюция объектов, и устанавливает ограничения на состояние объектов в поведенческой и статической моделях.

В3

В ранних версиях языка UML *диаграммы деятельности* рассматривались как разновидность *конечных автоматов* и даже могли использоваться вместо диаграмм конечных автоматов. С этой целью диаграммы деятельности были дополне-

ны обозначениями конечных автоматов и проведено отличие между *состояниями объектов* (в конечных автоматах) и *состояниями деятельности* (которые не моделируются конечными автоматами). В современной версии языка UML диаграммы деятельности определяют поведение, используя лишь модель потоков управления и данных, напоминающие сети Петри (Petri) (Ghezzi et. al, 2003).

В отличие от большинства методов моделирования в языке UML роль и место диаграмм деятельности в процессе разработки системы четко не очерчены. Семантика диаграмм деятельности позволяет использовать их на разных уровнях абстракции и на разных этапах жизненного цикла проекта. Их можно использовать на ранних этапах анализа для описания общего функционирования системы. Кроме того, они позволяют моделировать поведение прецедента использования или любой его части. Их можно также использовать при проектировании вместо принципиальной схемы конкретного метода и даже отдельных алгоритмов, содержащихся в методе.

Диаграммы деятельности можно рассматривать как "мостики" в модели системы. Они используются для графической визуализации потоков сообщений, данных и процессов в других элементах моделирования.

B5

Объекты взаимодействуют друг с другом, обмениваясь сообщениями. Как правило, сообщение от одного объекта вызывает метод (операцию) другого объекта. Сигнатура сообщения содержит список *фактических аргументов*. Вызываемый метод включает в эту сигнатуру соответствующий список *формальных аргументов*. В языке UML фактические аргументы сообщения называются *аргументами*, а формальные аргументы метода — *параметрами*.

B7

Пакет — это группа элементов моделирования. Он не имеет никакого отношения к способам их использования клиентами. Для того чтобы учесть способ использования элементов моделирования, в языке UML предусмотрена стереотипизированная версия компонента, называемая *подсистемой* (до появления версии UML 2.0 подсистема считалась стереотипизированной версией пакета).

Визуально подсистема является компонентом, стереотипизированным с помощью ключевого слова «`subsystem`». Семантически подсистема скрывает свои элементы моделирования и раскрывает клиентам только свои открытые сервисы. Клиенты запрашивают сервисы подсистемы с помощью *предлагаемых интерфейсов*. В реальности подсистема — это концепция, и она не может воплощаться в виде экземпляров, т.е. не может иметь конкретной реализации. Сервисы подсистемы реализуются (воплощаются) в виде классов, принадлежащих подсистеме.

Моделирование доступа к подсистеме через предлагаемые интерфейсы означает, что реализация всех классов в этой подсистеме скрыто за ними. Это дает моде-

лям реализации важное преимущество. Если интерфейс не определен в подсистеме, то другие компоненты становятся зависимыми от реализации классов в подсистеме. Такие зависимости негативно влияют на адаптивность системы (т.е. снижает ее понятность, эксплуатационную надежность и масштабируемость).

Объяснение упражнений с нечетными номерами

У1

Модифицированное взаимодействие объектов показано на рис. 3.23. Два потока имеют номера 1 и 2. Поток поставок имеет два зависимых сообщения, имеющих номера 1.1. и 1.2.

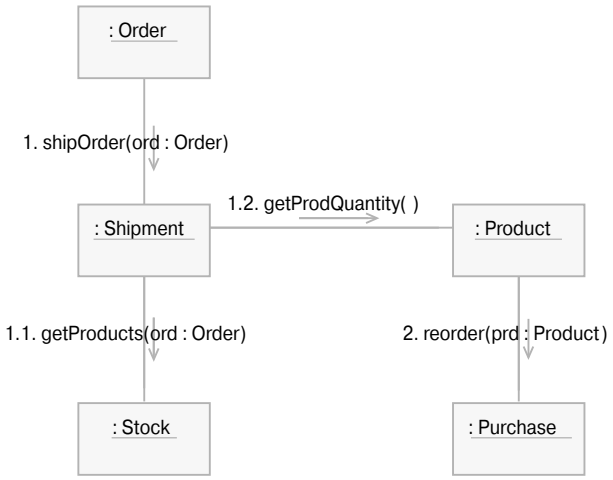


Рис. 3.23. Взаимодействие объектов при поставке и пополнении запасов

Помимо недостаточно полного объяснения процедуры создания объектов классов `Shipment` и `Purchase`, на этом рисунке остаются и другие неразъясненные детали. Например, модель ничего не говорит о том, как метод `getProdQuantity()` пересчитывает объекты класса `Product` и что именно делают эти объекты, посылая сообщения `reorder()`.

У3

На модифицированной диаграмме классов (рис. 3.24) показаны два новых класса (`Lecture` и `Tutorial`), являющихся подклассами класса `CourseOffering`. Подклассы имеют свои собственные ассоциации с классом `Teacher`.

Поскольку объект класса `Teacher` может как читать лекции, так и вести семинары, роли имеют разные имена — `lect_in_charge_of` и `tut_in_charge_of`. Это важный факт, поскольку для учета этих ролей в реализованных классах необходимо предусмотреть два атрибута, именами которых являются имена ролей.

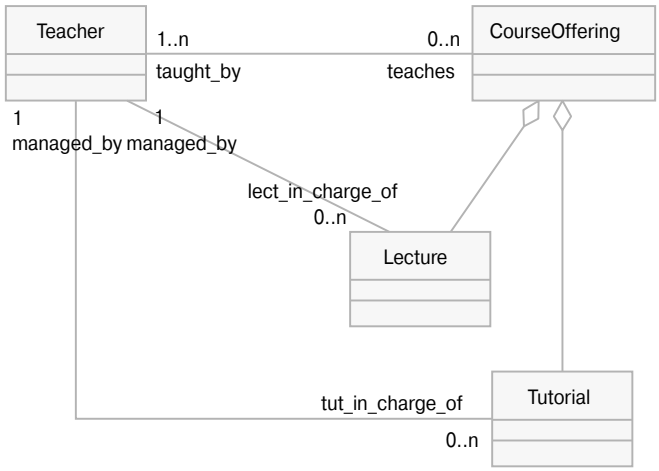


Рис. 3.24. Модифицированная диаграмма классов, демонстрирующая замену ассоциированного класса

У5

Для простоты изложения на рис. 3.25 показаны типы возвращаемых значений, хотя на диаграммах коммуникации они обычно не указываются. Сообщения `iterateChapters` и `iterateParagraphs` представляют собой высокоуровневые описания требований, предусматривающих, что объекты класса `Book` должны просматривать много глав, а объекты класса `Chapter` должны просматривать много разделов. Подробности этой циклической операции пропущены. Поскольку сообщения `iterateChapters` и `iterateParagraphs` представляют собой лишь абстракции, а не сообщения методам, после их имен не показаны параметры, образующие список ее аргументов.

Объект класса `Finder` вызывает операцию `search()` и применяет ее к объекту класса `Book`. В результате он возвращает список номеров глав и разделов, содержащих строку, переданную в виде аргумента `what` (что), как объект класса `List` (Список).

Однако текст книги содержится в объектах класса `Paragraph`. Соответственно объекты класса `Chapter` просматривают объекты класса `Paragraph` и применяют к каждому из них операцию `search()`. Эта операция возвращает значение истина/ложь, а объект класса `Chapter` формирует массив номеров разделов, содержащих искомую строку. Этот массив возвращается объекту класса `Book`, и на основе этого массива объект класса `Book` создает список номеров глав и разделов, а затем возвращает их объекту класса `Finder`.

У7

Возможны разные (и произвольные) решения. Одно из них приведено на рис. 3.26. Обратите внимание на то, что это решение невозможно реализовать на языке Java, так как он не поддерживает множественное наследование.

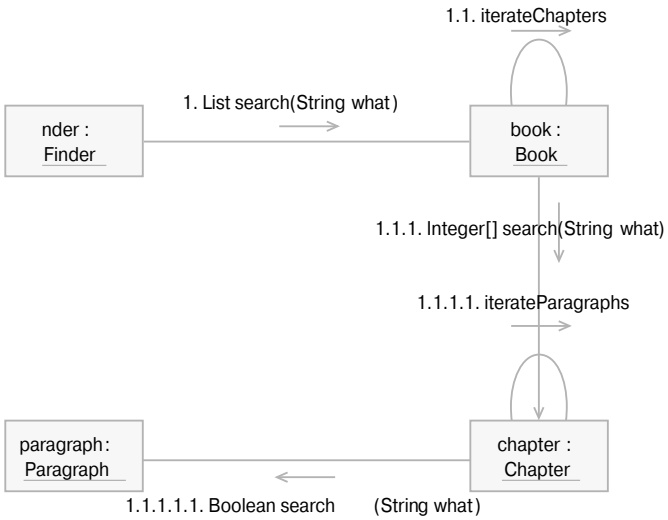


Рис. 3.25. Диаграмма взаимодействия объектов

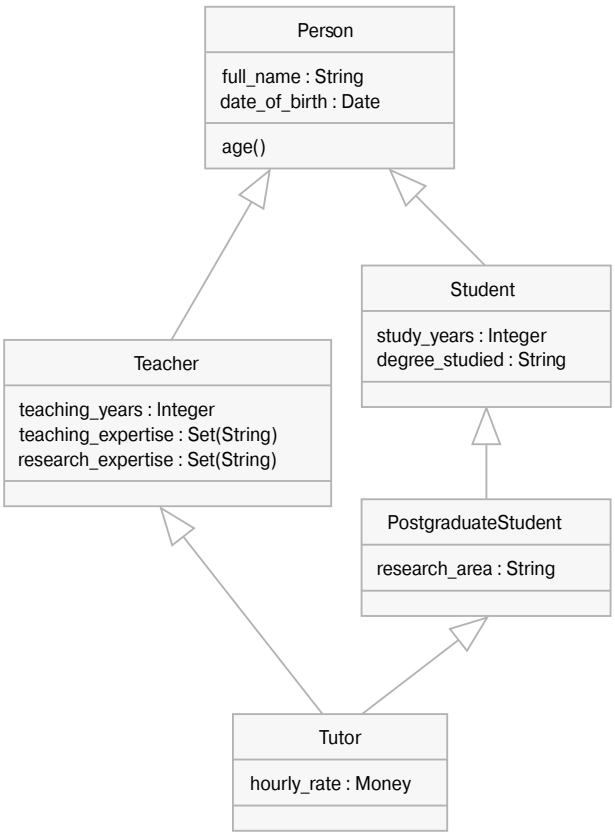


Рис. 3.26. Множественное наследование

Атрибуты `teaching_expertise` и `research_expertise` в классе `Teacher` называются *параметризованными*. Компоненты `Set` (Множество) и `String` (Строка) являются классами. Класс `String` — это параметр класса `Set`. Значениями этих двух атрибутов являются множества строк.

Объяснение упражнений с нечетными номерами. Магазин видеокассет

MB1

Расширенная диаграмма прецедентов использования показана на рис. 3.27. Прецедент `Прокат видеокассеты` дополнен прецедентами `Распечатка квитанции`, `Проверка возраста` и `Принятие платежа`. Между прецедентами использования `Сканирование членской карточки` и `Сканирование видеокассеты` нет прямых отношений. Модель предполагает, что прецедент `Прокат видеокассеты` имеет информацию о результатах сканирования в начале транзакции.

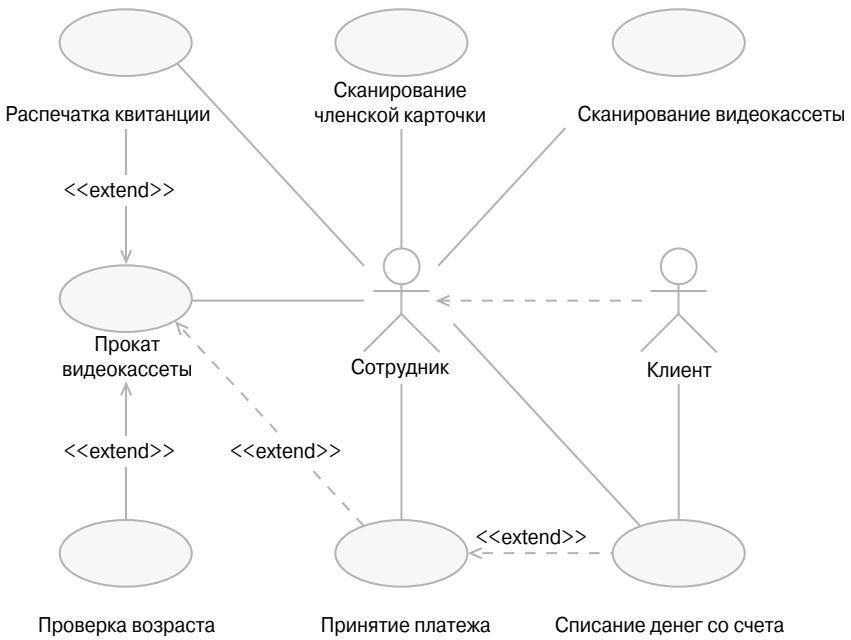


Рис. 3.27. Расширенная диаграмма прецедентов использования системы управления магазином видеокассет

MB3

На рис. 3.28 показано, что введение класса `Sale` (Продажа) порождает новые ассоциации, но с каждой из них связан определенный трюк. В новой модели необходима ассоциация между классами `Customer` и `Payment`. Платеж может охватывать несколько заказов, и возможно (по крайней мере не запрещено), что эти заказы относятся к разным клиентам. Следовательно, без ассоциации `Payment–Customer` невозможно идентифицировать клиента по платежу.

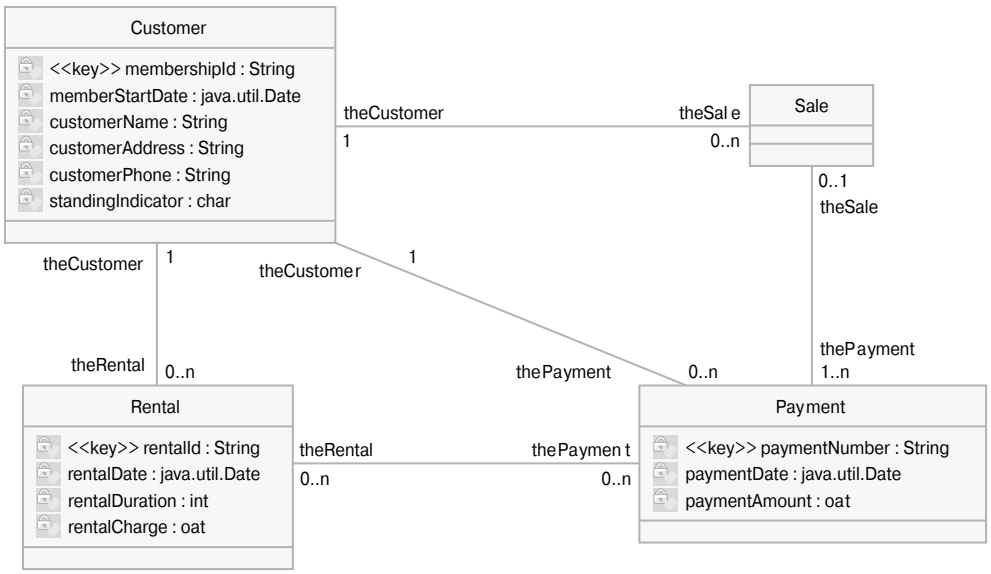


Рис. 3.28. Модифицированная и расширенная диаграмма классов

MB5

На рис. 3.29 показана модель конечного автомата для класса `Video`. Обратите внимание на использование события со сторожевым условием при переходе в состояние `Повреждено`, а также на сторожевое условие, сопровождающее действие при переходе в состояние `На продажу`. Все остальные переходы сопровождаются только событиями.

В модели не показано ни начальное, ни конечное состояние. Эти состояния не важны для наших целей.

На рис. 3.30 показана альтернативная модель. Класс `Video` описывает носитель, содержащий фильм или другой развлекательный материал. Когда видеокассета приобретается и поставляется в `Магазин видеокассет`, она переходит в состояние `Доступно для проката или продажи`. После события “Разместить на складе” видеокассета переходит в состояние `На складе`. Это абстрактное состояние.

В реальности видеокассета может находиться в одном из трех состояний — `Напрокат`, `На продажу` или `Повреждено`. Внутренние переходы между этими тремя состояниями на диаграмме не представлены (чтобы не загромождать рисунок).

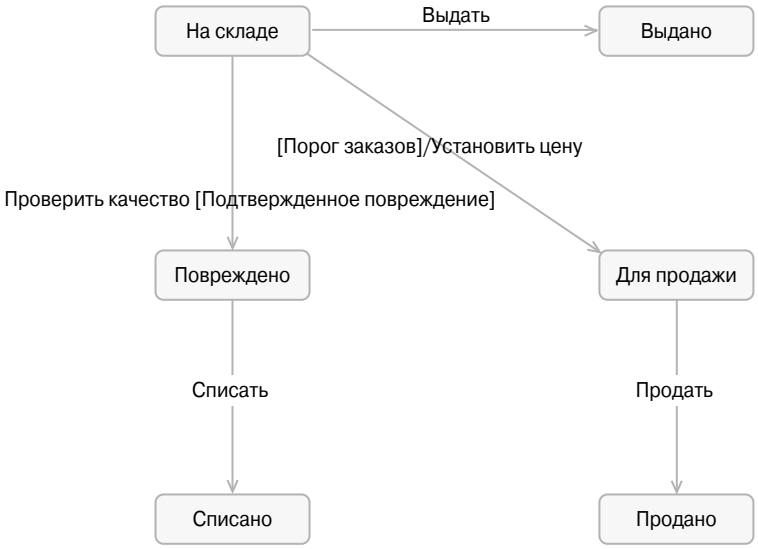


Рис. 3.29. Диаграмма конечного автомата для класса `Video`

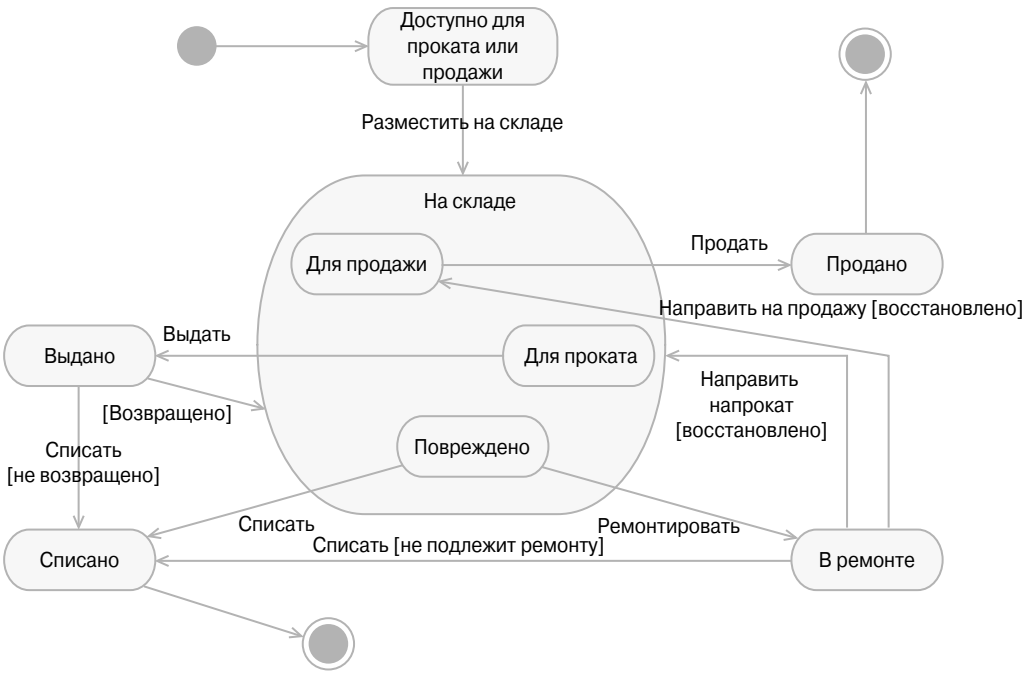


Рис. 3.30. Альтернативная диаграмма конечного автомата для класса `Video`

Видеокассета, находящаяся в состоянии Выдано, может быть возвращена или нет. Если выполняется условие [Возвращено], то происходит переход обратно в состояние На складе. В противном случае происходит событие "Списать", после чего видеокассета переходит в состояние Списано. В это состояние можно перейти также из состояния Повреждено или В ремонте.

Из состояния В ремонте есть три возможных перехода. Если видеокассета находится в состоянии [Восстановлено], то она может быть доступна Для проката или Для продажи. В последнем случае видеокассета может перейти в состояние Продано.

ГЛАВА 4

Спецификация требований

Цели

4.1. Архитектурные прерогативы
4.2. Спецификация состояний
4.3. Спецификация поведения
4.4. Спецификации изменения состояний
Резюме
Ключевые термины
Многовариантные тесты
Вопросы
Упражнения. Магазин видеокассет
Упражнения. Управление взаимоотношениями с заказчиками
Упражнения. Зачисление в университет
Ответы на контрольные вопросы
Ответы к многовариантным тестам
Ответы на вопросы с нечетными номерами
Объяснение упражнений. Зачисление в университет

Цели

Требования необходимо уточнить с помощью графических и других формальных моделей. Для того чтобы полностью описать систему, нужно много моделей. Язык UML предлагает системному аналитику множество методов моделирования, позволяющих решить эту задачу. Этот процесс носит итеративный и поступательный характер. В достижении успеха моделирования существенную роль играют CASE-инструменты. Спецификации требований порождают три категории моделей: модели состояний, модели поведения и модели изменения состояний.

Входной информацией для процесса спецификации требований является неформальное описание пожеланий заказчика, а выходной — модели спецификации. Эти модели (см. главу 3) обеспечивают формальное определение разных аспектов (ракурсов) системы. Этап спецификации требований завершается созданием расширенного ("уточненного") *технического задания* (см. раздел 2.6). Новый документ часто называют *спецификацией* (specification document). Структура исходного документа остается неизменной, но его содержание значительно расширяется. В результате на этапе проектирования и реализации спецификация заменяет собой техническое задание (на практике этот расширенный документ часто по-прежнему называется *техническим заданием*).

Прочитав эту главу, читатели будут

- осознавать важность раннего архитектурного проектирования для успешного создания адаптивных систем;
- понимать большинство фундаментальных принципов, лежащих в основе архитектурных структур (метамоделей);
- знать архитектурную среду PCBMER, которая впоследствии будет использована для изучения прецедентов и примеров;
- владеть практическими навыками моделирования классов, ассоциаций, а также других отношений и интерфейсов;
- владеть практическими навыками моделирования прецедентов использования, деятельности, взаимодействий и операций;
- уметь моделировать изменения состояния объектов.

4.1. Архитектурные прерогативы

Спецификация требований (requirements specification) связана со *строгим* моделированием требований заказчика, определенных на этапе их выявления. Основной упор при этом делается на желательных услугах, предоставляемых системой (функциональных требованиях). Системные ограничения (нефункциональные требования) на этапе спецификации часто не уточняются, а лишь используются в качестве ориентира для оценки результатов моделирования. Ориентиры и оценки моделирования выражаются в форме архитектурных прерогатив.

Архитектура программного обеспечения (software architecture) определяет структуру и организацию взаимодействующих компонентов программного обеспечения и **подсистем** (subsystems), образующих систему. "Архитектура программного обеспечения учитывает и фиксирует намерения проектировщиков, касающиеся структуры и функционирования системы, тем самым предотвращая ее распад с течением времени. Она играет ключевую роль в достижении интеллектуального контроля над чрезмерной сложностью систем" (Kruchten et al., 2006). Продуманная архитектура программного обеспечения является необходимым и наиболее важным условием для обеспечения *адаптивности* (легкости сопровождения) системы, т.е. ее понятности, легкости эксплуатации и масштабируемости (см. раздел 2.2.1.2).

Следовательно, прежде чем начинать детализацию системных спецификаций, коллектив разработчиков должен согласовать архитектурные шаблоны и принципы, которыми они будут руководствоваться (Maziashek and Liong, 2005). Это чрезвычайно важно. Без ясной архитектурной концепции системы этап анализа неизбежно завершится разработкой спецификации нежизнеспособной системы (если она вообще будет работать!).

Основной целью моделирования программного обеспечения должна быть минимизация **зависимостей** между компонентами. Для этого разработчики не должны допускать нечеткости в определении связей между объектами, иначе система превратится в мешанину непонятно как взаимодействующих компонентов. Сложность такой системы по мере добавления новых объектов растет по экспоненциальному закону. Такое положение дел терпеть нельзя, и разработку следует немедленно прекратить, пока она не принесла беды. Уже на самых ранних этапах разработки следует создать четкую архитектурную модель, представленную в виде иерархии компонентов, подсистем и ограничений, наложенных на взаимодействие объектов.

4.1.1. Модель–представление–контроллер

Как всегда в проектировании программного обеспечения, существует множество разных вариантов архитектурных проектов системы. Несмотря на то что вполне можно представить себе проект, возникающий снизу вверх, начиная

с индивидуальных проектных решений, обычно архитекторы применяют подход сверху вниз, выводя конкретный проект из заранее определенной **архитектурной структуры** (**архитектурной метамодели**) (Maziaszek and Liong, 2005).

Большинство современных архитектурных структур и связанных с ними шаблонов описывается в рамках концепции *модель–представление–контроллер* (MVC), разработанной для языка Smalltalk-80 (Krasner and Pope, 1988). В языке Smalltalk-80 концепция MVC требует от программиста разделять классы приложения на три группы классов, производных от абстрактных классов Model, View и Controller и специализирующих их.

Объекты модели (model objects) представляют собой объекты данных — бизнес-сущностей и бизнес-ролей, относящихся к предметной области. Сведения об изменениях объектов модели сообщаются объектам представления и контроллера с помощью обработки событий. Для этого используется метод "издатель–подписчик". Модель является издателем и, следовательно, ничего не знает о своих представлениях и контроллерах. Представление и контроллер подписываются на модель, но, кроме этого, могут инициировать изменения модельных объектов. Для того чтобы облегчить выполнение этого задания, модель предоставляет необходимые интерфейсы, инкапсулирующие бизнес-данные и поведение.

Объекты представления — это объекты пользовательского интерфейса, которые задают состояние модели в формате, требуемом пользователем и поддерживаемом графическим пользовательским интерфейсом. Они выделяются из объектов модели. Объекты представления подписываются на модель, чтобы получать уведомления о ее изменениях и иметь возможность обновлять дисплей. Модели представления могут содержать компоненты (subviews), отображающие на дисплее их части. Как правило, каждому объекту представления ставится в соответствие объект контроллера.

Объекты контроллера (controller objects) символизируют события мыши и клавиатуры. Они реагируют на запросы, посланные представлением и являющиеся результатом взаимодействия пользователя с системой. Объекты контроллера распознают нажатия клавиш, щелчки мыши и другие события и преобразуют их в действия над объектами модели. Они служат посредниками между объектами представления и модели. Отделение входных данных пользователя от их визуального представления позволяет изменять реакцию системы на действия пользователя без изменения графического интерфейса, и наоборот — изменять графический интерфейс без изменения поведения системы.

На рис. 4.1 показана точка зрения пользователя на взаимосвязи между объектами MVC. Линии символизируют связи между объектами. События графического пользовательского интерфейса перехватываются объектами представления и передаются объектам контроллера для интерпретации и дальнейших действий. Смешение поведения представления и контроллера в одном объекте рассматривается как весьма нежелательное.

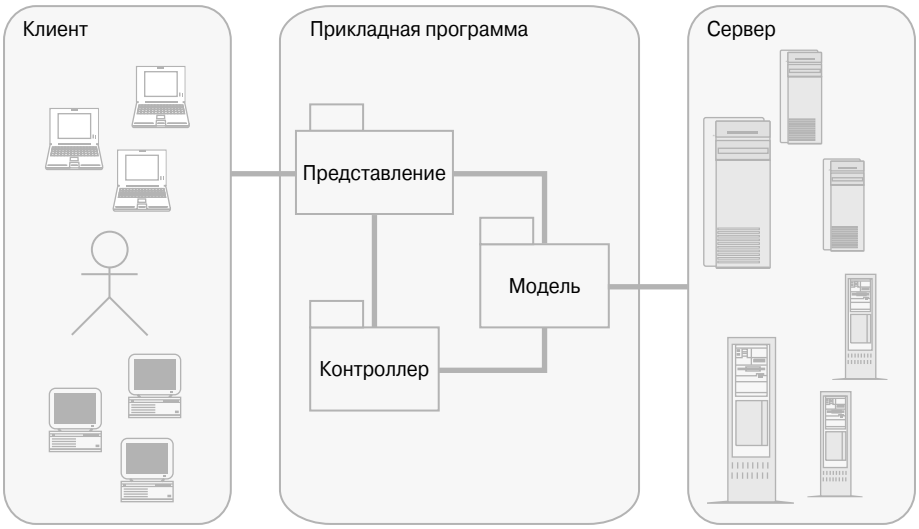


Рис. 4.1. Архитектура MVC

4.1.2. Архитектура Core J2EE

Концепция MVC лежит в основе практически всех современных архитектур, расширяющих ее до размера промышленных систем и систем для электронной коммерции. Одной из таких архитектур является Core J2EE (Alur et al., 2003; Roy-Faderman et al., 2004). Как показано на рис. 4.2, модель J2EE состоит из ярусов — три из них охватывают компоненты прикладных программ (презентацию, бизнес и интеграцию), а два остальных являются внешними по отношению к приложению (клиент и информационная система предприятия (Enterprise Information System — EIS)).



Рис. 4.2. Архитектура Core J2EE

Пользователь взаимодействует с системой посредством *яруса клиента* (client tier), который может быть программируемым клиентом (например, клиентом Java Swing или аплетом), клиентом HTML Web-браузера, мобильным клиентом WML или даже клиентом Web-сервиса, основанного на языке XML. Процесс, поддерживающий интерфейс пользователя, может быть запущен на клиентской машине (программируемый клиент), или на Web-сервере, или на сервере приложения (например, приложения Java JSP/сервлет).

Ярусом информационной системы предприятия (EIS tier) может быть любая система, поставляющая информацию, например, база данных предприятия, внешняя база данных в системе электронной коммерции или внешний сервис SOA. При этом данные могут быть распределены по многим серверам.

Пользователь получает доступ к приложению через *ярус представления* (известный также как *Web*-ярус или *ярус представления со стороны сервера*). В Web-приложении этот ярус состоит из пользовательского интерфейса или процессов, запускаемых на Web-сервере или сервере приложения. В рамках архитектуры MVC ярус представления состоит из компонентов представления и контроллера.

Ярус бизнеса (business tier) содержит части логики приложения, которые еще не воплощены в виде компонентов контроллера в ярусе представления. Он отвечает за оценку и соблюдение бизнес-правил и транзакций. Кроме того, он управляет бизнес-объектами, загруженными ранее с яруса информационной системы предприятия в кэш-память приложения.

Ярус интеграции (integration tier) несет исключительную ответственность за установление и поддержание связей с источниками данных. Этот ярус знает, как устанавливать связь с базами данных посредством интерфейса **Java Database Connectivity** (JDBC) или интегрироваться с внешними системами с помощью службы сообщений **Java Messaging Service** (JMS).

Архитектура Core J2EE является обобщающей и поясняющей, а не управляющей. Она вводит "разделение ответственности" между тремя ярусами прикладных программ. Кроме того, она требует, чтобы **компоненты** презентации могли общаться с компонентами интеграции только через ярус бизнеса, и наоборот. Однако она не сохраняет строгий иерархический порядок, поскольку разрешает двунаправленные связи (вызовы методов), а значит, допускает циклические вызовы.

Тем не менее существует множество методов, разработанных для платформы J2EE, носящих регуляторный характер и предназначенных для решения вопросов, связанных со сложностью разработки и интеграции информационных систем предприятий и электронной коммерции. Сначала появились технологии Jakarta Struts, позволяющие реализовать шаблоны MVC. Затем их расширили технологии Spring Framework и серверы приложения (например, JBoss или Websphere Application Server). Впоследствии произошла интеграция электронной коммерции с реализациями службы сообщений JMS в серверах приложения.

4.1.3. Презентация–контроллер–компонент–посредник–сущность–ресурс

В предыдущем издании книги, а также в работе (Maciaszek and Liong, 2005) была описана архитектурная модель *PCMEF*. Эта аббревиатура обозначала пять иерархических уровней классов: представление–управление–посредник–сущность–базис (presentation–control–mediator–entity–foundation). Эти слои можно моделировать как подсистемы или пакеты. Недавно появилась модель **PCBMER**, расширяющая модель PCMEF за счет дополнительного иерархического уровня: представление–контроллер–компонент–посредник–сущность–ресурс (presentation–controller–bean–mediatir–entity–resource) (Maciaszek, 2006). Архитектура PCBMER соответствует современным тенденциям архитектурного проектирования. Например, она хорошо согласуется с моделью Core J2EE, описанной в предыдущем разделе.

Архитектура *Core PCBMER* продемонстрирована на рис. 4.3. Эта модель позаимствовала имена внешних ярусов у модели Core J2EE (ярусы клиента и информационной системы предприятия). Ярусы представлены в виде узлов UML. Пунктирные стрелки означают *отношения зависимости*. Таким образом, например, представление зависит от контроллера и от **компонента**, а контроллер зависит от компонента. Иерархия PCBMER не является строго линейной, и ярус более высокого уровня может иметь несколько смежных ярусов более низкого уровня (эти ярусы могут быть *внутренними листами*, т.е. не иметь смежных нижних ярусов).

Односторонняя зависимость не означает, что объекты не могут передавать сообщения в противоположном направлении. На самом деле они могут это делать. Однако такая связь должна быть обеспечена с помощью метода, минимизирующего зависимости, например посредством интерфейсов или протоколов "издатель/подписчик".

Архитектура PCBMER, адаптированная к разработке, требует, чтобы проектировщики относили каждый класс в системе к одной из шести подсистем. Это автоматически улучшает проект, поскольку каждый класс вносит свой вклад в достижение цели, поставленной перед подсистемой. Этот класс становится более связным и целенаправленным.

Соединение классов, обеспечивающее предоставление услуг клиентами системы, разрешается только по линиям зависимости. Для этого могут понадобиться более длинные пути коммуникаций, чем нужно, но благодаря этому удается исключить возможность появления сетей взаимодействующих объектов. Архитектура PCBMER обеспечивает оптимальное сочетание **связности** и связанности объектов, удовлетворяющих всем требованиями системы.

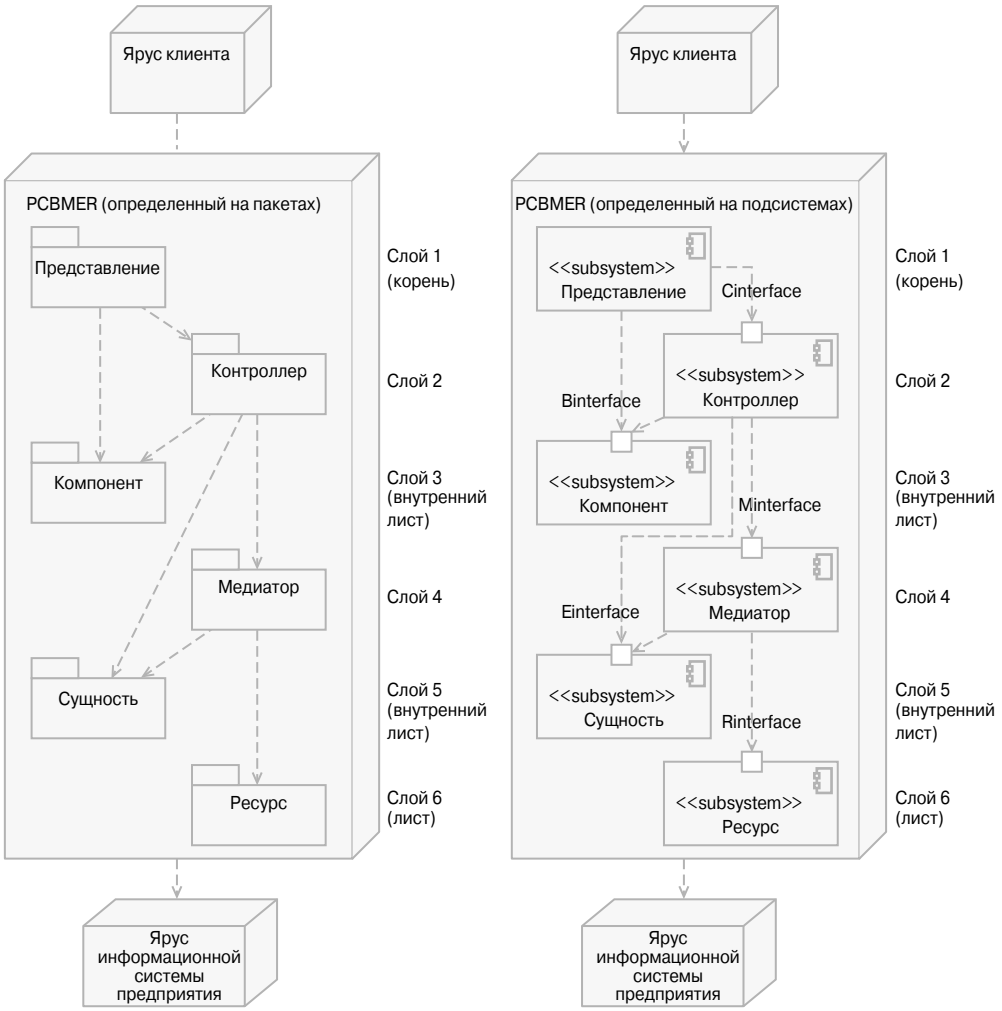


Рис. 4.3. Архитектура Core PCBMER

4.1.3.1. Уровни архитектуры PCBMER

На рис. 4.3 показаны два варианта представления модели *Core PCBMER* — на основе пакетов и подсистем UML (см. раздел 3.6.1). Напомним, что *пакет* группирует элементы моделирования (классы и т.п., включая другие пакеты) под одним именем. Услуги, предоставляемые пакетами, являются результатом услуг, предоставленных его элементами моделирования. С другой стороны, понятие *подсистемы* используется для моделирования крупномасштабных компонентов, т.е. подсистема — это разновидность компонента. *Компонент* (см. раздел 3.6.2) также группирует элементы моделирования (классы и т.п., включая другие компоненты) под одним именем, причем его реализации в среде могут быть взаимозаменяемыми. Услуги,

обеспечиваемые компонентами, полностью инкапсулированы и представляются в виде набора **портов**, определяющих **предлагаемые и требуемые интерфейсы**.

К *уровню компонентов* (bean layer) относятся классы данных и объекты значений, предназначенные для отображения на экране дисплея. Если компоненты не вводятся пользователем, то они создаются из объектов-сущностей (уровень сущностей). Модель Core PCBMER не регламентирует доступ к объектам компонентов — посредством передачи сообщений или обработки сообщений, — поскольку уровень компонентов не зависит от других уровней.

Уровень представления (presentation layer) описывает экран дисплея и элементы пользовательского интерфейса, на которых отражаются компоненты. Этот уровень обеспечивает согласованность представления при изменениях компонентов, поэтому он зависит от уровня компонентов. Эта зависимость может быть реализована двумя способами — прямым вызовом методов (передачей сообщений) с помощью *модели вытягивания* (pull model) или обработки событий, после которой происходит передача сообщений с помощью *модели проталкивания* (push model), или *модели вытягивания–проталкивания* (push–and–pull model). (Иногда эти модели называют активной и пассивной соответственно. — *Примеч. ред.*)

Уровень контроллера (controller layer) отражает логику приложения. Объекты контроллера отвечают на запросы пользовательского интерфейса, исходящие от представления и являющиеся результатом взаимодействия пользователя с системой. В программируемом клиенте графического пользовательского интерфейса запросы могут поступать от меню или кнопок. В клиенте Web-браузера запросы пользовательского интерфейса появляются в виде запросов "HTTP GET and POST".

Уровень сущностей (entity layer) соответствует *контроллеру* и *медиатору*. Он содержит классы, представляющие "бизнес-объекты". Они хранят (в памяти) объекты, извлеченные из базы данных или созданные с целью хранения в базе данных. Многие классы сущностей представляют собой контейнерные классы.

Уровень посредника (mediator layer) устанавливает канал связи между классами сущностей и классами ресурсов. Этот уровень управляет деловыми транзакциями, устанавливает бизнес-правила, инициирует бизнес-объекты на уровне сущностей и управляет кэш-памятью приложения. С архитектурной точки зрения посредник преследует две цели. Во-первых, он изолирует уровни сущностей и ресурсов, так чтобы изменения на этих уровнях можно было выполнять независимо друг от друга. Во-вторых, он обеспечивает связь между уровнями контроллера и сущностей/ресурсов, когда контроллер запрашивает данные, не зная, где хранятся данные — в памяти или в базе данных.

Уровень ресурсов (resource layer) обеспечивает все связи между внешними источниками персистентными данными (базами данных, Web-службами и т.д.). На этом уровне устанавливаются связи между базами данных и SOA-серверами, а также конструируются запросы на персистентные данные и инициируются транзакции с базами данных.

4.1.3.2. Принципы PCBMER

Архитектура Core PCBMER имеет много явных преимуществ. Одним из основных является *разделение ответственности* (separation of concerns) между уровнями, позволяющее их либо совершенно независимую, либо предсказуемую и управляемую модификацию. Например, уровень представления, обеспечивающий пользовательский интерфейс в Java-приложении, можно заменить интерфейсом мобильных телефонов, продолжая использовать существующую реализацию уровней контроллера и компонентов. Иначе говоря, одна и та же пара уровней контроллера и компонентов может поддерживать несколько пользовательских интерфейсов одновременно.

Второе важное преимущество — *исключение циклов* в отношениях зависимости, порождающее шестиуровневую иерархию с отношениями зависимости, направленными исключительно сверху вниз. Циклы могут привести к вырождению иерархии в сетевую структуру. По этой причине циклы исключены как между уровнями PCBMER, так и внутри каждого уровня.

Третье преимущество заключается в том, что архитектура PCBMER обеспечивает высокую степень *устойчивости*. Более высокие уровни зависят от уровней, расположенных ниже. Следовательно, поскольку нижние уровни устойчивы (не подвергаются значительным изменениям, в частности, в отношении интерфейсов), то изменения, происходящие на верхних уровнях, не приводят к серьезным последствиям. Напомним также, что более нижние уровни можно снабдить дополнительными функциональными свойствами (вместо изменения существующих функциональных свойств), и это не должно влиять на существующие характеристики более высоких уровней.

Архитектура PCBMER устанавливает и другие свойства и ограничения, не показанные на рис. 4.3. Основные *архитектурные принципы* PCBMER (Maciaszek, 2006; Maciaszek and Liong, 2005) приведены ниже.

- *Принцип зависимости сверху вниз* (downward dependency principle — DDP). Принцип DDP утверждает, что основная структура зависимостей должна быть направлена сверху вниз. Объекты, расположенные на верхних уровнях, зависят от объектов нижнего уровня. Следовательно, нижние уровни *устойчивее* верхних. Интерфейсы, абстрактные классы, доминантные классы и аналогичные устройства должны инкапсулировать устойчивые уровни и допускать расширение по мере необходимости.
- *Принцип уведомления снизу вверх* (upward notification principle — UNP). Принцип UNP декларирует слабую связанность связей между уровнями, направленных сверху вниз. Этого можно достичь с помощью *асинхронной коммуникации* (asynchronous communication), основанной на *обработке событий* (event processing). Объекты, расположенные на верхних уровнях, действуют как подписчики (*наблюдатели*), желающие знать об изменениях состояния

объектов на нижних уровнях. Когда объект (*издатель*), находящийся на нижнем уровне, изменяет свое состояние, он посылает уведомление своим подписчикам. В ответ подписчики могут связаться с издателем (в направлении сверху вниз), чтобы синхронизировать свое состояние в состояние издателя.

- *Принцип связей между соседями* (neighbor communication principle — NCP).
  Принцип NCP требует, чтобы уровень взаимодействовал непосредственно с соседним уровнем в соответствии с прямыми зависимостями между уровнями. Этот принцип гарантирует, что система не выродится в сеть взаимодействующих объектов. Для соблюдения этого принципа передача сообщений между отдаленными объектами (т.е. не являющимися соседями) основывается на *делегировании* (delegation) или *пересылке* (forwarding). (Второй вариант означает передачу ссылки самому себе, а первый — нет.) В более сложных сценариях для группировки интерфейсов, обеспечивающих взаимодействие отдаленных объектов, можно использовать *осведомленный пакет* (acquaintance package).
- *Принцип явной ассоциации* (explicit association principle — EAP).
  Принцип EAP явно регламентирует обмен разрешенными сообщениями между классами. Если проект соответствует парадигме PCBMER, то зависимости сверху вниз между классами (по принципу DDP) подтверждаются соответствующими ассоциациями. Ассоциации, возникающие в результате применения принципа DDP, являются однонаправленными (в противном случае возникли бы циклические зависимости). Однако следует помнить, что не *все* ассоциации между классами возникают вследствие передачи сообщений. Например, для реализации ссылочной интеграции между классами на уровне сущностей могут понадобиться двунаправленные ассоциации.
- *Принцип исключения циклов* (cycle elimination principle — CEP).
  Принцип CEP гарантирует устранение *циклических зависимостей* между уровнями и классами (Maciaszek and Liong, 2005). Это объясняется тем, что такие зависимости нарушают принцип разделения ответственности и усложняют повторное использование компонентов. Циклы можно устранить, поместив классы, нарушающие принцип CEP, в новый уровень или пакет, созданный специально для этой цели, или введя новые линии связи в цикле, чтобы обеспечить коммуникацию с помощью интерфейса.
- *Принцип именования классов* (class naming principle — CNP).
  Принцип CNP позволяет распознать по *имени класса* уровень/пакет, которому принадлежит этот класс. Перед именем каждого класса в архитектуре PCBMER приписывается первая буква имени уровня. Например, класс `EV-ideo` — это класс из уровня сущностей (entity layer). Тот же принцип применяется к интерфейсам. Перед именем каждого интерфейса указываются две прописные буквы — первая буква, `I`, означает, что это — имя интер-

фейса (interface)), а вторая буква идентифицирует уровень. Таким образом, `ICVideo` — это интерфейс на уровне контроллера.

- *Принцип пакета связей* (acquaintance package principle — APP).
  Принцип APP является следствием принципа NCP. Пакет связей состоит из *интерфейсов*, которые объект передает вместо конкретных объектов в качестве аргументов при вызове методов. Эти интерфейсы можно реализовать на любом уровне архитектуры PCBMER. Принцип APP позволяет эффективно организовать связь между отдаленными уровнями, централизовав управление зависимостями в отдельном пакете связей.

Контрольные вопросы 4.1

КВ1. Назовите необходимое и самое важное условие, обеспечивающее адаптивность (легкость сопровождения) программного обеспечения.

КВ2. Какие объекты в архитектуре MVC представляют события, связанные с мышью и клавиатурой?

КВ3. Какой ярус в архитектуре Core J2EE отвечает за установление и поддержку соединений с источниками данных?

КВ4. Какой уровень в архитектуре PCBMER отвечает за установление и поддержку соединений с источниками данных?

4.2. Спецификация состояний

Состояние (state) объекта определяется значениями его атрибутов и ассоциациями. Например, объект `BankAccount` (Банковский счет) может находиться в состоянии "превышение кредитного лимита", если значение атрибута `balance` (баланс) отрицательно. Поскольку состояния объекта определяются структурами данных, модели структур данных называются *спецификациями состояний* (state specifications).

Спецификация состояний отражает *структурный*, или *статический ракурс системы* (см. раздел 3.3). Здесь основной задачей является определение *классов* проблемной области, а также их *атрибутов* и *отношений* с другими классами. Вначале *операции* классов обычно не рассматриваются. Они выводятся из моделей *спецификации поведения* (behavior specifications).

В типичной ситуации сначала определяются *классы сущностей* (бизнес-объекты). К ним относятся классы, определяющие предметную область и *персистентно* существующие в базе данных. Классы, обслуживающие события, порожденные пользователем (*управляющие классы*), классы, обеспечивающие рендеринг графического интерфейса пользователя (*классы представления*), а также классы,

управляющие данными, предназначенными для этого рендеринга (*классы компонентов*), не устанавливаются до тех пор, пока не станут известны функциональные характеристики системы. Другие категории классов, например *классы ресурсов* и *классы-посредники*, также устанавливаются позднее.

4.2.1. Моделирование классов

Модель классов — это краеугольный камень разработки объектно-ориентированной системы. Именно классы описывают состояние и функционирование системы. К сожалению, классы трудно выявить, а их свойства не всегда очевидны. В одной и той же предметной области два аналитика никогда не создадут одинаковый набор классов и их свойств. Несмотря на то что модели классов могут отличаться, конечный результат и степень удовлетворенности пользователя могут быть в равной мере достаточными (или в равной мере недостаточными).

Моделирование классов — это не детерминированный процесс. Не существует общего рецепта поиска и определения наилучших классов. Этот процесс в значительной мере носит *итеративный* и *поступательный* характер. Успешное проектирование классов зависит от следующих характеристик аналитика.

- Знания в области моделирования классов.
- Понимание проблемной области.
- Опыт в области аналогичных и успешных проектов.
- Способность смотреть вперед и предвидеть последствия решений.
- Готовность к пересмотру модели с целью устранения недостатков и т.д.

Последний момент связан с использованием CASE-инструментов. Широкомасштабное применение CASE-технологии может стать препятствием на пути разработки системы в технологически незрелых организациях (см. раздел 1.1.2.3.2). Однако использование CASE-средств для повышения *личной производительности труда* (personal productivity) всегда оправдано.

4.2.1.1. Выявление классов

Как уже говорилось, два разных аналитика не могут прийти к идентичным моделям классов для одной и той же проблемной области, и даже не могут думать о классах одинаково. Литература изобилует рекомендациями по *выявлению классов* (class discovery). Аналитики могут поначалу даже следовать одному из этих подходов, однако последующие итерации, как правило, обязательно приводят к использованию нешаблонных и даже произвольных механизмов.

Барами (Bahrami, 1999) изучил главные особенности четырех основных подходов к идентификации классов, перечисленных ниже.

- Метод именных групп.
- Метод общих шаблонных классов.

- Метод прецедентов использования.
- Метод CRC (class–responsibility–collaborators).

Барами (1999) описал каждый из подходов в своей работе, но лишь последний подход бесспорно является оригинальным. Далее мы обобщим эти подходы и приведем примеры, в которых используется *комплексный подход* (mixed approach).

4.2.1.1.1. Метод именных групп

Метод *именных групп* (noun phrase approach) основан на предположении, что аналитик читает формулировки технического задания в поисках имен существительных. Каждое имя существительное рассматривается как *потенциальный класс* (candidate class). Затем список всех классов разделяется на следующие три группы.

- Релевантные классы.
- Нечеткие классы.
- Нерелевантные классы.

К *нерелевантным* (irrelevant) относятся классы, выходящие за рамки проблемной области. Аналитик не в состоянии описать их предназначение. Опытные практики не включают нерелевантные классы в первоначальный список потенциальных классов. Это позволяет избежать формальных шагов по идентификации и исключению нерелевантных классов.

К *релевантным* (relevant) относятся классы, безусловно принадлежащие проблемной области. Имена существительные, представляющие имена этих классов, часто встречаются в техническом задании. Кроме того, аналитик может обосновать значимость и предназначение этих классов, используя знания о предметной области и аналогичных системах, изложенные в справочных руководствах, документах и описаниях коммерческих программных пакетов.

К *нечетким* (fuzzy) относятся классы, которые нельзя уверенно и безоговорочно признать релевантными. Они составляют наибольшую проблему. Их необходимо проанализировать более глубоко, а затем либо включить в список релевантных классов, либо исключить из списка нерелевантных. Окончательная классификация этих классов, собственно, и отличает хорошую модель классов от плохой.

Метод именных групп предполагает наличие полного и корректного технического задания. На практике это предположение редко соответствует действительности. Но даже если оно выполняется, кропотливое изучение больших объемов текста не обязательно должно привести к исчерпывающему и точному результату.

4.2.1.1.2. Метод общих шаблонных классов

Метод общих шаблонных классов (common class patterns approach) позволяет вывести потенциальные классы на основе теории родовой классификации объектов. *Теория классификации* описывает методы разделения мира объектов на удобные группы с целью более подробного изучения.

Барами (1999) приводит следующий перечень групп (шаблонов) для выявления потенциальных классов.

- *Класс понятий* (concept class). *Понятие* — это идея, которую разделяет значительная группа людей. Без понятий люди не способны эффективно общаться или вообще понимать друг друга. Например, `Reservation` (Резервирование) — это концептуальный класс, относящийся к системе резервирования мест в авиакомпаниях.
- *Класс событий* (events class). *Событие* — это нечто, происходящее моментально относительно выбранной временной шкалы. Например, `Arrival` (Прибытие) — это событийный класс, относящийся к системе резервирования мест в авиакомпаниях.
- *Класс организаций* (organization class). *Организация* — это любой вид целенаправленного объединения или совокупности сущностей. Например, `TravelAgency` (Бюро путешествий) — это класс, относящийся к системе резервирования мест в авиакомпаниях.
- *Класс "людей"* (people class). Под "*людьми*" здесь понимается роль, которую человек играет в той или иной системе, а не физическое лицо. Например, `Passenger` (Пассажир) — это класс, относящийся к системе резервирования мест в авиакомпаниях.
- *Класс местоположения* (places class). *Местоположение* описывает размещение объектов, связанных с информационной системой. Например, `Travel-Office` (Офис бюро путешествий) — класс местоположения, относящийся к системе резервирования мест в авиакомпаниях.

Дж. Рамбо и соавторы (J. Rumbaugh et al., 2005) предлагают другую схему классификации.

- *Физический класс* (physical class), например `Airplane` (Самолет).
- *Бизнес-класс* (business class), например `Reservation` (Резервирование).
- *Логический класс* (logical class), например `FlightTimetable` (Расписание рейсов).
- *Прикладной класс* (application class), например `ReservationTransaction` (Операция резервирования).
- *Компьютерный класс* (computer class), например `Index` (Индекс).
- *Поведенческий класс* (behavioral class), например `ReservationCancellation` (Отмена резервирования).

Метод общих шаблонных классов дает аналитикам общие принципы, но не предлагает никакого систематического процесса, посредством которого можно было бы выделить обоснованное и полное множество классов. Этот подход можно с успехом использовать для определения начального множества классов или для проверки того, следует ли включать некоторые классы (полученные другими

способами) в множество или, напротив, исключить их из него. Однако метод общих шаблонных классов слишком слабо связан со специфическими требованиями пользователей, чтобы претендовать на исчерпывающее решение.

Особая опасность, связанная с методом общих шаблонных классов, заключается в неверном истолковании имен классов. Например, что означает `Arrival` (Прибытие)? Означает ли это прибытие на взлетно-посадочную полосу (время приземления), прибытие к терминалу (время высадки), прибытие в зал возврата багажа (время таможенного досмотра) и т.д.? Аналогично, слово `Reservation` в среде североамериканских индейцев имеет совершенно иное значение (имеется в виду резервация. — *Примеч. ред.*).

4.2.1.1.3. Метод прецедентов использования

Метод прецедентов использования играет особую роль в языке UML (см. раздел 1.5.2). Графическая модель прецедентов использования сопровождается неформальными описаниями, а также диаграммами деятельности и взаимодействий (см. разделы 3.2 и 3.4). Эти дополнительные описания и модели определяют этапы (и объекты), необходимые в каждом прецеденте использования. На основе этой информации можно прийти к обобщениям, позволяющим выявить потенциальные классы.

Метод прецедентов использования обладает особенностями, присущими подходу снизу вверх. После выявления прецедентов и частичного определения *моделей взаимодействия* объекты, используемые в этих диаграммах, приводят к идентификации классов.

В действительности этот подход в чем-то похож на метод именных групп. Их объединяет тот факт, что прецеденты определяют требования. Оба подхода направлены на изучение формулировок, изложенных в техническом задании, чтобы выявить потенциальные классы. Способ представления этих формулировок — повествовательный или графический — имеет второстепенное значение. В любом случае на этапе анализа большую часть прецедентов использования можно описать только в текстовой форме без диаграмм взаимодействия.

Метод прецедентов использования страдает от тех же недостатков, что и метод именных групп. Будучи по сути подходом снизу вверх, он обеспечивает точность описания за счет полноты и корректности моделей прецедентов использования. Частичные модели прецедентов использования приводят к неполным моделям классов. Более того, модели классов соответствуют определенным функциональным свойствам и не всегда отражают *все* важные понятия предметной области. Каковы бы ни были цели и средства, это приводит к *функциональному подходу* (function-driven approach), который сторонники объектно-ориентированного подхода предпочитают называть *проблемно-ориентированным* (problem-driven approach)).

4.2.1.1.4. Метод CRC

Метод CRC (class–responsibility–collaborators — класс–ответственность–сотрудники) представляет собой нечто большее, чем метод выявления классов, — это привлекательный способ интерпретации, анализа и изучения объектов (Wirfs-Brock and Wilkerson, 1989; Wirfs-Brock et al., 1990). Метод CRC включает в себя сеансы "мозгового штурма", проведение которых облегчается за счет использования специально подготовленных карточек. Карточки состоят из трех разделов: *имя класса* записывается в верхнем разделе, *обязанности* класса перечислены в левом разделе, а *сотрудники* перечислены в правом. Обязанности — это услуги (операции), которые класс готов выполнить в интересах других классов. Для выполнения многих обязанностей необходимо участие (обслуживание) со стороны других классов. Такие классы перечисляются как сотрудники.

Метод CRC — это живой процесс, во время которого разработчики "играют в карты", заполняя карточки именами классов и назначая им "обязанности" и "сотрудников" в ходе исполнения сценария обработки информации (например, сценария использования). В тех случаях, когда возникает потребность в некоей услуге, а существующие классы не предоставляют ее, создается новый класс, которому назначаются соответствующие "обязанности" и "сотрудники". Если класс становится "слишком загруженным", он разделяется на несколько меньших классов.

Метод CRC идентифицирует классы на основе анализа передачи сообщений между объектами при выполнении задач обработки информации. Акцент делается на равномерном распределении "интеллекта" в системе. Некоторые классы могут быть выведены из технического проекта и не являются "бизнес-объектами". В этом смысле метод CRC можно использовать для верификации классов, выявленных с помощью других методов. Метод CRC также полезен при определении свойств классов, вытекающих из обязанностей и сотрудников класса.

4.2.1.1.5. Комплексный подход

На практике процесс выявления классов в разное время обычно опирается на разные подходы. Зачастую он включает элементы всех четырех подходов, рассмотренных выше. Немаловажными факторами при этом выступают общая эрудиция эксперта, его опыт и интуиция. Процесс идет ни снизу вверх, ни сверху вниз — он все время идет "из середины". Подобный подход к выявлению классов называется *комплексным* (mixed approach).

Один из возможных сценариев заключается в следующем. Начальное множество классов можно сформировать на основе общих знаний и опыта эксперта. При этом дополнительно можно руководствоваться методом общих шаблонных классов. Остальные классы можно добавить, основываясь на анализе обобщенного описания проблемной области с использованием метода именных групп. Если в распоряжении аналитика имеются прецеденты использования системы, то мож-

но воспользоваться методом прецедентов использования, чтобы добавить новые и проверить существующие классы. Наконец, метод CRC позволяет применить "мозговой штурм" для проверки правильности списка выявленных классов.

4.2.1.1.6. Некоторые правила выявления классов

Ниже приведен далеко не полный перечень руководящих принципов или правил, которым должен следовать аналитик при выборе потенциальных классов. Эти принципы относятся только к *классам сущностей* (см. раздел 4.1.3.1).

- Для каждого класса должно быть ясно сформулировано его *предназначение* (statement of purpose) в системе.
- Каждый класс — это шаблон описания *множества объектов* (set of objects). Классы *одиночек* (singleton classes), допускающие существование только одного объекта, весьма маловероятны среди бизнес-объектов. Подобные классы обычно являются частью общих знаний о приложении и, как правило, жестко запрограммированы в программах приложения. Например, если система спроектирована для единственной организации, существование класса `Organization` (Организация) может быть не оправданно.
- Каждый класс (т.е. класс-сущность) должен содержать *набор атрибутов* (set of attributes). Хорошим приемом является установление идентифицирующих атрибутов (*ключей*), чтобы помочь нам судить о мощности класса (т.е. ожидаемом количестве объектов данного класса в базе данных). Следует, однако, помнить о том, что класс не обязательно должен обладать пользовательским ключом. Объекты классов можно идентифицировать с помощью *идентификаторов объектов* (object identifiers — OID).
- Каждый класс должен отличаться от *атрибута*. Представляется ли понятие классом или атрибутом, зависит от области приложения. Цвет автомобиля обычно воспринимается как атрибут класса `Car` (Автомобиль). Однако на фабрике по производству красок `Color` (Цвет) — это определенно класс со своими собственными атрибутами (яркостью, насыщенностью, прозрачностью и т.д.).
- Каждый класс содержит *набор операций*. Однако на данном этапе мы не касаемся вопросов идентификации операций. Операции, входящие в *интерфейс* класса (услуги, предоставляемые классом системе), являются логическим следствием формулировки предназначения класса (см. пункт 1).

4.2.1.1.7. Примеры выявления классов

Пример 4.1. Зачисление в университет

Рассмотрите следующие требования к системе, управляющей зачислением в университет, и выделите потенциальные классы.

- Для получения каждой университетской степени существует несколько обязательных и выборочных курсов.
- Каждому курсу соответствует заданный уровень и значение условных баллов.
- Курс может быть составной частью системы получения произвольного количества степеней.
- Каждая степень определяет минимальное общее значение условных очков, требуемое для получения степени. Например, для степени бакалавра (специальность "вычислительные и информационные системы") требуется 68 баллов, включая обязательные курсы.
- Студенты могут составлять из дисциплин программы обучения, приспособленные к их индивидуальным потребностям и обеспечивающие им получение степени, на которую они претендуют.

Проанализируем требования, перечисленные в примере 4.1, с целью выделения потенциальных классов. В первом утверждении подходящими классами являются классы `Degree` (Степень) и `Course` (Курс), которые удовлетворяют пяти правилам, перечисленным ранее. Пока неясно, следует ли класс `Course` конкретизировать классами `CompulsoryCourse` (Обязательный курс) и `ElectiveCourse` (Факультативный курс). Например, ясно, что курс является обязательным или факультативным в зависимости от научной степени. Различие между обязательными и факультативными курсами может быть зафиксировано с помощью ассоциации или даже атрибута класса. Таким образом, классы `CompulsoryCourse` и `ElectiveCourse` рассматриваются как нечеткие.

Второе утверждение идентифицирует только атрибуты класса `Course`, а именно `course_level` (уровень курса) и `credit_point_value` (количество условных баллов). Третье утверждение характеризует ассоциацию между классами `Course` и `Degree`. Четвертая формулировка вводит атрибут `min_total_credit_points` (минимальное общее количество условных очков) в качестве атрибута класса `Degree`.

Последнее утверждение позволяет нам выделить три новых класса: `Student` (Студент), `CourseOffering` (Предлагаемый курс) и `StudyProgram` (Программа обучения). Первые два, безусловно, являются релевантными классами, а вот

StudyProgram можно превратить в ассоциацию между классами Student и CourseOffering. Поэтому StudyProgram классифицируется как нечеткий класс. Это рассуждение отражено в табл. 4.1.

Таблица 4.1. Потенциальные классы для системы, управляющей зачислением в университет

Релевантные классы	Нечетные классы
Course	CompulsoryCourse
Degree	EfectiveCourse
Student	StudyProgram
CourceOffering	

Пример 4.2. Магазин видеокассет

Рассмотрите следующие требования к системе, управляющей работой магазина видеокассет, и выделите потенциальные классы.

- Магазин видеокассет имеет большую коллекцию современных и популярных видеофильмов. Конкретный фильм может храниться на видеокассетах или дисках.
- Для каждого фильма установлен конкретный период проката (исчисляемый в днях) с соответствующей платой за прокат за этот период.
- Магазин должен быть в состоянии немедленно дать ответ на любой запрос по наличию фильмов в хранилище, а также количеству кассет или дисков (текущие условия по каждой ленте и диску должны быть известны и зафиксированы).

Первое утверждение содержит несколько имен существительных, но только некоторые из них можно превратить в потенциальные классы. Магазин видеокассет — это не класс, поскольку он составляет всю систему (в базе данных может быть только один объект этого класса). Аналогично, понятия хранилища и коллекции слишком общие, чтобы рассматривать их в качестве классов, по крайней мере на этом этапе. Релевантными классами представляются MovieTitle (Название фильма), VideoTape (Видеокассета) и VideoDisk (Видеодиск).

Второе утверждение говорит о том, что каждое название фильма имеет собственный тариф за прокат. Однако не ясно, что понимается под "фильмом" — наименование фильма или же носитель фильма (кассета или диск)? Необходимо прояснить это требование с заказчиками. Тем временем можно объявить класс RentalRates (Тариф за прокат) как нечеткий и не хранить информацию о периоде проката и плате за прокат в классе для названия фильма или его носителя.

Третье условие описывает информацию о состоянии видеокассеты и диска. Однако атрибуты наподобие `video_condition` (состояние видеокассеты) или `number_currently_available` (количество, имеющееся в наличии) могут быть объявлены в абстрактном классе верхнего уровня (назовем его `VideoMedium` (Носитель видеоинформации)), а затем унаследованы конкретным подклассом (например, `Videotape`). Итоги рассуждений отражены в табл. 4.2.

Таблица 4.2. Потенциальные классы для системы, управляющей работой магазина видеокассет

Релевантные классы	Нечетные классы
`MovieTitle`	`RentalRates`
`VideoMedium`	
`Videotape`	
`VideoDisk`	

Пример 4.3. Управление взаимоотношениями с заказчиками

Рассмотрите следующие требования к системе управления взаимоотношениями с заказчиками и выделите потенциальные классы.

- Система поддерживает функции "постоянного контакта" с текущей и потенциальной клиентской базой, так чтобы откликаться на ее нужды и заполучать новые контракты на приобретение наших товаров.
- Система хранит имена, номера телефонов, обычные почтовые и курьерские адреса и т.д. организаций и контактных лиц в этих организациях.
- Система позволяет сотрудникам планировать задания и мероприятия, которые необходимо провести в отношении контактных лиц. Сотрудники планируют задания и мероприятия для других сотрудников или для себя.
- Задание — это группа мероприятий, которые осуществляются для достижения определенного результата. Результатом может быть превращение потенциального клиента в клиента, организация доставки товара или решение проблемы клиента. К обычным типам мероприятий относятся телефонный звонок, визит, отправка факса, устройство обучения и т.д.

Первое утверждение содержит понятия "клиент", "контракт" и "товар". Наша общая эрудиция и опыт подсказывают нам, что это типичные классы. Однако понятия контракта и товара не входят в рамки системы управления взаимоотношениями с заказчиками и должны быть отброшены.

Customer (Клиент) — это релевантный класс, однако мы можем предпочесть назвать его Contact (Контакт), имея в виду, что не все контакты осуществляются с настоящими клиентами. Различие между реальным и потенциальным клиентом может как оправдать, так и не оправдать введение таких классов, как CurrentCustomer (Реальный клиент) и ProspectiveCustomer (Потенциальный клиент). Поскольку уверенности у нас нет, мы объявляем эти классы нечеткими.

Второе утверждение проливает свет на приведенные выше рассуждения. Нам необходимо провести различие между контактной *организацией* и контактным *лицом*. Имя Customer не кажется достаточно удобным для названия класса. Помимо прочего, понятие Customer подразумевает только реального клиента и, кроме того, может означать как организацию, так и контактное лицо. Новое предложение состоит в том, чтобы назвать классы следующим образом: Organization (Организация), Contact (Контакт) (подразумевается контактное лицо), CurrentOrg (Реальная организация) (т.е. организация, являющаяся реальным клиентом) и ProspectiveOrg (Потенциальная организация) (т.е. организация, являющаяся потенциальным клиентом).

Во втором утверждении упоминается несколько атрибутов классов. Однако обычный и курьерский почтовые адреса представляют собой составные атрибуты и применимы к обоим классам: Organization и Contact. Поэтому PostalAddress и CourierAddress — законные нечеткие классы.

В третьей формулировке вводятся три релевантных класса — Employee (Сотрудник), Task (Задание) и Event (Мероприятие). Это предложение объясняет суть плановой деятельности. Последнее утверждение посвящено дальнейшему прояснению смысла и взаимоотношений между классами, однако здесь не вводится ни одного нового класса. Потенциальные классы для этого примера приведены в табл. 4.3.

Таблица 4.3. Потенциальные классы для системы, управляющей взаимоотношениями с заказчиками

Релевантные классы	Нечетные классы
Organization	CurrentOrg
Contact	ProspectiveOrg
Employee	PostalAddress
Task	CourierAddress
Event	

4.2.1.2. Спецификация классов

Сформированный перечень потенциальных классов необходимо уточнить, разместив их на диаграмме и определив их свойства. Некоторые свойства можно ввести и отобразить в виде графических пиктограмм, представляющих классы на

диаграмме классов. Многие другие свойства, включенные в спецификацию класса, имеют только текстовое представление. CASE-инструменты, как правило, обладают возможностями редактирования, позволяющими легко вводить или модифицировать подобную информацию посредством диалоговых окон, снабженных вкладками, или с помощью аналогичных способов.

Как объяснялось в начале главы, классы задаются на определенном уровне абстракции. Более развитые возможности моделирования языка UML здесь не используются — они рассматриваются в главе 5 и последующих главах.

4.2.1.2.1. Именование классов

Каждому классу необходимо присвоить имя. При работе с некоторыми CASE-инструментами, помимо имени, классу можно также присвоить код, возможно, несовпадающий с его именем. Код должен удовлетворять соглашениям об именах, установленным целевым языком программирования или системой управления базами данных. Для генерации программного кода на основе проектной модели используется именно код класса, а не его имя.

Рекомендуется использовать общепринятые соглашения об именах классов. Соглашение PCBMER (см. раздел 4.1.3.2) предусматривает, что имя класса должно начинаться с прописной буквы, означающей архитектурный слой (подсистему/пакет), которому принадлежит данный класс. Собственно имя класса, следующее за обозначением архитектурного слоя, также должно начинаться с прописной буквы. Для составных имен в качестве первой буквы каждого слова также используется прописная буква (вместо того, чтобы отделять слова знаком подчеркивания или дефисом). Это всего лишь рекомендуемое соглашение, однако среди разработчиков оно нашло довольно много приверженцев.

Имя класса должно быть именем существительным в единственном числе (например, `Course`) либо, по возможности, сочетанием прилагательного и существительного в единственном числе (например, `CompulsoryCourse`). Ясно, что класс представляет собой шаблон для множества объектов и использование в качестве имен существительных во множественном числе не несет никакой дополнительной информации. Иногда существительное в единственном числе не отражает истинного предназначения класса. В подобных ситуациях допустимо использование существительных во множественном числе (например, `RentalConditions` (Условия проката) в примере 4.2).

Имя класса должно быть осмысленным и отражать истинную природу класса. Оно должно заимствоваться из словаря пользователей (а не из жаргона разработчиков).

Лучше использовать более длинные имена, чем шифровать их. Можно с уверенностью сказать, что имена, состоящие более чем из тридцати символов, слишком громоздки (а некоторые программные среды их просто не воспринимают, если CASE-инструменты работают с именами классов, а не кодами классов). Возможно также использование более длинных описательных имен в дополнение к именам и кодам классов.

4.2.1.2.2. Выявление и спецификация атрибутов классов

Графическая пиктограмма, представляющая класс, состоит из трех отделений (имя класса, атрибуты, операции) (см. приложение А, раздел А.3). Спецификация атрибутов классов принадлежит к *спецификации состояний* и рассматривается в этом разделе. Спецификация операций рассматривается ниже в этой главе, в разделе 4.3, посвященном спецификации поведения.

Выделение атрибутов осуществляется параллельно с выделением классов. Идентификация атрибутов — это побочный эффект установления классов. Это не означает, что выявление атрибутов представляет собой простую задачу. Наоборот, это процесс, требующий значительных усилий и многократных итераций.

В начальных моделях спецификации определяются только атрибуты, являющиеся существенными для понимания *состояний* (см. раздел 3.5), в которых могут находиться объекты класса. Остальные атрибуты можно до поры до времени игнорировать (однако аналитик должен быть уверен в том, что установленная, но проигнорированная на определенном этапе информация не будет по ошибке утеряна и будет зафиксирована впоследствии). Маловероятно, что *все* атрибуты класса будут приведены в техническом задании, однако важно не включать в спецификацию те атрибуты, которые не вытекают из требований. Остальные атрибуты можно будет добавить на последующих итерациях.

Для *имен атрибутов* мы рекомендуем придерживаться простого соглашения: в именах атрибутов использовать только строчные буквы, а в составных словах второе слово начинать с прописной буквы (например, `streetName`). В качестве альтернативы слова в составных именах можно разделять знаком подчеркивания (например, `street_name`).

4.2.1.2.3. Примеры выявления и спецификации атрибутов классов

Пример 4.4. Зачисление в университет

Обратитесь к примеру 4.1 (см. раздел 4.2.1.1.7) и рассмотрите следующие дополнительные требования, изложенные в техническом задании.

- Выбор студентами учебных курсов может быть ограничен из-за конфликтов расписания, а также из-за ограничений, наложенных на количество студентов, которое может быть набрано на текущий предлагаемый курс.
- Предлагаемая студентом программа обучения вводится в интерактивную систему зачисления. Система проверяет программу на непротиворечивость и сообщает о любых проблемах. Проблемы требуется решать при помощи научного руководителя. Окончательная программа должна быть согласована с представителем заведующего кафедрой, а затем направлена секретарю учебного заведения.

В первом утверждении, указанном в примере 4.4, упоминаются конфликты расписания, однако мы не знаем достоверно, как следует моделировать эту проблему. Возможно, что речь здесь идет о прецеденте, который процедурно определяет конфликты расписания.

Вторую часть этой же формулировки можно смоделировать за счет введения атрибута `enrolment_quota` (квота набора) в класс `CourseOffering`. Теперь также ясно, что класс должен обладать атрибутами `year` (год) и `semester` (семестр).

Вторая формулировка укрепляет нас во мнении о необходимости введения класса `StudyProgram` (Программа обучения)). Поскольку класс `StudyProgram` сочетает в себе ряд дисциплин, предлагаемых к изучению в текущий момент, он также должен обладать атрибутами `year` и `semester`.

Ближайшее рассмотрение нечетких классов `CompulsoryCourse` (Обязательный курс) и `ElectiveCourse` (Факультативный курс) приводит нас к выводу, что учебный курс является обязательным или факультативным в зависимости от научной степени, которую хочет получить студент (см. табл. 4.1). Один и тот же курс может быть обязательным по отношению к одной степени, факультативным по отношению к другой и вообще не допустимым применительно к некоторым другим степеням. По этой причине элементы `CompulsoryCourse` и `ElectiveCourse` не являются классами в полном смысле этого слова. (Заметим, что здесь мы не затрагиваем область моделирования классов с использованием обобщения; см. приложение А, раздел А.7.)

На рис. 4.4 представлена модель классов, соответствующая проведенным нами рассуждениям. Кроме того, на рисунке используются *стереотипы* (stereotype) `<<PK>>` и `<<CK>>` для обозначения первичных и потенциальных ключей соответственно. Это уникальные идентификаторы объектов для рассмотренных классов. Здесь же заданы типы данных для атрибутов. Типы данных соответствуют языку Java.

Классы `StudyProgram` и `CourseOffering` пока не имеют идентифицирующих атрибутов. Они будут введены в эти классы после установления ассоциативных связей между классами (см. приложение А, раздел А.5).

Первое утверждение в примере 4.5 разъясняет, что условия проката зависят от содержания носителя, например `EMovie`, `EGame` и `EMusic`. Кроме того, они зависят от вида носителя: `EVideotape`, `EDVD` или `ECD`.

Из формулировки второго требования вытекает необходимость двух классов: `EEmployee` и `ECustomer`. Ассоциации между этими классами и классом `ERental`, управляющим операциями, связанными с прокатом, будут определены позднее.

Основные обязанности класса `ERental` определяются в третьем пункте. Многие единицы проката могут обрабатываться в рамках одной транзакции. Кроме того, этот утверждение устанавливает четыре категории содержания носителей: фильмы, телевизионные программы, игры и музыка.

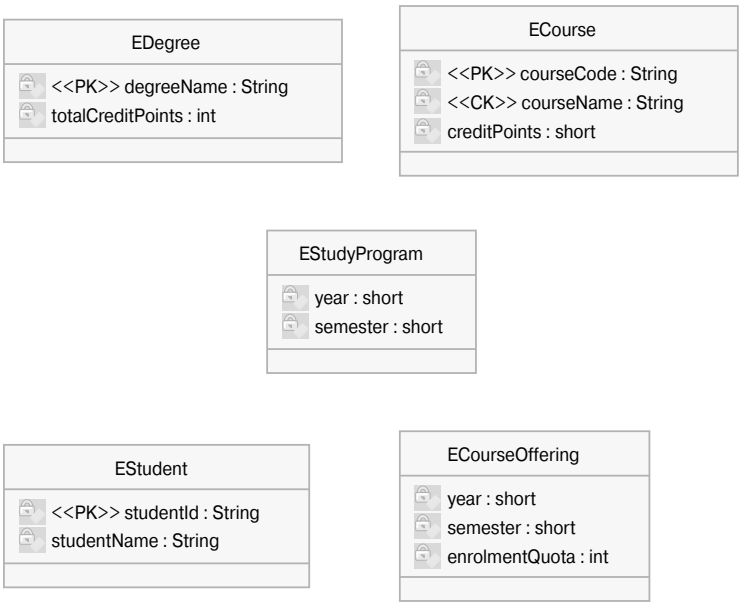


Рис. 4.4. Спецификации классов для системы управления зачислением в университет

Пример 4.5. Магазин видеокассет

Обратитесь к комбинированным требованиям, перечисленным в примере 4.2 (см. раздел 4.2.1.1.7) и рассмотрите дополнительные требования.

- Тариф за прокат зависит от вида носителя и его содержания. Содержанием может быть фильм или музыка. Носителем может быть видеокассета, DVD и игровой CD. Музыкальный CD представляет собой носитель музыки, но некоторые музыкальные диски могут иметь видеоформат, например VCD или DVD-A.
- Система хранит информацию о сотрудниках и определяет их ответственность за выполнение операций, выполняемых для заказчика.
- Для проката в течение разного срока выполняются разные операции. Детали отдельной операции могут быть связаны с несколькими носителями одновременно (поскольку все единицы выдаются на один и тот же период времени). Носитель может содержать фильм, телепрограммы, игры или музыку.
- Работники магазина стремятся запомнить коды наиболее популярных лент. Они часто используют коды, а не названия фильмов. Это полезная практика, поскольку разные режиссеры могли снять фильмы с одинаковыми названиями.

Последний пункт добавляет в класс `EMovie` атрибуты кода фильма `movieCode` (в качестве ключевого атрибута) и `director` (режиссер). Остальные атрибуты были рассмотрены в примере 4.2 (см. раздел 4.2.1.1.7).

На рис. 4.5 показана модель классов для системы, управляющей магазином видеокассет, построенная в результате обсуждения требований примеров 4.2 и 4.5. Атрибут `EMovie.isInStock` является *производным*. Атрибут `EVideoMedium.percentExcellentCondition` является *внутриклассовым* (*статическим*) (см. приложение А, раздел А.3.3). Этот атрибут может содержать долю объектов `EVideoMedium`, значение атрибута `videoCondition = "E"` (отлично). Несмотря на то что этот факт на диаграмме не отражен, для вычисления доли требуемых объектов необходимо ассоциировать с этим атрибутом некую внутриклассовую операцию (например, `computePercentageExcellentCondition`).

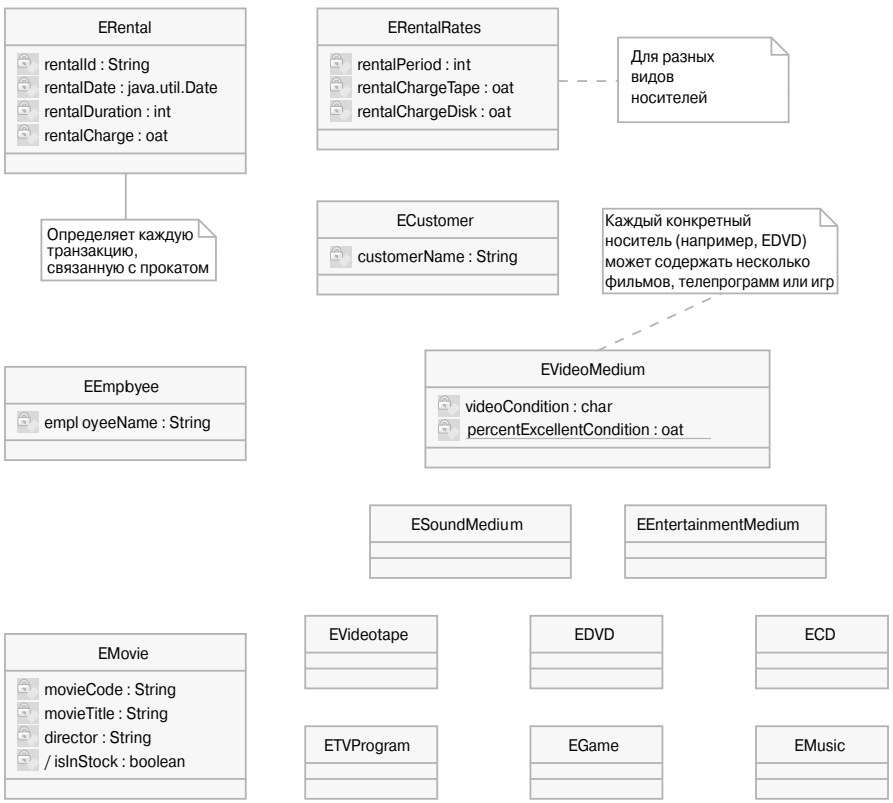


Рис. 4.5. Спецификации классов для системы управления магазином видеокассет

Пример 4.6. Управление взаимоотношениями с заказчиками

Обратитесь к примеру 4.3 (см. раздел 4.2.1.1.7) и рассмотрите следующую дополнительную информацию.

- Клиент считается реальным, если с ним заключен контракт на поставку продукции или предоставление услуг. Однако функции управления контрактами выходят за рамки нашей системы.
- Система позволяет создавать различные отчеты о контактах на основе почтового и курьерского адресов (например, находить всех клиентов по почтовому коду).
- Дата и время создания задания фиксируются. Можно также сохранить значение "дохода", ожидаемого от выполнения задания.
- Мероприятия для сотрудника отображаются на экране его компьютера в виде страницы календаря (один день на страницу). Приоритет каждого мероприятия (низкий, средний, высокий) визуально выделяется на экране.
- Не со всеми мероприятиями связано понятие "срок исполнения" — некоторые из них являются "бессрочными" (они могут выполняться в любое время в течение дня, на который они запланированы).
- Время создания мероприятия не может изменяться, а срок исполнения — может.
- По завершении мероприятия дата и время его завершения фиксируются.
- Система также хранит отличительные признаки для сотрудников, которые создают задания и планируют мероприятия, а также для сотрудников, получающих задания и проводящих мероприятия.

Анализ первого утверждения в примере 4.6 показывает, что понятие реального клиента выводится на основе ассоциации между классами `Organization` и `Contract`. Эта ассоциативная связь может быть довольно динамичной. Следовательно, потребность в классах `CurrentOrg` и `ProspectiveOrg` отпадает. Более того, система не касается управления контрактами и не отвечает за поддержку класса `Contract`. Лучший вариант, который просматривается в данном случае, — смоделировать решение с помощью производного атрибута `Organization.is_current`, который обозначает реальную организацию и при необходимости может изменяться подсистемой управления контрактами.

Второе утверждение наводит на мысль о необходимости введения двух классов для адресов: `PostalAddress` и `CourierAddress`.

Оставшиеся формулировки требований дают дополнительную информацию о содержании атрибутов классов. Кроме того, здесь можно высказать несколько соображений, связанных с ассоциативными отношениями и ограничениями целостности (они будут рассмотрены позже).

Спецификация классов для системы управления взаимоотношениями с клиентами представлена на рис. 4.6. Как видно на рисунке, отношения между классами в модели отсутствуют. Поэтому, к примеру, восьмое утверждение, которое связывает сотрудников с заданиями и мероприятиями, не нашло отражения в модели.

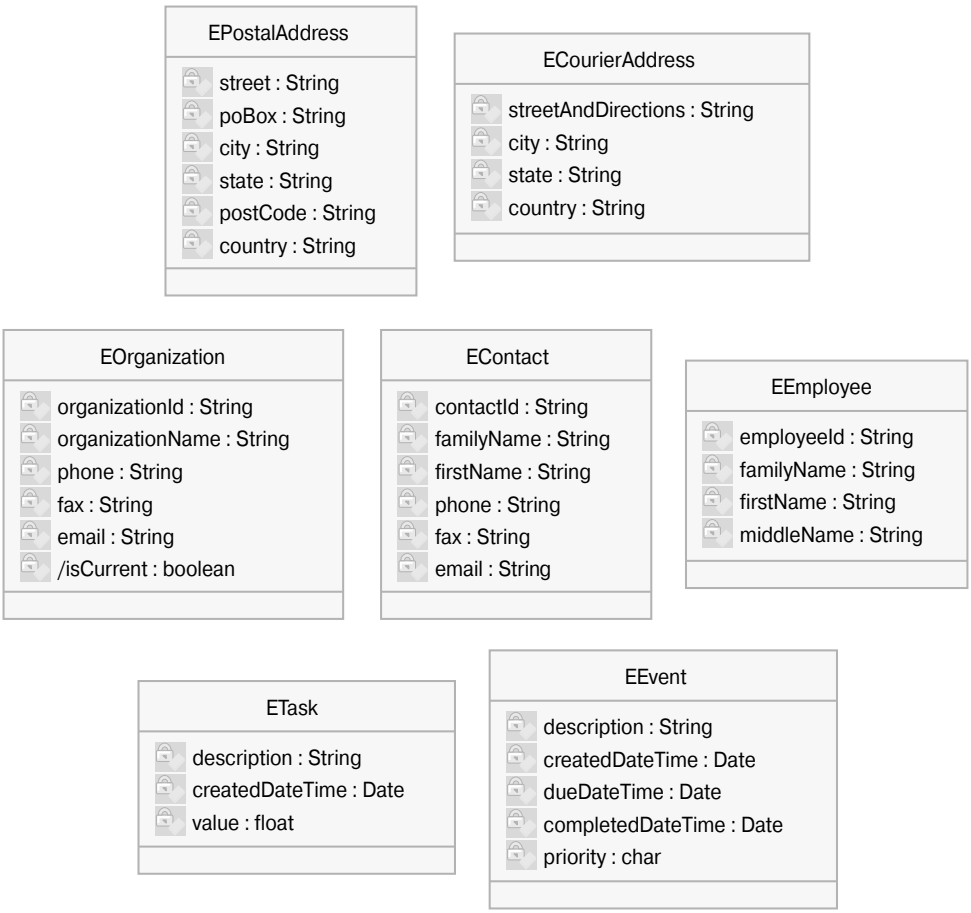


Рис. 4.6. Спецификации классов для системы управления взаимоотношениями с клиентами

Пример 4.7. Прямой маркетинг по телефону

Обратитесь к разделу 1.6.4 (задача 4) и разделу 2.5, в котором представлена бизнес-модель требований для системы прямого маркетинга по телефону, и рассмотрите пример 2.3. В частности, изучите диаграмму бизнес-классов на рис. 2.16 (см. раздел 2.5.4). Примите к сведению следующую дополнительную информацию.

- Каждая кампания имеет свое название, которое, как правило, используется при ее упоминании. Кроме того, кампания обладает уникальным внутренним кодом для внутренних ссылок. Каждая кампания проводится в течение заданного периода времени. Вскоре после завершения кампании проводится розыгрыш лотереи и объявляются владельцы выигрышных лотерейных билетов.
- Все лотерейные билеты пронумерованы. В рамках кампании все билеты обладают уникальными номерами. Общее количество билетов, выпущенных для кампании, количество билетов, проданных до настоящего времени, и текущее состояние каждого билета известны (например, "реальный", "заказан", "оплачен", "выигрышный").
- Для оценки производительности операторов фиксируется продолжительность звонков и успешные результаты (т.е. звонки, завершившиеся заказом билетов).
- В системе поддерживается обширная информационная база, касающаяся благотворителей.
- Помимо обычных сведений, относящихся к контактам (адрес, телефонный номер и т.д.), эта информация включает исторические подробности, такие как дата первого и последнего участия благотворителя в кампаниях вместе с общим количеством кампаний, в которых они участвовали. В системе также хранятся данные обо всех известных предпочтениях и ограничениях, связанных с благотворителем (например, таких как нежелательное время для звонков и обычно используемая им при покупке билетов кредитная карточка).
- Звонки должны обрабатываться в соответствии с определенными приоритетами. Звонки, на которые не получен ответ или зафиксирован ответ автоответчика, требуется перепланировать, чтобы дозвониться позже. Важно изменять время повторных попыток дозвониться.
- Можно пытаться дозвониться снова и снова до тех пор, пока не будет исчерпан лимит попыток. Этот лимит может отличаться для различных типов звонков. Например, лимит для обычного "звонка-приглашения" может отличаться от лимита для "звонка-напоминания" благотворителю о невыполненном платеже.

- Возможные результаты звонков классифицируются, чтобы облегчить ввод данных в систему. Типичными видами результата являются: “успех” (т.е. билеты заказаны), “неудача”, “перезвонить позже”, “нет ответа”, “занято”, “автоответчик”, “факс”, “неверный номер”, “разъединение“.

Первая формулировка, указанная в примере 4.7, позволяет установить несколько атрибутов класса `ECampaign` (Кампания). Класс `ECampaign` содержит следующие атрибуты: `campaignCode` (код кампании) (первичный ключ), `campaignTitle` (название кампании), `dateStart` (дата начала) и `dateClosed` (дата закрытия). Последнее предложение в первом пункте касается призов кампании. При его ближайшем рассмотрении можно прийти к выводу о том, что `EPrize` (Приз) — самостоятельный класс: приз разыгрывается, ему присуще такое свойство, как победитель, и он должен обладать другими явно не выраженными свойствами, такими как описание, ценность и место в ряду других призов кампании.

Мы вводим класс `Prize` в модель классов вместе с атрибутами, о которых говорилось выше: `prizeDescr` (Описание приза), `prizeValue` (Ценность приза) и `prizeRanking` (Место приза). Заметим, что дата розыгрыша призов совпадает для всех призов кампании, и введем атрибут даты розыгрыша `dateDrawn` в класс `ECampaign`. Приз выигрывает благотворитель. Мы можем зафиксировать этот факт позже в связи с введением ассоциации классов `EPrize` и `ESupporter`.

Второе условие гласит, что у каждого билета есть номер, однако он не уникален среди всех билетов (он уникален только в рамках кампании). Мы вводим атрибут `ticketNumber`, однако не придаем ему статус первичного ключа для класса `ECampaignTicket`. Два других атрибута в классе — `ECampaignTicket` —`ticketValue` (Цена билета) и `ticketStatus` (Статус билета). Общее количество билетов и количество проданных билетов определяются атрибутами `numTickets` и `numTicketsSold` в классе `Ecampaign`.

Третье утверждение обнаруживает несколько открытых вопросов. Какие структуры данных требуются нам для расчета производительности труда операторов системы? Для того чтобы ответить на этот вопрос, нам требуется определить некоторые показатели, с помощью которых можно выразить производительность труда. Один из возможных вариантов состоит в том, чтобы подсчитывать среднее количество звонков в час и среднее количество успешных звонков в час. Затем можно вычислить показатель производительности, разделив количество успешных звонков на их общее количество. Введем в класс `ETelemarketer` соответствующие атрибуты: `average_per_hour` и `success_per_hour`.

Для вычисления показателей производительности труда оператора системы необходимо запоминать продолжительность каждого звонка. Хранилищем для этой информации служит класс `ECallOutcome` (Результат звонка). Добавим в этот класс атрибуты, задающие начало и конец кампании: `startTime` и `endTime`. Мы предполагаем, что результат каждого звонка связан с оператором посредством ассоциации.

Анализ четвертого пункта приводит к включению ряда атрибутов в класс ESupporter. К этим атрибутам относятся supporterId (первичный ключ), supporterName (имя спонсора), phoneNumber (номер телефона), mailingAddress (почтовый адрес), dateFirst (дата начала), dateLast (дата окончания), campaignCount (счетчик кампаний), preferredHours (предпочтительные часы) и creditCardAttributes (атрибуты кредитной карточки). Некоторые из этих атрибутов (mailingAddress и preferredHours) достаточно сложны для того, чтобы превратить их в дополнительные классы позже в процессе разработки. Пока мы храним их как атрибуты.

Пятое утверждение касается класса ECallScheduled (Запланированный звонок). Мы вводим в него атрибуты phoneNumber (номер телефона), priority (приоритет) и attemptNumber (количество попыток). У нас нет полного понимания того, как сделать так, чтобы последующие звонки производились в разное время дня. Очевидно, это должно быть возложено на алгоритм планирования, однако нам потребуется поддержка структур данных. К счастью, некоторый свет на эту проблему проливает следующая формулировка.

Результатом анализа шестого утверждения является введение класса ECallType (Тип звонка). Этот класс содержит следующие атрибуты: typeDescr (описание типа), callAttemptLimit (ограничение на количество попыток), а также alternateHours (альтернативные часы). Последний атрибут — это сложная структура данных, аналогичная атрибуту preferredHours класса ESupporter, которая, в конце концов, станет отдельным классом.

Последнее утверждение классифицирует результаты звонков. Это прямое указание на необходимость введения соответствующего класса EOutcomeType. Возможные типы результата могут храниться в атрибуте outcomeTypeDescr (описание типа результата). Не ясно, какие еще атрибуты можно включить в класс EOutcomeType, однако мы убеждены, что по мере изучения детализированных требований эти атрибуты будут установлены. Одним из них может быть атрибут followUpAction (следующий шаг), предназначение которого — хранить информацию о типичных последующих шагах применительно к каждому типу результата.

На рис. 4.7 представлена модель классов, завершающая приведенные выше рассуждения. На ней сохранены ассоциативные отношения, уже установленные в модели бизнес-классов (см. рис. 2.16), и не добавлена ни одна новая ассоциация.

4.2.2. Моделирование ассоциаций

Ассоциации соединяют объекты в системе и способствуют взаимодействию между ними. Без ассоциаций объекты также могут устанавливать связь и взаимодействовать друг с другом во время выполнения программы, но лишь за счет дополнительных вычислений. Например, объект может запросить информацию о другом объекте в процессе передачи сообщений. Кроме того, два объекта могут совместно использовать одни и те же атрибуты (это связано с целостностью реляционных баз данных).

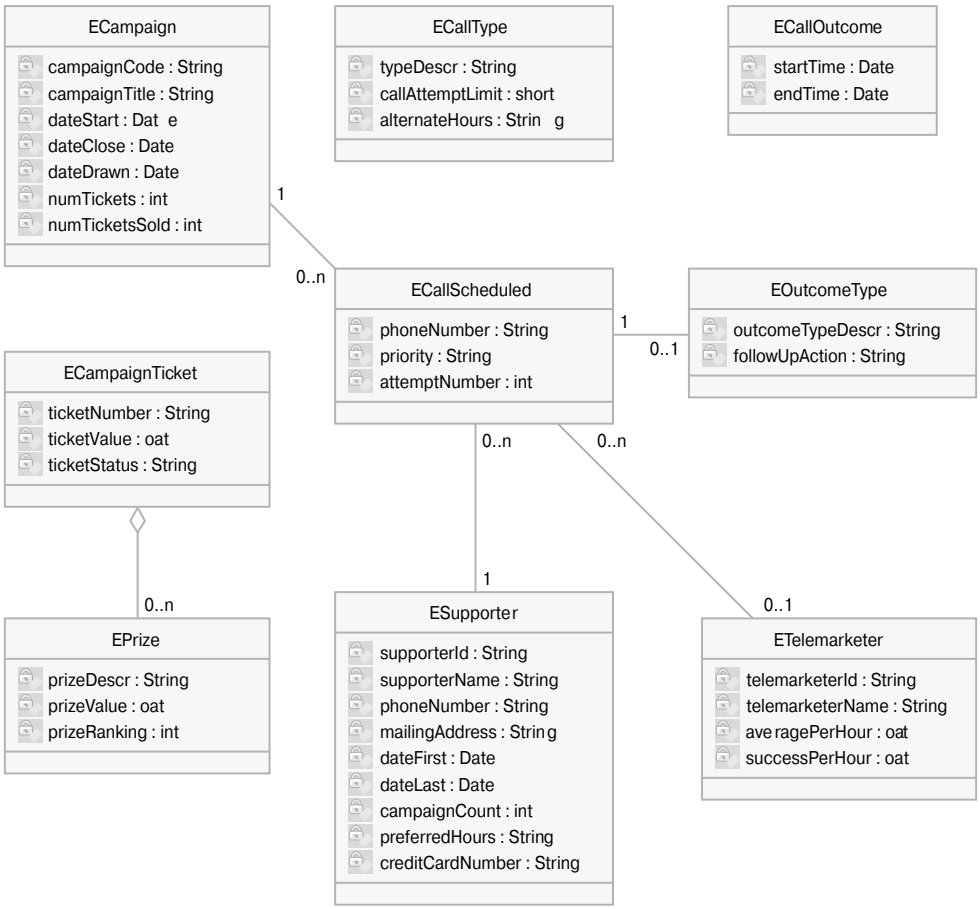


Рис. 4.7. Спецификации классов для системы управления прямым маркетингом по телефону

Ассоциации представляют собой наиболее существенный вид отношений моделей, в частности моделей постоянных бизнес-объектов. Ассоциации поддерживают выполнение прецедентов использования и, таким образом, обеспечивают совмещение спецификации *состояний* и *поведения*.

Среда PCBMER подтверждает важность явных ассоциаций в программе, устанавливая принцип EAP (см. раздел 4.1.3.2). Этот принцип требует, чтобы ассоциация между двумя объектами, вступающими в связь на этапе выполнения программы, устанавливалась на этапе ее компиляции.

4.2.2.1. Выявление ассоциаций

Поиск основных ассоциаций представляет собой побочный эффект процесса выявления классов. При определении классов аналитик принимает решение о его атрибутах, причем некоторые из этих атрибутов являются ассоциациями с дру-

гими классами. Атрибуты могут относиться к элементарным типам данных либо могут вводиться в качестве других классов, устанавливая таким образом отношения с другими классами. По существу, любой атрибут, относящийся к *неэлементарным типам данных*, должен моделироваться как ассоциация (или агрегация) по классу, представляющему этот тип данных.

"Холостой прогон" прецедентов использования позволяет выявить оставшиеся ассоциации и установить *пути взаимодействия* (collaboration paths) между классами, необходимые для выполнения прецедентов. Обычно ассоциации должны поддерживать эти пути взаимодействия.

Несмотря на то что ассоциации используются для описания передачи сообщений, существует различие между *циклом ассоциаций* (cycle of associations) и *циклом сообщений* (cycle of messages). Циклы ассоциаций, например простейший цикл, показанный на рис. A.14 (см. приложение A, раздел A.5.2), встречаются часто и не вызывают абсолютно никаких проблем. Циклы сообщений, в которых, например, несколько объектов обмениваются взаимными сообщениями, наоборот, небезопасны, поскольку они порождают зависимости, возникающие на этапе выполнения программ и плохо поддающиеся контролю.

Иногда *цикл ассоциаций* является *некоммутативным* (незамкнутым). Это позволяет полностью выразить базовую семантику (Maciaszek, 1990). Иначе говоря, по крайней мере одна из ассоциаций в цикле может быть *производной* (derived). Соответствующий пример показан на рис. 3.9 (см. главу 3, раздел 3.3.3). С семантической точки зрения *производная ассоциация* избыточна и должна быть исключена (хорошая семантическая модель не должна быть избыточной). Тем не менее довольно часто производные ассоциации остаются в проектной модели (например, из соображений эффективности).

4.2.2.2. Спецификация ассоциаций

Спецификация ассоциаций подразумевает выполнение следующих действий.

- Присваивание имен ассоциациям.
- Присваивание имен ассоциативным ролям.
- Установление кратности ассоциации (см. приложение A, раздел A.5.2).

Правила *именования ассоциаций* должны до некоторой степени соответствовать соглашениям по именованию атрибутов. Однако принцип CNP в подходе PCBMER, утверждающий, что перед именем ассоциации должна стоять прописная буква, обозначающая архитектурный слой (см. раздел 4.1.3.2), не применяется к *именам ассоциаций*. Это объясняется тем, что имена ассоциаций не реализуются в системе. Они используются только для моделирования.

И наоборот, *роли* ассоциаций представляются в реализованных системах с помощью атрибутов классов, связанных ассоциацией (см. приложение A, раздел A.3.1.1). Следовательно, правила *именования ролей ассоциаций* должны строго соответствовать именам атрибутов, т.е. первое слово должно начинаться со строч-

ной буквы, а все последующие — с прописной (см. раздел 4.2.1.2.2). Иногда по выбору разработчика перед именем ассоциации можно ставить определенный артикль *the*, например `–theOrderLine`.

Если два класса связаны только одним ассоциативным отношением, то задавать имена этой ассоциации и ее роли необязательно. CASE-инструменты могут различать каждую ассоциацию через системные идентификационные имена.

Имена ролей (role names) можно использовать для раскрытия более сложных ассоциаций, в частности *самоассоциативных отношений* (*рекурсивных ассоциаций*, связывающих объекты одного и того же класса). При задании ролевых имен их следует выбирать с учетом того, что в проектной модели они станут атрибутами классов, расположенных на противоположных концах ассоциативной связи.

Кратность (multiplicity) должна быть задана для обоих концов (ролей) ассоциации. Если вопрос кратности на этом этапе не ясен, нижнюю и верхнюю границы кратности можно опустить.

4.2.2.3. Пример спецификации ассоциации

Пример 4.8. Управление взаимоотношениями с заказчиками

Обратитесь к примеру 4.3 (см. раздел 4.2.1.1.7) и 4.6 (см. раздел 4.2.1.2.3). Требования, приведенные в этих примерах, позволяют нам выявить и специфицировать ассоциации на классах для системы управления взаимоотношениями с заказчиками.

Модель ассоциаций для системы управления взаимоотношениями с заказчиками показана на рис. 4.8. Для того чтобы продемонстрировать гибкость ассоциативного моделирования, имена ассоциаций и ролевые имена используются бессистемно. На рисунке также показан код класса `ETask`, который автоматически сгенерирован используемым CASE-инструментом (IBM Rational Rose). Обратите внимание на то, что если на графической модели указано имя роли, то оно используется генератором кода. В противном случае CASE-инструмент сам генерирует имена ролей, о чем свидетельствует имя переменной `theEvent` в сгенерированном коде, несмотря на то, что в модели имя роли ассоциации с классом `EEvent` не указано.

Кратность всех ассоциаций между классами `EPostalAddress` и `ECourierAddress`, с одной стороны, и классами `Organization` и `Contact`, с другой, равна единице. Однако ограничение "исключительное ИЛИ" `{Xor}` представляет собой множество двух пар ассоциаций. В языке UML *XOR-ограничение* демонстрируется с помощью пунктирной линии с именем `{Xor}`, соединяющей две или более ассоциации (фигурные скобки в языке UML обозначают ограничения, наложенные на модель). Поясним использование ассоциации между классами `EOrganization` и `EPostalAddress`.

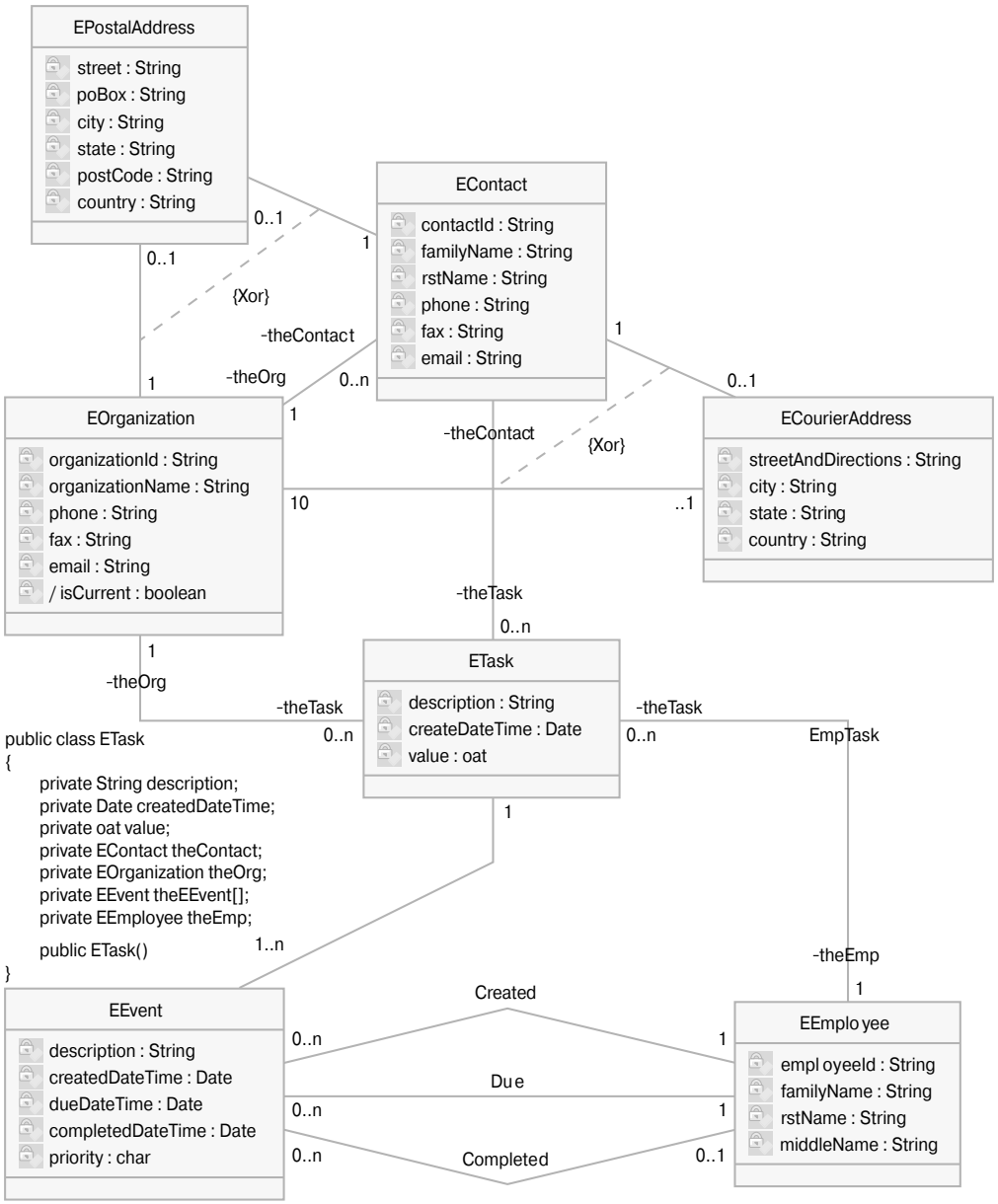


Рис. 4.8. Спецификации ассоциаций для системы управления взаимоотношениями с заказчиками

На одном конце ассоциации объект класса `EOrganization` связан максимум с одним объектом класса `EPostalAddress`, но только в том случае, если почтовый адрес организации известен. На противоположном конце ассоциации

конкретный объект класса `EPostalAddress` связан с объектом класса `EOrganization` *или* объектом класса `EContact`.

Кратность роли `–theContact` между классами `ETask` и `EContact` не задана. Требования не объясняют, должно ли задание быть непосредственно связано с контактом. Поэтому у нас нет уверенности в том, может ли оно быть связано более чем с одним контактом.

Наконец, существуют три ассоциации между классами `EEvent` и `EEmployee`. Эти ассоциации устанавливают, кто из сотрудников создает мероприятие, кто отвечает за его выполнение и кто завершает его. Во время создания мероприятия сотрудник, который будет его выполнять, неизвестен (поэтому кратность на конце ассоциации `Completed` со стороны работника равна "нуль или один").

4.2.3. Моделирование отношений агрегации и композиции

Агрегация и ее более строгая форма — *композиция* — реализует семантику "часть–целое" между составным классом (супермножеством) и компонентным классом (подмножеством) (см. приложение А, раздел А.6). В языке UML агрегация трактуется как ограниченная форма ассоциации. Это колоссальная недооценка роли агрегации в моделировании. Достаточно сказать, что агрегация, наряду с обобщением, является наиболее важным методом многократного использования функциональных возможностей в объектно-ориентированных системах.

Моделирующая способность языка UML значительно усилилась бы, если бы язык поддерживал четыре возможных семантики для агрегации (Maciaszek et al, 1996b).

1. Агрегация *ExclusiveOwns* (Безраздельное владение).
2. Агрегация *Owns* (Владение).
3. Агрегация *Has* (Содержит).
4. Агрегация *Member* (Участник).

Агрегация *ExclusiveOwns* устанавливает следующие свойства.

- Между компонентными классами и их составными классами установлено отношение *зависимости от существования* (existence-dependent), т.е. удаление составного объекта распространяется вниз по иерархии отношения, и связанные компонентные объекты также удаляются.
- Агрегация *транзитивна*, т.е. если объект *C1* является частью объекта *B1*, а объект *B1* — частью объекта *A1*, то объект *C1* является частью объекта *A1*).
- Агрегация *асимметрична* (*нерефлексивна*), т.е. если объект является частью объекта *A1*, то объект *A1* не является частью объекта *B1*.
- Агрегация *стационарна*, т.е. если объект *B1* является частью объекта *A1*, то он не может быть частью объекта $A_i (i \neq 1)$.

Агрегация *Owns* поддерживает первые три свойства агрегации *ExclusiveOwns*, к которым относятся:

- зависимость от существования;
- транзитивность;
- асимметричность.

Агрегация *Has* семантически слабее, чем агрегация типа *Owns*. Эта агрегация поддерживает следующие свойства:

- транзитивность;
- асимметричность.

Агрегация *Member* обладает свойством целенаправленного группирования независимых объектов, при котором не делается предположений относительно свойства зависимости от существования, транзитивности, асимметричности или стационарности. Это абстракция, в которой совокупность членов-компонентов рассматривается как составной объект более высокого уровня. Компонентный объект в агрегации *Member* может одновременно принадлежать нескольким составным объектам, поэтому кратность агрегации *Member* может иметь значение "*многие ко многим*".

Несмотря на то что агрегация получила признание как фундаментальная концепция моделирования, по меньшей мере наряду с *обобщением* (Smith and Smith, 1977), в объектно-ориентированном анализе и проектировании ей уделяется незначительное внимание (за исключением областей приложений, предусматривающих "идеальное сходство", наподобие систем мультимедиа). К счастью, эта тенденция в будущем может измениться благодаря вкладу и проницательности специалистов, работающих над шаблонами проектирования. Это ясно видно, например, из трактовки агрегации (композиции) в книге так называемой "Банды четырех" (GoF) (Gamma et al., 1995). ("Бандой четырех" в шутку называют авторов широко известной книги *Design Patterns — Elements of Reusable Object-Oriented Software* Эриха Гамму (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides). — *Прим. ред.*)

4.2.3.1. Выявление агрегаций и композиций

Поиск *агрегаций* ведется параллельно с поиском *ассоциаций*. Если ассоциация проявляет одно или более из четырех семантических свойств, рассмотренных выше, то ее можно моделировать как агрегацию. При объяснении отношения агрегации критерием являются слова "*содержит*" ("has") и "*является частью*" ("is part of"). Если иерархия описывается сверху вниз, то используется слово "содержит" (например, объект класса `Книга` "содержит" объект класса `Глава`). При интерпретации снизу вверх используется фраза "является частью" (например, объект класса `Глава` "является частью" объекта класса `Книга`). Если предложение, опи-

сывающее отношение, прочитывается вслух с использованием этих фраз и лишено смысла на естественном языке, то это отношение не является агрегацией.

Со структурной точки зрения агрегация часто связывает воедино большое количество классов, тогда как ассоциация степени выше двух бессмысленна (см. приложение А, раздел А.5.1). Когда требуется связать более двух классов воедино, отличным вариантом моделирования может быть агрегация *Member*. Однако следует отметить, что язык UML допускает *n-арные ассоциации* между тремя и более классами.

4.2.3.2. Спецификация агрегаций и композиций

Язык UML обеспечивает лишь ограниченную поддержку агрегации. Сильная форма агрегации называется в UML *композицией* (composition). В композиции составной объект может физически содержать компоненты ("по значению"). Компонент может принадлежать только одному составному объекту. Отношение композиции языка UML в большей или меньшей степени соответствует нашим агрегациям *ExclusiveOwns* и *Owns*.

Слабая форма агрегации в UML называется просто *агрегацией*. Это отношение "по ссылке" — составной объект физически не содержит компоненты. Один компонент может обладать несколькими ассоциативными или агрегативными связями в модели. Попросту говоря, агрегация в языке UML соответствует нашим агрегациям типа *Has* и *Member*.

Закрашенный ромб (solid diamond) в языке UML представляет композицию, *пустой ромб* (hollow diamond) используется для определения агрегации. В остальном спецификация агрегации совпадает с обозначениями для ассоциации.

4.2.3.3. Пример спецификации агрегации и композиции

Пример 4.9. Зачисление в университет

Обратитесь к примеру 4.1 (см. раздел 4.2.1.1.7) и 4.4 (см. раздел 4.2.1.2.3). Рассмотрите следующие дополнительные требования.

- Академическая характеристика студента должна быть доступна по требованию. Характеристика должна включать информацию об оценках, полученных студентом по каждому из курсов, на которые он записался (и которые он не покинул без взыскания, т.е. в первые три недели с начала семестра).
- За каждый курс отвечает один преподаватель, однако этот курс могут вести и другие преподаватели. В течение разных семестров за один курс могут отвечать разные преподаватели; кроме того, в течение разных семестров курс могут вести разные преподаватели.

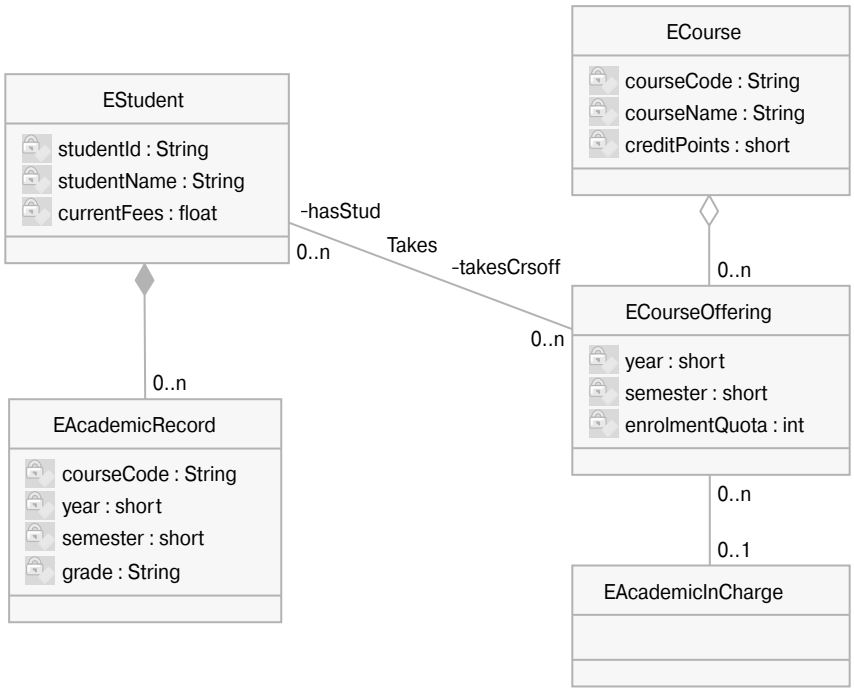


Рис. 4.9. Спецификации агрегаций для системы управления взаимоотношениями с заказчиками

Модель классов, в которой выделено отношение агрегации, показана на рис. 4.9. Класс `EStudent` "содержит" класс `EAcademicRecord` в отношении композиции языка UML (семантика "по значению"). Каждый объект класса `EAcademicRecord` физически содержится в объекте класса `EStudent`. Несмотря на существование ассоциации `Takes` (Выбирает), каждый объект класса `EAcademicRecord` включает атрибут `courseCode` (код курса). Это необходимо, поскольку ассоциация `Takes` реализуется с помощью атрибута `takesCrsoff` ("выбирает" курс) в классе `EStudent`, введенного как *коллекция*, например `Set[CourseOffering]`. Атрибут `takesCrsoff` не зависит от информации во вложенном объекте `EAcademicRecord`, хотя он безусловно связывает объект класса `EStudent` с объектом класса `ECourse`.

Класс `ECourse` "содержит" класс `ECourseOffering` — это описание агрегации в языке UML ("по ссылке"). Каждый объект класса `ECourseOffering` только логически содержится в одном из объектов класса `ECourse`. Объект класса `ECourseOffering` может также участвовать в других агрегациях и/или ассоциациях (например, с объектами классов `EStudent` и `EAcademicInCharge` (Преподаватель)).

4.2.4. Моделирование отношений обобщения

Общие *характеристики* (атрибуты и операции) одного класса и более можно погрузить в более общий класс. Это явление известно как *обобщение* (см. приложение А, раздел А.7). Отношение обобщения (generalization) соединяет обобщенный класс (*суперкласс*) с более специализированными классами (*подклассами*). Обобщение делает возможным *наследование* (*повторное использование*) характеристик суперкласса подклассом. В традиционных объектно-ориентированных системах наследование применяется к классам, а не к объектам (наследуются типы, а не значения).

Помимо наследования, обобщение преследует еще две цели (Rumbaugh et al., 2005).

- Заменимость.
- Полиморфизм.

В соответствии с принципом *заменимости* (substitutability) объект подкласса является законным значением переменной суперкласса. Например, если переменная объявлена с целью хранения объектов класса `Fruit` (Фрукт), то объект класса `Apple` (Яблоко) является допустимым значением.

В соответствии с принципом *полиморфизма* (polymorphism), изложенным в приложении А (см. раздел А.7.1), одна и та же операция может иметь разные реализации в разных классах. Вызывающий объект может вызвать операцию, не зная и не заботясь о том, какая из реализаций операции выполнится. Вызываемый объект знает, какому классу он принадлежит, и выполняет свою собственную реализацию.

Полиморфизм работает лучше в тех ситуациях, когда он используется в сочетании с наследованием. Зачастую полиморфная операция в суперклассе объявляется, а реализация для нее в этом классе отсутствует. Это значит, что операция получила лишь **сигнатуру** (имя и список формальных аргументов), а ее реализация должна быть обеспечена в каждом из подклассов. Подобная *операция* называется *абстрактной*.

Абстрактную операцию (abstract operation) не следует путать с **абстрактным классом** (abstract class). Последний является классом, у которого отсутствуют непосредственные экземпляры (однако его подклассы могут иметь экземпляры). Экземпляры класса `Vegetable` (Овощи) могут не существовать. Непосредственными экземплярами являются только объекты классов `Potato` (Картофель), `Carrot` (Морковь) и т.д.

На практике класс, обладающий абстрактной операцией, является абстрактным. Конкретный класс наподобие `Apple` не может иметь абстрактных операций. Несмотря на то что абстрактные операции вводятся на уровне *спецификаций поведения* (behavior specifications), абстрактные классы относятся к сфере *спецификаций состояний* (state specifications).

4.2.4.1. Выявление обобщений

Многие суперклассы/подклассы аналитик отмечает еще в процессе формирования первоначального перечня классов. Многие другие обобщения можно обнаружить при определении ассоциаций. Различные ассоциации (даже принадлежащие одному и тому же классу) могут связываться с классом на различных уровнях обобщения/специализации. Например, класс `Course` может быть связан с классом `Student` (`Student` *takes* `Course` — студент выбирает курс), кроме того, этот класс может быть связан с классом `TeachingAssistant` (`TeachingAssistant` *teaches* `Course` — ассистент *ведет* курс). Дальнейший анализ может показать, что класс `TeachingAssistant` является подклассом `Student`.

При поиске отношения обобщения критерием являются фразы "*может быть*" ("can be") и "*является разновидностью*" ("is a kind of"). При описании иерархии классов сверху вниз используется фраза "может быть" (например, объект класса `Student` "может быть" объектом класса `TeachingAssistant`, т.е. студент может быть ассистентом). При интерпретации отношения снизу вверх используется фраза "является разновидностью" (например, объект класса `TeachingAssistant` "является разновидностью" объекта класса `Student` — "ассистент — это разновидность студента"). Обратите внимание на то, что если также справедливо утверждение о том, что объект класса `TeachingAssistant` является разновидностью объекта класса `Teacher`, то возникает *множественное наследование* (см. приложение А, раздел А.7.3).

4.2.4.2. Спецификация обобщений

Отношение обобщения между классами показывает, что один класс совместно использует структуру или поведение, определенные в одном или более классов. Обобщение представляется в языке UML сплошной линией со *стрелкой*, указывающей на суперкласс.

Полная спецификация обобщения включает несколько мощных возможностей. Например, можно уточнить отношение обобщения, указав его *уровень доступа* (access), т.е. *право* обращения к другим классам, или определить, что необходимо делать в случае множественного наследования и т.д. Эти вопросы рассматриваются в разделе 5.2 главы 5.

4.2.4.3. Пример спецификации обобщений

На рис. 4.10 представлена модифицированная модель классов для системы управления магазином видеокассет. Иерархия обобщения выражает тот факт, что объект класса `EEntertainmentMedium` "может быть" объектом класса `EVideoMedium` (Носитель видеоинформации) или `ESoundMedium` (Носитель звука). Класс `EVideoMedium` "содержит" класс `ESoundMedium`. Кроме того, объект класса `EVideoMedium` "может быть" объектом класса `EVideotape`, `EDVD` или `ECD`. Класс `ECD` — это "разновидность" класса `ESoundMedium`, но

в некоторых случаях класс ECD может быть "разновидностью" класса EVideo-Medium. Другие сложности, связанные с классом ECD, такие как классификация игровых и звуковых компакт-дисков или различия между VCD и DVD-A, нами не рассматриваются. Вторая иерархия обобщений отражает тот факт, что объект класса EEntertainmentMedium "может быть" объектом класса EMovie, ETVProgram, EGame или EMusic. Классы EEntertainmentMedium, EEntertainmentItem, ETVProgram, EVideoMedium и ESoundMedium являются *абстрактными* (см. приложение А, раздел А.8). Объекты класса EVideo-Medium создаются лишь с помощью *конкретных классов* EVideotape, EDVD и ECD. *Абстрактная операция* computeExcellentCondition() должна быть объявлена (реализована) в этих конкретных классах.

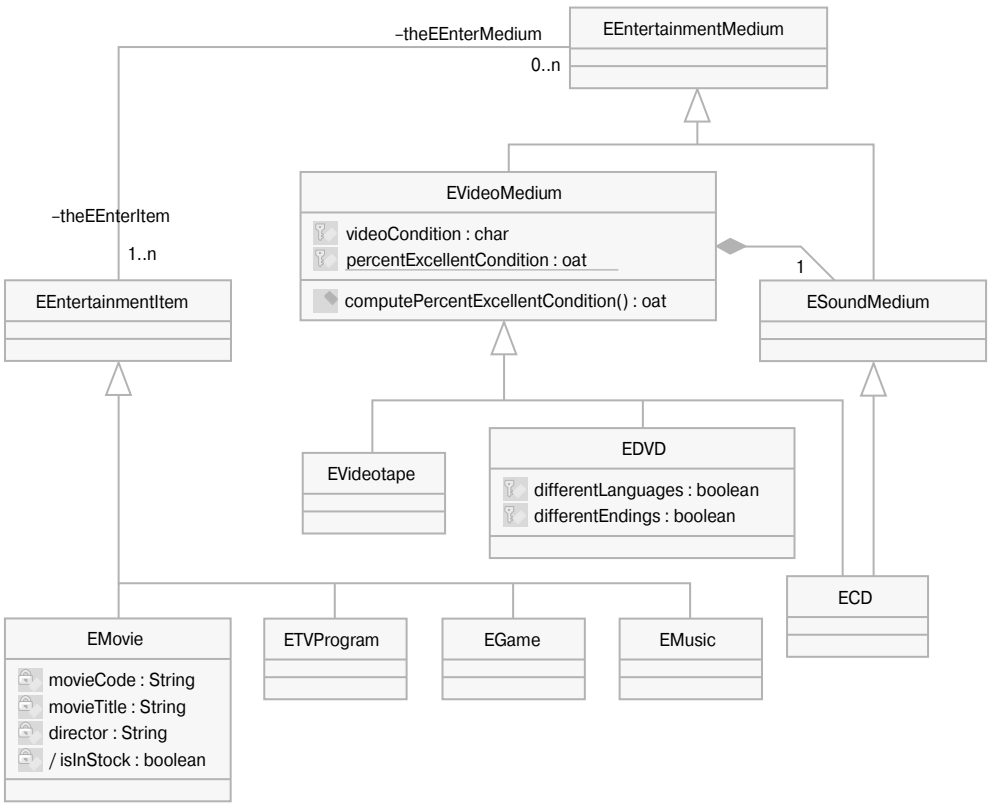


Рис. 4.10. Спецификации обобщений для системы управления магазином видеокассет

Конкретные классы наследуют все атрибуты своих суперклассов. Например, все объекты класса EVideotape содержат переменные videoCondition и percentExcellentCondition. Кроме того, они связаны с объектом класса EEntertainmentItem через объект theEEnterItem.

Пример 4.10. Магазин видеокассет

Обратитесь к примеру 4.2 (см. раздел 4.2.1.1.7) и 4.5 (см. раздел 4.2.1.2.3). Классы, обозначенные на рис. 4.5 (пример 4.5), образуют иерархию обобщений с корнем в классе `EEntertainmentMedium`. Кроме того, они образуют параллельную иерархию обобщений с корнем в классе, который можно было бы назвать `EEntertainmentItem` или `EEntertainmentItem-Category`.

Наша задача заключается в расширении модели, показанной на рис. 4.5, за счет включения в нее отношений обобщения между классами и установления базовой ассоциации между двумя иерархиями обобщений. Для того чтобы отобразить разницу между состояниями классов в иерархии обобщений, корень которой находится в классе `EEntertainmentMedium`, предположим, что емкость EDVD позволяет хранить несколько копий одного и того же фильма на разных языках или с разными развязками. Особенности расшифровки игровых и музыкальных дисков раскрывать не обязательно.

4.2.5. Моделирование интерфейсов

Несмотря на то что *интерфейсы* не имеют реализации, они обладают одними из наиболее мощных средств моделирования (см. приложение А, раздел А.9). Интерфейсы не имеют атрибутов (за исключением констант), ассоциаций или состояний. Они содержат лишь *операции*, но все операции неявно считаются открытыми и абстрактными. Операции *объявляются* (т.е. преобразуются в реализованные методы) в классах, реализующих свои интерфейсы.

Интерфейсы не имеют ассоциаций с классами, но они могут быть целью односторонней ассоциации, идущей от класса. Это происходит, когда атрибут, реализующий ассоциацию, имеет тип интерфейса, а не класса. Значением такого атрибута является ссылка на некий класс, реализующий интерфейс.

Интерфейс может иметь отношение обобщения с другим интерфейсом. Это значит, что интерфейс может расширять другой интерфейс, наследуя его операции.

4.2.5.1. Выявление интерфейсов

В отличие от других элементов моделирования, рассмотренных ранее, интерфейсы не выявляются на этапе анализа предметной области. Интерфейсы должны основываться на обоснованных принципах моделирования, определяющих надежные и легко поддерживаемые системы. Отделяя классы, *использующие* интерфейс, от классов, *реализующих* этот интерфейс, можно получить системы, которые легко понять, поддерживать и развивать.

Интерфейсы играют важную роль в архитектурных моделях, таких как PCBMER (см. раздел 4.1.3). С их помощью можно разрывать циклические зависимости в системе, реализовать схемы событий издатель/подписчик, скрывать реализацию от неавторизованных клиентов и т.д. В типичных ситуациях интерфейс раскрывает только небольшую часть функциональных свойств реального класса.

4.2.5.2. Спецификация интерфейсов

Интерфейсы — это классы, следовательно, их можно изобразить в виде типичного прямоугольника, символизирующего класс, и ключевого слова «`interface`». Вместо ключевого слова можно использовать небольшой кружок, расположенный в правом верхнем углу прямоугольника. Кроме того, интерфейс можно изобразить в виде маленького кружочка, а его имя указать чуть ниже.

Класс, *использующий* (*требующий*) интерфейс, можно обозначить с помощью пунктирной стрелки, указывающей на интерфейс. Для ясности рядом со стрелкой указывают ключевое слово «`use`». Класс может использовать (требовать) только часть операций из полного списка операций, поддерживаемых интерфейсом.

Класс, *воплощающий* (*реализующий*) интерфейс, обозначается пунктирной линией с треугольником на конце. Это обозначение напоминает символ обобщения, только линия при этом должна быть пунктирной. Для ясности рядом с этой линией указывается ключевое слово «`implement`». Этот класс должен *реализовать* (*предоставлять*) все операции, обеспечиваемые интерфейсом.

4.2.5.3. Примеры спецификации интерфейсов

Пример 4.11. Управление взаимоотношениями с заказчиками

Обратитесь к примеру 4.3 (см. раздел 4.2.1.1.7), 4.6 (см. раздел 4.2.1.2.3) и 4.8 (см. раздел 4.2.2.2.1). Рассмотрим, в частности, классы `EContact` и `EEmployee`, показанные на рис. 4.8 в примере 4.8. Эти два класса имеют общие атрибуты (`firstName`, `familyName`). Операции, обеспечивающие доступ к этим атрибутам, можно выделить в отдельный интерфейс.

Предположим, что нам необходимо вывести на экран информацию о просрочках. Соответствующий класс, находящийся на уровне представления, предназначен для вывода на экран списка просрочек вместе с именами заказчиков и сотрудников, а также дополнительной информацией (номерами телефонов и адресами электронной почты) о заказчике.

Наша задача — предложить такую модель, чтобы класс представления использовал только один или несколько интерфейсов, реализованных в классах `EEmployee` и `EContact`, для реализации функционального свойства “вывести на экран просрочки”.

На рис. 4.11 показаны два графических представления двух интерфейсов. Интерфейс `IAPerson` реализован классом `EEmployee`. Интерфейс `IAContact-Info` реализован в классе `EContact`. Поскольку класс `IAContactInfo` наследует свойства класса `IAPerson`, класс `EContact` также реализует операции класса `IAPerson`.

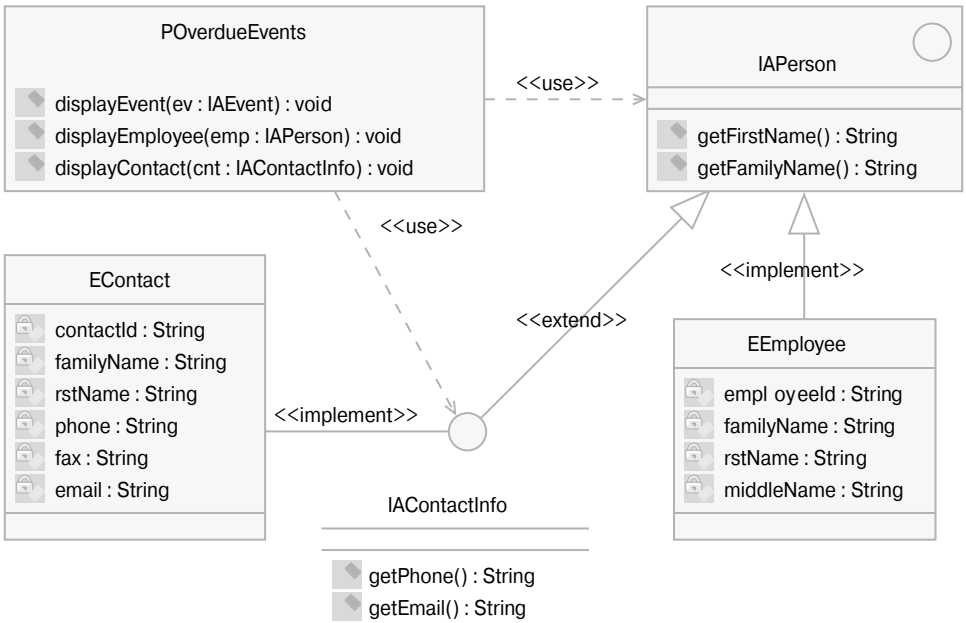


Рис. 4.11. Спецификации интерфейса для системы управления взаимоотношениями с заказчиками

Класс `POverdueEvent` является классом представления. Эта модель демонстрирует три операции в классе `POverdueEvent`. Операция `displayEmployee()` принимает объект класса `IAPerson` как аргумент и, следовательно, способна применить методы `getFirstName()` и `getFamilyName()` к заданному объекту класса `EEmployee`.

Аналогично, операция `displayContact()` получает объект класса `IAContactInfo`. Следовательно, она может вызывать методы, предлагаемые классом `IAContactInfo` и реализованные в классе `EContact`. Существуют четыре таких метода: `getPhone()`, `getEmail()`, `getFirstName()` и `getFamilyName()`.

4.2.6. Моделирование объектов

Моделирование связано с определением *систем*. Модель — это не действующая система, и поэтому она не содержит объектов-экземпляров. В любом случае количество объектов в нетривиальной системе может быть огромным, и представить

их графически невозможно. Тем не менее при моделировании классов мы часто представляем себе объекты и рассматриваем с их помощью трудные сценарии.

4.2.6.1. Спецификация объектов

Язык UML позволяет создавать графическое представление *объектов* (см. приложение А, раздел А.2.1). С помощью диаграммы объектов можно иллюстрировать структуры данных, включая отношения между классами. Объектные диаграммы можно использовать для иллюстрации более сложных структур данных и связей между объектами. Демонстрация связей позволяет прояснить взаимодействие объектов во время выполнения программной системы.

4.2.6.2. Пример спецификации объектов

Пример 4.12. Зачисление в университет

Задача в этом примере заключается в том, чтобы показать несколько объектов, представляющих классы из модели классов на рис. 4.9 (см. пример 4.9 в разделе 4.2.3.3).

На рис. 4.12 из примера 4.12 показана диаграмма объектов, соответствующая модели классов, представленной на рис. 4.9. На рисунке хорошо видно, что объект класса `Student` (Don Donaldson) физически содержит два объекта класса `AcademicRecord` (для курсов COMP224 и COMP326). К настоящему времени Дон Дональдсон записался на два курса: COMP225 и COMP325. Курс COMP325 предлагается во втором семестре 2007 года. Этот курс будет вести Рик Ричардс (Rick Richards).

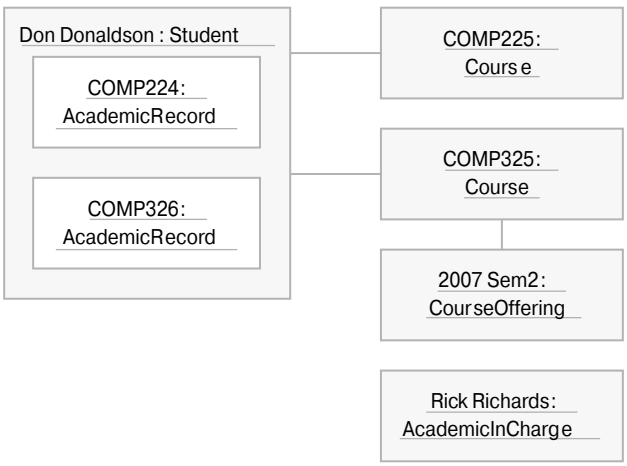


Рис. 4.12. Диаграмма объектов для системы, управляющей зачислением в университет

Контрольные вопросы 4.2

КВ1. Для чего используется метод CRC?

КВ2. Для чего используются имена ролей?

КВ3. Что означает транзитивность в контексте агрегации?

КВ4. Объект подкласса может быть законным значением переменной суперкласса. С каким принципом связано это утверждение?

4.3. Спецификация поведения

Поведение системы с точки зрения внешнего пользователя изображается в виде прецедентов использования (см. раздел 3.1 главы 3). Модели прецедентов использования можно разрабатывать на различных уровнях абстракции. Их можно применить к системе в целом для того, чтобы специфицировать основные функциональные блоки разрабатываемого приложения. Их также можно использовать для фиксации поведения пакетов UML, частей пакетов или даже класса внутри пакета.

На этапе *анализа* прецеденты использования фиксируют системные требования, концентрируясь на том, *что* делает или должна делать система. На этапе *проектирования* прецеденты использования можно использовать для спецификации поведения системы в том виде, в каком она должна быть реализована.

Поведение системы, закрепленное с помощью прецедентов использования, требует соответствующих вычислений и взаимодействия объектов для выполнения этих прецедентов. *Вычисления* можно моделировать с помощью диаграмм деятельности. *Взаимодействие* объектов можно задать с помощью диаграмм последовательностей или диаграмм коммуникации.

Спецификация поведения позволяет взглянуть на систему *с точки зрения ее функционирования* (operational view). Здесь основная задача состоит в том, чтобы определить *прецеденты использования* для области приложений и установить, какие классы участвуют в выполнении этих прецедентов. При этом необходимо идентифицировать операции классов и сообщения, передаваемые между объектами. Несмотря на то что взаимодействие объектов инициирует изменения состояния объектов, спецификации поведения дают *функциональный взгляд на застывшее состояние системы*. Изменения в состоянии объектов явным образом находят выражение в *спецификациях изменения состояний*.

Модели прецедентов использования должны нарабатываться итеративно и параллельно с моделями классов. Классы, определенные в спецификации состояний, подлежат дальнейшей детальной проработке, в ходе которой обозначаются наи-

более важные операции. Следует, однако, напомнить, что спецификация состояний определяет только классы сущностей ("бизнес-объектов"). По мере создания моделей поведения появляются другие уровни классов (см. раздел 4.1.3.1).

4.3.1. Моделирование прецедентов использования

Моделирование прецедентов тесно связано с определением требований (см. главу 2). Требования, изложенные в техническом задании, необходимо довести до прецедентов использования, зафиксированных в спецификациях. Если остальной процесс разработки направляется прецедентами, то процесс называется проблемно-ориентированным (см. разд. 4.2.1.1.3).

Подобно моделированию классов, моделирование прецедентов использования представляет собой *итеративный* и *поступательный* процесс. Первоначальную диаграмму прецедентов можно определить на основе требований верхнего уровня. Это может быть *модель бизнес-прецедентов использования* (см. раздел 2.5.2 главы 2). Для дальнейшего уточнения прецедентов следует руководствоваться более детальными требованиями. Если в течение жизненного цикла разработки требования пользователей подвергаются изменениям, то эти изменения следует отразить сначала в техническом задании, а потом в модели прецедентов использования. Затем изменения в прецедентах доводятся до других моделей (Hoffer et al., 2002; Rational, 2000).

При выявлении прецедентов использования аналитик должен убедиться в том, что он твердо придерживается принципов, лежащих в основе концепции прецедентов. Прецеденты использования представляют следующие компоненты общей модели системы (Hoffer et al., 1999; Kruchten, 2003; Rumbaugh et al., 2005; Quatrani, 2000).

- *Завершенный* фрагмент функциональных возможностей, включая *основной поток* логики управления, любые его вариации (*подпотоки*) и исключительные условия (*альтернативные потоки*).
- Фрагмент *внешне наблюдаемых* функций (отличных от внутренних).
- *Ортогональный* фрагмент функциональных возможностей (прецеденты могут совместно использовать объекты, но выполняться независимо друг от друга).
- Фрагмент функциональных возможностей, *инициируемый действующим лицом* (однако, будучи инициированным, прецедент может взаимодействовать с другими действующими лицами). При этом возможно, что действующее лицо окажется только на принимающем конце прецедента, инициированного (возможно, косвенным образом) другим субъектом.
- Фрагмент функциональных возможностей, предоставляющий действующему лицу ощутимый *полезный результат* (в пределах одного прецедента).

Выявление *прецедентов использования* основано на анализе следующих источников информации.

- Требования, определенные в техническом задании.
- Действующие лица и их цели применительно к системе.

Вопросы управления требованиями рассматривались в разделе 2.4 главы 2. Напомним лишь, что требования представляются в печатном виде. При поиске прецедентов использования нас интересуют только *функциональные требования*.

Прецеденты использования можно определить на основе анализа задач, выполняемых субъектами. Джекобсон (Jacobson, 1992) предлагает ответить на ряд вопросов, касающихся действующих лиц. Ответы на эти вопросы помогут идентифицировать прецеденты использования.

- Каковы основные задачи, выполняемые каждым действующим лицом?
- Должно ли действующее лицо получать доступ к информации в системе или модифицировать информацию?
- Должно ли действующее лицо информировать систему о любых изменениях в других системах?
- Следует ли информировать действующее лицо о непредвиденных изменениях в системе?

В ходе анализа прецеденты использования обращаются к личностным потребностям действующих лиц. В некотором роде это **прецеденты использования действующих лиц** (actor use case). Поскольку прецеденты использования определяют основные строительные конструкции системы, то существует необходимость в выделении **системных прецедентов использования** (system use case). Системные прецеденты извлекают все то общее, что присутствует в прецедентах использования действующих лиц, и дают возможность разработать общее решение, применимое (через механизм наследования) к данной области. "Действующим лицом" в системном прецеденте использования является разработчик/программист, а не пользователь. Системные прецеденты использования идентифицируются на этапе проектирования.

4.3.1.1. Спецификация прецедентов

Спецификация прецедентов включает графическое представление действующих лиц, прецедентов использования и четырех типов отношений, перечисленных ниже.

- Ассоциация.
- Включение.
- Расширение.
- Обобщение прецедента.

Отношение *ассоциации* устанавливает каналы связи между действующим лицом и прецедентом использования. В качестве стереотипов для отношений "*содержит*" и "расширяет" используются слова «include» и «extend». Отношение *обобщения* позволяет специализировать прецедент использования посредством изменения любого из аспектов базового прецедента.

Отношение включения «include» позволяет вынести общее поведение за пределы включаемого прецедента. Отношение «extend» обеспечивает контролируемую форму расширения поведения прецедента использования с помощью активизации другого прецедента в определенных точках расширения. Отношение «include» отличается от отношения «extend» в том, что "включаемый" прецедент использования является необходимым для завершения "активизирующего" прецедента.

На практике проект может легко столкнуться с проблемами, если прилагать слишком большие усилия для выявления отношений между прецедентами использования в целом и конкретными парами прецедентов. Кроме того, обобщенные прецеденты использования имеют тенденцию к таким тесным связям, что взаимосвязи станут превалировать и загромождать диаграмму, смещая акценты с надлежащей идентификации прецедентов в сторону отношений между прецедентами.

4.3.1.2. Пример спецификации прецедентов использования

Пример 4.13. Зачисление в университет

Обратитесь к постановке задачи для системы управления зачислением в университет (см. раздел 1.6.1 главы 1), а также к требованиям, определенным в примерах 4.1 и 4.4 (раздел 4.2.1). Наша задача состоит в определении прецедентов на основе анализа функциональных требований.

На рис. 4.13 показана обобщенная диаграмма прецедентов использования для системы, управляющей зачислением в университет (пример 4.13). Модель содержит четыре действующих лица и четыре прецедента использования. Каждый прецедент инициируется действующим лицом и является завершенным, внешне видимым и ортогональным фрагментом функциональных возможностей. Все действующие лица, за исключением действующего лица Студент, являются *инициирующими*. Действующее лицо Студент получает результаты экзаменов и инструкции по зачислению на курсы, и только после этого производится ввод и проверка программы обучения в следующем семестре (учебном периоде).

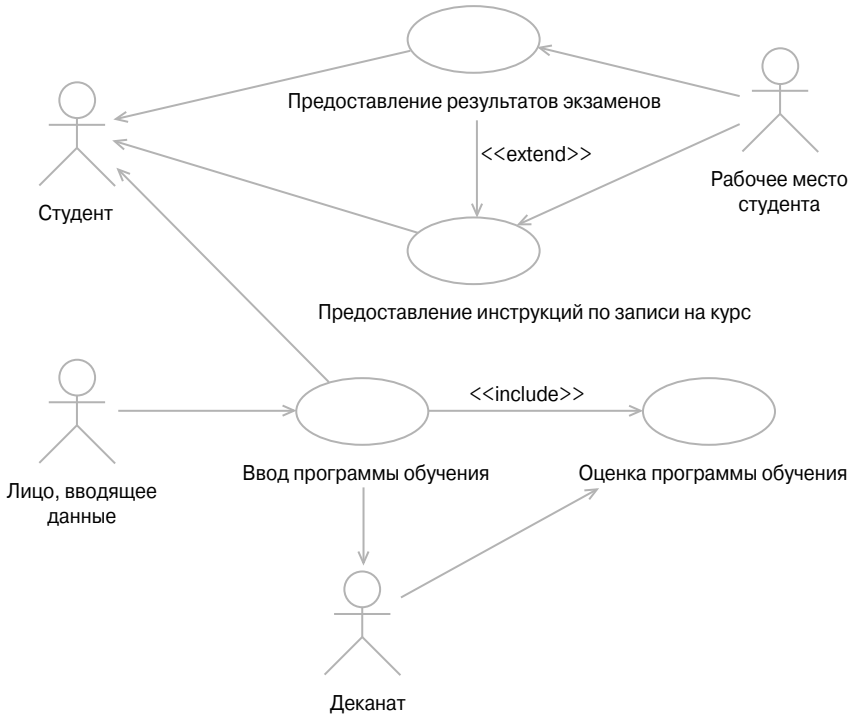


Рис. 4.13. Диаграмма прецедентов использования для системы, управляющей зачислением в университет

Прецедент использования `Предоставить результаты экзаменов` *может* "расширить" («`extend`») прецедент `Предоставить инструкции по записи на курс`. Первый прецедент не всегда расширяет последний прецедент. Например, для новых студентов результаты экзаменов неизвестны. Вот почему отношение моделируется с использованием стереотипа расширения («`extend`»), а не включения («`include`»).

Отношение «`include`» было установлено от прецедента `Ввести программу обучения` к прецеденту `Проверить программу обучения`. Отношение «`include`» означает, что первый из прецедентов всегда включает последний. Как только программа изучения введена, она проверяется на предмет конфликтов расписания, специальных согласований и т.д.

На диаграмме прецедентов использования системы управления взаимоотношениями с заказчиками представлены три действующих лица и пять прецедентов (рис. 4.14). Эта модель обладает интересной особенностью: действующие лица связаны отношением обобщения. `Менеджер по обслуживанию клиентов` — "разновидность" `Сотрудника по обслуживанию клиентов`, который в свою очередь является разновидностью `Сотрудника`. Обобщающая иерархия делает диа-

грамму более выразительной. Любое мероприятие, выполненное Сотрудником, может также выполнить Сотрудник по обслуживанию клиентов или Менеджер по обслуживанию клиентов. Следовательно, Менеджер по обслуживанию клиентов неявно входит в ассоциацию (и может инициировать) любой прецедент использования.

Пример 4.14. Управление взаимоотношениями с заказчиками

Обратитесь к постановке задачи для системы управления взаимоотношениями с заказчиками (см. раздел 1.6.3 главы 1) и к требованиям, определенным в примерах 4.3 и 4.6 раздела 4.2.1. Задача состоит в том, чтобы установить прецеденты использования на основе анализа функциональных требований.

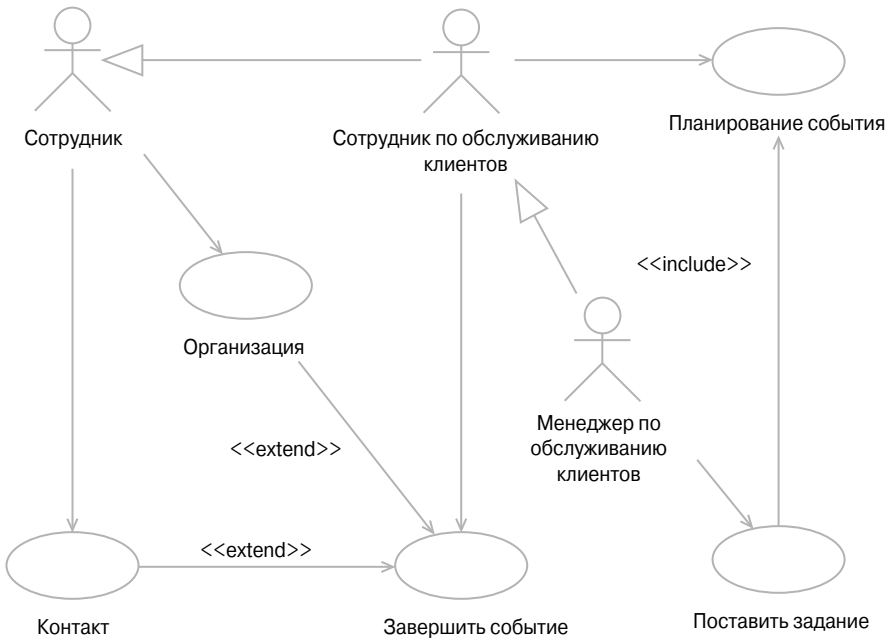


Рис. 4.14. Диаграмма прецедентов использования для системы управления взаимоотношениями с заказчиками

Прецедент Постановка задания "содержит" прецедент Планирование события, как следствие того требования, что задание нельзя создать, не запланировав первое мероприятие. Отношение «extend» обозначает тот факт, что завершение мероприятия может инициировать изменения деталей, связанных с организацией или контактом.

Пример 4.15. Магазин видеокассет

Обратитесь к постановке задачи для магазина видеокассет (см. раздел 1.6.2 главы 1) и к требованиям, определенным в примерах 4.2 и 4.5 раздела 4.2.1. Задача состоит в том, чтобы установить прецеденты использования на основе анализа функциональных требований. Для одного из прецедентов использования напишите неформальную спецификацию, содержащую резюме, действующих лиц, предпосылки, описание, исключения и постусловия.

На диаграмме прецедентов использования системы управления магазином видеокассет (пример 4.15) представлены только два действующих лица и шесть прецедентов использования (см. рис. 4.11). Второстепенные действующие лица наподобие `Клиента` или `Поставщика` не показаны. Эти действующие лица не инициируют ни одного прецедента. Все прецеденты инициирует `Сотрудник`. Между действующими лицами `Сканирующее устройство` и `Сотрудник` установлено отношение *зависимости*, которому аналитик присвоил стереотип в виде фразы «`depends on`» (зависит от).

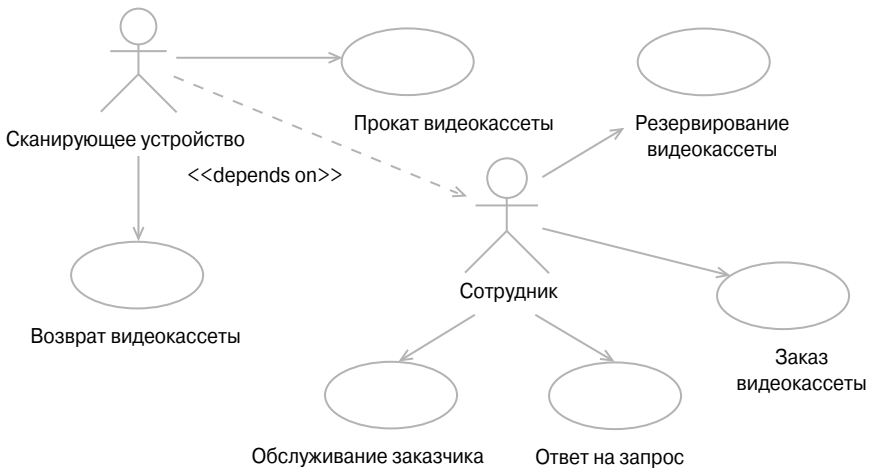


Рис. 4.15. Диаграмма прецедентов использования для системы управления магазином видеокассет

В табл. 4.4 иллюстрируется тот факт, что графическое представление прецедентов использования являет собой лишь один из аспектов полной модели прецедентов. Каждый прецедент на диаграмме в дальнейшем должен быть задокументирован в репозитории CASE-инструмента. В частности, необходимо его текстовое описание. В табл. 4.4 используется одна популярная структура для описательной

спецификации прецедентов (см. раздел 3.1.4 главы 3). Пример более реалистичного прецедента использования, в котором используется аналогичная структура, описан в разделе 6.5.3 главы 6.

Таблица 4.4. Описательная спецификация прецедента использования `Прокат видеокассет`

Прецедент использования	**Прокат видеокассет**
Краткое описание	Клиент желает взять напрокат видеокассету или диск, взятые с полки или заказанные заранее. Если клиент не имеет задолженностей, то заказ выдается только после получения платежа. Если видеокассета не была возвращена своевременно, то клиенту высылается напоминание
Действующие лица	`Сотрудник и Сканирующее устройство`
Предусловия	В наличии имеются видеокассеты или диски, которые можно взять напрокат. У клиентов есть клубные карточки. Сканирующее устройство работает правильно. Работники за прилавком знают, как обращаться с системой
Основной поток	Клиент может спросить работника о наличии видео (включая зарезервированные видеофильмы) или взять кассету или диск с полки. Видеоноситель и членская карточка сканируются, и работнику не сообщают никаких сведений о неплатежах или задержках, так что он не задает клиенту соответствующих вопросов. Если клиент не нарушает правил, то может взять максимум до восьми видеофильмов. Однако если статус клиента определен как "ненадежный", то его просят внести задаток за один период проката каждой кассеты или диска. После получения суммы задатка информация о наличии фильмов обновляется, и кассета или диск передаются клиенту вместе с чеком напрокат. Клиент расплачивается наличными с помощью кредитной карточки или электронного перевода. Каждая запись о прокате хранит (под учетным номером клиента) дату выдачи и срок возврата видеофильма вместе с идентификатором работника. Для каждого видеофильма, сданного напрокат, создается отдельная запись. Прецедент генерирует напоминание о просроченном возврате клиенту, если видеофильм не был возвращен в течение двух дней по истечении даты возврата, а также повторное напоминание по истечении следующих двух дней (и в этот момент клиент помечается как "нарушитель")
Альтернативные потоки	• `Сопровождение клиентов` может быть активизирован для выдачи новой карточки • Попытка взять напрокат слишком много видеофильмов • У клиента нет членской карточки. В этом случае прецедент Видеофильмы не выдаются из-за нарушения клиентом правил

Окончание табл. 4.4

Прецедент использования	Прокат видеокассет
	• Видеоноситель или членская карточка не могут быть отсканированы из-за их повреждения • Электронный перевод денег или платеж по кредитной карточке отклоняется
Постусловия	Видеофильмы сданы напрокат, и база данных обновлена соответствующим образом

Пример 4.16. Прямой маркетинг по телефону

Обратитесь к постановке задачи для системы управления прямым маркетингом по телефону, описанному в примерах 2.1–2.4 (см. разделы 2.1.1.2–2.5.2 главы 2) и к примеру 4.7 раздела 4.2.1.2.3. Установите прецеденты использования на основе анализа функциональных требований.

Решение примера 4.16, приведенное на рис. 4.16, состоит из нескольких прецедентов.

Действующие лица непосредственно взаимодействуют с тремя прецедентами: `Планирование и осуществление следующего звонка`, `Запись результатов звонка` и `Вывод деталей звонка`. Последний прецедент может быть, в свою очередь, "расширен" до прецедентов `Вывод информации о кампании`, `Вывод истории спонсора` и `Вывод подробностей о призе`. Прецедент `Обновления информации о спонсоре` "расширяет" прецедент `Вывод истории спонсора`. Прецедент `Регистрация заказа на билет` и `Планирование обратного вызова` может "расширить" прецедент `Запись результатов звонка`. Предполагается, что действующие лица могут независимо связываться с расширенными прецедентами использования, однако каналы связи для этих прецедентов явно не показаны.

4.3.2. Моделирование деятельности

Подобно традиционным *блок-схемам* и *структурным схемам* (flowcharts and structural charts), получившим распространение в рамках структурных методов процедурно-ориентированной разработки программ, *диаграммы деятельности* (activity diagrams) представляют логический поток управления в объектно-ориентированных программах (хотя и на более высоком уровне абстракции). Это позволяет отразить как *параллельное* (current control), так и *последовательное управление* (sequential control).

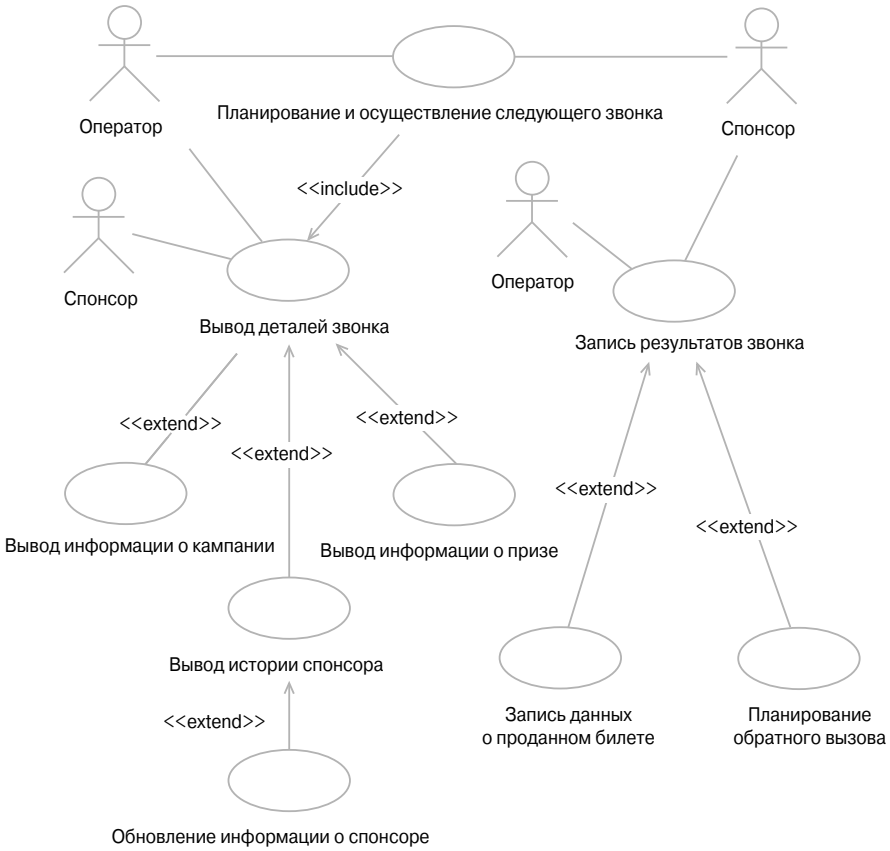


Рис. 4.16. Диаграмма прецедентов использования для системы управления прямым маркетингом по телефону

Модели деятельности широко используются в *проектировании*. Однако они также являются отличным методом иллюстрации вычислений или технологических процессов на уровне абстракции, приемлемом для *анализа*. *Графы операций* (activity graphs) можно использовать для демонстрации различных уровней детализации вычислений.

Модели деятельности могут быть особенно полезны для определения потоков *действий* (actions) в процессе выполнения прецедентов использования. Несмотря на то что действия (операции) выполняют объекты, на диаграммах деятельности классы этих объектов явно не указываются. Следовательно, граф операций можно построить, даже если разработка модели классов еще не завершена. В конце концов, каждый вид деятельности определяется одной или несколькими операциями в одном или нескольких взаимодействующих классах. Детальная проработка такого взаимодействия может быть осуществлена с использованием диаграмм взаимодействия (см. раздел 3.2 главы 3).

4.3.2.1. Выявление действий

Каждый прецедент использования можно моделировать с помощью одного или нескольких графов операций. Событие, источником которого служит действующее лицо, инициирующее прецедент использования, совпадает с событием, запускающим выполнение графа операций. Процесс выполнения последовательно переходит от одного *действия* к другому. Действие считается завершенным, когда заканчивается его вычисление. Внешние прерывания, инициируемые событиями, которые должны прервать действие, допускаются только в исключительных случаях. Если ожидается, что подобные события могут происходить часто, следует использовать диаграмму конечного автомата (см. раздел 3.5.2 главы 3).

Действия лучше всего выявлять на основе анализа предложений *описательной спецификации* прецедентов использования (см. табл. 4.4). Каждая фраза, содержащая глагол, может рассматриваться как потенциальное действие. Описание альтернативных потоков приводит к ветвлению и разделению потоков.

4.3.2.2. Спецификация действий

Деятельность состоит из *действий*. Осуществление деятельности сводится к выполнению составляющих ее действий. После выявления деятельности спецификация ее выполнения выглядит как процесс соединения действий *потоками управления* (control flows) и *потоками объектов* (object flows). Потоки объектов используются тогда, когда возникает необходимость показать, что представляют собой данные, являющиеся предметом действия, и как они должны обрабатываться. Параллельные потоки инициируются (*разделяются*) и *воссоединяются* с помощью *полос синхронизации* (synchronization bars). Альтернативные потоки создаются (*разветвляются*) и *сливаются* в *ромбах принятия решений* (decision diamond).

4.3.2.3. Пример спецификации видов деятельности

Пример 4.17. Магазин видеокассет

Обратитесь к примеру 4.15 раздела 4.3.1.2 и, в частности, к неформальной спецификации диаграммы деятельности `Прокат видеокассет`. Постройте диаграмму, которая должна демонстрировать последовательность действий, соединенных потоками управления (потоки объектов показывать не обязательно).

На рис. 4.17 представлена диаграмма деятельности в рамках прецедента использования `Прокат видеокассет`, описанного в примере 4.17. Нет ничего удивительного в том, что эта диаграмма отражает неформальную спецификацию пре-

цедента (см. табл. 4.4). Обработка начинается со сканирования клубной карточки клиента или видеокассеты/видеодиска. Эти два вида деятельности рассматриваются независимо друг от друга (что показано с помощью символа разделения).

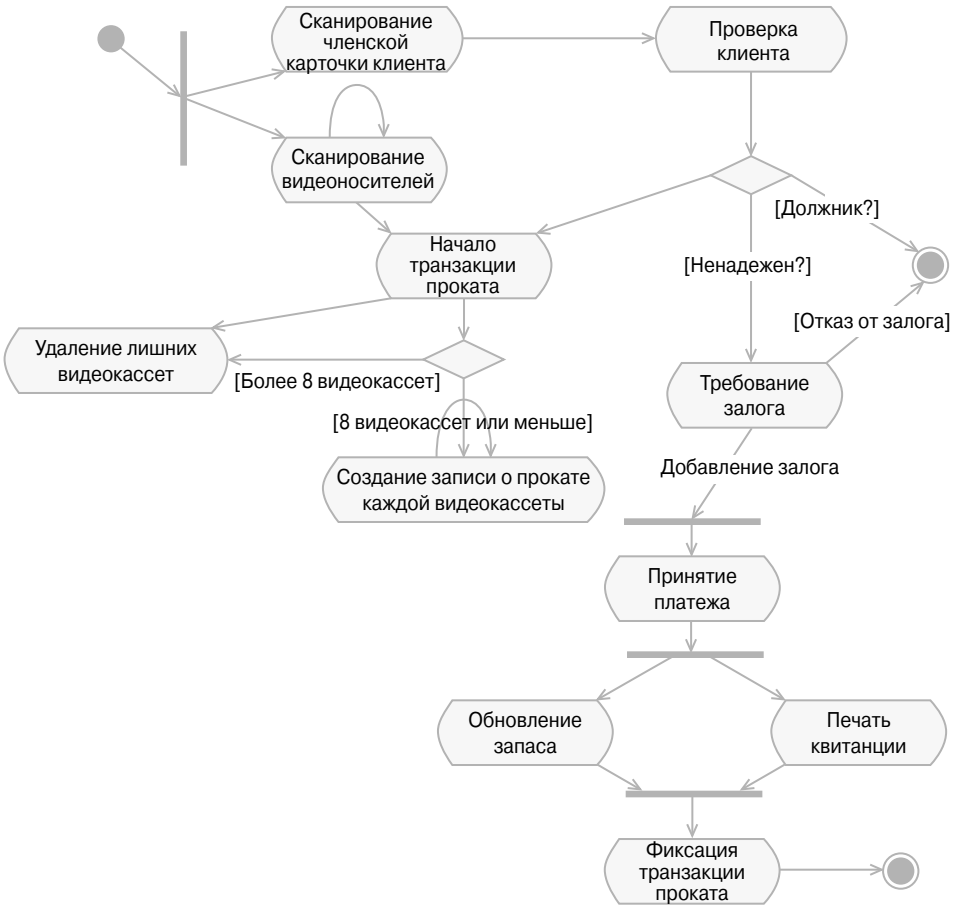


Рис. 4.17. Диаграмма деятельности для прецедента использования `Прокат видеокассет` *в магазине видеокассет*

Действие `Проверка клиента` предусматривает проверку истории заказчика и вычисление выручки, которую магазин может получить в результате одного или другого решения. Если клиент относится к должникам, то выполнение прецедента использования `Прокат видеокассеты` прекращается. Если клиент имеет хорошую репутацию, то начинается выполнение действия `Начало транзакции проката`.

Второе условие разветвления гарантирует, что один клиент не возьмет напрокат больше восьми видеокассет или видеодисков. После выполнения платежа второе разделение позволяет параллельно выполнять действия `Обновление`

запаса и Печать квитанции. Эти параллельные потоки соединяются в одно целое до начала выполнения действия Фиксация транзакции проката, но после завершения всего процесса обработки.

4.3.3. Моделирование взаимодействий

Диаграммы последовательностей (sequence diagram) и *диаграммы коммуникации* (communication diagrams) относятся к *диаграммам взаимодействия* (interaction diagram). Они отражают шаблоны **взаимодействий** между объектами, необходимых для выполнения прецедента использования, операции или других функциональных компонентов.

Диаграммы последовательностей показывают обмен сообщениями между объектами, упорядоченными в виде временной последовательности. Диаграммы коммуникации подчеркивают отношения между объектами, по которым происходит обмен сообщениями. Мы считаем диаграммы последовательностей более полезными для анализа, а диаграммы коммуникации — для проектирования.

Поскольку модели взаимодействия ссылаются на объекты, необходимо, чтобы по меньшей мере первая итерация моделирования состояний была завершена, а основные классы объектов определены. Несмотря на то что взаимодействия влияют на состояния объектов, диаграммы взаимодействия не отражают их в явном виде. Эта функция возложена на диаграммы конечных автоматов и спецификации изменения состояний (см. раздел 3.5 главы 3 и раздел 4.4).

Диаграммы взаимодействия можно использовать для определения *операций* (методов) в классах (см. раздел 3.4.3 главы 3 и раздел 4.3.4). Любое сообщение, направляемое объекту на диаграмме взаимодействия, должно быть обслужено некоторым методом, определенным в классе этого объекта.

4.3.3.1. Выявление последовательностей сообщений

Выявление последовательностей сообщений является логическим продолжением моделирования деятельности. *Действия*, представленные на *диаграмме деятельности*, отображаются в виде *сообщений* на *диаграмме последовательностей*. Если уровни абстракции, используемые для построения модели деятельности и модели последовательностей, совпадают, то такое отображение затруднений не вызывает.

4.3.2.2. Спецификация последовательностей сообщений

При спецификации сообщений полезно проводить различие между сообщением, представляющим собой *сигнал*, и сообщением, которое являет собой *вызов*: “Сообщение — это передача сигнала от одного объекта (отправителя) к одному или нескольким объектам (получателям) или вызов операции одного объекта (получателя) другим объектом (отправителем)” (Rumbaugh et al, 2005). *Сигнал* означает асинхронное взаимодействие объектов. Отправитель может продолжить вы-

полнение потока сразу после отправки сигнального сообщения. *Вызов* означает синхронное обращение к операции с условием возврата управления отправителю. С точки зрения объектно-ориентированной разработки сигналы означают *обработку событий* (event processing), а вызовы — *передачу сообщений* (message passing) (Maciaszek and Liong, 2005).

Методы, активизирующие сообщения, могут возвращать, а могут и не возвращать данные вызывающим объектам. Если значения не возвращаются, то *тип возвращаемого значения* является *пустым*. В противном случае тип возвращаемого значения может быть либо элементарным (например, `char`), либо неэлементарным (т.е. классом). Типы возвращаемых значений на диаграммах последовательностей обычно не указываются. При необходимости на диаграмме можно показать специальную линию возврата (пунктирную линию со стрелкой). Обратите внимание на то, что тип возвращаемого значения не совпадает с возвращаемым сообщением (*обратным вызовом*).

4.3.3.3. Пример спецификации последовательностей

Пример 4.18. Зачисление в университет

Обратитесь к примеру 4.9 раздела 4.2.3.3 и к примеру 4.12 раздела 4.2.6.2, а также к прецеденту использования `Ввод программы обучения`, показанному на рис. 4.13. Задача состоит в том, чтобы построить диаграмму последовательностей для этого прецедента.

Здесь мы не касаемся вопросов проверки корректности введенной программы обучения. Проверке обязательных условий, наличия конфликтов в расписании, специальных разрешений посвящен другой прецедент использования (`Проверка программы обучения`). Наша задача состоит только в том, чтобы убедиться, действительно ли в текущем семестре предлагается курс, на который желает записаться студент, и открыта ли еще на него запись (есть ли свободные места).

На рис. 4.18 и 4.19 показана диаграмма последовательностей для двух альтернативных сценариев решения задачи, описанной в примере 4.18, —централизованного и распределенного (Fowler, 2004). Диаграмма последовательностей, приведенная на рис. 4.18, описывает *централизованное* решение, напоминающее процедурный стиль программирования. В рамках этого подхода один класс (`CEnroll`) принимает на себя нагрузку, выполняя бо́льшую часть работы. Диаграмма последовательностей, показанная на рис. 4.19, описывает *распределенное* решение, в котором объекты сообща выполняют работу, а центрального пункта управления не существует. В этом случае управление обработкой информации распределено между многими объектами. Как правило, при объектно-ориентированном

проектировании (допускающем повторное использование и сопровождение программ) распределенный подход является более предпочтительным.

В обеих ситуациях выполнение прецедента использования начинает `Лицо, вводящее данные`. В языке UML инициатор обработки может оставаться неизвестным. Первое сообщение — так называемое *базовое* (found mesaage) — может исходить из неизвестного источника (Fowler, 2004). Однако многие CASE-инструменты не допускают существования неизвестных источников.

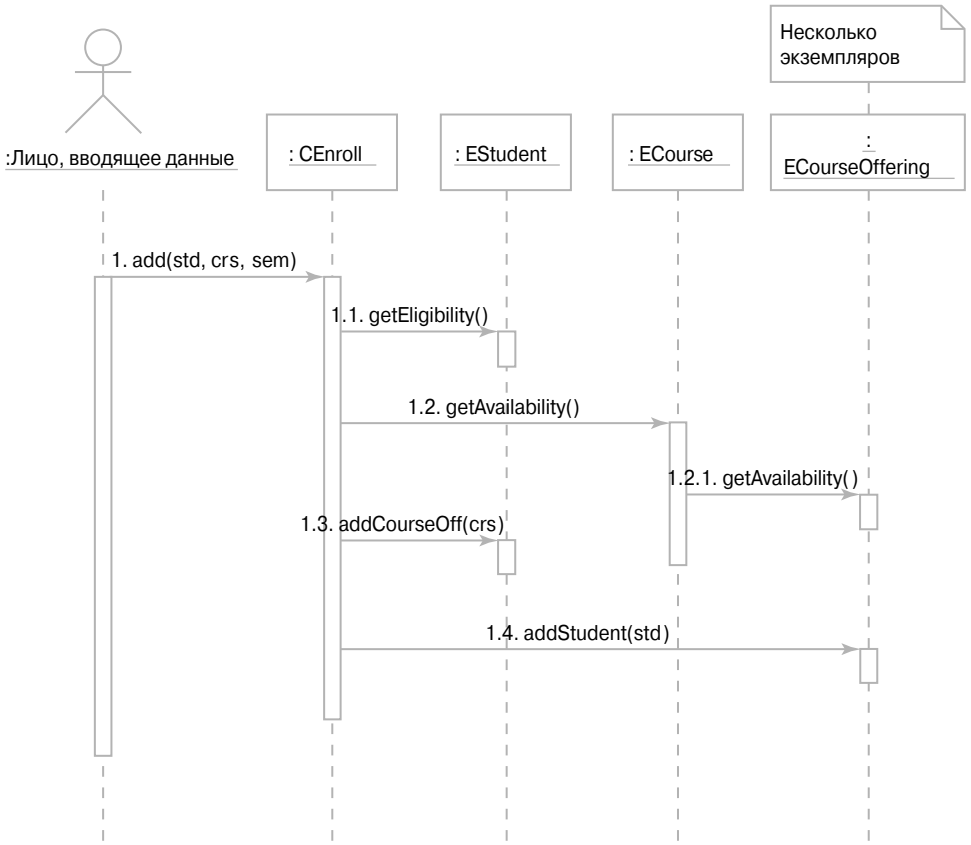


Рис. 4.18. Централизованная диаграмма последовательностей для системы зачисления в университет

Обработка начинается тогда, когда *управляющий* объект класса `CEnroll` получает запрос на зачисление (операция `add()`) некоего студента (аргумент `std`) на курс (аргумент `crs`) на протяжении определенного семестра (аргумент `sem`). В обоих вариантах решения объект класса `CEnroll` поручает продолжение обработки объекту класса `EStudent`. Однако в централизованном решении (см. рис. 4.18) объект класса `EStudent` должен лишь подтвердить свое право на

зачисление (например, подтвердив факт оплаты). С другой стороны, в распределенном решении (см. рис. 4.19) объекту класса `EStudent` поручается самому проверить свое право на зачисление и продолжить обработку в зависимости от результата.

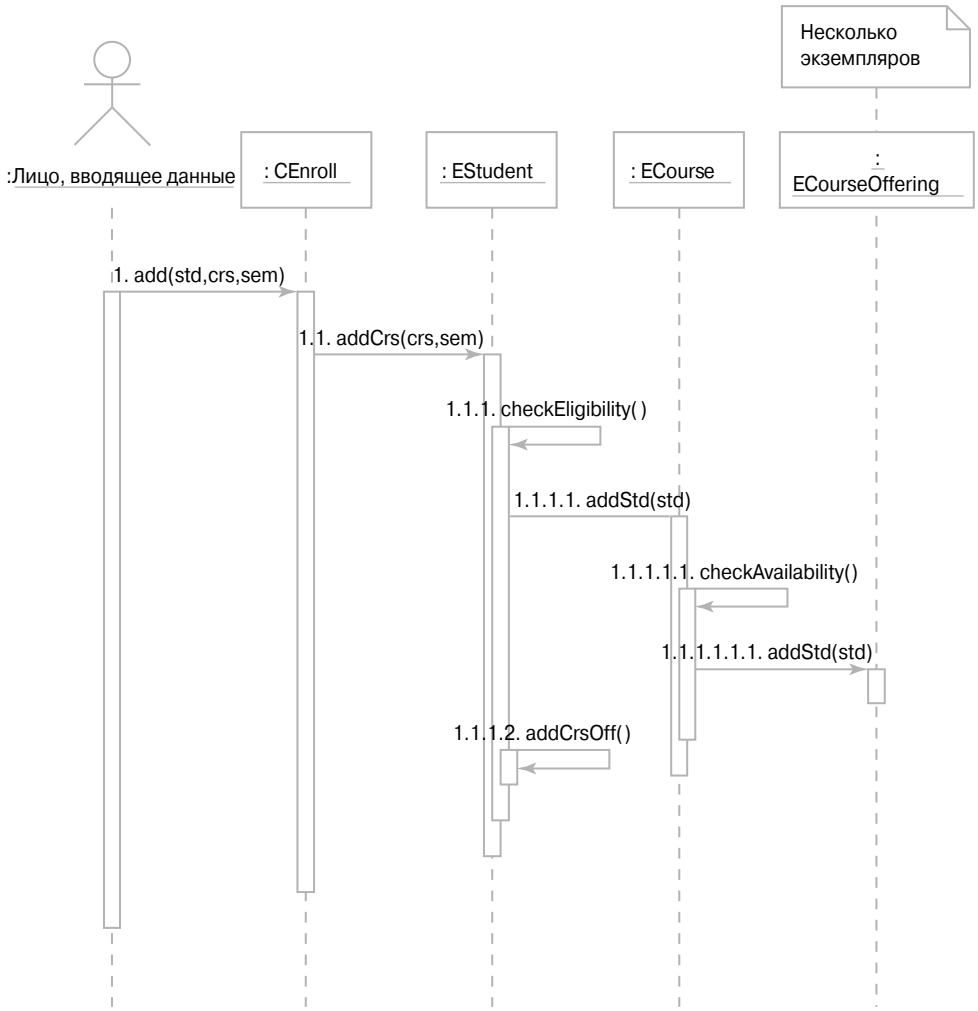


Рис. 4.19. Распределенная диаграмма последовательностей для системы зачисления в университет

Если объект класса `EStudent` *имеет* право на зачисление, то в централизованном решении управление передается объекту класса `CEnroll`. В распределенном решении объект класса `EStudent` должен послать запрос объекту класса `ECourse` на зачисление. Для этого объект класса `EStudent` передает операции `addCrs()` аргументы `crs` и `sem`.

В реальности объект класса `EStudent` включается в состав объекта класса `ECourseOffering`, а не `ECourse`, причем может существовать несколько объектов класса `ECourseOffering`. Зачисление студента на предлагаемый курс требует установления двунаправленной ассоциации между соответствующими объектами классов `EStudent` и `ECourseOffering`. Кроме того, в централизованном решении объект класса `ECourseOffering` устанавливает *связь* с объектом класса `EStudent` и для этого возвращается объекту этого класса.

4.3.4. Моделирование открытых интерфейсов

Открытый интерфейс (public interface) класса определяется набором *операций*, предлагаемых классом в качестве услуг другим классам в системе. Подобные операции объявляются открытыми. Объекты могут взаимодействовать друг с другом для выполнения прецедентов использования и действий только посредством открытых интерфейсов.

Открытые интерфейсы впервые определяются ближе к окончанию этапа анализа, когда спецификации состояний и поведения в основном определены. В ходе анализа определяются лишь *сигнатуры* всех открытых операций (имя операции, список формальных аргументов, тип возвращаемого значения), а в ходе проектирования определяются алгоритмы (например, псевдокоды) для *методов*, реализующих эти операции.

4.3.4.1. Выявление операций классов

Операции классов лучше всего определять на основе диаграмм последовательностей. Каждое *сообщение* в модели последовательностей должно быть обслужено *операцией* в целевом объекте (см. раздел 3.4.3 главы 3). Если модели последовательностей полностью построены, определение открытых операций выполняется автоматически.

Однако на практике диаграммы последовательностей — даже если они разработаны для всех прецедентов использования и видов деятельности — могут не обеспечить достаточного уровня детализации, позволяющего выявить все открытые операции. Кроме того, диаграммы последовательностей для операций, пересекающих границы прецедентов, могут оказаться неприменимыми (например, когда деловые операции охватывают более одного прецедента использования).

По этим причинам при выявлении операций могут оказаться полезными вспомогательные методы. Один из таких методов основан на том факте, что объекты отвечают за собственную "судьбу" и поэтому должны поддерживать четыре элементарных операции.

- Создать (Create).
- Читать (Read).
- Обновить (Update).
- Удалить (Delete).

Этот набор операций известен как *операции CRUD* (см. раздел 2.5.2 главы 2). Операции CRUD позволяют другим объектам отправлять сообщение объекту с требованием выполнить следующие действия.

- Создать новый экземпляр объекта.
- Получить доступ к состоянию объекта.
- Модифицировать состояние объекта.
- Уничтожить себя.

4.3.4.2. Спецификация операций классов

На этом этапе жизненного цикла разработки программного обеспечения необходимо модифицировать диаграмму классов, чтобы ввести в модель *сигнатуры операций* (operation signatures). Здесь можно также определить *область видимости операций*. По умолчанию она совпадает с *областью видимости экземпляра* (операция применяется к *объектам-экземплярам*). *Область видимости класса* (*статическая*) должна быть объявлена явно (с помощью символа $ перед именем операции). Область видимости класса устанавливает, что некая операция применима к *объектам класса* (см. раздел А.3.3 приложения А).

Другие свойства операции, такие как параллельность, полиморфное поведение и спецификация алгоритма, задаются позже в процессе проектирования.

4.3.4.3. Пример спецификации операций классов

Пример 4.19. Зачисление в университет

Обратитесь к примеру 4.18 раздела 4.3.3.3 и диаграммам последовательностей, приведенным на рис. 4.18 и 4.19. Определите операции по этой диаграмме последовательностей и введите их в классы `EStudent`, `ECourse` и `ECourseOffering`.

Операции, заданные в модели классов на рис. 4.20, получены непосредственно из диаграммы последовательностей, показанной на рис. 4.18. Пиктограммы перед атрибутами означают, что они *закрыты* (private). Пиктограммы перед операциями свидетельствуют о том, что они *открыты* (public).

Модель класса, представленная на рис. 4.20, не только полностью соответствует диаграмме последовательностей, показанной на рис. 4.18, но и дополняет ее типами значений, возвращаемых операциями. Установление ассоциации между объектами классов `EStudent` и `ECourseOffering` инициируется объектом класса `CEnroll`. Однако перед этим объект класса `CEnroll` должен получить объект класса `ECourseOffering` от операции `getAvailability()`, относящейся к классу `ECourse`.

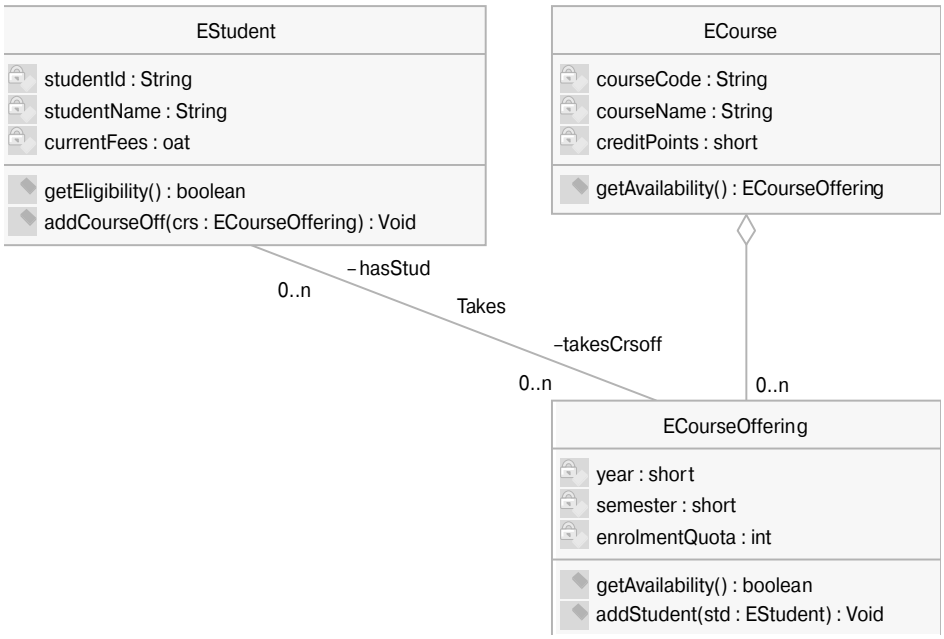


Рис. 4.20. Спецификации операций классов для централизованной системы зачисления в университет

В распределенной системе объект класса `CEnroll` лишь *инициирует* процесс зачисления, а все остальные процедуры выполняются взаимодействующими объектами. На рис. 4.21 показаны операции классов, определенные по диаграмме последовательностей, приведенной на рис. 4.19. Распределенный подход точнее соответствует духу объектной ориентации, предполагающему "использование многих небольших объектов, реализующих множество небольших методов, для создания множества возможностей для замещения и варьирования операций" (Fowler, 2004).

Контрольные вопросы 4.3

КВ1. Дают ли прецеденты использования возможность для представления параллельных потоков управления?

КВ2. Можно ли построить диаграммы деятельности до создания моделей классов?

КВ3. Сообщение может обозначать асинхронное взаимодействие между объектами. Как называется такое сообщение?

КВ4. Какой метод моделирования с помощью языка UML наиболее полезен для выявления операций классов?

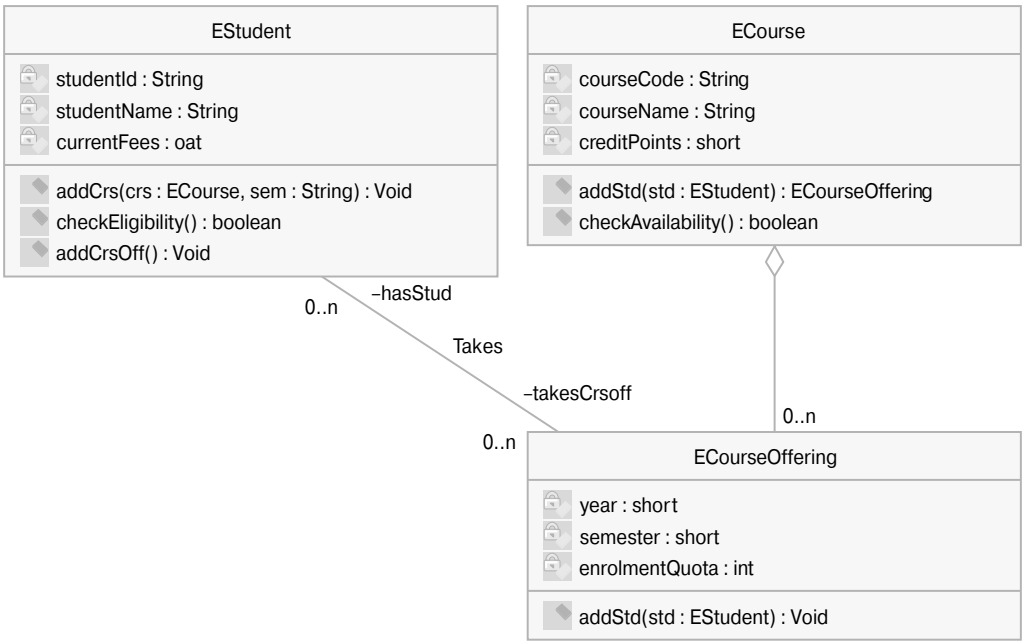


Рис. 4.21. Спецификации операций классов для распределенной системы зачисления в университет

4.4. Спецификации изменения состояний

Состояние объекта в определенный момент времени определяется значениями его атрибутов, включая атрибуты отношений. Спецификация состояний, помимо прочего, определяет атрибуты класса. Спецификация поведения определяет операции класса, некоторые из которых имеют *побочные эффекты*, т.е. изменяют состояние объекта. Однако, чтобы понять, как объект может изменять свое состояние во времени, необходим более внимательный взгляд на систему. Для этого следует использовать спецификации изменения состояний.

Важность спецификаций изменения состояний варьируется в зависимости от предметной области. Моделирование изменения состояний в бизнес-приложениях значительно менее важно, чем в инженерных проектах и системах реального времени. Многие *инженерные приложения и системы реального времени* (engineering and real-time applications) полностью определяются изменением состояний объектов. При моделировании подобных систем необходимо с первого дня сосредоточить внимание на изменении состояний. Что случится, если температура окажется слишком высокой? Что будет, если клапан не закроется? Что произойдет, если контейнер переполнится? И так далее.

В этой книге описываются в основном *бизнес-приложения* (business applications), для которых изменения состояния наблюдаются менее часто. В этом случае моделирование изменения состояний обычно осуществляется ближе к окончанию этапа анализа (и продолжается со значительно большей глубиной на этапе проектирования). Многие спецификации изменения состояний определяют *исключительные условия* в системе. Естественно, что исключения из обычного поведения системы моделируются после спецификации нормального поведения.

4.4.1. Моделирование состояний объектов

Моделирование состояний объектов осуществляется с помощью диаграмм конечных автоматов (см. раздел 3.5.2 главы 3). Диаграмма конечного автомата — это граф состояний и переходов. Модели состояний строятся для каждого класса, демонстрирующего *динамическое поведение*, представляющее интерес для аналитика. Не все классы на диаграмме классов относятся к этой категории.

Диаграммы конечных автоматов можно также использовать для описания динамического поведения других моделируемых элементов, например прецедентов использования, кооперации или операций. Это, однако, встречается нечасто, и некоторые CASE-средства могут не поддерживать подобные функциональные возможности.

4.4.1.1. Выявление состояний объектов

Процесс исследования состояний объектов основан на анализе содержимого атрибутов в классах и выявлении того, какие из этих атрибутов представляют особый интерес с точки зрения прецедентов использования. Не все атрибуты влияют на изменения состояний. Например, изменение телефонного номера клиентом не изменяет состояния объекта `Customer`. У клиента по-прежнему есть телефон, а номер телефона не важен. Однако удаление номера телефона может трактоваться как изменение состояния с точки зрения некоторых прецедентов использования, так как связаться с клиентом по телефону больше нельзя.

Аналогично, изменение номера телефона, которое включает изменение кода региона, может представлять интерес как изменение состояния, поскольку указывает на то, что клиент сменил свое географическое местоположение, и это изменение, возможно, необходимо зафиксировать и отразить на диаграмме конечного автомата.

4.4.1.2. Спецификация состояний объектов

Основные обозначения языка UML, касающиеся спецификации диаграмм конечных автоматов, были введены в разделе 3.5.2 главы 3. Для успешного применения этой системы обозначений аналитик должен понимать, как взаимосвязаны разные понятия, какие сочетания понятий являются неестественными или недопустимыми и какие сокращения возможны в рамках предлагаемой системы обозначений. CASE-инструменты могут накладывать определенные ограничения, но одновременно открывают интересные возможности.

Переходы между состояниями активизируются при наступлении определенных событий *или* выполнении определенных условий. Это означает, например, что линия перехода не нуждается в метке, указывающей *имя события*. Само условие (записанное в квадратных скобках) активизирует переход всякий раз, когда *деятельность* объекта, пребывающего в определенном состоянии, завершена и условие принимает значение "истина".

Некоторые события (например, `Дверь закрыта`) не связаны ни с какими действиями, а другие (например, `Дверь отрывается`) предполагают определенные действия и связаны с явной *деятельностью*. Такая деятельность в определенном состоянии называется *активной* (do activity). Активная деятельность может обозначаться внутри пиктограммы состояния с помощью ключевого слова `do/` (например, `do/закрыть дверь`).

В типичной ситуации переход запускается *сигнальным событием* (signal event) или *событием вызова* (call event). Сигнальное событие устанавливает явную, асинхронную однонаправленную связь между двумя объектами. Событие вызова устанавливает синхронную связь, в которой вызывающий объект ожидает ответа.

Существуют еще два типа пусковых событий: *событие изменения* (change event) и *временное событие* (time event). Событие изменения означает выполнение сторожевого условия (булевого выражения) за счет изменения сторожевого условия. Временное событие означает, что объект, пребывающий в определенном состоянии, удовлетворяет некоему временному условию, например, наступил определенный момент времени или прошло определенное количество времени. В некоторых моделях временные события, запускающие переход в зависимости от абсолютных или относительных измерений времени, оказываются весьма полезными.

Еще одно соображение, касающееся моделирования конечных автоматов, связано с возможностью спецификации *входных действий* (entry action) внутри пиктограммы текущего состояния (с помощью ключевого слова `entry/`) или рядом с линией, символизирующей вход в состояние. Аналогично, *выходное действие* (exit action) можно изображать внутри пиктограмм состояний (с помощью ключевого слова `exit/`)) или рядом с линией, символизирующей выход из состояния. Несмотря на то что это не влияет на семантику, выбор используемого метода может сказаться на удобстве восприятия модели (Rumbaugh et al., 1991).

4.4.1.3. Пример спецификации диаграммы конечного автомата

Пример 4.20. Магазин видеокассет

Обратитесь к классу `EMovie` примера 4.10 (см. рис. 4.10) из раздела 4.2.4.3. Задача состоит в спецификации диаграммы конечного автомата для класса `EMovie`.

Диаграмма конечного автомата для класса `EMovie` приведена на рис. 4.22 и демонстрирует различные способы спецификации переходов. Переходы между состояниями `В наличии` и `Нет в запасе` задают параметризованные имена событий наряду с именами действий и защитными условиями. С другой стороны, переход из состояния `Зарезервировано` в `Не зарезервировано` лишен явного пускового события. Этот переход запускается посредством завершения деятельности в состоянии `Зарезервировано`, в результате которого условие отказа в резервировании `[больше не резервировать]` принимает значение "истина".



Рис. 4.22. Диаграмма конечного автомата для класса `EMovie` *в системе управления магазином видеокассет*

Контрольные вопросы 4.4

КВ1. Представляет ли диаграмма конечного автомата последовательность изменений состояний?

КВ2. Всегда ли при активизации соответствующего перехода изображается изменение состояния?

Резюме

Эта глава является ключевой для всей книги. В ней объясняется чрезвычайно высокая важность архитектуры при разработке системы и описан язык UML во всей свой мощи. Архитектура PCBMER, описанная в этой главе, согласована с языком UML. Ее преимущества будут показаны в последующих главах.

Иллюстративным материалом в этой главе стали примеры, сформулированные в главе 1 и рассмотренные в главах 2 и 3. При их разборе были задействованы все часто используемые модели UML. Девиз этой главы можно было бы сформулировать так: "в аналитических моделях информационных систем однозначных решений вида "черное–белое", "нуль–единица", "истина–ложь" не существует". Для каждой проблемы существует множество потенциальных решений, нужно лишь найти среди них приемлемый и работоспособный вариант.

- Спецификация состояний описывает информационные системы со *статической* точки зрения *классов*, содержания их *атрибутов* и *отношений*. Существует много методов *выявления классов*, но ни один из них не является универсальным. Решить эту проблему можно лишь на основе комплекса методов, соответствующих знаниям и опыту аналитика. В языке UML классы задаются с помощью *диаграмм классов*. Эти диаграммы позволяют наглядно представить классы и три типа отношений между ними: *ассоциацию*, *агрегацию* и *обобщение*.
- Спецификация поведения описывает информационные системы с *функциональной* точки зрения. Движущей силой спецификации поведения и, конечно, анализа требований и проектирования системы в целом выступают прецеденты использования. *Диаграммы прецедентов использования* дают только наглядное представление — истинная сила прецедентов заключается в их *неформальных спецификациях*. Остальные поведенческие диаграммы являются производными от моделей прецедентов. Они включают в себя *диаграммы деятельности*, *диаграммы взаимодействия* и добавляют *операции* в классы.
- Спецификации изменения состояний описывают информационные состояния с *динамической* точки зрения. Объекты "бомбардируются" событиями, и некоторые из этих событий вызывают изменения состояний объектов. Изменения состояний моделируются с помощью диаграмм конечных автоматов.

Ключевые термины

Абстрактная операция (abstract operation). "Операция, не имеющая реализации, т.е. представляющая собой лишь спецификацию, но не метод. Реализация должна обеспечиваться любым конкретным производным классом" (Rumbaugh et al., 2005).

Абстрактный класс (abstract class). Класс, не имеющий непосредственной реализации. Он может иметь лишь косвенные экземпляры, реализуемые с помощью конкретных производных классов.

Архитектура MVC (Model-View-Controller). Архитектурный шаблон "модель–представление–контроллер".

Архитектура PCBMER (Presentation–Controller–Bean–Mediatir–Entity–Resource). Архитектурная модель "представление–контроллер–компонент–посредник–сущность–ресурс".

Архитектура программного обеспечения (software architecture). См. *Архитектура* в разделе "Ключевые термины" главы 1.

Архитектурная метамодель (architectural meta-model). Высокоуровневая модель, определяющая каркас архитектуры программного обеспечения для конкретного проекта.

Архитектурная структура (architectural framework). См. *Архитектурная метамодель*.

Зависимость (dependency). "Отношение, свидетельствующее о том, что для спецификации или реализации отдельных элементов модели или их наборов требуются другие элементы или наборы элементов. Это значит, что полная семантика зависимых элементов либо семантически, либо структурно зависит от определения поставляемых элементов" (UML, 2005).

Заменимость (substitutability). "Принцип, утверждающий, что если S — подтип типа T, то объекты типа T в компьютерной программе можно заменять объектами класса S (т.е. объекты типа S можно подставлять вместо объектов класса T) без изменения желательных свойств программы (корректности, выполняемой задачи и т.д.)" (`www.en.wikipedia.org/wiki/Substitutability`).

Интерфейс JDBC (Java Database Connectivity). Интерфейс соединения с базами данных в языке Java.

Компонент (bean). *Компонент* программного обеспечения, допускающий повторное использование и представляющий собой объект данных и операции (методы) доступа к этим данным. Он также является именем уровня в архитектуре PCBMER.

Компонент (component). См. *Компонент* в разделе "Ключевые термины" глав 1 и 3.

Подсистема (subsystem). Единица иерархической декомпозиции в крупных системах; разновидность компонента.

Подход CRC (Class–Responsibility–Collaborators). Подход класс–ответственность–сотрудники.

Полиморфизм (polymorphism). "Способность объектов принадлежать разным типам, чтобы отвечать на вызовы методов с одинаковыми именами, каждый из которых имеет свои собственные функциональные свойства, зависящие от типа дан-

ных. Программист (и программа) не знают заранее точный тип объекта, поэтому такое поведение можно реализовать на этапе выполнения программы (этот процесс называется поздним, или динамическим связыванием)" (`www.en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming`).

Порт (port). "Свойство классификатора, определяющее отдельную точку взаимодействия между ним и средой или между ним и его внутренними частями... Порт может определять услуги классификатора, которые он предоставляет (предлагает) среде, а также услуги которых он ожидает (требует) от среды" (UML, 2005). См. также *Классификатор* в разделе "Ключевые термины" главы 3.

Предлагаемые интерфейсы (provided interfaces). "Набор интерфейсов, реализованных классификатором... выражающим обязанности его экземпляров перед своими клиентами" (UML, 2005).

Прецедент использования действующего лица (actor use case). Прецедент использования, предусматривающий удовлетворение идентифицируемых потребностей действующего лица (см. также *Прецедент использования* в разделе "Ключевые термины" главы 3).

Прецедент использования системы (system use case). Прецедент использования, удовлетворяющий обобщенные потребности системы в целом. Его функциональные свойства могут повторно использоваться прецедентами использования действующих лиц с помощью наследования.

Принцип APP (Acquaintance Package Principle). Принцип осведомленного пакета в модели PCBMER.

Принцип CEP (Cycle Elimination Principle). Принцип исключения циклов в архитектуре PCBMER.

Принцип CNP (Class Naming Principle). Принцип именования классов в архитектуре PCBMER.

Принцип DDP (Downward Dependency Principle). Принцип зависимости сверху вниз в архитектуре PCBMER.

Принцип EAP (Explicit Association Principle). Принцип явной ассоциации в архитектуре PCBMER.

Принцип NCP (Neighbor Communication Principle). Принцип связей между соседями в архитектуре PCBMER.

Принцип UNP (Upward Notification Principle). Принцип уведомления снизу вверх в архитектуре PCBMER.

Связанность (coupling). См. *Зависимость*. Связанность обычно противопоставляется *связности*. Цель проекта — низкая *связность* и высокая связанность, однако низкая связанность часто приводит к более низкой (т.е. худшей) связности, и наоборот.

Связность (cohesion). "Показатель, измеряющий, насколько тесно связаны между собой линии кода, обеспечивающие конкретные функциональные свойства модуля. Связность относится к порядковому типу измерений и обычно выражается фразами "высокая связность" или "низкая связность" (`www.en.wikipedia.org/wiki/Cohesion`).

Сигнатура (signature). "Имя и параметр функционального свойства, например операции или сигнала. В сигнатуру могут входить типы возвращаемых значений (для операций, но не для сигналов)" (Rumbaugh et al., 2005).

Служба JMS (Java Messaging Service). Служба обмена сообщениями в языке Java.

Спецификация требований (requirements specification). Подробная, ориентированная на заказчиков спецификация функциональных и нефункциональных критериев, которым должна удовлетворять разрабатываемая информационная система. Спецификации требований записываются в техническом задании, и поэтому эти названия документа часто являются синонимами.

Требуемые интерфейсы (required interfaces). "Определяет услуги, необходимые классификатору, для того чтобы выполнять свои функции и обязанности перед своими клиентами" (UML, 2005).

Многовариантные тесты

MT1. Что такое MVC?

а. Модель.

б. Метамодель.

в. Среда программирования.

г. Все перечисленное выше.

MT2. Что из перечисленного ниже является именем яруса в архитектуре Core J2EE?

а. Бизнес.

б. Интеграция.

в. Представление.

г. Все перечисленное выше.

MT3. Что из перечисленного ниже является уровнем в архитектуре PCBMER, представляющем классы данных и объекты значений, предназначенные для отображения на экране дисплея?

а. Сущность.

б. Компонент.

в. Представление.

г. Ни один из перечисленных выше вариантов не подходит.

MT4. Как называется процесс отсылки уведомления объектам-подписчикам?

а. Обработка событий.

б. Пересылка.

в. Делегирование.

г. Ни один из перечисленных выше вариантов не подходит.

MT5. Какой из перечисленных ниже подходов к выявлению классов работает с концепцией нечеткого класса?

а. CRC.

б. Подход, основанный на общих шаблонных классах.

в. Подход, основанный на прецедентах использования.

г. Ни один из перечисленных выше вариантов не подходит.

MT6. В каких из категорий агрегаций наблюдаются свойства транзитивности и асимметрии?

а. Агрегация *Owns*.

б. Агрегация *Has*.

в. Агрегация *ExclusiveOwns*.

г. Все перечисленное выше.

MT7. Какое из перечисленных ниже отношений всегда связывает два прецедента использования?

а. Ассоциация.

б. Агрегация.

в. Расширение.

г. Все перечисленное выше.

MT8. Как называется событие, означающее, что сторожевое условие выполнено?

а. Изменение.

б. Временно*е* событие.

в. Сигнал.

г. Вызов.

Вопросы

B9. Рассмотрите семь принципов архитектуры PCBMER, описанные в разделе 4.1.3.2. Два из них относятся к иерархическим, а не сетевым архитектурам. Какие? Аргументируйте свой ответ.

B10. Рассмотрите рис. 4.23, на котором показаны классы, сгруппированные в три пакета, а также коммуникационные зависимости между этими классами.

Покажите, как можно ослабить эти зависимости с помощью интерфейса, включенного в каждый из этих пакетов. Каждый интерфейс должен определять открытые сервисы, предоставляемые соответствующим пакетом. Связи между пакетами осуществляются с помощью этих интерфейсов.

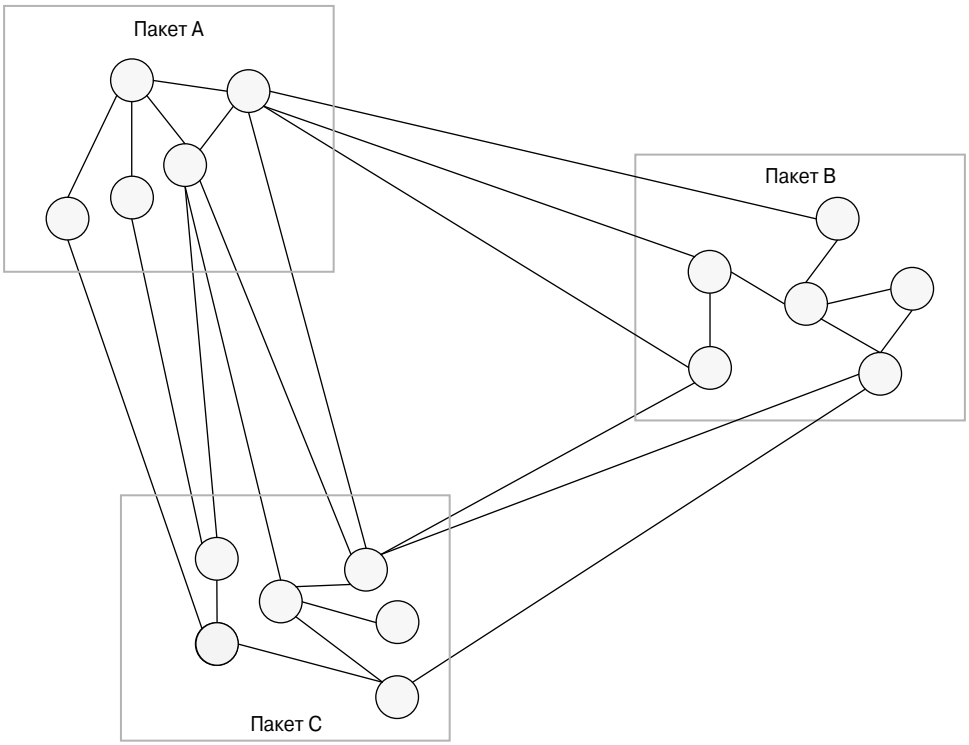


Рис. 4.23. Сеть взаимодействующих объектов. Источник: Maciazhek and Liong (2005). Воспроизводится с разрешения компании Pearson Education Ltd.

B11. Каким образом функциональные требования и требования к данным, выявленные на этапе установления требований, связаны с моделями состояний, поведения и изменения состояний, разрабатываемыми на этапе спецификации требований?

B12. Приведите доводы за и против использования CASE-инструментов для спецификации требований.

B13. Объясните, в чем заключаются основные различия четырех подходов к выявлению классов.

B14. Обсудите и сравните процессы, связанные с выявлением и спецификацией атрибутов классов и операций классов. Как следует моделировать атрибуты и операции: в параллель или по отдельности? Почему?

B15. Автоматическое генерирование кода для рис. 4.8 (см. пример 4.8, раздел 4.2.2.2.1) закончится неудачей. Почему?

B16. Обратитесь к модели классов, приведенной на рис. 4.10 (см. пример 4.10, раздел 4.2.4.3). Объясните, почему атрибут `EMovie.isInStock` моделируется как *производный*, а атрибут `EVideoMedium.percentageExcellentCondition` — как *статический*?

B17. Использование тернарных и производных ассоциаций в моделях анализа нежелательно. Почему? Приведите примеры.

B18. Обратитесь к модели классов, приведенной на рис. 4.8 (см. пример 4.8, раздел 4.2.2.2.1). Рассмотрите верхнюю часть модели, в которой определяются классы и ассоциации между классами `EPostalAddress`, `ECourierAddress`, `EOrganization` и `EContact`. Предложите альтернативные пути моделирования этих же требований. Можно ли придать модели бо́льшую гибкость, чтобы ее можно было приспособить к возможным изменениям в требованиях? Приведите доводы за и против альтернативных решений.

B19. Обратитесь к модели классов на рис. 4.9 (см. пример 4.9, раздел 4.2.3.3). Объясните, как изменится семантика модели, если класс `EAcademicRecord`, связанный с ассоциацией *Takes*, моделировать как "ассоциацию в роли класса"?

B20. Наследование без полиморфизма возможно, но не очень полезно. Почему? Приведите примеры.

B21. Какие из моделей UML полезны при спецификации поведения? Объясните, в чем заключаются сильные стороны каждой из этих моделей и каким образом они взаимосвязаны при определении поведения системы?

B22. Объясните, в чем состоит различие прецедентов использования и деловых функций (или деловых транзакций)?

B23. Объясните разницу между решением (ветвлением) и разделением на диаграмме деятельности. Приведите пример.

B24. Приведите пример отношений «`include`» и «`extend`» между прецедентами. В чем их основное различие?

B25. Приведите пример класса с несколькими атрибутами. Объясните, какие из атрибутов могут инициировать переход между состояниями, а какие из атрибутов индифферентны к изменениям состояний?

Упражнения. Магазин видеокассет

Дополнительные требования

Рассмотрите следующие дополнительные требования к системе управления магазином видеокассет.

- За кассеты и диски, возвращенные позже срока, взимается дополнительная плата за период, превышающий срок проката. Каждый носитель обладает уникальным идентификационным номером.
- Фильмы заказываются у поставщика, который, в общем случае, может поставить кассеты и диски в течение одной недели. Обычно один заказ делается на несколько фильмов.
- Зарезервировать можно те фильмы, которые заказаны у поставщика и/или все копии которых находятся в прокате. Можно также зарезервировать те фильмы, которых нет в запасе и которые не заказаны у поставщика; при этом с клиента требуется задаток за один период проката.
- Клиент может также сделать несколько предварительных заказов, однако для каждого зарезервированного фильма готовится отдельный запрос на резервирование. Резервирование может быть отменено из-за отсутствия реакции со стороны клиента, более точно, в течение одной недели с момента, когда клиенту было сообщено о возможности взять фильм напрокат. Если за фильм был внесен задаток, он записывается на счет клиента.
- База данных хранит обычную информацию о поставщиках и клиентах, т.е. адреса, телефонные номера и т.д. В каждом заказе поставщику указываются заказываемые фильмы, их количество, форматы кассеты/диска, а также дата ожидаемой доставки, отпускная цена, возможные скидки и т.д.
- Когда кассета или диск возвращается клиентом или поступает от поставщика, вначале удовлетворяются предварительные заказы. Работники магазина устанавливают контакт с клиентами, сделавшими предварительный заказ. Для правильной обработки резервирования фильмов информация, связанная с резервированием, обновляется дважды: после установления контакта с клиентом, когда ему сообщается о поступлении фильма, и после сдачи фильма клиенту напрокат. Эти шаги гарантируют правильное проведение операции резервирования.
- Клиент может взять несколько кассет или дисков, однако каждому взятому носителю ставится в соответствие отдельная запись. Для каждого выдаваемого напрокат фильма фиксируются дата и время выдачи, установленный и фактический срок возврата. Позже запись о прокате обновляется,

чтобы отразить факт возврата видеофильма и факт окончательного платежа (или возврата денег). Кроме того, запись хранит информацию о продавце, отвечающем за прокат фильма. Детальная информация о клиенте и по прокату хранится в течение года, чтобы можно было легко определить уровень доверия к клиенту. Старая информация по прокату сохраняется в течение года в целях проведения аудита.

- Все операции выполняются с использованием наличности, электронного перевода денег или кредитных карточек. От клиентов требуется внести плату за прокат при выдаче кассет/дисков.
- Если кассета или диск возвращены позже установленного срока (или не могут быть возвращены по каким-либо причинам), плата взимается либо со счета клиента, либо принимается непосредственно от клиента.
- Если кассета или диск задержаны более чем на два дня, клиенту отправляют уведомление о задержке. После отправки двух уведомлений о задержке одной и той же кассеты или диска клиента предупреждают о том, что он является "нарушителем" и при следующем его обращении в магазин руководство рассматривает вопрос о снятии с него статуса "нарушителя".

MB1. Рассмотрите дополнительные требования и вернитесь к примеру 4.2 (см. раздел 4.2.1.1.7). Какие новые классы можно получить на основе анализа расширенных требований?

MB2. Рассмотрите дополнительные требованиям, упражнение У1 и пример 4.10 (см. раздел 4.3.1.2). Расширьте модель классов, приведенную на рис. 4.10 (см. пример 4.10), чтобы учесть расширенные требования. Покажите классы и отношения.

MB3. Рассмотрите дополнительные требования и пример 4.15 (см. раздел 4.3.1.2). Изучите неформальную спецификацию для прецедента `Прокат видеокассет`, приведенную в табл. 4.4 (см. пример 4.15). Пропустите последний абзац раздела "Основной поток" таблицы (он относится к прецеденту `Обслуживание заказчика`). Разработайте отдельную диаграмму прецедентов для отображения дочерних прецедентов `Прокат видеокассет`.

Упражнения. Управление взаимоотношениями с заказчиками

УВЗ1. Вернитесь к примеру 4.8 (см. раздел 4.2.2.2.1). Постройте диаграмму конечного автомата для класса `EEvent`.

УВЗ2. Вернитесь к примеру 4.8 (см. раздел 4.2.2.2.1). Рассмотрите классы `EOrganization`, `EContact`, `EPostalAddress` и `ECourierAddress`. В качестве расширения модели, показанной на рис. 4.8, допустите, что организация имеет иерархическую структуру, т.е. может состоять из более мелких подразделений. Уточните модель класса, используя обобщение, и в то же время расширьте ее, чтобы учесть иерархию организации.

Упражнения. Зачисление в университет

Дополнительные требования

Рассмотрите следующие дополнительные требования к системе зачисления в университет.

- Университет разделен на факультеты, которые в свою очередь разделены на кафедры. Преподаватели работают на одной из кафедр.
- Присвоение большинства научных степеней является прерогативой одного факультета, но некоторые научные степени находятся в совместном ведении двух и более факультетов.
- Новые студенты получают по почте уведомление о приеме и инструкции по записи на курсы. Студенты, продолжающие учебу и имеющие право заново записаться на курсы, получают (также по почте) инструкции по записи вместе с уведомлением об экзаменационных результатах.
- Инструкции по записи на курсы включают расписание аудиторий для предлагаемого курса.
- В период записи на курсы студенты могут получить консультацию по вопросу составления программы обучения у научного руководителя на факультете, на котором они числятся.
- Студенты не ограничены в изучении только тех курсов, которые предлагаются их факультетом, и в любой момент могут сменить факультет, на котором они числятся (заполнив форму изменения программы, которую можно получить в деканате).

- Для того чтобы записаться на определенные курсы, студент должен сначала пройти обязательные курсы, получив требуемые оценки (простого прослушивания курса может быть недостаточно). Студент, который не смог удовлетворительно пройти курс, может попытаться сделать это вновь. Для записи на курс в третий раз или в случае требования освобождения от обязательных курсов необходимо получить согласие представителя соответствующего деканата.
- Если студенты записываются на учебные курсы, которые в совокупности дают за год менее чем 18 условных баллов, то они относятся к категории студентов заочной формы обучения (большинство курсов дают 3 балла).
- Для того чтобы получить возможность записаться на программу курсов, которые в данном семестре дают в совокупности более 14 баллов, требуется специальное разрешение.
- За каждым курсом обучения закреплен ответственный преподаватель, однако к нему могут быть привлечены дополнительные преподаватели. В каждом семестре за курс может отвечать другой преподаватель, а каждый курс в течение семестра могут вести разные преподаватели.

ЗУ1. Обратитесь к приведенным выше дополнительным требованиям и к примеру 4.1 (см. раздел 4.2.1.1.7). Какие новые классы можно вывести на основе анализа расширенных требований?

ЗУ2. Обратитесь к приведенным выше дополнительным требованиям, к упражнению ЗУ1 и примеру 4.9 (см. раздел 4.2.3.3). Расширьте модель классов, приведенную на рис. 4.9, чтобы учесть расширенные требования. Покажите классы и отношения.

ЗУ3. Обратитесь к приведенным выше дополнительным требованиям, к упражнению, примеру 4.13 (см. раздел 4.3.1.3). Предложите способ расширить модель прецедентов, приведенную на рис. 4.13 (пример 4.13), чтобы учесть расширенные требования.

ЗУ4. Обратитесь к приведенным выше дополнительным требованиям и к примеру 4.18 (см. раздел 4.3.3.3). Каким образом следует расширить диаграмму последовательностей (см. раздел 3.4.2), расширяющую диаграмму последовательностей, приведенную на рис.4.18 (пример 4.18), чтобы включить проверку обязательных условий записи на курсы — студент записывается на курс только в том случае, если он прошел обязательные курсы?

ЗУ5. Обратитесь к примеру 4.18 (раздел 4.3.4.3) и решению к приведенному выше упражнению ЗУ4. Воспользуйтесь расширенной диаграммой последовательностей для введения новых операций в релевантные классы, включая два класса, приведенные на рис. 4.20.

Ответы на контрольные вопросы

Контрольные вопросы 4.1

КВ1. Соответствие определенной архитектурной модели.

КВ2. Объекты контроллера.

КВ3. Ярус интеграции.

КВ4. Уровень ресурсов.

Контрольные вопросы 4.2

КВ1. Метод CRC (класс–ответственность–сотрудники) предназначен для выявления классов.

КВ2. Имена ролей можно использовать для объяснения более сложных ассоциаций, в частности *самоассоциаций* (рекурсивных ассоциаций, связывающих между собой объекты одного и того же класса).

КВ3. Транзитивность означает, что, если объект подмножества C является частью объекта подмножества B, а B является частью другого объекта подмножества A, то объект C является частью объекта A.

КВ4. Принцип заменимости.

Контрольные вопросы 4.3

КВ1. Нет. Представить параллельные потоки управления могут лишь диаграммы деятельности.

КВ2. Да. Это возможно, поскольку диаграммы деятельности не визуализируют явно классы объектов, выполняющих действие.

КВ3. Сигнал.

КВ4. Диаграммы последовательности.

Контрольные вопросы 4.4

КВ1. Нет. Диаграммы конечных автоматов не отражают последовательности изменений состояний, даже если такие последовательности ожидаются.

КВ2. Не обязательно. Можно указать входное действие, которое должно быть полностью завершено до изменения состояния.

Ответы к многовариантным тестам

МТ1. г

МТ2. г

МТ3. б

МТ4. а

МТ5. г (ни один вариант не подходит)

МТ6. г

МТ7. в

МТ8. а

Ответы на вопросы с нечетными номерами

В1

Эти два принципа — NCP и APP.

Принцип NCP запрещает непосредственную связь между удаленными уровнями. Это порождает иерархические последовательности связей. Например, если `A` и `C` — два удаленных уровня, но уровень `B` является соседним по отношению к уровням `A` и `C`, то уровень `A` должен иметь связь с уровнем `B`, чтобы получить доступ к услугам уровня `C`. В сети уровни `A` и `C` имели бы прямой контакт.

Принцип APP смягчает строгость принципа NCP. Он иногда неудобен и порождает длинные цепочки сообщений между удаленными уровнями, особенно если расстояние между ними велико. Включение отдельного уровня интерфейсов в соответствии с принципом APP позволяет передавать сообщения между удаленными уровнями, избегая нежелательных зависимостей.

По общему признанию, принцип UNP также допускает иерархии и препятствует образованию сетей. Однако основное преимущество принципа UNP заключается в том, что он сокращает зависимость между объектами, исключая зависимости снизу вверх, а не просто предотвращает образование сетей. По этой причине принцип UNP не включен в ответ.

В3

Требования к функциям и данным, зафиксированные на этапе *определения требований* (requirements determination), формулируются в виде *перечня услуг* (service

statement). На этапе *спецификации требований* (requirement specification) они моделируются в виде диаграмм состояний, поведения и изменения состояний.

Модели спецификации описывают разные аспекты одного и того же набора требований. Конкретное требование можно моделировать с помощью всех трех диаграмм, каждая из которых описывает особый ракурс проекта.

Можно сказать, что *модели состояния* (state models) выражают большинство *требований к данным* (data requirements), а *модели поведения* (behavior models) — большинство функциональных требований. *Модели изменения состояний* (state change models) относятся как к требованиям, предъявляемым к данным и функциям, так и к системным ограничениям (нефункциональным требованиям).

Кроме того, классы в модели состояния можно дополнить подробным проектом операций. Окончательная модель состояний фиксирует всю внутреннюю работу, а также выражает требования, предъявляемые к данным и функциям.

B5

Существуют четыре подхода к выявлению классов.

- Метод именных групп.
- Метод общих шаблонных классов.
- Метод прецедентов использования.
- Метод CRC (class–responsibility–collaborators).

Метод *именных групп* (noun phrase approach) является самым ранним и самым быстрым. Поиск именных групп в техническом задании можно выполнять с помощью лексических анализаторов. Однако итоговый словарь классов может оказаться обманчивым и неточным.

Метод общих шаблонных классов (common class patterns approach) оказывается удобным, если его сочетать с другими подходами, но самостоятельно он не в состоянии привести к полному списку классов. Этому подходу недостает системных ориентиров, содержащихся в списке требований (метод именных групп) и наборе прецедентов использования (метод прецедентов использования) и вырабатываемых на семинарах с пользователями (метод CRC).

Метод прецедентов использования (use case-driven approach) требует предварительных инвестиций в разработку прецедентов использования, анализ которых позволяет выявить классы. Перед созданием окончательного списка классов множества классов, выявленных при изучении всех прецедентов использования, необходимо объединить. В этот список следует включать только классы, которые непосредственно запрашиваются прецедентами использования. Это может помешать дальнейшей эволюции системы, поскольку модели классов должны строго соответствовать функциональным свойствам системы, описанным в текущих прецедентах использования.

Подход CRC (CRC approach) является наиболее объектно-ориентированным среди четырех перечисленных методов. Он идентифицирует классы, необходимые для

реализации обсуждаемых прецедентов использования. Основное внимание в этом подходе уделяется поведению класса (операциям), что может привести к выявлению так называемых "поведенческих" классов (в противоположность "информационным" классам). Однако этот метод может также определить атрибуты каждого класса. В этом смысле подход CRC относится к относительно низкому уровню абстракции и может использоваться в сочетании с тремя предыдущими подходами.

B7

Автоматическое генерирование может оказаться неудачным, поскольку ассоциации между классами `EEvent` и `EEmployee` не имеют уникальных ролевых имен, а CASE-инструмент не способен генерировать экземпляры переменных с разными именами для ассоциаций с одним и тем же классом. Например, невозможно создать три переменные с именем `theEEmployee` в классе `EEvent`. Необходимы переменные `theCreatedEmp`, `theDueEmp` и `the CompletedEmp`.

B9

Семантическое моделирование должно быть однозначным. Каждая модель должна иметь только одну интерпретацию. *Тернарные ассоциации* (ternary associations) сами по себе порождают многочисленные интерпретации и, следовательно, нежелательны. Рассмотрим тернарную ассоциацию `FamilyUnit` между классами `Man`, `Woman` и `Residence` (Maciaszek, 1990). Предположим, что кратность ассоциации равна единице на всех концах, за исключением того факта, что членство класса `Residence` в ассоциации является необязательным, т.е. ассоциация `FamilyUnit` может существовать и без объекта класса `Residence`.

Такие тернарные ассоциации порождают сложные вопросы. Может ли один и тот же человек участвовать в нескольких ассоциациях `FamilyUnit`, если он был женат дважды. Может ли в одном и том же доме жить несколько семей? Что если семья разделена и дом принадлежит одному из супругов? Что если человек заключил новый брак и привел в старый дом новую супругу (супруга)?

Решением задачи является отказ от тернарных ассоциаций и замена их тремя бинарными ассоциациями — `Man–Women`, `Man–Residence` и `Women–Residence`. При *обобщении* ассоциация `Person–Residence` может заменить две последние ассоциации, что еще больше упростит модель.

Хорошая аналитическая модель не должна быть избыточной. *Производную ассоциацию* (derived association) можно удалить из модели без какой-либо потери информации. Производная ассоциация представляет собой особый вид ограничений в модели, который не обязательно явно изображать на диаграмме классов.

Рассмотрим следующий *цикл ассоциаций* (cycle of associations). Ассоциация `Department–Employee` является ассоциацией "один ко многим", как и ассоциации `Department–ResearchProject` и `ResearchProject–Employee` (Maciaszek, 1990). Изучение цикла показывает, что ассоциация `Department–`

ResearchProject является лишней. Мы всегда можем выяснить, какой исследовательский проект выполняет подразделение, найдя его сотрудников и пройдя по их связям с исследовательскими проектами.

Несмотря на то что ассоциация Department–ResearchProject в модели не является обязательной, аналитик может решить оставить ее, обосновывая это требованиями ясности проекта. Эту аргументацию можно усилить, сославшись на повышение производительности системы благодаря существованию прямой связи между классами Department и ResearchProject. В заключение отметим, что эта ассоциация является обязательной, если подразделение может сначала заключать контракт на выполнение исследовательского проекта и лишь после этого подбирать его участников.

B11

Этот вопрос довольно сложен. Если модель отражает один и тот же набор пользовательских требований, ее семантика не должна изменяться. Если новая модель не может отразить конкретную семантику, то недостающую часть следует выразить иначе, например с помощью описательного приложения. Иначе говоря, если новая модель является более выразительной, дополнительную семантику следует выводить из описательного приложения к исходной модели.

Рассмотрим модифицированную модель, показанную на рис. 4.24. Каждый экземпляр класса ассоциации EAcademyRecord имеет ссылку на объекты класса EStudent и ECourseOffering, а также два атрибута grade и courseCode. Идентичность каждого экземпляра класса EAcademyRecord обеспечивается ссылкой на него (что соответствует двум ролям ассоциации, –hasSTud и –takesCrsoff). Атрибут оценки показывает, сколько баллов может набрать студент, прослушав конкретный курс.

Обосновать наличие атрибута courseCode в классе EAcademicRecord намного сложнее. Можно сказать, что этот атрибут не является необходимым, поскольку каждый экземпляр класса ECourseOffering (на который указывает ссылка –takesCrsoff в объекте EAcademicRecord) является компонентом отдельного экземпляра ECourse. Следовательно, можно проследить за "указателем" от класса ECourseOffering к классу ECourse, а затем выяснить значение атрибута courseCode.

Альтернативная схема изображена на рис. 4.25. Здесь класс ECourseOffering содержит класс ECourse. Это агрегация "по ссылке", так как класс ECourse является изолированным, который в полной модели, вероятно, связан с другими отношениями. Факт, что атрибут courseCode теперь является новой частью класса ECourseOffering, также следует из анализа класса EAcademicRecord (атрибут courseCode оттуда удален).

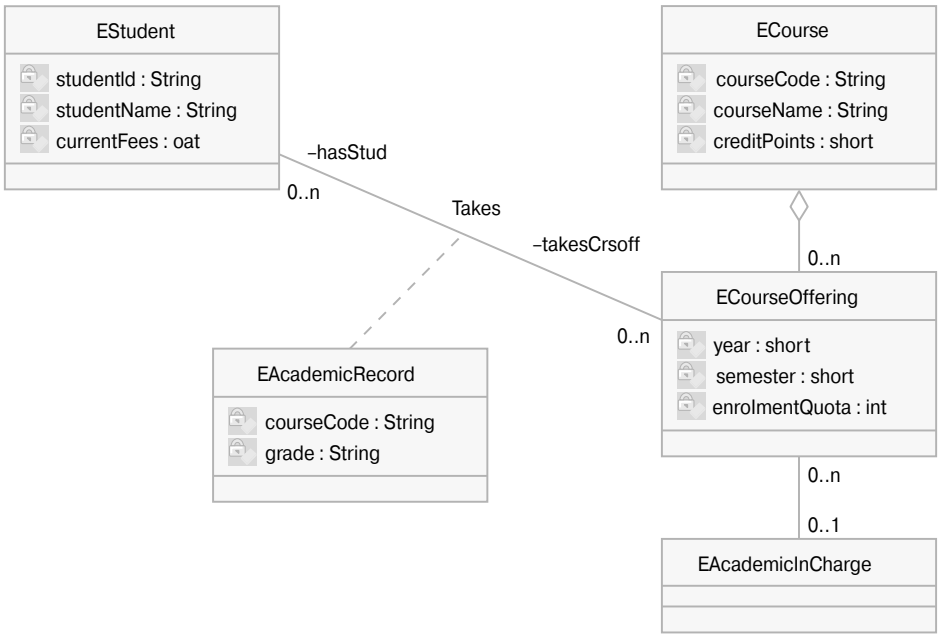


Рис. 4.24. Использование класса ассоциации для системы, управляющей зачислением в университет

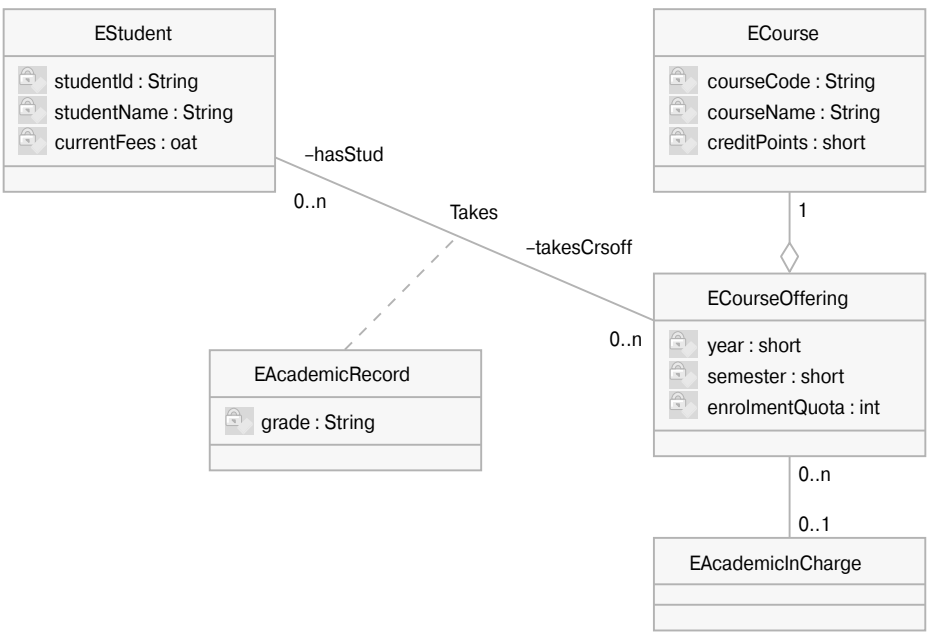


Рис. 4.25. Использование класса ассоциации после изменения направления ассоциации в системе, управляющей зачислением в университет

Основная разница между моделями, показанными на рис. 4.24 и 4.25, с одной стороны, и моделью, показанной на рис. 4.9, с другой стороны, заключается в том, что первые модели имеют ту же самую семантику, но с меньше избыточностью (без дублирования атрибутов). Класс `EAcademicRecord` не содержит атрибуты `year` и `semester`. Без атрибута `courseCode` также можно обойтись. Однако информационное содержание класса `EAcademicRecord` должно быть выведено из ассоциированных объектов, а не храниться как часть объекта класса `EStudent` (напомним, что в модели, приведенной на рис. 4.9 объект класса `EAcademicRecord` содержался в объекте класса `EStudent` "по значению").

B13

К моделям UML, которые лучше всех подходят для *спецификации поведения* (behavior specification), относятся:

- диаграммы деятельности;
- диаграммы последовательности;
- диаграммы коммуникации.

Диаграммы деятельности (activity diagrams) лучше всего подходят для вычислительных моделей, состоящих из последовательных и параллельных этапов. В зависимости от степени детализации этих действий в модели диаграмму деятельности можно использовать для вычислительных моделей на разных уровнях абстракции. Диаграмма деятельности может моделировать вычисления в прецеденте использования или в операции класса.

Диаграмма последовательностей (sequence diagram) демонстрирует обмен сообщениями между объектами, упорядоченными во времени. Как и диаграммы деятельности, их можно использовать с разной степенью детализации, описывая обмен обобщенными сообщениями на уровне всей системы или подробными сообщениями на уровне операций в классах. В отличие от диаграмм коммуникации, на диаграммах последовательностей не показываются отношения между объектами.

Диаграммы коммуникации (communication diagram) дополняют диаграммы последовательностей и показывают обмен сообщениями между объектами. При необходимости сообщения можно нумеровать, чтобы показать их порядок следования во времени. Диаграммы коммуникации больше подходят для проектирования, поскольку их можно использовать для моделирования прецедента использования или операций.

Все три модели имеют свои роли и спецификации поведения. Их можно использовать для демонстрации разных аспектов одного и того же функционального свойства, но чаще они используются для описания разных функциональных свойств или одного и того же поведения на разных уровнях абстракции. Диаграммы последовательностей чаще используются для *анализа*, а диаграммы последовательностей — для проектирования.

B15

Решение (*ветка*) разделяет поток управления, исходящий из определенного действия, на несколько потоков, каждый из которых имеет отдельное *сторожевое условие*. Для того чтобы гарантировать наступление события, вызывающего разделение, сторожевые условия должны учитывать все возможности.

Ветвление — это поток управления, исходящий из одного действия и приводящий к нескольким действиям. Если действие-источник активно и возникает поток управления, то все целевые действия становятся активными (т.е. не имеют сторожевых условий).

На рис. 4.26 приведены примеры решения и ветвления в банковском приложении, управляющем клиентскими счетами.

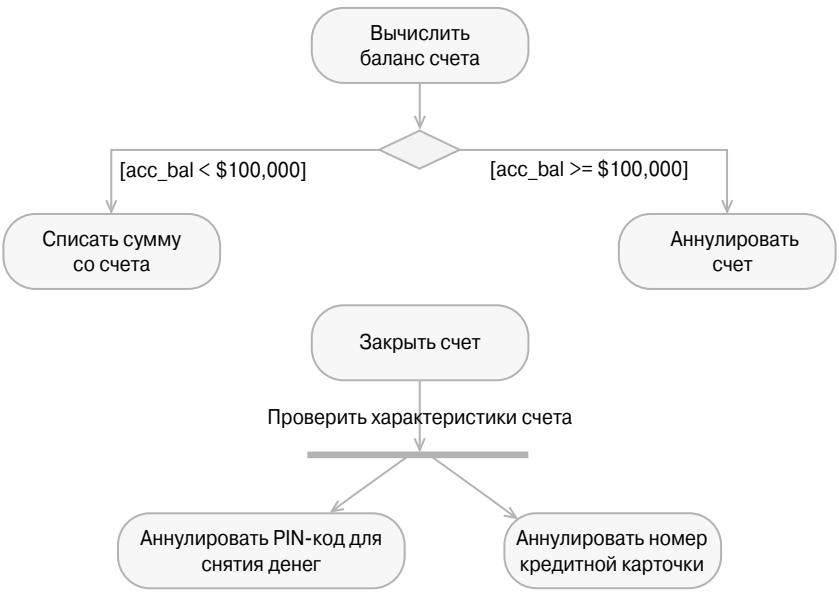


Рис. 4.26. Решение (ветка) и ветвление на диаграмме деятельности

B17

Рассмотрим класс ECampaign в системе прямого маркетинга по телефону (см. рис. 4.7). Атрибуты этого класса показаны на рис. 4.27.

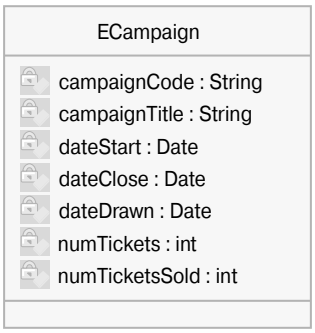


Рис. 4.27. Класс ECampaign *в системе прямого маркетинга по телефону*

Изменение состояний в классе можно анализировать с разных точек зрения. Для каждого ракурса можно построить отдельную модель состояний. В зависимости от выбранной точки зрения одни атрибуты рассматриваются, а другие игнорируются. Например, с точки зрения использования системы прямого маркетинга по телефону для продажи лотерейных билетов среди спонсоров объект класса ECampaign может находиться в состоянии Билеты в наличии или Все билеты проданы. Для того чтобы определить эти состояния, мы интересуемся лишь атрибутами numTickets и numTicketsSold. Остальные атрибуты можно игнорировать.

Аналогично, если известно, что объект класса ECampaign (Кампания) может находиться в состоянии Активна или Завершена, необходимо обратить внимание на атрибуты dateStart и dateClose. Если к тому же нас интересует моделирование состояния Завершена из-за отсутствия билетов, то следует дополнительно рассмотреть атрибуты numTickets и numTicketsSold.

Объяснение упражнений. Зачисление в университет

ЗУ1

Список предлагаемых классов приведен в табл. 4.5. Имена классов, установленных в примере 4.1 (см. табл. 4.1 раздела 4.2.1.1.7), подчеркнуты. Имя нечеткого класса `FullTimeStudent` приведено в квадратных скобках, чтобы указать на то, что он является противоположностью класса `PartTimeStudent`.

Таблица 4.5. Дополнительные классы — кандидаты на включение в систему зачисления в университет

Релевантные классы	Нечеткие классы
Course	CompulsoryCourse
Degree	ElectiveCourse
Student	StudyProgram
CourseOffering	NewStudent
Division	ContinuingStudent
Department	AcceptanceForm
Academic	EnrolmentIstructions
	ExaminationResults
	ClassTimeable
	AcademicAdviser
	RegistrationDivision
	PrerequisiteCourse
	SpecialApproval
	(Special Permission)
	HeadDelegate
	PartTimeStudent
	[FullTimeStudent]
	AcademicInCharge

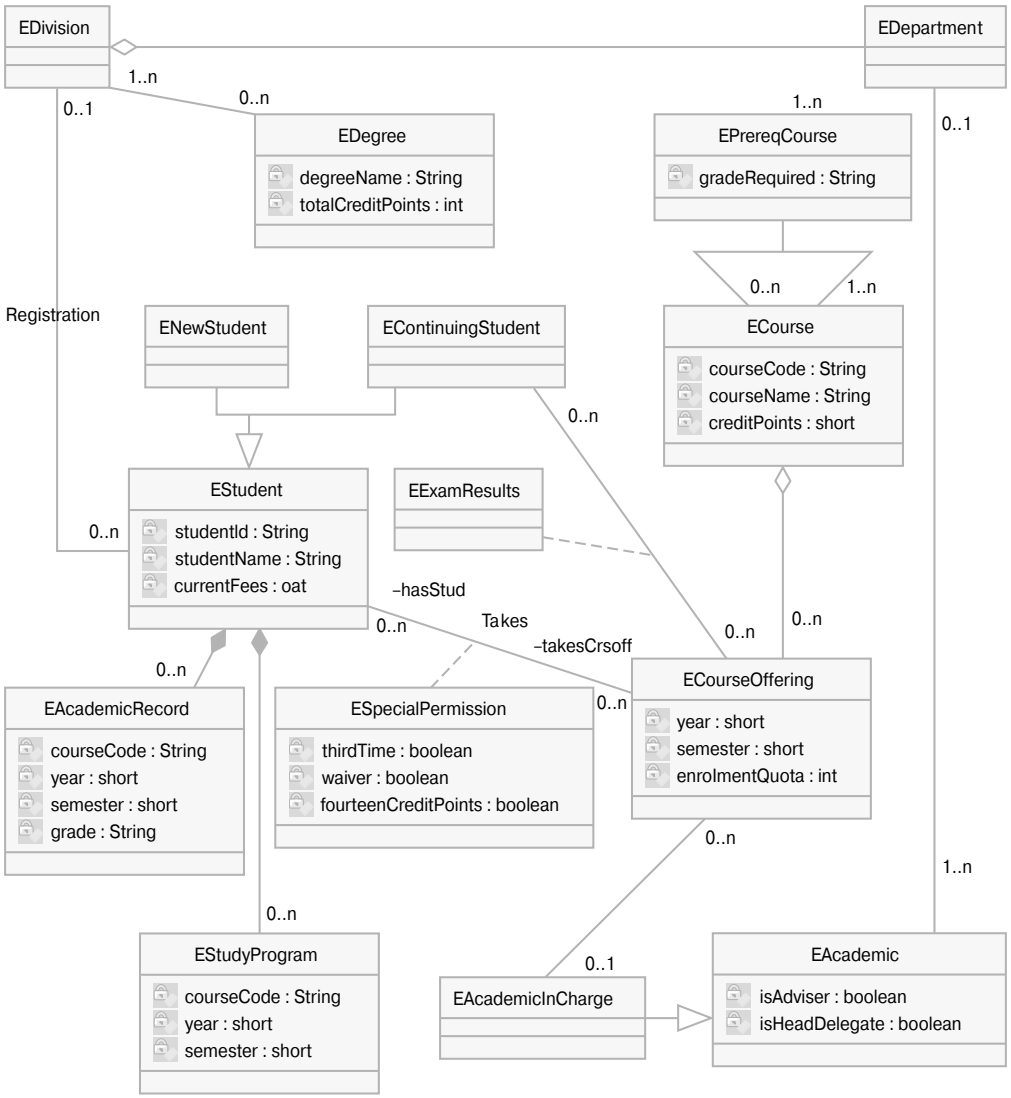


Рис. 4.28. Модель классов для системы, управляющей зачислением в университет

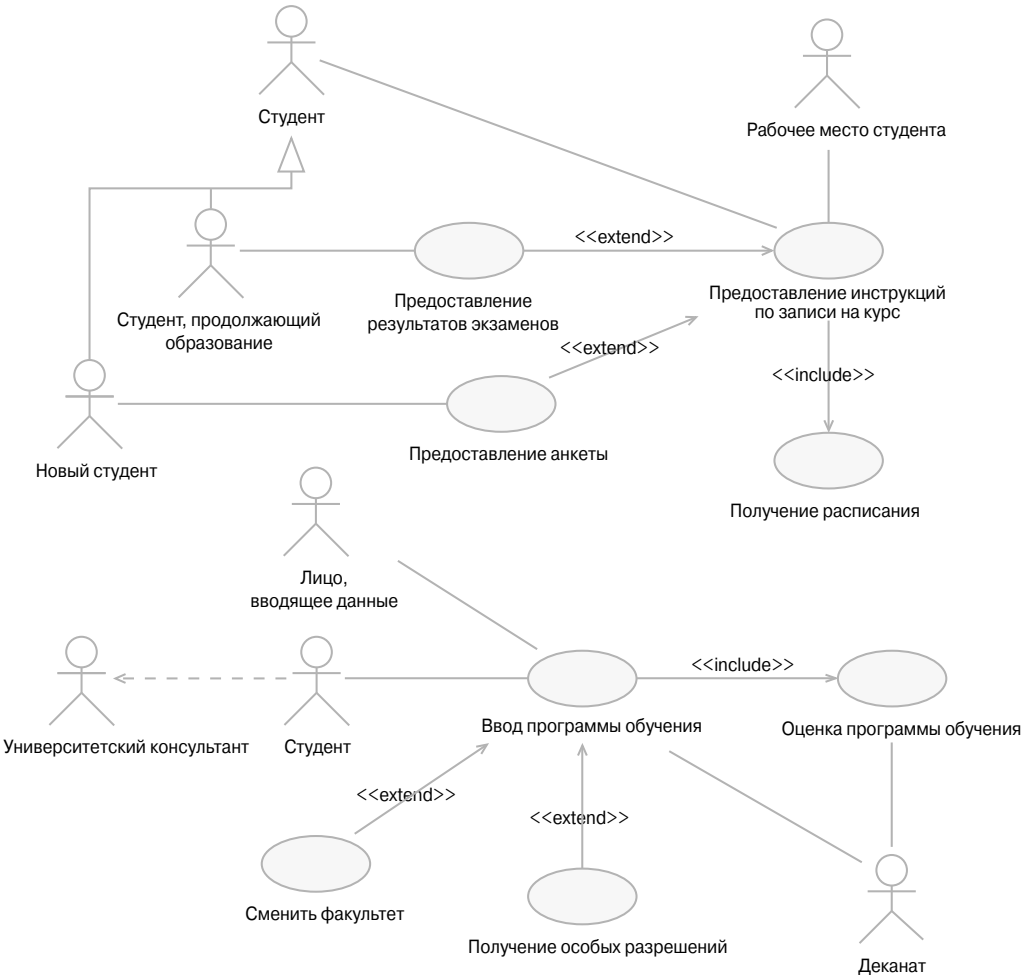


Рис. 4.29. Модель прецедентов использования системы, управляющей зачислением в университет

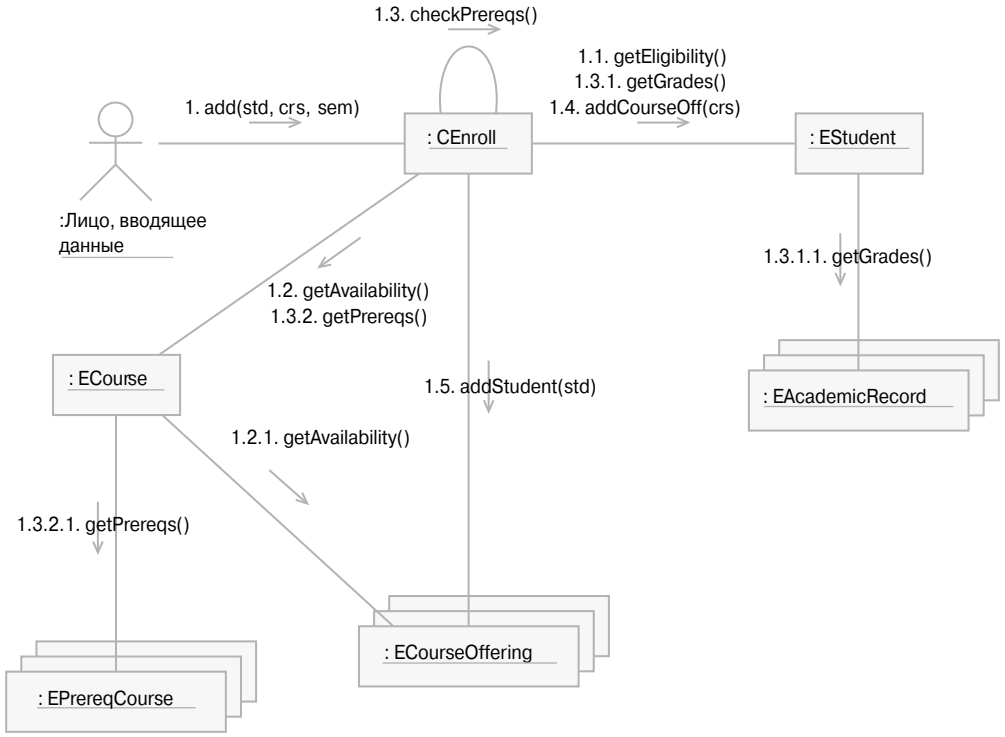


Рис. 4.30. Диаграмма коммуникации для системы, управляющей зачислением в университет

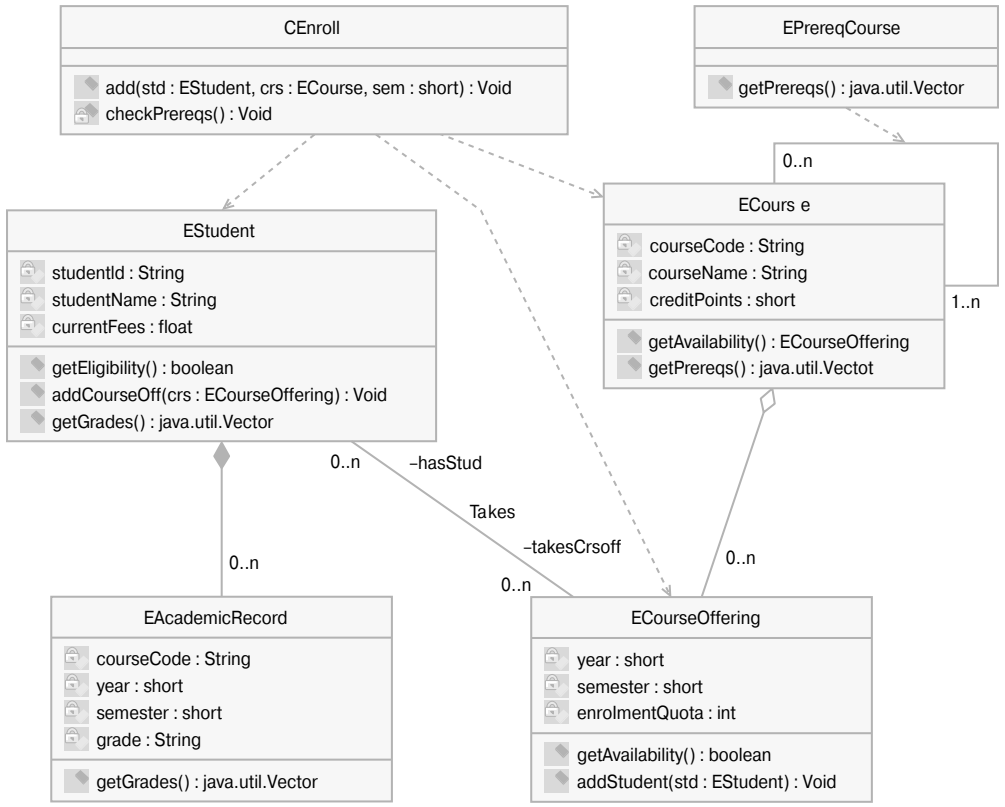


Рис. 4.31. Добавление операций в классы системы, управляющей зачислением в университет

ГЛАВА 5

Переход от анализа к проектированию

Цели

5.1. Углубленное моделирование классов
5.2. Углубленное моделирование обобщения и наследования
5.3. Углубленное моделирование агрегации и делегирования
5.4. Углубленное моделирование взаимодействий
Резюме
Ключевые термины
Многовариантные тесты
Вопросы
Упражнения
Упражнения. Регистрация времени
Упражнения. Затраты на рекламу
Ответы на контрольные вопросы
Ответы к многовариантным тестам
Ответы на вопросы с нечетными номерами
Объяснение упражнений с нечетными номерами
Объяснение упражнений. Регистрация времени

req_analysis_system design.indb 355 05.06.2008 14:43:11

Цели

В предыдущих двух главах картина визуального объектно-ориентированного моделирования нарисована в розовом свете. Примеры были нетребовательными, язык визуального проектирования — простым и привлекательным для использования, зависимости между моделями — очевидными. Мы вольно обращались с методами моделирования и не слишком заботились об альтернативных решениях.

Реалии разработки программного обеспечения куда сложнее. Как мы заметили в начале книги, простых решений для сложных проблем не существует. Объекты составляют основу современной технологии решения сложных проблем. А раз так, то объекты обязаны обеспечить и техническую глубину, соответствующую уровню сложности, к которой они обращены.

Эта глава призвана дать критическую оценку объектной технологии и ее пригодности к решению сложных проблем. Мы вводим передовые концепции в моделирование классов, иерархию классов, наследование и делегирование. На протяжении всей главы мы сравниваем, выносим решения, высказываем мнение и предлагаем альтернативные решения. Из-за своего технического характера многие темы, рассматриваемые в этой главе, переносятся прямо на системное проектирование. Это согласуется с универсальным характером UML, а также итеративным и наращиваемым процессом объектно-ориентированной разработки ПО.

Прочитав главу, читатели будут

- уметь настраивать язык UML для удовлетворения конкретных потребностей и технологических требований проекта;
- знать свойства языка UML для моделирования на более низких абстрактных уровнях;
- понимать, что не следует применять слепо некоторые мощные объектные технологические концепции;
- осознавать компромиссы моделирования, особенно в отношении обобщения и агрегации;
- владеть практическими навыками создания высокоуровневых моделей взаимодействия.

5.1. Углубленное моделирование классов

Концепции моделирования анализа, рассмотренные ранее, были достаточны для выработки завершенных моделей анализа. Однако обеспечиваемый этими концепциями уровень абстракции не исчерпывает всех возможных деталей, допу-

стимых в рамках моделирования анализа (т.е. деталей, которые не касаются аппаратных/программных решений, но обогащают семантику модели). UML включает нотацию для ряда дополнительных концепций, о которых мы упоминаем только вскользь или которых не касаемся вообще.

Дополнительные концепции моделирования включают **стереотипы** (stereotypes), **ограничения** (constraints), **производную информацию** (derived information), **видимость** (visibility), *квалифицированные ассоциации* (qualified associations), *ассоциативные классы* (association classes), *параметризованные классы* (parametrized class) и некоторые другие. Эти концепции необязательны. Многие модели могут быть вполне приемлемы без них. Их применение требует осторожности и точности, чтобы любой, кто попробует разобраться с моделями впоследствии, мог без труда понять намерения разработчика.

5.1.1. Механизмы расширения

Стандарт UML содержит набор концепций моделирования и обозначений, которые могут применяться в любых проектах, связанных с разработкой программного обеспечения. Однако эта универсальность означает, что для некоторых специфических и неортодоксальных проектов стандартных средств языка UML может оказаться недостаточно. Стандарт UML, как и любой стандарт, представляет собой "наименьший общий знаменатель". Было бы нецелесообразно усложнять язык UML экзотическими свойствами, предназначенными для узкоспециализированных предметных областей, инфраструктур программного обеспечения или языков программирования.

В то же время существует необходимость выйти за пределы встроенных возможностей языка UML. Для этой цели в языке UML предусмотрены механизмы расширения, такие как стереотипы, ограничения, **определения дескрипторов** и **меченые значения**. Согласованный набор расширений материализует в языке UML концепцию **профиля**, которая, в свою очередь, расширяет ссылочную **метамодель** (т.е. язык UML), ориентируясь на разные прикладные области или технологические платформы. Например, можно создавать профили языка UML для моделирования баз данных, разработки хранилищ данных или Web-систем.

5.1.1.1. Стереотипы

Стереотип (stereotype) расширяет существующий элемент моделирования в языке UML, изменяя его семантику. По существу стереотип не является новым элементом модели и не влияет на структуру UML. Он лишь обогащает содержание существующей нотации, расширяя и уточняя модель. Существуют различные способы применения стереотипов.

Обычно стереотипы *метятся* в моделях с помощью имен, заключенных в угловые кавычки, например «`global`», «`PK`», «`include`». Кроме того, стереотип можно изобразить с помощью *пиктограммы*.

Некоторые распространенные стереотипы являются встроенными — они заранее определены в UML. Некоторые CASE-инструменты включают готовые пиктограммы для встроенных стереотипов. Большинство CASE-инструментов обеспечивают возможность создания новых пиктограмм по желанию аналитика. На рис. 5.1 показаны примеры классов, стереотипизированных с помощью пиктограммы и метки соответственно, а также пример класса, не являющегося стереотипом.

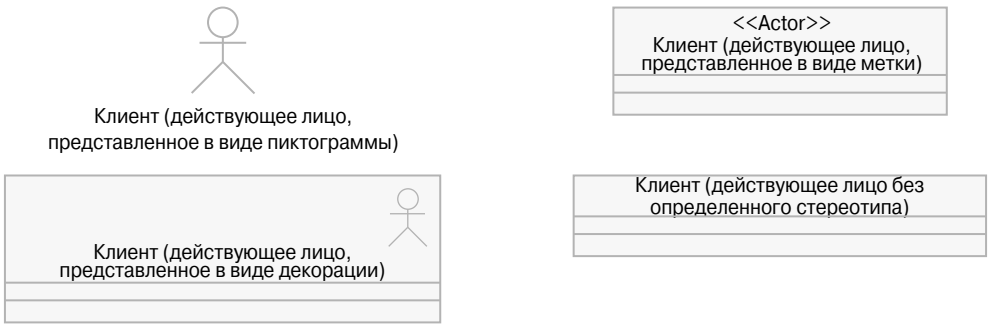


Рис. 5.1. Графическое представление стереотипов

Ограничение, гласящее, что стереотип расширяет семантику, но не структуру UML, не имеет большого значения. Главная отличительная черта любой объектно-ориентированной системы заключается в том, что она *полностью состоит из объектов* (объектами являются классы, атрибуты, методы и т.д.). Следовательно, стереотипизирование класса может привести в итоге к созданию нового элемента моделирования, который вводит новую категорию объектов.

5.1.1.2. Комментарии и ограничения

"**Комментарий** (comment) — это текстуальная аннотация, сопровождающая набор элементов... Компонент дает возможность сопровождать элементы разными примечаниями. Комментарии не имеют семантического значения, но могут содержать информацию, полезную для моделирования... Комментарии указываются в прямоугольнике, правый верхний угол которого загнут (так называемый "символ примечания"). Этот прямоугольник содержит текст комментария. Связь с каждым из аннотированных элементов указывается отдельной пунктирной линией" (UML, 2005).

"*Ограничение* — это условие, выраженное на естественном или машинном языке для объявления семантики элемента... Ограничение выражает дополнительную семантическую информацию, связанную с ограничиваемыми элементами. Ограничение — это условие, которое должно выполняться для корректного проектирования системы... Ограничение представляет собой текстовую строку в фигурных скобках ({}). Для элементов, примечание к которым выражается текстовой строкой (например, атрибутов и т.д.), строка ограничения может следовать вслед за тексто-

вой строкой в скобках. Если ограничение относится к отдельному элементу (классу, ассоциативному пути и т.п.), то оно указывается возле него, желательно ниже имени... Если ограничение относится к двум элементам (двум классам или двум ассоциациям), то оно указывается в виде пунктирной линии, проведенной между элементами, помеченными строкой ограничения (в скобках)" (UML, 2005).

Поскольку ограничения также могут выражаться в виде текстовых строк, разница между комментарием и ограничением заключается не столько в виде представления, сколько в семантических последствиях. Комментарий не имеет никакого семантического значения для модели. *Ограничение* имеет семантическое значение для модели и (в идеале) должно записываться на формальном языке. Фактически язык UML предлагает специализированный язык для этой цели — *Object Constraint Language* (OCL).

На диаграмме модели указываются только простые комментарии и ограничения. Более сложные комментарии и ограничения (с более длинными пояснениями) хранятся в CASE-хранилище как текстовые документы.

Некоторые виды ограничений в языке UML предусмотрены заранее. На рис. 4.8 (см. раздел 4.2.2.2 главы 4) показан пример использования встроенного *XOR-ограничения*. Ограничения, введенные проектировщиками, образуют *механизм расширения*.

Стереотипы часто путают с ограничениями. Действительно, разница между ними со временем стерлась. *Стереотип* часто используется для введения нового *ограничения* в модель — иногда представляющего ценность для проектировщика, но не имеющего непосредственной поддержки в языке UML.

Пример 5.1. Прямой маркетинг по телефону

Обратитесь к примеру 4.7 (см. разд. 4.2.1.2.3 главы 4). Рассмотрите классы `ECampaign и ECampaignTicket` (см. рис. 4.7) и установите отношение между ними. Добавьте в класс `ECampaign` операции для вычисления количества проданных и оставшихся билетов, а также для определения продолжительности кампании и количества дней, оставшихся до ее закрытия.

Добавьте в диаграмму информацию о том, что класс `ECampaign` отличается от бонусной кампании. Кроме того, напишите на диаграмме напоминание об операциях, которые следует добавить в класс `ECampaignTicket`.

Напомним, что часть второго требования, указанного в примере 4.7, гласит: "Все билеты пронумерованы. На протяжении отдельной кампании каждый билет имеет уникальный номер". Включите это требование в список ограничений. (Решая пример 4.7 (рис. 4.7), мы не стали фиксировать указанное выше ограничение, а лишь включили атрибуты `campaignCode` и `ticketNumber` как часть составного первичного ключа для класса `CampaihnTicket`.)

Простое расширение модели класса, удовлетворяющее требованиям примера 5.1, приведено на рис. 5.2. На рисунке показаны два комментария и одно ограничение.

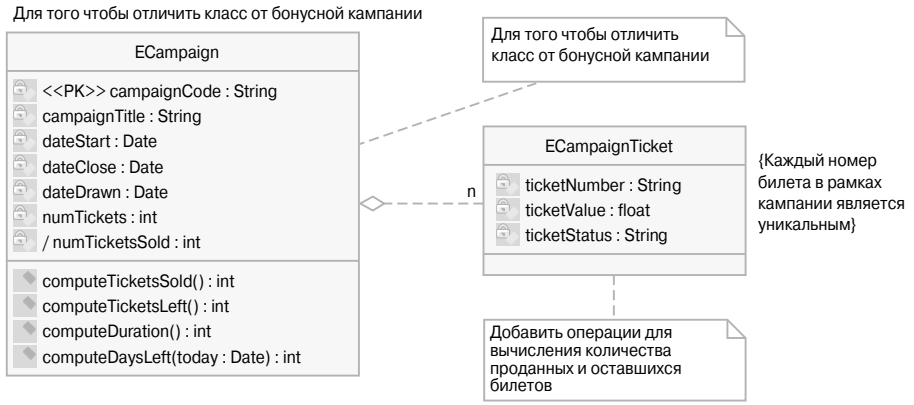


Рис. 5.2. Два комментария и ограничение для системы прямого маркетинга по телефону

Пример 5.2. Управление взаимоотношениями с заказчиками

Обратитесь к примеру 4.8 (см. разд. 4.2.2.2.1 главы 4). Предположим, что в систему введено новое ограничение: сотрудники не должны планировать мероприятия для себя. Это значит, что сотрудник, запланировавший мероприятие, должен отличаться от сотрудника, отвечающего за проведение мероприятия.

Расширьте соответствующую часть модели классов (см. рис. 4.8) таким образом, чтобы включить новое требование.

Решение примера 5.2 (рис. 5.3) выражается в виде ограничения, наложенного на ассоциативные линии. Они изображаются в виде прерывистой стрелки-указателя зависимости, проведенной от ассоциации `Создано` к ассоциации `Подлежит выполнению`. Это ограничение *привязано* к стрелке.

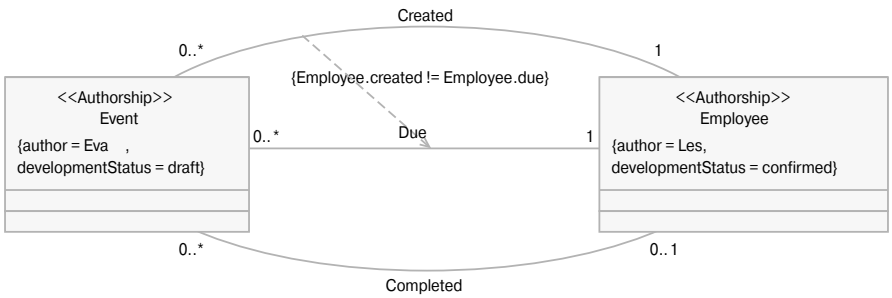


Рис. 5.3. Ограничения, наложенные на ассоциацию в системе управления взаимоотношениями с заказчиками

5.1.1.3. Примечания и дескрипторы

Понимание концепции дескрипторов требует различать *определение дескриптора* и *меченое значение*. *Определение дескриптора* (tag definition) — это свойство стереотипа. Оно показывается в виде прямоугольника, символизирующего атрибут класса, содержащего определение стереотипа. "*Меченое значение* (tagged value) — это пара "имя–значение", которую можно присоединить к элементу модели, использующему стереотип, содержащий определение дескриптора" (Rumbaugh et al., 2005).

Подобно примечаниям, *дескрипторы* (tag) представляют произвольную текстовую информацию в модели и записываются в фигурных скобках.

`tag = значение` — например:

```
{analyst = Les, developmentStatus = confirmed}
```

Поскольку дескриптор можно представить лишь в виде атрибута, определенного в стереотипе, стереотип с определениями дескрипторов должен быть определен в элементе модели (например, пакета) до того, как меченое значение будет приписано к конкретному экземпляру данного элемента модели.

Аналогично стереотипам и ограничениям, некоторые дескрипторы в UML определены заранее. Обычно дескрипторы используются для отображения информации об управлении проектом.

Пример 5.3. Управление взаимоотношениями с заказчиками

Обратитесь к примеру 4.8 (см. разд. 4.2.2.2.1 главы 4) и приведенному выше примеру 5.2. Определите стереотип `Authorship` и сделайте так, чтобы его можно было применить к любому элементу модели. Объявите дескрипторы `author` и `developmentStatus`. Примените определенные стереотипы к классам `Event` и `Employee`. Укажите некоторые меченые значения.

Предположим, что появилось новое требование: сотрудник, запланировавший задание, должен также создать первое событие для этого задания в рамках той же самой транзакции. Расширьте соответствующую часть модели класса (см. рис. 4.8 и 5.3) так, чтобы учесть новое требование. Используйте для этого примечание об ограничении, определенное в классе `Task`.

Расширенная модель для примера 5.3 показана на рис. 5.4. Стереотип `Authorship` (Авторство) с определениями дескрипторов показан в левом верхнем углу. Для выражения ограничения, присоединенного к элементу `Task` (Задание), используется примечание. (В идеале примечание об ограничении должно быть присоединено ко всем трем ассоциациям, но это требование трудно реализовать с помощью CASE-инструментов. Причина заключается в том, что ограничение относится не к примечанию, а к элементу модели, например, к классу или ассоциации.)

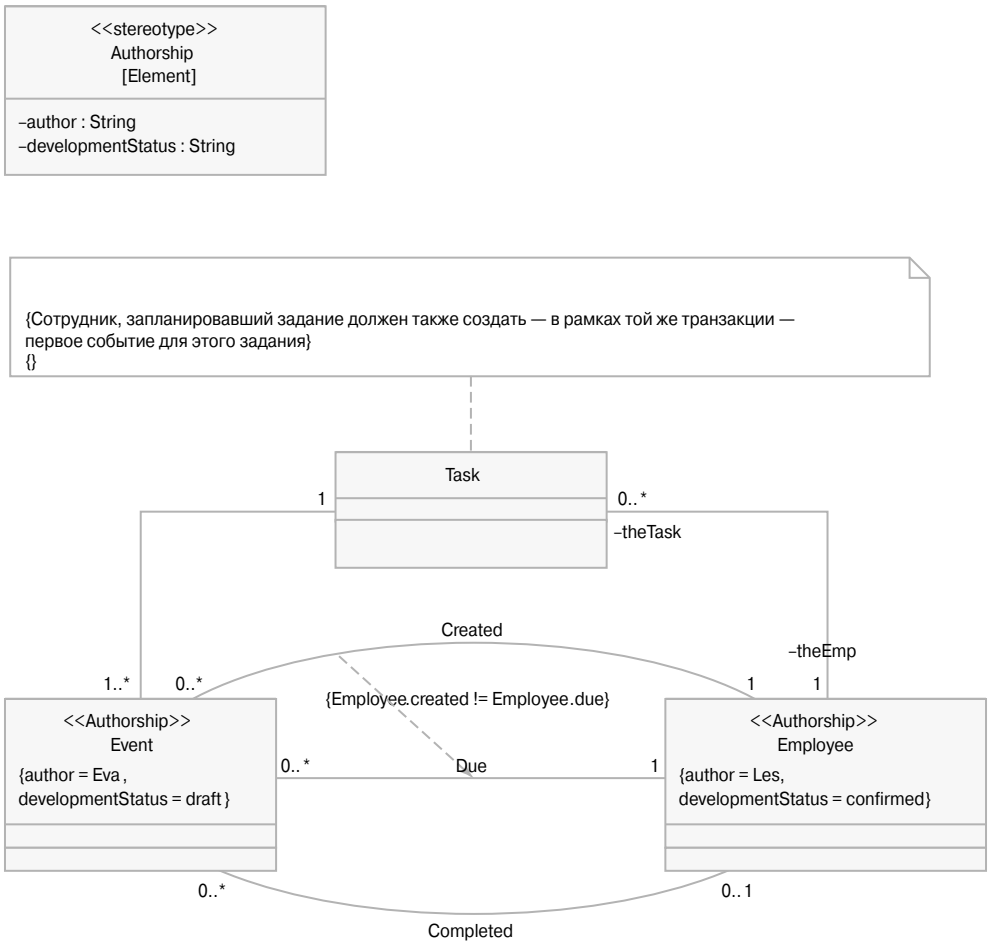


Рис. 5.4. Дескриптор и ограничения, наложенные на систему управления взаимоотношениями с заказчиками

5.1.2. Видимость и инкапсуляция

Концепция *видимости* (visibility) и связанное с ней понятие *инкапсуляции* (encapsulation) рассматриваются в приложении А в ракурсе видимости классов, т.е. *видимости атрибутов* (см. раздел А.3.1.2 приложения А) и *видимости операций* (см. раздел А.3.2.2 приложения А). Видимость класса означает, что другой класс имеет доступ к его элементам. В приложении А описывается *открытая* (public) и *закрытая* (private) видимость *атрибутов* и *операций*. В стандарте UML предусмотрены еще два вида видимости — *защищенная* (protected) и *пакетная* (package).

Точно так же можно определить видимость *класса*, *интерфейса*, *ассоциации* или *пакета* (в противоположность видимости элементов класса, т.е. *атрибутов* и *операций*). В *ассоциации* видимость применяется к *именам ролей* (см. раздел А.3.1.1

приложения А). Поскольку имена ролей (называемые в языке UML 2.0 *именами полюсов ассоциации*) реализуются как атрибуты классов, видимость имен ролей означает, что результирующие атрибуты можно использовать в выражениях доступа для перехода вдоль ассоциации. В *пакете* понятие видимости применяется к элементам, содержащимся в пакете, например к *классам*, *ассоциациям* и *вложенным пакетам*. Это значит, что другие пакеты имеют доступ к элементам данного пакета.

Перечислим полный набор маркеров видимости для атрибутов и операций классов:

+ — для открытой видимости;
– — для закрытой видимости;
— для защищенной видимости;
~ — для пакетной видимости.

CASE-инструменты часто заменяют эти довольно невыразительные обозначения более яркими и привлекательными графическими маркерами. Пример альтернативных обозначений приведен на рис. 5.5.

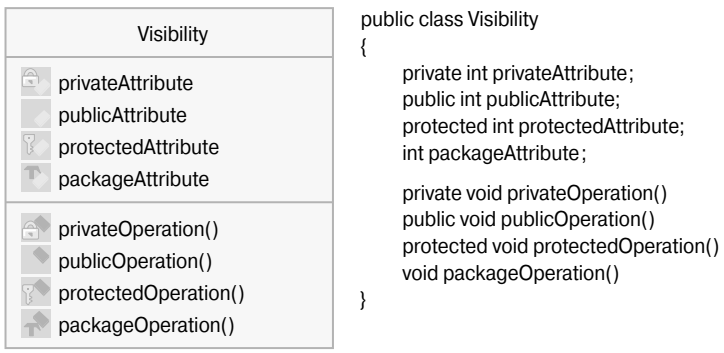


Рис. 5.5. Обозначения видимости в CASE-инструменте и соответствующий код на языке Java

5.1.2.1. Защищенная видимость

Защищенная видимость применяется в контексте наследования. *Закрытые свойства* (атрибуты и операции) базового класса, доступные лишь объектам самого класса, не всегда удобны. Часто необходимо открыть доступ к закрытым свойствам базового класса *объектам производного класса* (подкласса базового класса).

Рассмотрим иерархию классов, где `Person` — это (не абстрактный) базовый класс, а `Employee` — производный класс. Если `Joe` — объект класса `Employee`, то — по определению обобщения — `Joe` должен иметь доступ к свойствам (по меньшей мере к некоторым атрибутам) класса `Person` (например, к операции `getBirthDate()`).

Для того чтобы дать возможность производному классу осуществлять свободный доступ к свойствам его базового класса, эти (в противном случае — закрытые)

свойства необходимо определить в базовом классе как *защищенные*. (Напомним, что понятие видимости относится к классам. Если `Betty` — еще один объект класса `Employee`, то он должен иметь доступ к любому свойству объекта `Joe`: открытому, защищенному или закрытому.)

Пример 5.4. Прямой маркетинг по телефону

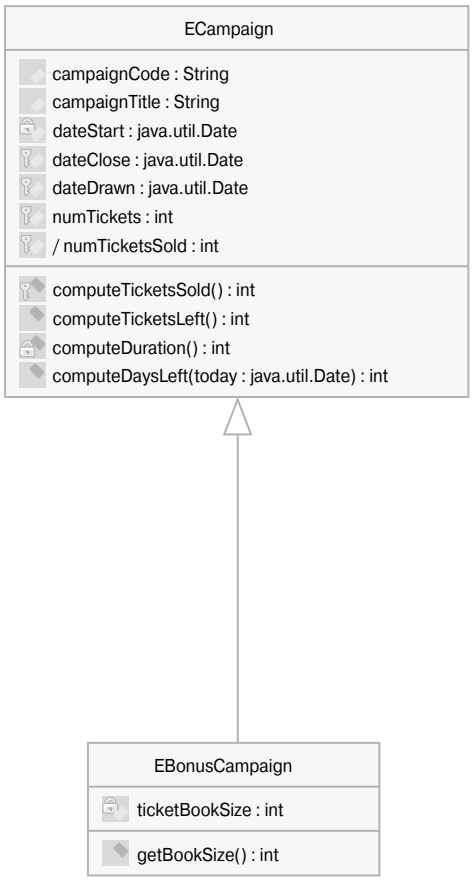
Обратитесь к постановке задачи 4 (см. раздел 1.6.4 главы 1) и примеру 4.7 (см. раздел 4.2.1.2.3 главы 4). Постановка задачи содержит следующее замечание: "Эти схемы включают специальные *бонусные кампании* для вознаграждения приверженцев, закупающих лотерейные билеты в массовом количестве, для привлечения новых жертвователей и т.д.". Это замечание еще не нашло отражения в модели.

Предположим, что одна из призовых кампаний включает "билетные книжки": если благотворитель покупает целую книжку билетов, ему бесплатно вручается дополнительный билет основной кампании.

Задача заключается в следующем.

- Обновить модель класса, включив в нее класс `EBonusCampaign`.
- Изменить видимость атрибутов класса `ECampaign` (см. рис. 5.2) таким образом, чтобы они были видимы классу `EBonusCampaign` за исключением атрибута `dateStart`. Сделать атрибуты `campaignCode` и `campaignTitle` видимыми для остальных классов модели.
- Добавить в класс `ECampaign` следующие вычислительные операции: `computeTicketsSold()` (вычислить количество проданных билетов), `computeTicketsLeft()` (вычислить количество оставшихся билетов), `computeDuration()` (вычислить продолжительность кампании), `computeDaysLeft()` (вычислить количество оставшихся дней кампании).
- Выяснить, что у внешних классов нет нужды в операции `computeTicketsSold()`, — им требуется только выполнять операцию `computeTicketsLeft()`. Кроме того, операция `computeDuration()` используется только операцией `ECampaign.computeDaysLeft()`.
- Класс `EBonusCampaign` хранит атрибут `ticketBookSize` (размер билетной книжки), а операция доступа к нему называется `getbookSize()`.

На рис. 5.6 показана модель классов и код на языке Java, соответствующие примеру 5.4. Операция `computeDuration()` в классе `ECampaign` является закрытой. Операция `computeDuration()` — защищенной, а остальные две — открытыми. Операция `getBookSize()` специфична для класса `EBonusCampaign` и является открытой.




```
public class ECampaign
{
    public String campaignCode;
    public String campaignTitle;
    private Date dateStart;
    protected Date dateClose;
    protected Date dateDrawn;
    protected int numTickets;

    protected int computeTicketsSold()
    {
    return 0;
    }

    public int compute Tickets Left()
    {
    return 0;
    }

    private int computeDuration()
    {
    return 0;
    }

    public int computeDaysLeft(Date today)
    {
    return 0;
    }
}

public class EBonusCampaign extends
ECampaign
{
    private int ticketBookSize;

    public int getBookSize()
    {
    return 0;
    }
}
```

Рис. 5.6. Использование защищенной видимости в системе прямого маркетинга по телефону

5.1.2.2. Видимость унаследованных свойств классов

Как указывалось в предыдущих разделах, понятие *видимости* применяется к объектам на различных уровнях детализации. Обычно имеется в виду, что видимость применяется к *элементарным объектам* (primitive objects) — атрибутам и операциям. Однако видимость можно задать и по отношению к другим "контейнерам". Это приводит к полной неразберихе с правилами замещения.

Рассмотрим, к примеру, ситуацию, когда видимость определяется в *иерархии наследования* на уровне базового класса и на уровне *свойств* базового класса. Пусть класс `B` — подкласс класса `A`. Класс `A` содержит смесь атрибутов и операций; одни из них являются открытыми, другие — закрытыми, а некоторые — защищенными. Вопрос звучит так: "К какому типу видимости относятся унаследованные свойства в классе `B`?"

Ответ на этот вопрос зависит от уровня видимости, установленного базовому классу A при объявлении его в производном классе B. Базовый класс можно объявить открытым (`class B: public A`), защищенным (`class B: protected A`) или закрытым (`class B: private A`).

Типичная реализация приведенного выше сценария выглядит следующим образом (Horton, 1997).

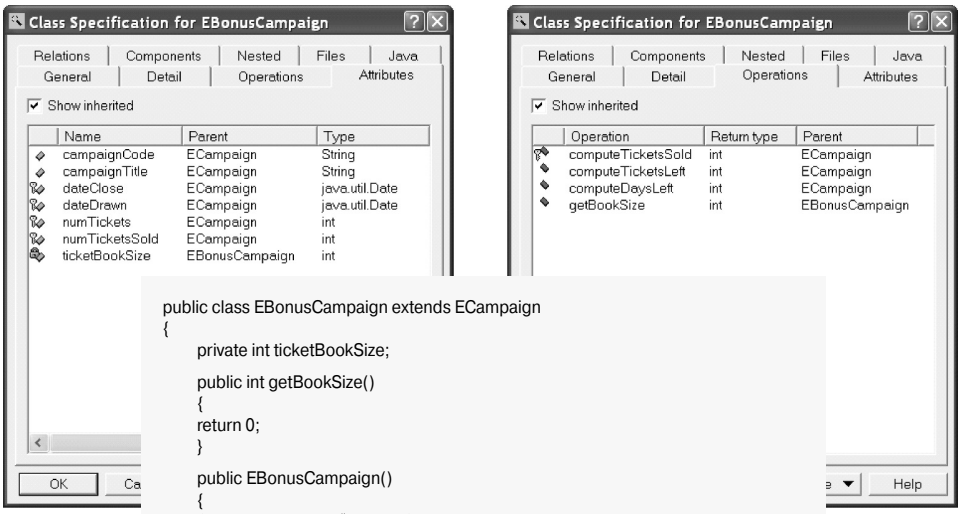
- Закрытые свойства (атрибуты и операции) базового класса A не видимы для объектов класса B, независимо от того, как базовый класс определен в классе B.
- Если базовый класс A определен как `public`, видимость унаследованных свойств в производном классе B не изменяется (открытые свойства остаются открытыми, а защищенные — защищенными).
- Если базовый класс A определен как `protected`, видимость унаследованных свойств в производном классе B изменяется на `protected`.
- Если базовый класс A определен как `private`, видимость унаследованных свойств с типом видимости `public` и `protected` в производном классе B изменяется на `private`.

Отметим, что в контексте проведенного выше обсуждения понятие *наследования реализации* означает следующее: если в базовом классе A существует свойство x, то оно существует и во всех классах, производных от класса A. Однако *наследование* свойства не обязательно означает, что это свойство *доступно* в производных классах. В частности, закрытые свойства базового класса остаются закрытыми в базовом и не доступными в производном классе. Это продемонстрировано на рис. 5.7, на котором показаны (доступные) свойства, унаследованные от класса `EBonusCompany`, представленного на рис. 5.6.

5.1.2.3. Видимость в пакетах и дружественных классах

Во многих ситуациях некоторые классы должны иметь прямой доступ к свойствам другого класса, в то время как остальные классы в системе остаются закрытыми. Язык Java поддерживает такие возможности с помощью *пакетов*. Язык C++ решает эту задачу, определяя *дружественные операции и классы*.

Пакетная видимость в языке Java предусмотрена по умолчанию. Если перед атрибутом или операцией в программе на языке Java не указаны ключевые слова `private`, `protected` или `public` (или ключевое слово `public` перед всем классом), то их видимость по умолчанию считается пакетной. Пакетная видимость означает, что все остальные классы в пакете имеют прямой доступ к этому атрибуту и операции. Однако для всех классов в *других* пакетах этот атрибут, операция или класс остаются закрытыми.




```
public class EBonusCampaign extends ECampaign
{
    private int ticketBookSize;

    public int getBookSize()
    {
    return 0;
    }

    public EBonusCampaign()
    {
    System.out.println("EBonusCampaign constructor);
    }

    public static void main(String[] args)
    {
    EBonusCampaign x    =  new EBonusCampaign();
    x.computeDuration();
    //The above will fail (x cannot access computeDuration)
    }
}
```

Рис. 5.7. Доступность свойств производного класса в системе прямого маркетинга по телефону

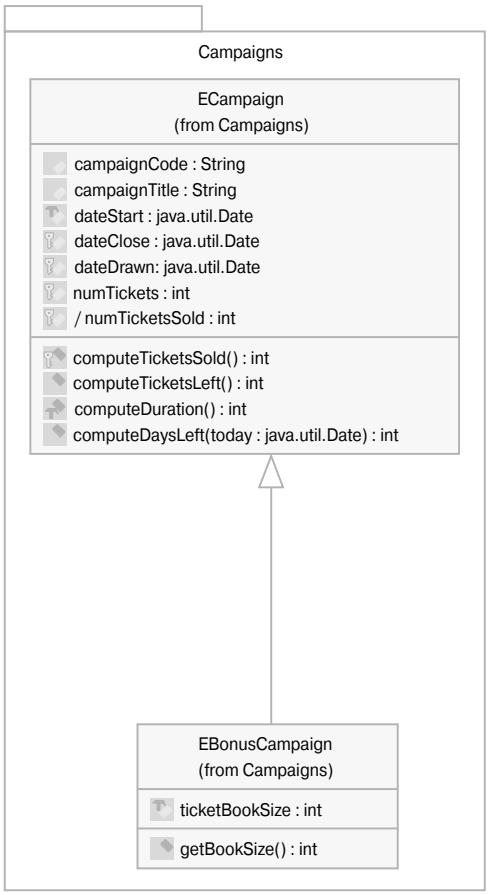
Защищенная (и открытая) видимость также является пакетной, но не наоборот. Это значит, что другие классы в том же самом пакете могут иметь доступ к защищенным свойствам, но *производные* классы не имеют доступа к свойствам, имеющим пакетную видимость, если производные и базовый классы относятся к разным пакетам.

На рис. 5.8 показан вариант модели, представленной на рис. 5.6. В этом варианте закрытый метод `computeDuration()` и закрытые данные-члены `dateStart` и `ticketBookSize` имеют пакетную видимость. Более того, класс `EBonusCampaign` также имеет пакетную, а не открытую видимость.

Как и пакетная видимость в языке Java, концепция *дружественных операций и классов* в языке C++ позволяет решить проблему, когда существует несколько связанных классов и одному из них необходим доступ к закрытым свойствам другого класса. Типичным примером являются два класса, `Book` (Книга) и `BookShelf` (Книжная полка), а также операция `putOnBookShelf()` в классе `Book`.

Для того чтобы решить эту проблему, можно объявить операцию `putOnBookShelf()` *дружественной* по отношению к классу `BookShelf`.

```
friend void Book:putOnBookShelf()
```




```
package Campaigns;

import java.util.Date;

public class ECampaign
{
      public String campaignCode;
      public String campaignTitle;
      Date dateStart;
      protected Date dateClose;
      protected Date dateDrawn;
      protected int numTickets;

      protected int computeTicketsSold()
      {
      return 0;
      }

      public int compute TicketsLeft()
      {
      return 0;
      }

      int computeDuration()
      {
      return 0;
      }

      public int computeDaysLeft(Date today)
      {
      return 0;
      }
}

package Campaigns;

class EBonusCampaign extends ECampaign
{
      int ticketBookSize;

      public int getBookSize()
      {
      return 0;
      }
}
```

Рис. 5.8. Использование пакетной видимости в системе прямого маркетинга по телефону

Дружественными могут быть как *класс*, так и *операция* другого класса. Дружественность не носит взаимного характера. Класс, объявивший другой класс дружественным, не обязательно является дружественным по отношению к нему.

Дружественная операция или класс объявляется *внутри* класса, предоставляющего другому классу права друга. Однако дружественная операция не является свойством класса, поэтому к ней неприменимы атрибуты видимости. Это также означает, что в определении дружественного класса нельзя упоминать атрибуты класса просто по имени, — все они должны уточняться именем класса (как если бы дружественная операция была обычной внешней операцией).

В языке UML дружественные отношения показаны с помощью пунктирной линии *отношения зависимости* (dependency relationship), исходящей из класса или операции и направленной к классу, который предоставляет права друга. Стереотип «`friend`» привязан к стрелке зависимости. По общему признанию, нотация UML не полностью воспринимает и поддерживает семантику дружественности.

Пример 5.5. Прямой маркетинг по телефону

Обратитесь к примеру 4.7 (см. раздел 4.2.1.2.3 главы 4). Рассмотрите отношение между классами `ECampaign` и `ECallScheduled`.

Объекты класса `ECallScheduled` очень активны, и при выполнении операций им требуются специальные права. В частности, они выполняют операцию `getTicketsLeft()` (рис. 5.9), которая устанавливает, остались ли какие-то билеты, чтобы заказ благотворителя мог быть удовлетворен. Существенно, что эта операция имеет прямой доступ к свойствам класса `ECampaign` (таким как `numTickets` и `numTicketsSold`).

Наша задача — объявить операцию `getTicketsLeft()` дружественной классу `ECampaign`.

Обозначения языка UML для отношения дружественности в системе прямого маркетинга по телефону, описанной в примере 5.5, показаны на рис. 5.9. Отношение зависимости, стереотипизированное ключевым словом «`friend`», означает, что класс `ECallScheduled` зависит от класса `ECampaign`, являясь дружественным по отношению к нему. Это отражает тот факт, что класс `ECampaign` предоставляет статус дружественного классу `ECallScheduled` (операция `getTicketsLeft()` объявлена дружественной в классе `ECampaign`, поскольку класс `ECampaign` должен сам называть своих друзей).

```
class ECampaign{
public:
    friend void ECallScheduled::getTicketsLeft();
};
```

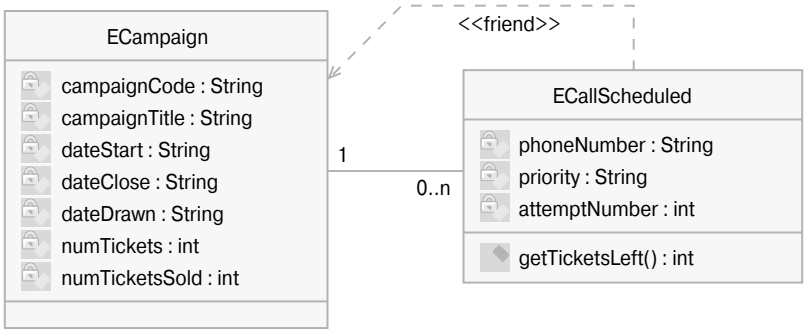


Рис. 5.9. Отношение дружественности в системе прямого маркетинга по телефону

5.1.3. Производная информация

Производная информация (derived information) представляет собой разновидность *ограничения*, которое чаще всего накладывается на *атрибуты* или *ассоциации*. Производная информация вычисляется на основе других элементов модели. Строго говоря, производная информация является избыточной — при необходимости ее можно вычислять.

Несмотря на то что производная информация не обогащает семантику *модели анализа* (analysis model), она придает ей большую четкость (поскольку в модели явно указывается, что некая величина вычисляется). Решение о том, приводить ли в модели анализа производную информацию, разработчик должен принимать самостоятельно, поскольку эта информация последовательно учитывается во всей модели.

Знание, что определенная информация является производной, имеет большее значение в *проектной модели* (design model), в которой необходимо оптимизировать доступ к информации. В проектных моделях можно также решить, хранить ли производную информацию (после ее вычисления) или динамически вычислять каждый раз, когда в ней возникает необходимость. Это не новое свойство — в прежних сетевых базах данных оно было известно как концепция *актуальных* (хранимых) и *виртуальных* данных.

В языке UML производная информация обозначается с помощью косой черты (`/`) перед именем производного атрибута или ассоциации.

5.1.3.1. Производный атрибут

Мы уже использовали производные атрибуты ранее в нескольких диаграммах, правда, не приводя никаких объяснений. Например, атрибут `/numTicketsSold` на рис. 5.2 — производный в классе `ECampaign`. Значение атрибута `/numTicketsSold` вычисляется операцией `computeTicketsSold()`. Эта операция прослеживает связи агрегации, идущие от объекта класса `ECampaign` к объектам класса `ECampaignTicket`, и проверяет каждый атрибут `ticketStatus`. Если атрибут `ticketStatus` принимает значение "продан", то к счетчику проданных билетов добавляется единица. После обработки всех билетов вычисляется текущее значение количества проданных билетов `numTicketsSold`.

5.1.3.2. Производная ассоциация

Производная ассоциация — более спорная тема. Типичным случаем производной ассоциации можно считать ассоциацию, образованную тремя классами, уже соединенными двумя ассоциациями, но не имеющими третьей ассоциации, замыкающей цикл. Зачастую необходимость в третьей ассоциации возникает в связи с требованием семантической корректности модели (*коммутативность цикла*). Если третья ассоциация не представлена в модели явно, она может быть выведена из двух других ассоциативных связей.

Пример 5.6. База данных о заказах

Рассмотрите простую базу данных о заказах с классами `Customer`, `Order` и `Invoice`. Предположим, что заказ всегда формируется одним клиентом, а каждый счет-фактура генерируется для отдельного заказа.

Выведите ассоциацию между этими тремя классами. Существует ли возможность ввести производную ассоциацию в модель?

Между классами `Customer` и `Invoice` возможно ввести производную ассоциацию. На рис. 5.10 она обозначена именем `/CustInv`. Эта ассоциация выводится благодаря несколько необычному бизнес-правилу, по которому кратность ассоциации между классами `Order` и `Invoice` равна "один к одному".

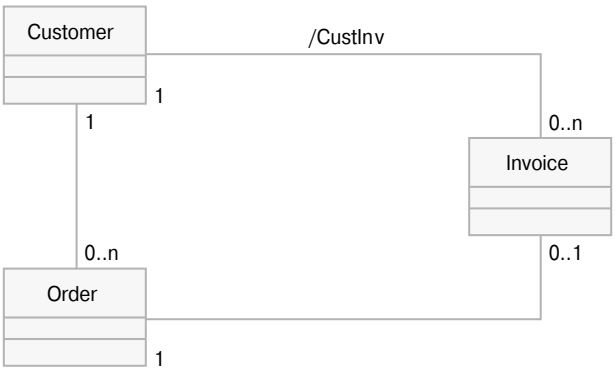


Рис. 5.10. Производная ассоциация

Производная ассоциация не вносит в модель какую-либо новую информацию. Всегда можно связать клиента с определенным счетом-фактурой, отыскав единственный заказ для каждого счета-фактуры, а затем одного клиента для каждого заказа.

5.1.4. Квалифицированная ассоциация

Концепция *квалифицированной ассоциации* (qualified association) вызывает споры среди специалистов. Некоторые охотно пользуются этим понятием, другие не воспринимают его совсем. Неизвестно, можно ли построить полную и достаточно выразительную модель классов без использования квалифицированной ассоциации. Однако если уж использовать квалифицированную ассоциацию, то делать это следует последовательно.

На одном полюсе бинарной квалифицированной ассоциации имеется "отделение" для атрибута (*квалификатора*). Ассоциация редко квалифицируется на обоих полюсах. Это "отделение" содержит один или несколько атрибутов, служа-

щих ключами индексирования для прослеживания ассоциативной связи от квалифицированного *класса-источника* (source class) к *целевому классу* (target class) на противоположном полюсе ассоциации.

Например, ассоциация между классами `Flight` (Рейс) и `Passenger` (Пассажир) имеет кратность "многие ко многим". Однако, если класс `Flight` квалифицировать с помощью атрибутов `seatNumber` (место) и `departure` (пункт отправления), то кратность ассоциации снижается до значения "один к одному" (рис. 5.11). Составной ключ индекса, введенный с помощью квалификатора (`flightNumber` + `seatNumber` + `departure`), может быть связан только с одним объектом `Passenger` либо не связан вообще ни с одним объектом этого класса.

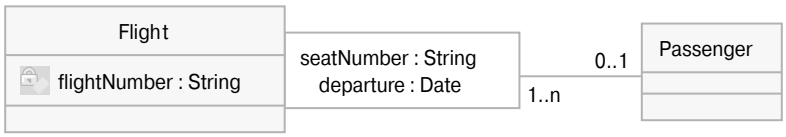


Рис. 5.11. Квалифицированная ассоциация

При прослеживании связи в прямом направлении кратность ассоциации представляет количество целевых объектов, связанных с составным ключом (квалифицированный объект + значение квалификатора). При прослеживании связи в обратном направлении кратность ассоциации описывает количество объектов, обозначенных составным ключом (квалифицированный объект + значение квалификатора) и связанных с каждым целевым объектом (Rumbaugh et al., 2005).

Уникальность в идентификации объектов, введенная с помощью квалификатора, часто представляет собой важную семантическую информацию, которую невозможно эффективно получить другими способами (такими как ограничения или включение дополнительных атрибутов в целевой класс). В общем случае дублировать атрибут квалификации в целевом классе нежелательно.

5.1.5. Ассоциативный или материализованный класс

В приложении А (см. раздел А.5.4) рассматривается пример *ассоциативного класса* (association class) — ассоциации, которая сама является классом. Ассоциативный класс обычно используется в тех случаях, когда между двумя классами существует ассоциация "многие ко многим" и каждый экземпляр ассоциации (связь) обладает собственными значениями атрибутов. Для того чтобы обеспечить возможность хранить эти атрибуты, требуется *ассоциативный класс*.

На первый взгляд простая концепция ассоциативного класса таит в себе хитрое ограничение. Рассмотрим ассоциативный класс *C* между классами *A* и *B*. Ограничение состоит в том, что для каждой пары связанных экземпляров классов *A* и *B* может существовать только один экземпляр класса *C*.

Если подобное ограничение неприемлемо, то специалист по моделированию должен *материализовать* ассоциацию, заменив класс *C* обычным классом *D* (Rumbaugh et al., 2005). *Материализованный класс* (reified class) *D* допускает две бинарные ассоциации с классами *A* и *B*. Класс *D* не зависит от классов *A* и *B*. Каждый экземпляр *D* обладает своей собственной идентичностью, так что при необходимости можно создать несколько экземпляров этого класса для того, чтобы установить связь между одними и теми же экземплярами классов *A* и *B*.

Различия между *ассоциативным и материализованным классами* чаще всего возникают в контексте моделирования временно*й* (исторической) информации. Примером такого приложения является база данных сотрудников, в которой записаны их предыдущие и нынешние ставки.

5.1.5.1. Модель с ассоциативным классом

Пример 5.7. База данных о сотрудниках

Каждому сотруднику в организации присвоен уникальный идентификатор `empId`. Имя сотрудника хранится в виде имени, фамилии и инициалов.

Каждому сотруднику в штатном расписании установлен оклад. Для каждого уровня существует диапазон окладов, т.е. минимальная и максимальная зарплата. Диапазоны окладов для данного уровня никогда не изменяются. Если возникает необходимость изменить минимальный или максимальный оклад, создается новый уровень зарплаты. Кроме того, в организации хранятся начальная и конечная даты введения каждого уровня зарплаты.

В организации хранятся все предыдущие значения зарплаты сотрудника, включая начальную и конечную даты установления ему определенного уровня зарплаты. Также фиксируются любые изменения зарплаты сотрудника в рамках одного и того же уровня.

Разработайте модель классов для базы данных о сотрудниках, используя ассоциативный класс.

Пример 5.7 порождает проблемы. Нам требуется класс для хранения подробной информации о сотруднике (`Employee`), а также класс для хранения информации об уровнях зарплаты (`SalaryLevel`). Проблема состоит в моделировании текущих назначений зарплаты работникам, а также хронологии этих назначений. На первый взгляд кажется естественным использовать ассоциативный класс `SalaryHistoryAssociation`.

На рис. 5.12 представлена модель классов, в которой использован ассоциативный класс `SalaryHistoryAssociation`. Это решение неверно. Идентичность

объектов SalaryHistoryAssociation выводится на основании составного ключа, созданного с использованием ссылок на первичные ключи классов Employee и SalaryLevel (т.е. empId и levelId).

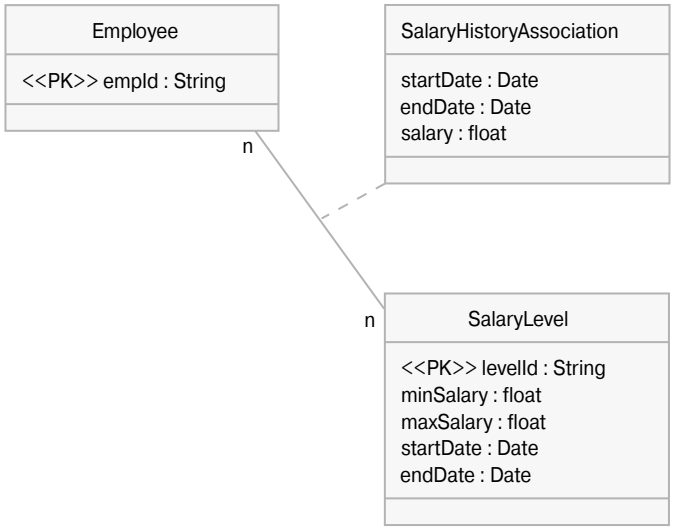


Рис. 5.12. Неправильное использование ассоциативного класса

Никакие два объекта класса SalaryHistoryAssociation не могут иметь одинаковых составных ключей (т.е. одинаковых связей с объектами класса Employee и SalaryLevel). Это также означает, что проект диаграммы на рис. 5.12 не обеспечивает выполнение следующего требования: любые изменения зарплаты сотрудника в рамках одного и того же уровня также фиксируются. Решение, показанное на рис. 5.12, не может рассматриваться как удовлетворительное — требуется более корректная модель.

5.1.5.2. Модель, использующая материализованный класс

Ассоциативный класс не может иметь дублирующихся ссылок на объекты ассоциированных классов. *Материализованный класс* не зависит от ассоциированных классов, и подобное ограничение на него не распространяется. Первичный ключ материализованного класса не использует атрибутов, обозначающих связанные классы.

Пример 5.8. База данных о сотрудниках

Обратитесь к примеру 5.7 (см. раздел 5.1.5.1). Создайте модель классов для базы данных о сотрудниках, используя материализованный класс.

Решение примера 5.8 продемонстрировано на рис. 5.13. Оно основано на использовании материализованного класса `SalaryHistoryReified`. Этот класс не имеет явно помеченного первичного ключа. Однако можно предположить, что этот ключ содержит атрибуты `empId` и `segNum`. Атрибут `segNum` хранит последовательные номера изменений оклада для каждого сотрудника. Каждый объект класса `SalaryHistoryReified` принадлежит отдельному объекту класса `Employee` и связан с единственным объектом класса `SalaryLevel`. Теперь модель учитывает изменения зарплаты сотрудников, относящихся к одному и тому же уровню.

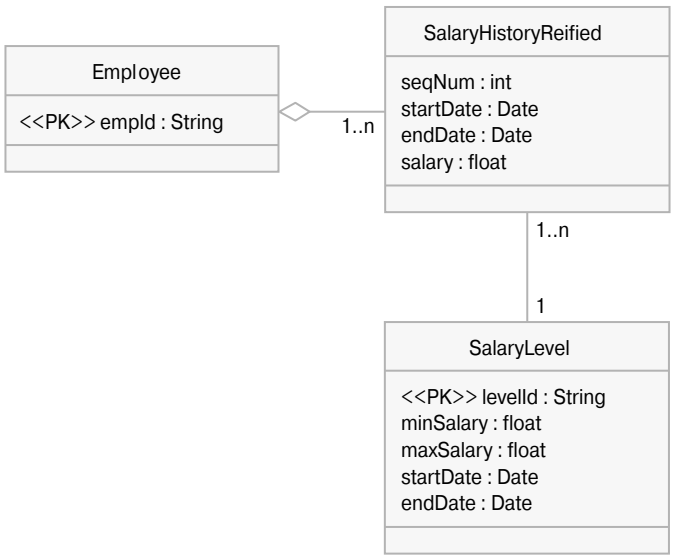


Рис. 5.13. Правильное решение с помощью материализованного класса

Обратите внимание на то, что модель, приведенную на рис. 5.13, следует уточнить. В частности, весьма вероятно, что предположение "диапазоны окладов для заданного уровня никогда не изменяются" в будущем будет ослаблено.

Контрольные вопросы 5.1

- Назовите наиболее важный механизм расширения в языке UML.
- Как называются имена ролей в языке UML 2.0?
- Какой уровень видимости предусмотрен в языке Java по умолчанию, т.е. когда уровень видимости явно не указан?
- Может ли материализованный класс заменить ассоциативный без потери семантики?

5.2. Углубленное моделирование обобщения и наследования

Существуют три основных вида отношений между классами: ассоциация, агрегация и обобщение. Обобщение и наследование обсуждаются в приложении А (см. раздел А.7), но внимательный читатель мог заметить, что полезность обобщения для моделей анализа подвергается сомнению. *Обобщение* (generalization) — полезная и мощная концепция, но она также может стать источником множества проблем из-за сложности механизмов наследования, в частности в крупных программных проектах.

Понятия обобщения и наследования связаны, но не идентичны. Важно понимать разницу между ними. Ценой неточности в определении этих понятий является часто явно демонстрируемое в литературе недопонимание их различия. Конечно, до тех пор, пока это различие не осознано, легко впасть в бессмысленную и безосновательную дискуссию, касающуюся доводов за и против обобщения и наследования.

Обобщение — это семантическое отношение между классами. Оно устанавливает, что *интерфейс* подкласса должен включать все (открытые, пакетные и защищенные) свойства суперкласса. *Наследование* — это "механизм, с помощью которого более специфические элементы вбирают в себя структуру и поведение, определенные более общими элементами" (Rumbaugh et al., 2005).

5.2.1. Обобщение и заменимость

С точки зрения семантики моделирования обобщение вводит в модель дополнительные классы, разделяет их на общие и более специфические классы и устанавливает отношения "суперкласс–подкласс". Несмотря на то что обобщение вводит новые классы, оно может уменьшить общее количество отношений *ассоциации* и *агрегации* в модели (поскольку ассоциации и агрегации характерны для более общих классов и подразумевают существование связей с объектами более специфичных классов).

Исходя из требуемой семантики, ассоциация или агрегация может связывать класс с наиболее общим классом иерархии обобщения (см. диаграммы классов на рис. 3.11 и 4.10). Поскольку подкласс может *заменять* родительский класс, объекты подкласса обладают всеми отношениями ассоциации и агрегации суперкласса. Это позволяет зафиксировать ту же семантику модели с помощью меньшего количества отношений ассоциации/агрегации. Хорошая модель подразумевает верный выбор компромисса между глубиной обобщения и уменьшением количества отношений ассоциации/агрегации, являющегося следствием обобщения.

При продуманном использовании обобщение способствует повышению уровня выразительности, понятности и абстрактности системных моделей. Источни-

ком преимуществ обобщения служит *принцип заменимости* (см. раздел 4.2.4 главы 4) — объект подкласса может быть использован вместо объекта суперкласса в любом месте программы, где объект подкласса имеет доступ к объекту суперкласса. К сожалению, существуют такие способы использования механизма наследования, которые могут свести на нет все преимущества принципа заменимости.

5.2.2. Наследование или инкапсуляция

Инкапсуляция (encapsulation) требует, чтобы доступ к состоянию объекта (т.е. его атрибутам) был возможен только через операции интерфейса объекта. Результатом применения инкапсуляции является высокая степень независимости данных, так что изменения, затрагивающие инкапсулированные структуры данных, не требуют внесения изменений в существующие программы. Но в какой мере идея инкапсуляции осуществима в приложениях?

В действительности инкапсуляция *ортогональна* наследованию и возможностям запросов, и поэтому необходим определенный компромисс между инкапсуляцией и последними двумя свойствами. На практике объявить все данные как *закрытые* невозможно.

Наследование подрывает основы инкапсуляции, открывая подклассам непосредственный доступ к *защищенным* атрибутам. Для вычислений, охватывающих объекты, принадлежащие разным классам, может потребоваться, чтобы они были *дружественными* друг другу или содержали элементы, имеющие пакетную видимость. Это еще больше нарушает инкапсуляцию. Следует также помнить, что инкапсуляция касается понятия класса, а не объекта, и в большинстве программных сред (за исключением языка Smalltalk) объект не может скрыть ничего от другого объекта того же класса.

Наконец, пользователи, осуществляющие доступ к базам данных с помощью средств языка SQL, вполне справедливо желают обращаться непосредственно к атрибутам, а не работать с методами доступа к данным, затрудняющими формирование запросов. Это требование особенно справедливо в отношении приложений хранилищ данных, связанных с интерактивной аналитической обработкой запросов (OnLine Analytical Processing — OLAP).

Итак, приложения должны разрабатываться таким образом, чтобы был достигнут требуемый уровень инкапсуляции и при этом достигался компромисс в отношении наследования, незапланированных запросов и вычислительных требований.

5.2.3. Наследование интерфейса

Когда *обобщение* используется с целью обеспечения заменимости, оно может служить синонимом *наследования интерфейса* (*выделения подтипа*, *наследования типа*). Это не только “безопасный”, но и самый желательный вид наследования

(см. раздел А.9 приложения А). Помимо других преимуществ, наследование интерфейса позволяет реализовать множественное наследование в языках, не поддерживающих его непосредственно (например, в языке Java).

Подкласс наследует типы атрибутов и сигнатуру операций (имя операции плюс формальные аргументы). Говорят, что подкласс *поддерживает* интерфейс суперкласса. Реализация унаследованных операций откладывается на более позднее время.

Существует разница между понятиями *интерфейса* и *абстрактного класса* (см. раздел А.9.1 приложения А). Она заключается в том, что абстрактный класс может обеспечить частичную реализацию некоторых операций, в то время как чистый интерфейс откладывает определение всех операций.

Рис. 5.14 соответствует рис. А.26 (см. раздел А.9.2 приложения А). Этот рисунок демонстрирует альтернативную визуализацию интерфейса "леденец на палочке" и код на языке Java.

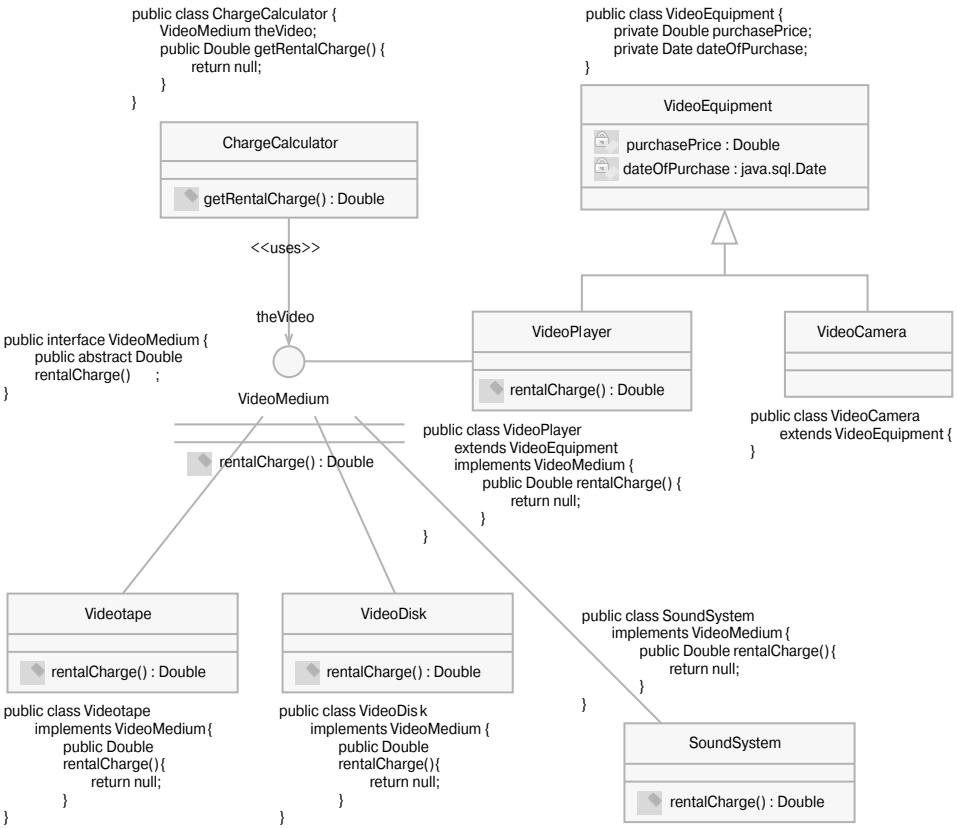


Рис. 5.14. Наследование интерфейса и реализации

5.2.4. Наследование реализации

Как отмечалось в предыдущем разделе, *обобщение* может подразумевать *заменимость* и реализовываться путем *наследования интерфейса*. Однако обобщение может поддерживать (сознательно или нет) *повторное использование кода* (code reuse) с помощью *наследования реализации*. Это очень мощная, иногда даже опасная по силе, интерпретация обобщения. Кроме того, эта интерпретация обобщения принята "по умолчанию".

Наследование реализации (implementation inheritance) — называемое также *выделением подтипа* (subclassing), *наследованием кода* (code inheritance) или *наследованием класса* (class inheritance), — объединяет свойства суперкласса в подклассах и позволяет при необходимости *замещать* их новыми реализациями. *Замещение* (overriding) может означать включение (вызов) метода суперкласса в метод подкласса и расширение его за счет введения новых функциональных возможностей. Оно также может означать полную замену метода суперкласса методом подкласса. Наследование реализации допускает совместное использование *описаний свойств* (property descriptions), *повторное использование кода* и *полиморфизм* (polymorphism).

При моделировании с помощью обобщения необходимо четко отдавать себе отчет в том, какой именно вид наследования подразумевается. Использование *наследования интерфейса* безопасно только в том случае, если оно касается наследования *фрагментов контракта — сигнатуры операций*. Наследование реализации касается наследования кода — наследования *фрагментов реализации* (Harmon and Watson, 1998; Szypersky, 1998). Если тщательно не контролировать и не ограничивать наследование реализации, оно может принести больше вреда, чем пользы. Теперь мы приступим к обсуждению доводов за и против наследования реализации.

5.2.4.1. Правильный способ использования наследования реализации — наследование посредством расширения

Язык UML довольно жестко регламентирует увязку наследования с обобщением и надлежащее использование наследования реализации (Rumbaugh et al., 2005). Единственным правильным способом наследования является инкрементное определение класса. Подкласс обладает бо́льшим количеством свойств (атрибутов и/или методов), чем его суперкласс. Подкласс — это *разновидность* суперкласса. Подобное наследование называется *расширяющим* (extension inheritance).

На рис. 5.23 представлен пример расширяющего наследования. Каждый объект `Employee` (Сотрудник) — это *разновидность* объекта `Person` (Личность), а объект `Manager` — это разновидность объекта `Employee`. Это не означает, что объект `Manager` одновременно является экземпляром трех классов (см. обсуждение множественного наследования в разделе А.7.4 приложения А). Объект `Manager` — это экземпляр класса `Manager`.

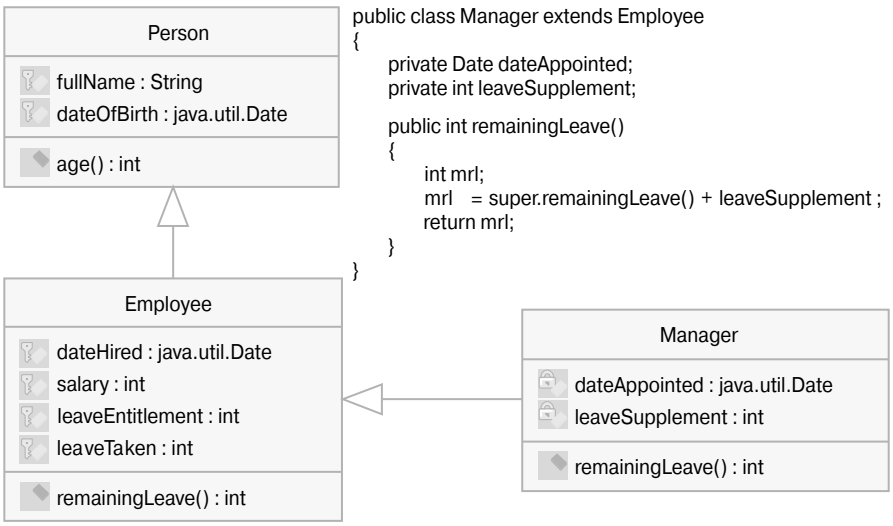


Рис. 5.15. Расширяющее наследование

Метод `remainingLeave()` может быть вызван из объекта `Manager` или `Employee`. Его выполнение зависит от того, из какого объекта он вызван.

Обратите внимание на то, что класс `Person` на рис. 5.15 не является абстрактным. Могут существовать некоторые объекты класса `Person`, которые представляют собой просто некую личность, т.е. не являющиеся сотрудниками (объектами класса `Employee`).

Расширяющее наследование требует внимательного отношения к использованию *замещения* свойств. Допустимым считается только придание свойствам большей определенности (например, ограничение допустимых значений или более эффективная реализация операции), но никак не изменение значения свойства. Если замещение приводит к изменению значения свойства, то объект подкласса нельзя больше подставить вместо объекта суперкласса.

5.2.4.2. Проблематичный способ использования наследования реализации — наследование посредством ограничения

При использовании расширяющего наследования определение подкласса расширяется за счет введения новых свойств. Однако наследование можно также использовать в качестве ограничивающего механизма, посредством которого некоторые унаследованные свойства подавляются (замещаются) в подклассе. Подобное наследование называется *ограничивающим* (restriction inheritance) (Rumbaugh et al., 1991).

На рис. 5.16 приведены два примера ограничивающего наследования. Поскольку наследование нельзя блокировать выборочно, класс `Circle` (Окружность) должен унаследовать свойства главной и дополнительной осей — `minor_axis` и `ma-`

jor_axis — от эллипса Ellipse (Эллипс) и заменить их атрибутом diameter (диаметр). Аналогично, класс Penguin (Пингвин) должен унаследовать операцию fly (летать) у класса Bird (Птица) и заменить ее операцией swim (плавать).

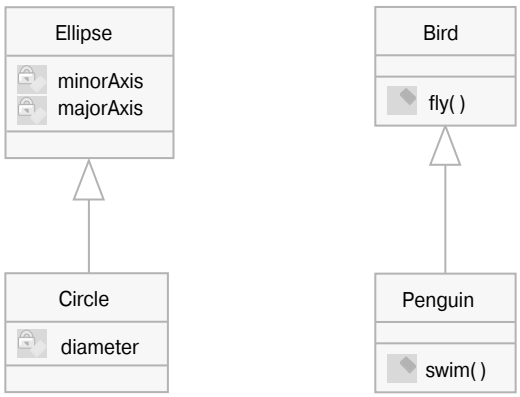


Рис. 5.16. Ограничивающее наследование

Ограничивающее наследование проблематично. С точки зрения обобщения подкласс не включает в себя все свойства суперкласса. Объект суперкласса попрежнему может быть заменен объектом подкласса при условии, что тот, кто использует объект, знает о замещенных (подавленных) свойствах.

При ограничивающем наследовании свойства класса используются (с помощью наследования) для реализации другого класса. Если замещение не носит всеобъемлющего характера, то ограничивающее наследование может принести определенные выгоды. Однако в общем случае ограничивающее наследование вызывает проблемы при сопровождении. Возможна даже ситуация, когда ограничивающее наследование подавляет наследуемые методы, заменяя их пустыми.

5.2.4.3. Неверный способ использования наследования реализации — удобное наследование

Если в процессе системного моделирования оказывается, что наследование нельзя отнести к расширяющему или ограничивающему, оно воспринимается как "плохая новость". Подобный тип наследования встречается в тех случаях, когда два или более класса обладают аналогичными реализациями, но при этом отсутствует отношение таксономии между понятиями, представленными этими классами. Один из классов произвольно выбирается в качестве прообраза другого. Этот вид наследования называется *удобным* (convenience inheritance) (Maciazhek et. al, 1996a; Rumbaugh et al., 1991).

На рис. 5.17 приведены два примера удобного наследования. Класс LineSegment (Отрезок) определен как подкласс класса Point (Точка). Очевидно, что *отрезок* — это не *точка*, и поэтому в данном случае обобщение в том виде, как

оно было определено ранее, неприменимо. Однако наследование все же можно использовать. Конечно, в классе `Point` можно определить атрибуты координат `xCoordinate` и `yCoordinate` и операцию перемещения `move()`. Класс `LineSegment` может унаследовать эти свойства и определить дополнительную операцию изменения размеров `resize()`. Операцию `move()` необходимо заместить. Аналогично, во втором примере класс `Car` (Автомобиль) наследует свойства класса `Motorbike` (Мопед) и добавляет некоторые новые свойства.

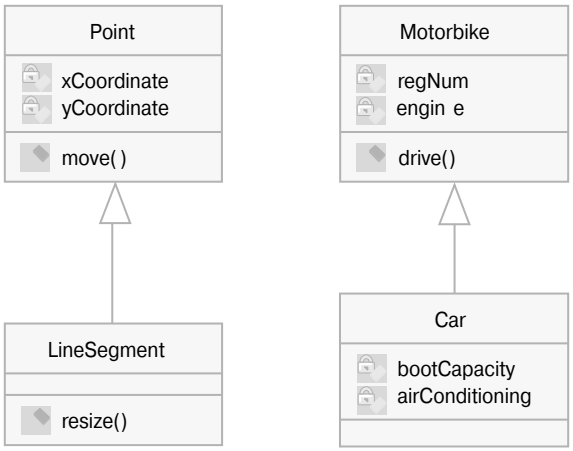


Рис. 5.17. Удобное наследование

Удобное наследование неприемлемо. Оно семантически некорректно и приводит к масштабному замещению. Принцип заменимости, как правило, не работает, поскольку объекты не принадлежат к подобным типам (класс `LineSegment` не является разновидностью класса `Point`, а класс `Car` не является разновидностью класса `Motorbike`).

На практике, к большому сожалению, разработчики часто используют удобное наследование, поскольку многие объектные среды программирования поощряют неразборчивость при использовании наследования реализации. Многие языки оснащены неисчислимыми средствами "усиления программирования" с помощью наследования, в то время как поддержка других свойств объектов (особенно агрегации) отсутствует.

5.2.4.4. Недостатки наследования реализации

Сказанное выше вовсе не означает, что запрещение использования *удобного наследования* служит гарантией успеха. Использование *наследования реализации* — рискованное дело по многим меркам. При отсутствии надлежащего контроля и управления наследование может использоваться чрезмерно или неверно и может создать проблемы, которые требуют первоочередного решения. Это осо-

бенно справедливо для разработки крупномасштабных систем, которые включают сотни классов и тысячи объектов, отличаются динамизмом изменения состояний объектов и эволюционным характером структур классов (как, например, в случае типичных бизнес-приложений).

Основные факторы риска связаны с перечисленными ниже концептуальными трудностями (Szyperski, 1998).

- Изменчивость базового класса.
- Замещение и обратные вызовы.
- Множественное наследование реализации.

5.2.4.4.1. Изменчивый базовый класс

Проблема *изменчивости базового класса* (*суперкласса*) касается ситуации, при которой подклассы достигли определенного уровня зрелости и надежности, в то время как *реализация* их суперкласса (или суперклассов при множественном наследовании) продолжается. Это серьезная проблема в любом случае, особенно в ситуации, когда суперкласс можно получить из внешних источников, находящихся вне контроля коллектива разработчиков системы.

Рассмотрим ситуацию, при которой суперклассы приложения образуют часть операционной системы, СУБД или графического пользовательского интерфейса. Если для разработки своего приложения вы приобретаете объектную СУБД, в действительности вы покупаете библиотеку для реализации типичных функций СУБД, таких как сохранение постоянных объектов, управление транзакциями, обеспечение параллельности, восстановление после сбоев и т.д. Если разрабатываемые классы являются потомками библиотеки классов, то последствия от внедрения новой версии библиотеки непредсказуемы (если при разработке модели наследования для приложения не проявить осторожность, то это несомненно так и есть).

С проблемой изменчивости базового класса трудно справиться, не объявив открытые интерфейсы неизменяемыми или, по меньшей мере, не установив контроль над наследованием реализации из суперклассов. Изменения в реализации суперклассов (для которых может даже отсутствовать исходный текст программ) непредсказуемым образом воздействуют на подклассы прикладной системы. Это справедливо даже тогда, когда интерфейс суперкласса остается неизменным. Ситуация может еще более усугубиться, если изменения также сказываются на интерфейсах. Ниже приведены примеры подобных ситуаций (Szyperski, 1998).

- Изменение сигнатуры метода.
- Разделение метода на два или более новых метода.
- Объединение существующих методов в виде более крупного метода.

Практический вывод из сказанного можно сформулировать следующим образом. Для того чтобы справиться с проблемой изменчивости базового класса, разработчики, проектирующие суперкласс, должны заранее иметь представление о том,

как будут действовать люди при повторном использовании суперкласса сегодня и в будущем. Конечно, узнать это, не прибегая к помощи волшебного шара, нельзя. Как гласит известная шутка: "сумасшествие наследуется — оно достается вам от ваших детей" (Gray, 1994). В разделе 5.3 рассматриваются некоторые альтернативные методы объектной разработки, которые, не будучи основаны на наследовании, все же обеспечивают требуемые функциональные возможности объектов.

5.2.4.4.2. Замещение, нисходящие и восходящие вызовы

Наследование реализации допускает выборочное замещение унаследованного программного кода. Ниже перечислены пять методов, с помощью которых метод подкласса может повторно использовать код его суперкласса.

- Подкласс может наследовать интерфейс и реализацию метода без внесения каких-либо изменений в реализацию.
- Подкласс может наследовать код и включить его (вызвать его) в свой собственный метод с той же сигнатурой.
- Подкласс может наследовать код и затем полностью заместить его новой реализацией с той же сигнатурой.
- Подкласс может наследовать пустой код (т.е. декларация метода отсутствует), а затем ввести реализацию для метода.
- Подкласс может наследовать только интерфейс метода (т.е. мы имеем случай наследования интерфейса), а затем ввести реализацию метода.

Из этих пяти методов первые два доставляют наибольшие трудности в случае склонности программного кода базового класса к эволюции. Пятый метод выказывает полное безразличие к наследованию. Последние два метода представляют особые случаи — четвертый случай тривиален, а пятый не касается наследования реализации.

Пример 5.9. Прямой маркетинг по телефону

Вернитесь к задаче 4, в которой описана система прямого маркетинга по телефону (см. раздел 1.6.4 главы 1), и к примеру 5.4 (см. раздел 5.1.2.1). Модифицируйте модель классов и отношение обобщения между классами `ECampaign` и `EBonusCampaign` (см. рис. 5.6), чтобы включить операции, иллюстрирующие два первых способа повторного использования кода, и покажите нисходящие и восходящие вызовы в отношении обобщения.

Для того чтобы продемонстрировать первый способ повторного использования кода, рассмотрите операцию `computeTicketSold()` в классе `ECampaign`, наследуемую без модификации классом `EBonusCampaign`. Реализация операции `computeTicket()` содержит вызов операции `computeTicketLeft()`. Операция `computeTicketLeft()` существует в классе `ECampaign`, а ее замещенная версия — в классе `EBonusCampaign`.

Для того чтобы продемонстрировать второй способ повторного использования кода и восходящий вызов, рассмотрите операцию getDateClose() в классе ECampaign и ее замещенную версию в классе EBonusCampaign. При вызове из класса ECampaign операция getDateClose() возвращает дату завершения кампании, т.е. объекта класса ECampaign. Однако при вызове из классе EBonusCampaign она возвращает наиболее позднюю дату из двух возможных для классов ECampaign и EBonusCampaign.

На рис. 5.18 показаны модель и код, описывающие решение примера 5.9. Класс CActioner (Акционер) вызывает операции из классов ECampaign и EBonusCampaign. Он имеет ссылку (-theECampaign) на суперкласс EBonusCampaign, но хранит ее в виде ссылки на тип ECampaign. В реальности присваивание объекта класса EBonusCampaign ссылке на класс ECampaign выполнятся на этапе выполнения программы (Maciaszek and Liong, 2005), а не на этапе компиляции, как показано на рис. 5.18.

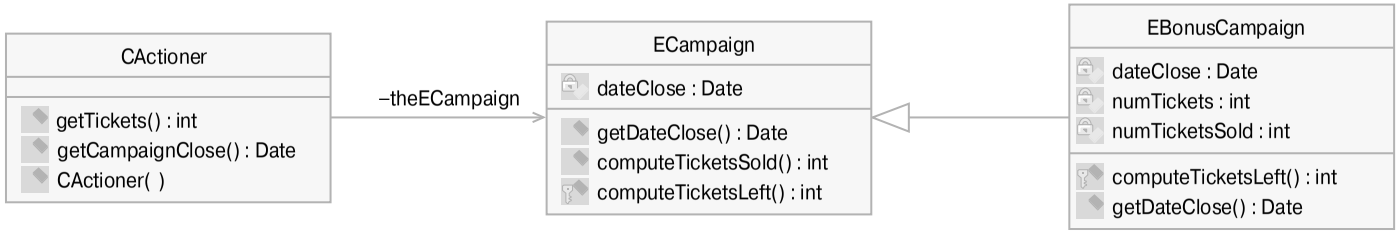
Когда операция getTickets() вызывается объектом класса CActioner, она вызывает операцию computeTicketsSold() из класса ECampaign. Здесь используется принцип заменимости — объект класса EBonusCampaign, на который ссылается ссылка -theECampaign, заменяется объектом класса ECampaign (поскольку класс ECampaign не содержит свою собственную замещенную версию операции computeTicketsSold()).

Далее, операция computeTicketsSold() вызывает операцию computeTicketsLeft(). Однако операция computeTicketsLeft() в классе EBonusCampaign замещается. Следовательно, вместо нее будет вызвана ее замещенная версия. Этот пример демонстрирует *нисходящий вызов* (down-call) подкласса из суперкласса. В результате вычисляется количество билетов, оставшихся в объекте класса EBounsCampaign, которое возвращается объекту CActioner.

Обратите внимание на то, что инициализация ссылки theECampaign и последующие нисходящие вызовы образуют зависимость класса CActioner от класса EBonusCampaign на этапе выполнения программы. На этапе компиляции эта зависимость не фиксируется. Класс CActioner имеет ассоциацию с классом ECampaign, но не с классом EBonusCampaign. Такие динамические зависимости трудно отслеживать. Они противоречат принципу EAP модели PCBMER (см. раздел 4.1.3.2 главы 4).

Когда операция getCampaignClose() вызывается объектом класса CActioner, она вызывает операцию getDateClose() из объекта, на который ссылается переменная -theECampaign. Этот объект принадлежит классу EBonusCampaign. Соответственно, вызывается операция getDateClose() из класса EBonusCampaign. Интересно, что операция getDateClose() обеспе-

req_analysis_system design.indb 386




```
public class CActioner
{
    private ECampaign theECampaign;

    public CActioner()
    {
        theECampaign    = new EBonusCampaign();
    }

    public void getTickets()
    {
        int tnum;
        tnum   = theECampaign.computeTicketsSold();
        return tnum;
    }

    public void getCampaignClose()
    {
        Date cdate;
        cdate    = theECampaign.getDateClose();
        return cdate
    }
}
```

```
public class ECampaign
{
    public Date getDateClose()
    {
        Date cdate      = dateClose;
        return cdate;
    }

    public int computeTicketsSold()
    {
        int tnum;
        tnum   = computeTicketsLeft();
        return tnum;
    }

    protected int computeTicketsLeft()
    {
        return 0; //not executed in the example
    }
}
```

```
public class EBonusCampaign
        extends ECampaign
{
    protected int computeTicketsLeft()
    {
        int tnum;
        tnum   = numTickets - numTicketsSold;
        return tnum;
    }

    public Date getDateClose()
    {
        if (super.getDateClose() < dateClose)
            Date cdate      = dateClose;
        else
            Date cdate      = super.getDateClose();
        return cdate;
    }
}
```

Рис. 5.18. Замещение, нисходящие и восходящие вызовы

05.06.2008 14:43:13

чивает расширение метода, определенного в классе `Ecampaign`, — он содержит обращение к суперклассу. Этот пример демонстрирует *восходящий вызов* (*обратный вызов*) суперкласса из подкласса.

Несмотря на простоту обратного вызова, комбинация восходящих и нисходящих вызовов создает запутанные циклические зависимости между классами. Кроме того, эти зависимости возникают на этапе выполнения программ, и, следовательно, их сложно отслеживать.

Пример 5.9 демонстрирует влияние замещения на проблему изменчивости базового класса. Он также показывает, что наследование реализации приводит к возникновению сетей, образованных линиями взаимодействия между объектами, которые, как было показано ранее, в больших системах отличаются неустойчивостью (см. раздел 4.1 главы 4). При наследовании реализации передача сообщений может возникать повсеместно, порождая как *нисходящие*, так и *восходящие вызовы* по иерархии наследования.

Как заметил Шиперский (Szyperswki, 1998): "В сочетании с наблюдаемым состоянием это приводит к циклической семантике, характерной для параллельных систем. Произвольный граф вызовов, формируемый сетью взаимодействующих объектов, разрушает классическую схему иерархии уровней и делает циклические вызовы нормой".

Справедливости ради, следует заметить, что обратные вызовы возможны не только между объектами, связанными отношениями наследования, а всюду, где существуют ссылки между объектами. Вновь приведем замечание Шиперского (Szyperswki, 1998): "При наличии ссылок на объекты… всякий вызов метода может рассматриваться как потенциальный восходящий вызов, и каждый метод может быть потенциально связан с обратным вызовом". Наследование только вносит свой вклад в общую проблему, но, надо заметить, вклад существенный.

5.2.4.4.3. Множественное наследование реализации

Множественное наследование описывается в приложении А (см. раздел А.7.3). В разделе А.9 проведено различие между *множественным наследованием интерфейса* (множественным выделением подтипов) и *множественным наследованием реализации* (множественным наследованием от суперклассов). Множественное наследование интерфейса допускает слияние контрактов интерфейсов. Множественное наследование реализации позволяет объединить *фрагменты реализации*.

Множественное наследование реализации в действительности не создает каких-либо дополнительных проблем наследования реализации. Оно скорее усугубляет проблемы, вызванные изменчивостью базового класса, замещением и обратными вызовами. Помимо требования блокировать наследование любых дублирующихся фрагментов (когда два или более суперкласса определяют одну и ту же операцию), оно может также вызвать необходимость переименовать операции всякий раз, когда повторяющиеся имена совпадают (а в действительности должны обозначать разные операции).

В этом контексте стоит напомнить о присущем системам росте сложности из-за множественного наследования — росте, вызванном отсутствием поддержки *множественной классификации* в объектных системах (см. раздел А.7.4 приложения А). Любые ортогональные ветви наследования, сходящиеся в одном суперклассе, должны объединяться в дереве наследования на более низком уровне иерархии с помощью специально созданных "объединяющих" классов (см. рис. А.22 приложения А).

Проблемы, связанные с множественным наследованием реализации, привели к тому, что в некоторых языках программирования этот механизм был исключен (например, в языке Java). Вместо него язык Java рекомендует использовать *множественное наследование интерфейса* (см. раздел А.9 приложения А).

Контрольные вопросы 5.2

- Какие принципы определяют полезность обобщения?
- Как наследование нарушает инкапсуляцию?
- Какую концепцию можно использовать вместо множественного наследования реализации?

5.3. Углубленное моделирование агрегации и делегирования

Агрегация представляет собой третий метод связывания классов в моделях анализа (см. раздел А.6 приложения А). В сравнении с двумя другими методами (*традиционной ассоциацией* и *обобщением*) агрегации уделялось меньше всего внимания. Тем не менее агрегация представляет собой наиболее мощный из известных методов управления сложностью больших систем с помощью распределения классов по иерархическим уровням абстракции.

Агрегация (и ее более сильный вариант — *композиция*) — это отношение включения. *Составной класс* (composite class) содержит один или несколько *компонентных классов* (component classes). Компонентный класс является элементом одного или более составного класса (хотя они могут существовать самостоятельно). Несмотря на то что агрегация получила признание как фундаментальная концепция моделирования, по меньшей мере одновременно с обобщением, в объектно-ориентированном анализе и проектировании ей уделяется лишь незначительное внимание (за исключением областей приложений наподобие систем мультимедиа).

В средах программирования (включая большинство объектных СУБД) агрегация реализуется так же, как традиционная ассоциация, — с помощью запроса ссылок между составными и компонентными объектами. Несмотря на то что структура времени компиляции для агрегации аналогична структуре ассоциации,

во время выполнения они ведут себя по-разному. Агрегация обладает более строгой семантикой, и ответственность за то, чтобы структуры времени выполнения удовлетворяли этой семантике, лежит (к сожалению) на программисте.

5.3.1. Расширение семантики агрегации

Несмотря на то что современные среды программирования игнорируют агрегацию, методы разработки объектных приложений включают агрегацию среди прочих возможностей моделирования, однако отводят ей последнее место. Кроме того (или вследствие недостаточной поддержки в средах программирования), методы разработки объектных приложений не стремятся придать агрегатным конструкциям строгую семантическую интерпретацию, зачастую трактуя их просто как особую форму ассоциации.

Как указывалось в разделе 4.2.3 главы 4, можно выделить четыре возможных семантики для агрегации (Maciaszek et. al, 1996).

1. Агрегация *ExclusiveOwns* (Безраздельное владение).
2. Агрегация *Owns* (Владение).
3. Агрегация *Has* (Содержит).
4. Агрегация *Member* (Участник).

Язык UML признает только две семантики агрегации, а именно *агрегацию* (*ссылочную семантику*) и *композицию* (*семантику значений*) (см. раздел А.6 приложения А). Теперь будет показано, каким образом можно использовать стереотипы и ограничения, чтобы расширить существующую нотацию UML для представления четырех видов агрегации, указанных выше.

5.3.1.1. Агрегация *ExclusiveOwns*

Агрегацию *ExclusiveOwns* в языке UML можно представить как композицию, стереотипизированную ключевым словом «`ExclusiveOwns`» и дополнительно ограниченную с помощью ключевого слова *frozen* (Fowler, 2003). Ограничение *frozen* (заморожен) применяется к *компонентному классу*. Оно констатирует, что объект *компонентного класса* не может быть *заново соединен* (в течение своего жизненного цикла) с другим составным объектом. Компонентный объект может быть удален вовсе, но не может переключиться на другого владельца.

На рис. 5.19 показаны два примера агрегации *ExclusiveOwns*. Левая часть рисунка представляет пример моделирования агрегации в UML с использованием семантики значений (закрашенный ромб), а правая часть — пример моделирования агрегации в UML с использованием ссылочной семантики (пустой ромб).

Объект `Chapter` (Глава) является частью по меньшей мере одного объекта `CopyrightedBook` (Книга, защищенная авторским правом). Будучи включенным (по значению) в составной объект, он не может быть повторно соединен с другим объектом `CopyrightedBook`. Это соединение заморожено.

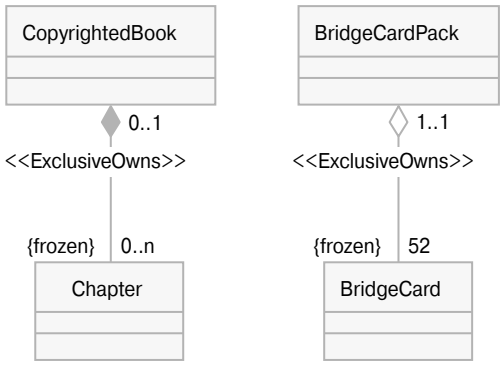


Рис. 5.19. Агрегация ExclusiveOwns

Объект `BridgeCardPack` (Колода карт для игры в бридж) содержит пятьдесят две карты. Для моделирования принадлежности в UML используется ссылочная семантика. Каждая карта для бриджа (объект `BridgeCard`) принадлежит одной колоде карт (`BridgeCardPack`) и не может быть повторно соединена с другой колодой.

5.3.1.2. Агрегация *Owns*

Аналогично агрегации *ExclusiveOwns*, агрегацию *Owns* можно выразить в языке UML с помощью семантики значений композиции (закрашенный ромб) или ссылочной семантики агрегации (пустой ромб). В каждый момент времени компонентный объект принадлежит одному составному объекту, однако он может быть *заново соединен* с другим составным объектом. При удалении составного объекта его компонентные объекты также удаляются.

На рис. 5.20 показаны два примера агрегации *Owns*. Объект класса `Water` (Вода) может быть соединен с другим объектом класса `Jug` (Кувшин). Аналогично, объект класса `Tire` (Шина) может быть соединен с другим объектом класса `Bicycle` (Велосипед). Благодаря зависимости по существованию разрушение объекта класса `Jug` или `Bicycle` распространяется вниз на их компонентные объекты.

5.3.1.3. Агрегация *Has*

Для моделирования агрегации *Has* в языке UML обычно используется ссылочная семантика агрегации (пустой ромб). Агрегация *Has* не содержит зависимости по существованию — удаление составного объекта не распространяется автоматически вниз на компонентные объекты. Агрегацию *Has* отличают такие свойства, как транзитивность и асимметричность.

Пример агрегации *Has* показан на рис. 5.21. Если объект класса `Trolley` (Тележка) *содержит* несколько объектов класса `BeerCrate` (Ящик пива), а каждый объект класса `BeerCrate` *содержит* несколько объектов класса `BeerBottle` (Бутылка пива), то объект класса `Trolley` содержит объекты класса `Beer-`

`Bottle` (*транзитивность*). В то же время объект класса `Trolley` тележка содержит объекты класса `BeerCrate`, но не наоборот (*асимметричность*).

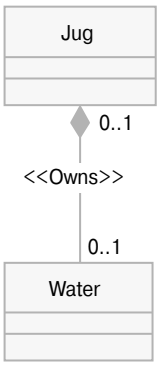
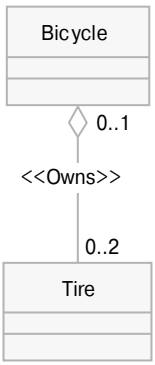


Рис. 5.20. Агрегация Owns



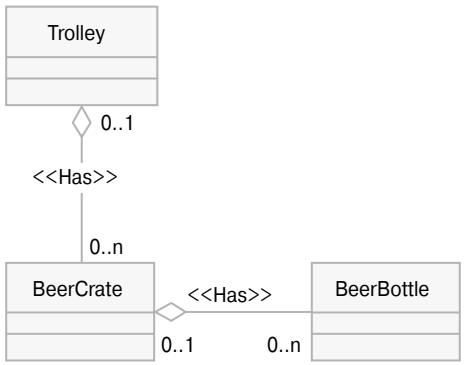


Рис. 5.21. Агрегация Has

5.3.1.4. Агрегация Member

Агрегация *Member* допускает отношения с кратностью "многие ко многим". В отношении свойств зависимости по существованию, транзитивности, асимметрии и ограничения `frozen` не делается никаких специальных предположений. При необходимости любое из этих четырех свойств можно выразить в UML с помощью ограничения. Из-за кратности "многие ко многим" агрегацию *Member* можно моделировать в UML только с помощью ссылочной семантики агрегации (пустой ромб).

На рис. 5.22 показаны четыре отношения агрегации *Member*. Объект класса `JADSession` (см. раздел 2.2.3.3 главы 2), описывающий совещание JAD, состоит из одного объекта класса `Moderator` (Модератора) и одного или нескольких объектов класса `Scribe` (Секретарь), `User` (Пользователь) и `Developer` (Разработчик). Каждый компонентный объект может быть членом нескольких объектов класса `JADSession`.

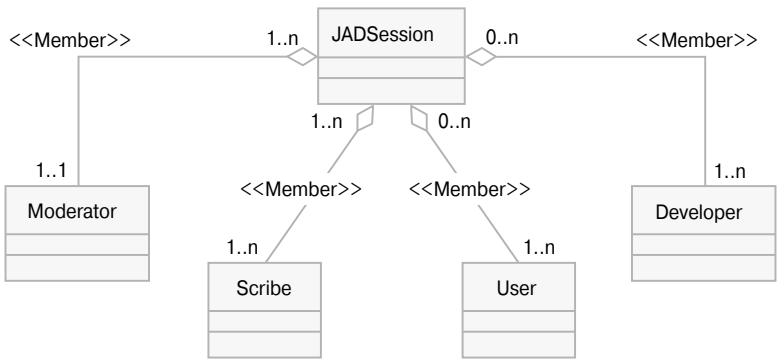


Рис. 5.22. Агрегация Member

5.3.2. Агрегация как альтернатива обобщению

Обобщение (generalization) — это отношение "суперкласс–подкласс". *Агрегация* (aggregation) больше напоминает отношение "супермножество–подмножество". Вопреки этому различию, обобщение можно представить как агрегацию.

Рассмотрим рис. 5.23. Невыполненные заказы клиентов могут быть отложены в ожидании дальнейших действий. Например, заказ может быть выполнен после пополнения запаса или в конкретные сроки, определенные клиентом.

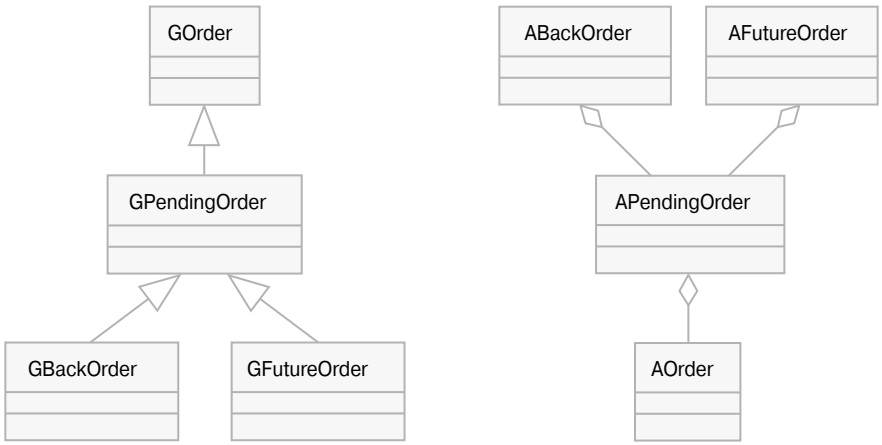


Рис. 5.23. Сравнение обобщения и агрегации

Модель, показанная на рис. 5.23, *слева*, представляет собой обобщение заказов клиентов. Класс `GOrder` (Заказ) может быть классом `GPendingOrder` (Отложенный заказ). Класс `GPendingOrder` может быть классом `GBackOrder` (Невыполненный заказ) или классом `GFutureOrder` (Заказ на будущее). Наследование гарантирует разделение атрибутов и операций вниз по дереву обобщения.

Аналогичную семантику можно смоделировать с помощью агрегации, показанной на рис. 5.23, *справа*. Классы `ABackOrder` и `AFutureOrder` включают в себя атрибуты и операции класса `APendingOrder`, которые в свою очередь включают класс `AOrder`.

Несмотря на то что две модели, показанные на рис. 5.23, отражают одну и ту же семантику, между ними существуют определенные различия. Одно из них вытекает из замечания, что модель обобщения основана на понятии класса, в то время как модель агрегации фактически сконцентрирована на понятии объекта.

Конкретный объект класса `GBackOrder` является также объектом классов `GPendingOrder` и `GOrder`. Для объекта класса `GBackOrder` существует один *идентификатор объекта* (OID). С другой стороны, конкретный объект класса `ABackOrder` состоит из трех отдельных объектов, каждый из которых обладает собственным идентификатором объекта, — собственно объект класса `ABack-`

`Order`, содержащийся в нем объект класса `APendingOrder` и содержащийся в нем объект класса `AOrder`.

Обобщение использует *наследование* для реализации своей семантики. Агрегация использует *делегирование* для повторного использования реализации компонентных объектов. Использование делегирования рассматривается в следующем разделе.

5.3.2.1. Делегирование и системы-прототипы

Вычислительная модель наследования основана на понятии *класса*. Однако в основу вычислительной модели можно положить понятие объекта. Объектно-ориентированная вычислительная модель структурирует объекты в виде *иерархий агрегации* (aggregation hierarchies). Всякий раз, когда *составной объект* (*внешний объект*) не в состоянии выполнить задание самостоятельно, он может вызвать методы одного из его *компонентных объектов* (*внутренних объектов*), — это и называется *делегированием* (delegation).

При подходе, основанном на делегировании, функциональные возможности системы реализуются с помощью включения (*клонирования*) функций существующих объектов во вновь требуемые функции. Существующие объекты трактуются как *прототипы* для создания новых *объектов*. Идея состоит в том, чтобы сначала отыскать требуемые функции в существующих объектах (*внутренних объектах*), а затем реализовать необходимые функции во *внешних объектах*. Внешние объекты запрашивают услуги внутренних объектов по мере необходимости. Системы, построенные подобным способом из существующих объектов-прототипов, называются *прототипами* (prototypical systems).

Объект может быть связан отношением *делегирования* с любым другим идентифицируемым и видимым объектом в системе (Lee and Tepfenhart, 1997). Когда внешний объект получает сообщение и не в состоянии выполнить задачу самостоятельно, он делегирует ее выполнение внутреннему объекту. При необходимости внутренний объект может переслать сообщение любому из своих внутренних объектов.

Интерфейсы внутреннего объекта могут быть видимы или невидимы объектам, отличным от внешнего объекта. Для контроля уровня видимости внутренних объектов можно использовать четыре типа агрегации (см. раздел 5.3.1). Например, внешний объект может раскрыть интерфейс внутреннего объекта как свой собственный в более слабой форме агрегации (например, в такой, как агрегация *Has* или *Member*). При более сильной форме агрегации внешний объект может скрыть интерфейс своего внутреннего объекта от внешнего мира (вводя, таким образом, более строгую форму *инкапсуляции*).

5.3.2.2. Сравнение делегирования и наследования

С помощью *делегирования* можно моделировать *наследование* и наоборот. Это значит, что одни и те же функциональные возможности системы можно реализовать как с помощью *наследования*, так и *делегирования*. Согласие в этом вопро-

се впервые было достигнуто на конференции в США (Орландо, штат Флорида), в 1987 году и теперь известно как Орландское соглашение (Stein et al., 1989).

Мы уже касались изъянов *наследования реализации*. В связи с этим возникает настоятельный вопрос: позволяет ли делегирование избежать недостатков, присущих наследованию реализации. Ответ на этот вопрос не очевиден (Szyperski, 1998).

С точки зрения *повторного использования* кода делегирование сильно приближено к наследованию. Внешний объект повторно использует реализацию внутреннего объекта. Разница состоит в том, что — в случае наследования — после завершения обслуживания управление всегда возвращается объекту, который получает исходное сообщение (запрос на выполнение задачи).

В случае делегирования, после того как управление передано от внешнего объекта внутреннему объекту, оно остается у последнего. Любая *авторекурсия* (self-recursion) должна быть явно запланирована и спроектирована в рамках делегирования. При использовании наследования реализации авторекурсия всегда случайна — она не запланирована и наложена как программная "заплата" (Szyperski, 1998). Одним из нежелательных последствий незапланированного/"заплатанного" повторного использования является проблема *изменчивости базового класса*.

Еще одним потенциальным преимуществом делегирования является то, что разделение и повторное использование можно определить динамически во время выполнения приложения. В системах, ориентированных на использование наследования, разделение и повторное использование обычно определяются статически при создании объекта. При этом достигается компромисс между безопасностью и скоростью выполнения *предусмотренного разделения* наследования и гибкостью *непредусмотренного разделения* делегирования.

В пользу делегирования можно привести тот аргумент, что непредусмотренное разделение более естественно и ближе к человеческому способу мышления (Lee and Tepfenhart, 1997). Объекты объединяются естественным способом для формирования масштабных решений и могут эволюционировать непредвиденным образом. В следующем разделе приводится другая точка зрения на эти же вопросы.

5.3.3. Агрегация и холоны — интеллектуальное орудие

В работах Мацияшека (Maciaszek et al., 1996a, 1996b) с целью преодоления сложности объектных моделей был предложен новый подход для описания архитектуры программного обеспечения, основанный на интерпретации естественных систем, предложенной Артуром Кёстлером (Koestler, 1967, 1978). Центральной концепцией является идея так называемых *холонов* ("holons"), которые интерпретируются как объекты, являющиеся одновременно и частью и целым. Более точно они рассматриваются как саморегулируемые сущности, которые проявляют одновременно взаимозависимые свойства части и независимые свойства целого.

Живые системы обладают иерархической организацией. Структурно они представляют собой агрегации полуавтономных элементов, которые обладают как не-

зависимыми свойствами целого, так и взаимозависимыми свойствами части. По Артуру Кёстлеру, они представляют собой агрегации холонов (от греческого слова *holos* — целое). Суффикс "-он" означает частицу или часть (как в словах протон или нейтрон) (Koestler, 1967).

Части и целое в абсолютном смысле не существуют в живых организмах и даже в социальных системах. Холоны образуют иерархические уровни соответствующей сложности. Например, в биологических организмах различаются иерархии атомов, молекул, органоидов, клеток, тканей, органов и систем органов. Подобные иерархии холонов называются *холархиями* (holarchies).

Каждый уровень холархии скрывает свою сложность от вышележащего уровня. Если смотреть *сверху вниз*, то холон представляет собой нечто законченное и уникальное, целое. Если смотреть *снизу вверх*, то холон представляет собой элементарную компоненту, часть. Каждый уровень холархии содержит множество холонов, например атомы (водород, углерод, кислород и т.д.), клетки (нервные волокна, клетки крови и т.д.).

Если смотреть *изнутри*, то холон предоставляет услуги другому холону. Если смотреть *снаружи*, то холон запрашивает услуги у других холонов. Холархии отличаются незавершенностью. Не существует абсолютных холонов-"листьев" или холонов-"вершин", за исключением тех, которые мы специально обозначаем таким образом для удобства интерпретации. Благодаря этим характеристикам сложные системы могут эволюционировать из простых систем.

Отдельные холоны, таким образом, представляются четырьмя характеристиками.

- Его внутренние правила (взаимодействия между ними могут формировать уникальные шаблоны).
- Самоутверждающаяся агрегация подчиненных холонов.
- Тенденция к интеграции по отношению к высшим холонам.
- Отношения с соседними холонами.

Удачные системы упорядочены в виде холархий, скрывающих сложность в последовательных нижних уровнях, и в то же время обеспечивают больший уровень абстракции в рамках более высоких уровней ее структур. Эта концепция соответствует семантике агрегации.

Агрегация предусматривает разделение — позволяет каждому классу оставаться инкапсулированным и концентрироваться на специфическом поведении (кооперация и услуги) класса способом, который не связан с реализацией его родительских классов (как это имеет место для обобщения). В то же время агрегация позволяет свободное перемещение между стратифицированными уровнями во время выполнения.

Баланс между интеграцией и самоутверждением объектов (холонов) достигается за счет требования, согласно которому объекты должны "признавать интерфейсы друг друга" (Gamma et al., 1995). Инкапсуляция не нарушается, поскольку

объекты взаимодействуют только посредством своих интерфейсов. Эволюция системы облегчается, поскольку взаимодействие объектов не запрограммировано жестко в реализации с помощью механизмов, аналогичных наследованию.

Со структурной точки зрения агрегация дает возможность моделировать большие сообщества объектов с помощью группирования их в виде различных множеств и установления между ними отношения "часть–целое". С функциональной точки зрения агрегация позволяет просматривать иерархию объектов (холонов) сверху вниз и снизу вверх.

Однако агрегация не позволяет моделировать необходимые возможности взаимодействия между холонами одного уровня так, чтобы они могли просматривать структуру изнутри и извне. Этот структурный и функциональный разрыв можно преодолеть с помощью отношений обобщения и ассоциации.

В рамках рекомендуемого подхода агрегация обеспечивает "вертикальное" решение и обобщение "горизонтального" решения для разработки объектных приложений. Агрегация становится преобладающей концепцией моделирования, которая определяет общую структуру системы. Эта структура может быть формализована за счет использования множества проектных шаблонов (Gamma et al., 1995), в особенности поддержки подхода на основе холонов и применения четырех видов агрегации. Мы надеемся, что рассмотренный выше подход послужит читателям интеллектуальным орудием в их собственных изысканиях (Maciaszek, 2006).

Контрольные вопросы 5.3

- Как агрегация реализуется в типичной программной среде?
- Какой вид агрегации должен указываться с ограничением frozen?
- Какая агрегация используется для реализации компонентных объектов?

5.4. Углубленное моделирование взаимодействий

Основы моделирования взаимодействий изложены в главе 3 (см. раздел 3.4) и главе 4 (см. раздел 4.3.3). Рассмотрим более сложные свойства диаграмм взаимодействия и покажем, как их гибкость и точность позволяют гладко перейти на уровень абстракции, требуемый детализированным проектом.

Из двух диаграмм взаимодействий *диаграмма последовательностей* более популярна, чем *диаграмма коммуникаций*. Они выражают одинаковую информацию, но диаграммы последовательностей отражают последовательность сообщений, а диаграммы коммуникаций — отношения между объектами. CASE-инструменты лучше приспособлены для визуализации диаграмм последовательностей и лучше

поддерживают тонкости моделирования взаимодействий в диаграммах последовательностей. По отношению к диаграммам коммуникации это не всегда правда. Замечательным преимуществом диаграмм коммуникации является то, что их можно использовать для презентаций на совещаниях. В таких ситуациях диаграммы коммуникации требуют намного меньше памяти, и их легче исправлять, чем диаграммы последовательностей.

5.4.1. Линии жизни и сообщения

Две из наиболее заметных концепций в диаграммах взаимодействия — *линии жизни* и *сообщения*. *Линия жизни* (lifeline) представляет участника взаимодействия: он символизирует существование объекта в конкретный момент на протяжении взаимодействия. *Сообщение* отражает существование связи между жизненными линиями в ходе взаимодействия. Линия жизни или объект, получающий сообщение, активизирует соответствующую операцию/метод. Момент, в который поток управления фокусируется на объекте, в языке UML 2.0 называется *выполнением спецификации* (ранее он назывался *активацией*).

Обозначения языка UML 2.0 для линий жизни и сообщений продемонстрированы на рис. 5.24. *Линии жизни* изображаются в виде именованных прямоугольников, расширенных вертикальными линиями (как правило, пунктирными). Прямоугольник, символизирующий линию жизни, может иметь имя, чтобы отражать следующие элементы:

- неименованный экземпляр класса — `:Class1`;
- именованный экземпляр класса — `:Class2`;
- класс, т.е. экземпляр метакласса, — `:Class3`, чтобы показать вызовы статических методов внутри классов;
- интерфейс — `:Interface`.

На рис. 5.24 показаны разные типы сообщений, разрешенных при моделировании взаимодействий.

- *Синхронные сообщения*, в которых вызывающие блоки, т.е. блоки, ожидающие ответа, представлены закрашенными стрелками — например, `doX`, `doA`, `doC`, `doD`.
- *Асинхронные сообщения* (asynchronous messages), в которых вызывающей стороной является не блок и, следовательно, допускаются многопоточные вычисления. Они представлены в виде открытых стрелок — например, `doB`.
- *Сообщения о создании объекта* (object creation messages), часто (но не всегда) сопровождаемые ключевыми словами, например `new` или `create`. Они представлены в виде открытой стрелки — например, `new(a,b)`, где `a` и `b` — параметры, передаваемые конструктору класса `Class2`.

- *Ответные сообщения* (reply messages), передающие результаты взаимодействия вызывающему модулю, активизировавшему действие. Они представляются в виде пунктирной линии с открытой стрелкой и часто сопровождаются описанием возвращаемого значения — например, `dValue`.

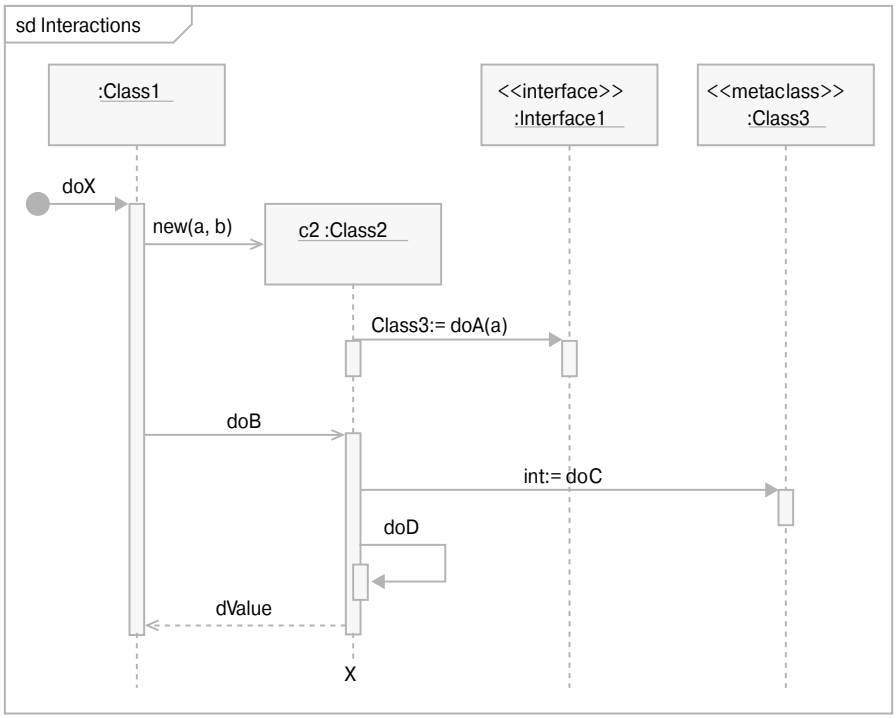


Рис. 5.24. Линии жизни и сообщения

Обратите внимание на то, что *ответное сообщение* — это один из двух способов представления возвращаемого значения. Второй способ — указание возвращаемой переменной в синтаксисе сообщения, например `Class3 = doA()` или `int = doC` (Larman, 2005). В зависимости от уровня абстракции, на котором формируется диаграмма, возвращаемые значения и ответы могут указываться или нет.

На рис. 5.24 также показано *найденное сообщение* (found message) — `doX`, — представляющее собой сообщение без указания отправителя. Иначе говоря, источник найденного сообщения находится за пределами видимости модели.

Разрушение объекта обозначается на диаграмме прописной буквой `X`. Обычно предполагается, что объекты, созданные в модели взаимодействия, уничтожаются в рамках той же самой модели (например, `c2:Class2`). Для языков, в которых не предусмотрена автоматическая сборка мусора (таких, как C++), уничтожение объекта должно инициироваться другим объектом с помощью отдельного сообщения «`destroy`».

Если линия жизни представлена с помощью интерфейса (`:Interface1`) или абстрактного класса, возникает естественное усложнение: вызванный метод выполняется в классе, реализующем интерфейс, или в конкретном классе, производном от абстрактного класса. В обоих вариантах рекомендуется создавать отдельные диаграммы взаимодействия для соответствующих реализаций интерфейса или для каждого полиморфного конкретного класса (Larman, 2005).

5.4.1.1. Учет базовой технологии

Даже если моделирование взаимодействий производится на достаточно высоком уровне абстракции, необходимо учитывать *программную технологию*, выбранную для разработки проекта. Современные языки программирования представляют собой скорее *среды программирования* (programming environment) с готовыми компонентами, библиотеками классов, файлами конфигураций XML, пользовательскими дескрипторами, соединениями с базами данных и т.п. Следовательно, б*о*льшая часть работы, выполняемой приложением, связана не с созданием нового кода, а с повторным использованием программ.

Оказывается, что детали взаимодействия пользовательского кода и среды трудно описать с помощью системы обозначения языка UML, поэтому нужны некоторые элементы, которые могли бы "заполнить пробел". Как минимум, необходимо указать используемую технологию с помощью стереотипов, сопровождающих линии жизни в диаграммах последовательности. (Предполагается, что читатель знаком с основами технологий программирования. В любом случае в примерах, сценариях и упражнениях приводится краткое описание технологий.)

В языке Java основными технологиями создания Web-приложений являются Java Server Pages (JSP), сервлеты и JavaBeans. *Сервлет* (servlet) — это программа на языке Java, которая разворачивается и выполняется на Web-сервере. Как правило, сервлет не имеет графического пользовательского интерфейса, и его можно отнести к уровню контроллера. Графический пользовательский интерфейс поставляется клиентами сервлета, например страницей сервера или аплетом.

Java Server Pages (JSP) — это страницы на языке HTML с фрагментами кода на языке Java. Для того чтобы выполнить приложение, действующее лицо запрашивает страницу JSP, набирая на клавиатуре адрес URL, например `www.myserver.com/myJSP.jsp`. Страницы JSP относятся к уровню презентации.

Компоненты *JavaBeans* — это классы на языке Java, способные хранить данные и выполнять определенные правила, позволяющие пользователю получать и отправлять данные с помощью методов `get()` и `set()` соответственно. Язык Java имеет механизм компонентов, позволяющих страницам JSP формировать значения, автоматически получаемые от компонента и отправляемые компоненты. Компоненты JavaBeans относятся к слою компонента.

Диаграмма последовательностей, соответствующая сценарию, описанному в примере 5.10, показана на рис. 5.25. Проект использует принцип CNP модели

PCBMER, в соответствии с которым перед каждым именем класса указывается первая буква пакета или подсистемы. Например, имя `PRequest` означает, что класс принадлежит пакету или подсистеме презентации.

Пример 5.10. Конвертация валют

Вернитесь к задаче 7, в которой описывается конвертация валют (см. раздел 1.6.7 главы 1). Предположим, что приложение состоит из двух Web-страниц: одна — для ввода данных и другая — для вывода результатов. Приложение должно получать текущий валютный курс из базы данных.

Постройте диаграмму последовательностей для приложения, конвертирующего валюту. Разработайте модель для технологии Java, состоящую из страниц JSP, сервлетов и компонентов JavaBeans. Следуйте принципам модели PCBMER, но можете не использовать уровень сущностей (см. раздел 4.1.3.1 главы 4). Показывать параметры и типы возвращаемых значений не обязательно.

Опишем диаграмму последовательностей, представленную на рис. 5.25. Страница `PRequest.jsp` посылает сервлету — классу `CCalculator` — запросы и получает от него ответы. Класс `CCalculator` просит класс `MMediator` выполнить операцию `getRate()`, а класс `MMediator` делегирует этот запрос классу `RQuery`. Класс `RQuery` получает валютный курс из базы данных и возвращает его классу `CCalculator`. Класс `CCalculator` создает и заполняет экземпляр класса `BBean`. Кроме того, класс `CCalculator` объявляет страницу JSP (в нашем примере — `PResult`), на которую должны выводиться результаты. Страница `PResult` обращается к данным, хранящимся в компоненте `BBean`, и передает их Web-браузеру. В итоге класс `CCalculator` получает от страницы `PResult` сообщения `startOver()` и передает их странице `PRequest`.

Представление результатов и другие детали, например доступ пользователя и оформление запроса, относятся к сфере ответственности Web-контейнера. Таким образом, класс `CurrencyConverter` просто сообщает Web-контейнеру, что страница `PResult` запрашивает ответ, и просит передать управление странице `PRequest`, когда пользователь хочет начать новые вычисления.

5.4.1.2. Визуализация информации о технологии в моделях взаимодействия

Линии жизни, снабженные стереотипами и ссылками на программную технологию, полезны, но не могут ясно показать, как пользовательский и технологический код взаимодействуют при выполнении приложения. Как указывалось ранее, создание моделей взаимодействия для пользовательского кода без учета технологических

деталей оставляет значительные пробелы в моделях и может оказаться бесполезным. С другой стороны, отражение технологических деталей в моделях взаимодействия может оказаться проблемой, поскольку язык UML не зависит от технологии. Даже если существует технологический профиль языка UML (см. раздел 5.1.1) или он разработан в рамках проекта, мощность и постоянные изменения технологии не позволяют точно показать, что именно происходит в ходе выполнения кода.

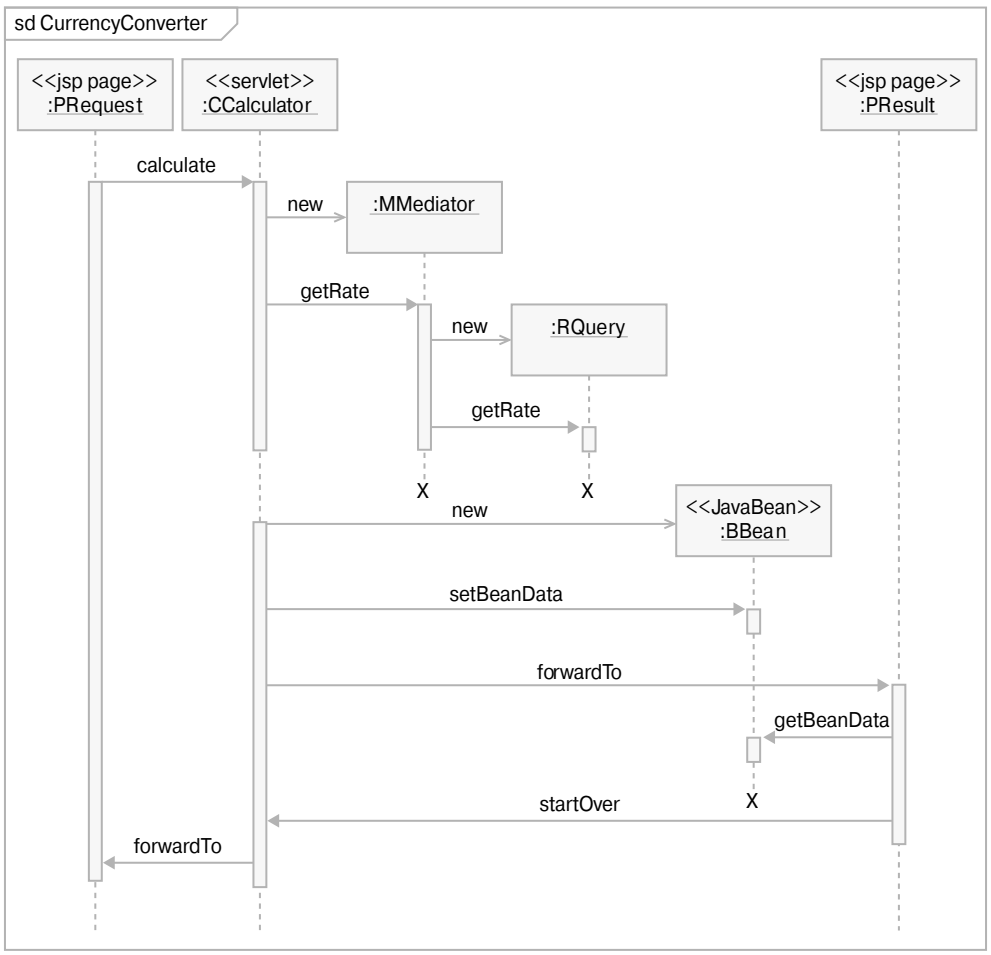


Рис. 5.25. Диаграмма последовательностей, использующая технологию JSP/servlet/JavaBeans для описания системы конвертации валют

С учетом этой оговорки рассмотрим простой пример приложения, состоящего из страницы JSP, создающей форму HTML, и сервлета, реализующего логику приложения и отражающего результаты вычислений с помощью Web-браузера. Технология JSP/сервлет позволяет использовать преимущества библиотеки классов

javax.servlet.*. Кроме того, она использует файл *дескриптора развертывания* (deployment descriptor) web.xml, содержащий инструкции о развертывании приложения. Помимо прочего, файл web.xml содержит имя сервлета, который должен быть вызван в соответствии с адресом URL. На этот адрес пользователи должны сослаться или непосредственно ввести его с клавиатуры в окне Web-браузера, чтобы отобразить начальную страницу JSP. Следовательно, когда пользователи укажут эту страницу, Web-сервер (*контейнер*) будет знать, какой сервлет следует вызвать.

Таким образом, сервлет связан с вводом названия страницы JSP и вычислением результатов. Выполняя эти задачи, сервлет основывается на реализации интерфейсов javax.servlet.* с именами HttpServletRequest (для ввода) и HttpServletResponse (для вывода). Во многих случаях класс сервлета состоит лишь из одного метода — doGet() или doPost(), — способного принять ввод, провести вычисления и сгенерировать вывод. Метод doGet() используется для получения запроса GET, в котором информация посылается как часть адреса URL, а метод doPost() используется для получения запроса POST, в котором информация посылается как часть сообщения, а не адреса URL. Запросы POST обычно используются, когда необходимо передать сервлету значения полей в форме HTML.

Диаграмма последовательностей, реализующая сценарий, описанный в примере 5.11, показана на рис. 5.26. Несмотря на то что эта диаграмма вполне проста и очевидна, некоторые элементы моделирования необходимо описать дополнительно. Например, взаимодействия с действующим лицом The User моделируются с помощью асинхронных сообщений, хотя, за исключением сообщения submitForm, другие сообщения представляют собой лишь вводимые данные. Аналогично, визуализация передачи сообщений от страницы JSP к сервлету, закодированная в файле web.xml, несколько произвольна. Вызовы интерфейсов HttpServletRequest и HttpServletResponse демонстрируют, как интерфейс Java/J2EE API (или, скорее, его реализация с помощью Web-сервера или сервера приложения) облегчает программирование Web-приложений.

Пример 5.11. Конвертация валют

Вернитесь к задаче 7, в которой описывается конвертация валют (см. раздел 1.6.7 главы 1), а также к примеру 5.10 из предыдущего раздела. Рассмотрим упрощенную реализацию системы, конвертирующей деньги из одной валюты (например, австралийские доллары) в другую (например, американские доллары). Пользователь должен ввести с клавиатуры сумму денег, подлежащих конвертации, и текущий курс этих валют. Получив эту информацию, приложение вычисляет сумму в целевой валюте и отражает результаты в трех полях: сумма конвертируемых денег, обменный курс и сумма денег, вычисленных в результате конвертации.

Постройте диаграмму последовательностей, демонстрирующую описанный сценарий. Разработайте модель для технологии Java, состоящей только из страниц JSP и сервлетов. Для того чтобы показать взаимодействие с пользователем, создайте действующее лицо `The User`. Для моделирования отображения между страницей JSP и сервлетом используйте действующее лицо `web.xml`. Кроме того, покажите линии жизни интерфейсов `HttpServletRequest` и `HttpServletResponse`. Показывать параметры методов и типы возвращаемых значений не обязательно.

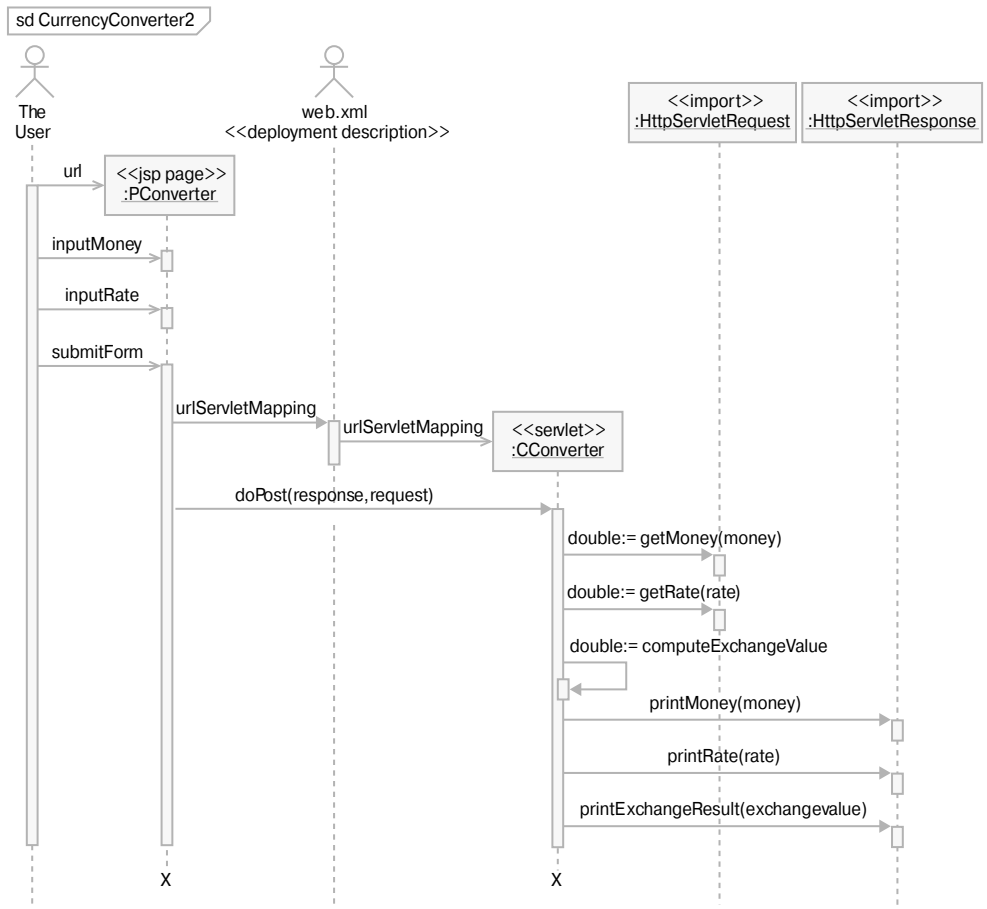


Рис. 5.26. Подробная диаграмма последовательностей, использующая технологию JSP/servlet для описания системы конвертации валют

5.4.2. Фрагменты

Часть взаимодействия называется **фрагментом взаимодействия** (interaction fragment). Взаимодействия могут содержать более мелкие фрагменты взаимодействия, называемые *комбинированными фрагментами* (combined fragments). Семантика комбинированного фрагмента определяется *оператором взаимодействия* (interaction operator). В языке UML 2.0 предусмотрено большое количество операторов, наиболее важные из которых перечислены ниже (Larman, 2005; UML, 2005).

- `alt`. Альтернативный фрагмент логического выражения многовариантного ветвления, выраженный в виде сторожевого условия.
- `opt`. Фрагмент, выполняемый, когда сторожевое условие является истинным.
- `loop`. Фрагмент цикла, повторяемый много раз в зависимости от условия цикла.
- `break`. Фрагмент прерывания, который выполняется вместо оставшегося вложенного фрагмента, если условие выхода является истинным.
- `parallel`. Параллельный фрагмент, допускающий перемежающееся выполнение функций.

На рис. 5.27 показано, как представляются фрагменты на диаграмме последовательностей. В модели показан *альтернативный фрагмент* — `alt`, который содержит *опциональный фрагмент* — `opt`. *Опциональный фрагмент* выполняется только в условии `else` альтернативного фрагмента и только если сторожевое условие `y<=0` является истинным.

Пример 5.12. Конвертация валют

Вернитесь к задаче 7, в которой описывается конвертация валют (см. раздел 1.6.7 главы 1). Рассмотрим настольную реализацию системы, конвертирующей деньги из одной валюты (например, австралийские доллары) в другую (например, американские доллары). Приложение состоит из одного языка класса Java — `CurrencyConvertor`.

Фрейм содержит три поля: для суммы в австралийских долларах, для суммы в американских долларах и для обменного курса. Кроме того, он содержит три кнопки (`btn`): `USD to AUD`, `AUD to USD` и `Close` (для выхода из приложения). Класс содержит метод `actionPerformed()`. Когда пользователь щелкает на одной из трех кнопок, происходит вызов этого метода из объекта `ActionEvent`. Метод `getSource()`, вызываемый из объекта `ActionEvent`, позволяет системе определить, на какой из кнопок щелкнул пользователь и какие вычисления следует выполнить.

Постройте диаграмму последовательностей, демонстрирующую описанный сценарий. При определении кнопки, на которой щелкнул пользователь, используйте альтернативный фрагмент условной логики.

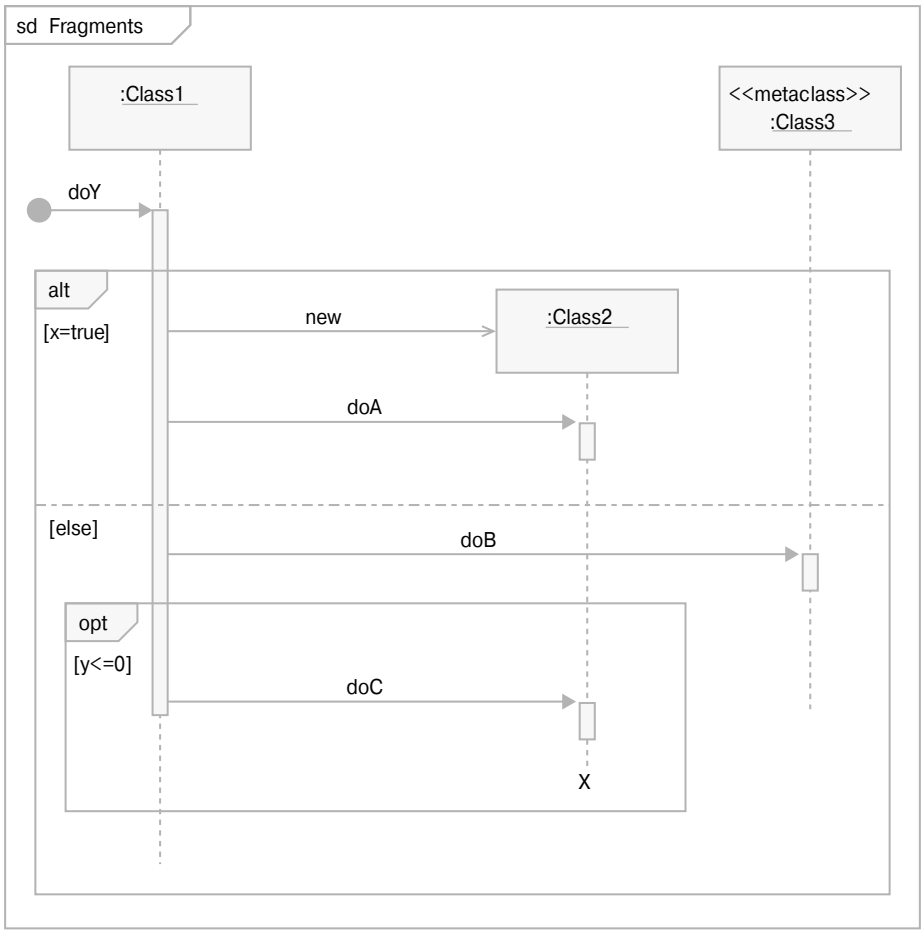


Рис. 5.27. Фрагменты

Диаграмма последовательностей, реализующая сценарий, описанный в примере 5.12, показана на рис. 5.28. Для обработки взаимоисключающих условий, соответствующих трем кнопкам, используется трехвариантный альтернативный фрагмент (условие `[else]` реагирует на событие `Close` выполнением метода `exit`). Все приложение состоит из одного класса — `CurrencyConverter`. Объект `ActionEvent` предоставляется библиотекой Swing.

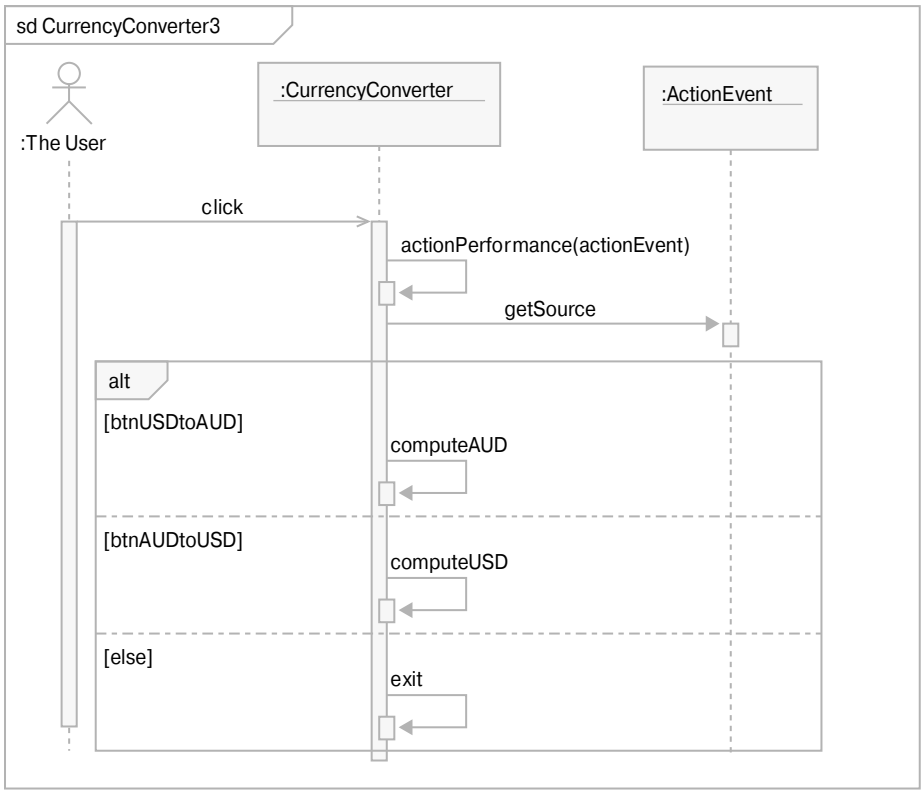


Рис. 5.28. Диаграмма последовательностей, использующая технологию Swing для описания системы конвертации валют

5.4.3. Использование взаимодействия

Кроме комбинированных фрагментов, взаимодействия могут содержать другие взаимодействия. Такие ссылки на взаимодействие называются *использованием взаимодействия* (interaction use). Объемлющее взаимодействие помечается дескриптором `sd` (sequence diagram — диаграмма последовательностей), как показано на рис. 5.24–5.28. Использование взаимодействия помечается дескриптором `ref` (reference — ссылка) и ссылается на другое взаимодействие `sd`, созданное отдельно.

Понятие использования взаимодействия позволяет извлекать и разделять общее поведение. Благодаря этому обеспечивается возможность повторного использования кода. Кроме того, полезно упростить сложную модель взаимодействий, разделив ее на множество отдельно определенных вариантов использования взаимодействия. Простой пример моделирования взаимодействий приведен на рис. 5.29.

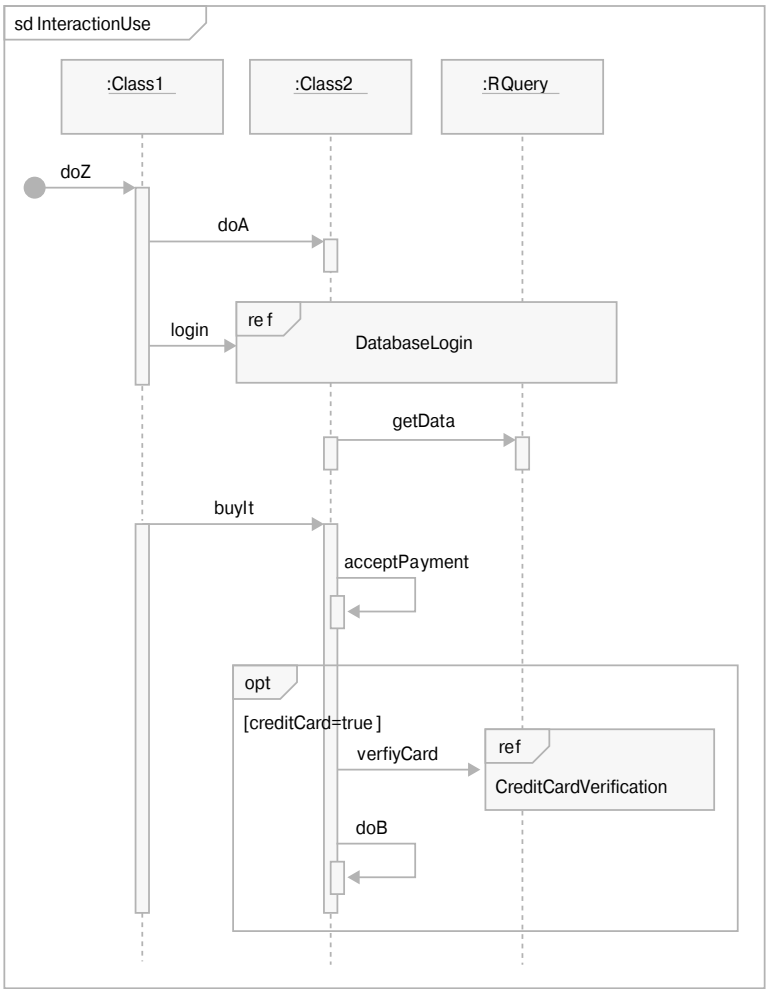


Рис. 5.29. Использование взаимодействий

Контрольные вопросы 5.4

- Какая концепция моделирования взаимодействий используется для определения многопоточного выполнения?
- Какая концепция моделирования взаимодействий используется для определения взаимодействия с неизвестным отправителем?
- Какому архитектурному слою принадлежит сервлет?
- Какой дескриптор используется для обозначения использования взаимодействия?

Резюме

В этой главе было завершено рассмотрение анализа требований и перекинут мостик к анализу и проектированию систем. Была тщательно исследована поддержка объектной технологии для разработки крупномасштабных систем. Местами глава представляется технически сложной, но обеспечивает проникновение в сущность объектной технологии, которое не так просто найти в книгах по анализу и проектированию систем. Это глубокое проникновение во многом помогает раскрыть слабые стороны и недостатки объектной технологии.

Основным методом расширения языка UML является использование *стереотипов*, которые дополняются *ограничениями* и *дескрипторами*. Они позволяют моделировать свойства, выходящие за пределы предопределенных в UML. За счет изобретательности в этой главе удалось широко использовать стереотипы.

Открытая и *закрытая* видимость, рассмотренные в предыдущей главе, обеспечивают только базовую поддержку *инкапсуляции*. *Защищенная* видимость позволяет управлять инкапсуляцией в рамках структур наследования. Понятие *дружественности* и *пакетная видимость* позволяют "прорваться" сквозь инкапсуляцию для обработки специфических ситуаций. *Видимость класса* (в противоположность видимости отдельных атрибутов и операций) представляет собой еще одну важную концепцию, касающуюся наследования.

Язык UML включает дополнительные концепции моделирования, способные повысить выразительность моделей классов. К ним относятся *производные атрибуты*, *производные ассоциации* и *квалифицированные ассоциации*. Одним из наиболее интригующих моментов в моделировании классов является выбор между *ассоциативным* и *материализованным классами*.

Концепция *обобщения* и *наследования* представляет собой в моделировании "палку о двух концах". С одной стороны, она способствует повторному использованию программного обеспечения и повышает выразительность, понятность и уровень абстракции моделей системы. С другой стороны, при неверном использовании всех перечисленных преимуществ она потенциально склонна к саморазрушению.

Концепция *агрегации* и *делегирования* представляет собой важную альтернативу обобщению и наследованию в моделировании. *Делегирование* и *системы-прототипы* обладают дополнительными преимуществами поддержки иерархических архитектурных структур. Абстрактное понятие *холона* дает интересный взгляд на способ построения сложных систем.

Диаграммы последовательностей являются предпочтительным средством моделирования взаимодействия и обеспечивают хорошую поддержку сложных задач моделирования. *Комбинированные фрагменты* были добавлены в модели взаимодействия в языке UML 2.0 для того, чтобы учесть детальную логику программирования. Сложные модели можно создавать с помощью *использования взаимодействий*.

Ключевые термины

Видимость (visibility). "Перечисление, значения которого (открытая, защищенная, закрытая или пакетная) означают, может ли элемент модели, к которой они относятся, быть доступным в рамках объемлющего пространства имен" (Rumbaugh et al., 2005).

Использование взаимодействия (interaction use). "Ссылка на взаимодействие в определении другого взаимодействия" (Rumbaugh et al., 2005).

Комментарий (comment). Аннотация, сопровождающая элемент или набор элементов модели. Комментарий не влияет на семантику модели.

Метакласс (metaclass). "Класс, экземплярами которого являются классы... обычно используемые для создания метамоделей" (Rumbaugh et al., 2005).

Метамодель (metamodel). "Модель, определяющая язык для выражения других моделей" (Rumbaugh et al., 2005).

Меченое значение (tagged value). "Пара "имя–значение", которую можно присоединить к элементу модели, использующему стереотип, содержащий определение дескриптора" (Rumbaugh et al., 2005).

Ограничение (constraint). "Условие, выраженное на естественном или машинном языке для объявления семантики элемента" (UML, 2005).

Определение дескриптора (tag descriptor). Свойство стереотипа, выраженное в виде атрибута в прямоугольнике класса, содержащем объявление стереотипа.

Производная информация (derived information). "Элемент, который вычисляется на основе других элементов и включается в модель для ее прояснения или для упрощения проектирования, даже если он не имеет дополнительной семантической ценности" (Rumbaugh et al., 2005).

Профиль (profile). "Определяет ограниченные расширения ссылочной метамодели для адаптации метамодели к конкретной платформе или предметной области" (UML, 2005).

Стереотип (stereotype). "Определяет способ расширения существующего метакласса, облегчает использование платформы или специальной терминологии, ориентированной на предметную область, или системы обозначений вместо или в дополнение к используемой в метаклассе" (UML, 2005).

Фрагмент взаимодействия (interaction fragment). "Структурная часть взаимодействия" (Rumbaugh et al., 2005).

Многовариантные тесты

МТ1. Что из перечисленного ниже не является механизмом расширения в языке UML?

а. Ограничение.

б. Стереотип.

в. Производный атрибут.

г. Меченое значение.

МТ2. Что из перечисленного ниже является синонимом наследования интерфейса?

а. Выделение подтипа.

б. Заменимость.

в. Полиморфизм.

г. Ничто из перечисленного выше.

МТ3. Наследование, при котором происходит замещение унаследованных свойств в подклассе, называется...

а. Расширяющим наследованием.

б. Удобным наследованием

в. Ограничивающим наследованием.

г. Все перечисленные варианты не верные.

МТ4. В рамках какой концепции всегда появляется авторекурсия?

а. Делегирование.

б. Наследование интерфейса.

в. Пересылка.

г. Наследование реализации.

МТ5. Как в языке UML 2.0 называется время, в течение которого поток управления фокусируется на объекте?

а. Использование взаимодействия.

б. Спецификация выполнения.

в. Линия жизни.

г. Все перечисленные варианты не верные.

МТ6. Какой из следующих операторов определяет параллельный фрагмент, позволяющий перемежающееся выполнение вложенных функциональных свойств?

а. `Opt`.

б. `Loop`.

в. `Alt`.

г. Ни один из перечисленных выше.

Вопросы

B1. Что в языке UML называется “профилем”? Приведите пример.

B2. Иногда класс позволяет порождать только неизменяемые объекты, т.е. объекты, которые не могут изменяться после реализации. Каким образом подобное требование можно представить в UML-модели?

B3. Объясните различие между ограничением и примечанием.

B4. Обозначают ли термины “инкапсуляция” и “видимость” одно и то же понятие? Объясните вашу точку зрения.

B5. Видимость унаследованных свойств в производном классе зависит от уровня видимости, который предоставлен базовому классу в определении этого производного класса. Какова будет видимость, если базовый класс объявлен как закрытый? Каковы последствия этого факта для остальной модели? Приведите пример.

B6. Понятие дружественности применимо к классу или операции. Объясните, в чем различие этих случаев. Приведите пример (отличный от изложенного в этой книге), когда требуется использование свойства дружественности.

B7. В чем заключаются преимущества использования производной информации для моделирования?

B8. Когда материализованный класс должен заменять ассоциативный? Приведите пример (отличный от приведенного в этой книге).

B9. В чем заключается принцип заменимости? Обоснуйте свой ответ.

B10. Объясните разницу между наследованием интерфейса и наследованием реализации. Используйте в ответе понятия наследования от интерфейсного, абстрактного или конкретного класса.

B11. В чем состоит проблема изменчивости базового класса? Каковы основные причины изменчивости базовых классов?

B12. Объясните различия между агрегациями *ExclusiveOwns* и *Owns*. Какие преимущества при моделировании дает разделение агрегации на два этих вида?

B13. Сравните наследование и делегирование. В чем их схожесть и различие?

B14. Как правило, обработка сводится к перебору всех объектов в коллекции (например, массиве или списке) и посылке каждому из них одного и того же сообщения. Как это можно смоделировать с помощью диаграмм последовательностей? Приведите пример.

B15. Можно ли с помощью сочетания диаграмм деятельности и последовательностей создать полезное обозначение для моделирования? Обоснуйте свой ответ.

Упражнения

У1. Обратитесь к рис. А.14 (см. раздел А.5.2 приложения А). Предположим, что преподаватель, который *руководит* курсом, должен также *вести* этот курс. Измените диаграмму на рис. А.14, чтобы отразить этот факт.

У2. Обратитесь к рис. А.20 (см. раздел А.7 приложения А) и рис. А.21 (см. раздел А.7.1 приложения А). Объедините оба рисунка в виде одной модели классов. Разработайте схему видимости для модели классов. Обоснуйте свой ответ.

У3. Обратитесь к рис. А.16 (см. раздел А.5.4 приложения А). Предположим, что система должна отслеживать успеваемость студентов по изучению нескольких курсов одной и той же дисциплины. Это связано с ограничением, которое гласит, что студент может провалить один и тот же курс не более трех раз (запись на этот же курс в четвертый раз не разрешается). Расширьте диаграмму на рис. А.16 таким образом, чтобы учесть в модели это ограничение. Используйте при этом материализованный класс. Введите в модель любые предположения и поясните их.

У4. Обратитесь к примеру 4.10 (см. раздел 4.2.4.3 главы 4). Перерисуйте диаграмму на рис. 4.10, используя агрегацию вместо обобщения. Приведите аргументы за и против новой модели.

У5. Обратитесь к рис. 4.18 (см. раздел 4.3.3.3 главы 4). Уточните диаграмму последовательностей на рис. 4.18 с помощью комбинированных фрагментов и других усовершенствованных понятий, связанных с моделированием взаимодействий.

У6. Обратитесь к рис. 4.18 (см. раздел 4.3.3.3 главы 4). Уточните диаграмму последовательностей на рис. 4.19 с помощью комбинированных фрагментов и других усовершенствованных понятий, связанных с моделированием взаимодействий.

Упражнение. Регистрация времени

Дополнительная информация

Вернитесь к задаче 6, в которой описывается система регистрации времени (см. раздел 1.6.6 главы 1). Рассмотрим функцию, с помощью которой сотрудник использует возможность запуска секундомера в программе Time Logger, чтобы начать новую запись. Эта функция возлагается на прецедент использования "*Создать запись — запуск секундомера*". Окно графического интерфейса, поддерживающее этот подпоток, показан на рис. 5.30.

Окно секундомера представляет собой немодальное диалоговое окно. Это позволяет пользователю получить доступ к записям о времени в первичном окне браузера, а также к другим свойствам программы Time Logger. Экран дисплея, показанный на рис. 5.30, показывает секундомер в состоянии запуска, в которое он переходит после старта из меню `Stopwatch`.

Окно имеет кнопки для запуска/остановки секундомера. Если секундомер запущен, пользователь может использовать записи списка для выбора и поля `Description` (Описание) для заполнения информации о своих действиях. Если пользователь щелкнул на кнопке `Stop`, программа Time Logger добавляет новую запись в строку браузера.

Параметр `Duration` (Продолжительность) вычисляется по содержимому полей `Start` (Начало), `Now` (Текущее время) и `Pause Duration` (Продолжительность паузы). Поле `Now` (Текущее время) не редактируется. Продолжительность паузы управляется кнопками `Pause` (Пауза) и `End Pause` (Конец паузы).

Кнопка `Reset` (Сброс) сбрасывает показания секундомера без сохранения записи в базе данных. Кнопка `Hide` скрывает секундомер и переводит его в фоновый режим работы. Скрытый секундомер можно вновь показать, используя меню `Stopwatch` (Секундомер).

Кнопки с пиктограммами, на которых изображены знак "плюс", карандаш и знак "минус", позволяют создавать, обновлять и удалять соответствующие элементы списка.

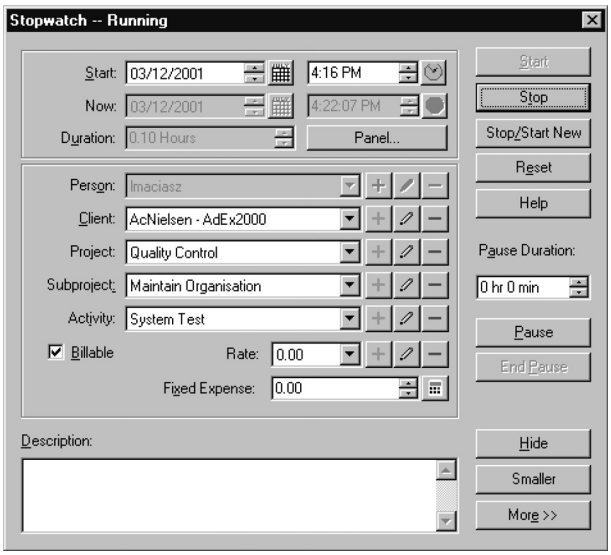


Рис. 5.30. Окно остановки отсчета в системе регистрации времени

PB1. Разработайте высокоуровневую диаграмму коммуникации для подпотока "Создать запись — запуск секундомера". На диаграмме следует показать только основные классы и сообщения, которыми они обмениваются. Нумеровать сообщения не обязательно. Сигнатуру сообщений указывать не обязательно. Проект должен соответствовать модели PCBMER, но классы медиатора и компонентов не являются обязательными. Объясните предположения и потенциальные недоразумения, а также неоднозначные сообщения.

PB2. Основываясь на решении упражнения PB1, разработайте диаграмму классов для подпотока "Создать запись — запуск секундомера". На диаграмме следует показать операции, но не обязательно показывать атрибуты. Классы должны быть связаны необходимыми статическими отношениями. Если между классами нет статических отношений, но они взаимодействуют между собой на этапе выполнения программы, следует использовать отношения зависимости. Объясните предположения и потенциальные недоразумения, а также неоднозначные части модели.

Упражнение. Затраты на рекламу

Дополнительная информация

Вернитесь к задаче 5, в которой описывается система учета затрат на рекламу (см. раздел 1.6.5 главы 1). Вспомните также решения упражнений, приведенных в конце главы 2. Рассмотрите функцию, с помощью которой пользователь поддерживает списки категорий и соответствующих рекламируемых товаров. Эта функция относится к сфере ответственности прецедента использования "Поддержка связи категория–товар". Часть окна графического интерфейса, поддерживающего этот подпоток, показана на рис. 5.31.

Окно `Maintain Category-Product Links` (Поддержка связи категория–товар) состоит из двух панелей: `Categories` (Категории) и `Products for [Active Category]` (Товары для [активной категории]), в данном случае — `Products for consolidated loans`. В окне `Categories` содержится браузер дерева. Категорию можно раскрыть или свернуть, щелкнув на знаке "плюс" или "минус" соответственно. Категории, содержащие подкатегории, идентифицируются пиктограммой папки. Категории на листах дерева (не имеющих подкатегорий) показаны с помощью пиктограммы заметки.

Выбор (подсветка) категории, не имеющей подкатегорий, а также список товаров в категории показаны в правой части окна.

Двойной щелчок на любой категории открывает окно `Update Category` (Обновить категорию). Выпадающее меню содержит пункт `Reassign From` (Перенести), который используется для переноса товара в другую категорию.

ЗР1. Разработайте высокоуровневую диаграмму коммуникации для подпотока "Поддержка связи категория–товар". На диаграмме следует показать только основные классы и потоки сообщений между ними. Нумеровать сообщения не обязательно. Сигнатуру сообщений указывать не обязательно. Проект должен соответствовать модели PCBMER, но классы медиатора и компонентов не являются обязательными. Объясните предположения и потенциальные недоразумения, а также неоднозначные сообщения.

ЗР2. Основываясь на решении упражнения ЗР1, разработайте диаграмму классов для подпотока "Поддержка связи категория–товар". На диаграмме следует показать операции, но не обязательно показывать атрибуты. Классы должны быть связаны необходимыми статическими отношениями. Если между классами нет статических отношений, но они взаимодействуют

между собой на этапе выполнения программы, следует использовать отношения зависимости. Объясните предположения и потенциальные недоразумения, а также неоднозначные части модели.

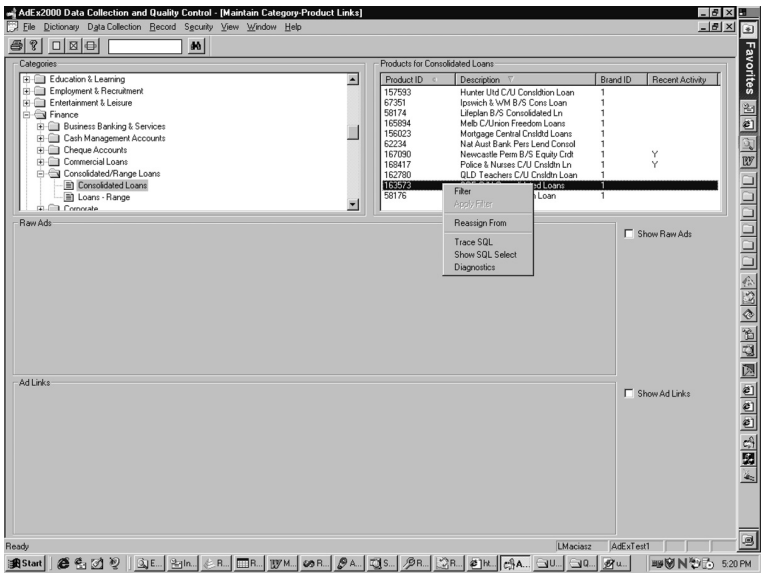


Рис. 5.31. Окно `Maintain Category-Product Links` *в системе учета расходов на рекламу. Источник: компания Nielsen Media Research, Sydney, Australia*

Ответы на контрольные вопросы

Контрольные вопросы 5.1

КВ1. Стереотип.

КВ2. Имена полюсов ассоциации.

КВ3. Пакетная видимость.

КВ4. Да, материализованные классы всегда могут заменить класс ассоциации без потери семантики (но не наоборот).

Контрольные вопросы 5.2

КВ1. Принцип заменимости.

КВ2. Открывая подклассам прямой доступ к защищенным атрибутам.

КВ3. Наследование интерфейса.

Контрольные вопросы 5.3

КВ1. Реализуется как обычная ассоциация с помощью запроса ссылок между составными и компонентными объектами.

КВ2. Агрегация *ExclusiveOwns*.

КВ3. Делегирование.

Контрольные вопросы 5.4

КВ1. Асинхронные сообщения.

КВ2. Найденное сообщение.

КВ3. Уровень контроллера.

КВ4. Дескриптор `ref` (ссылка).

Ответы к многовариантным тестам

МТ1. в

МТ2. а

МТ3. в

МТ4. г

МТ5. б

МТ6. г (этот оператор называется параллельным)

Ответы на вопросы с нечетными номерами

В1

"*Профиль* охватывает часть языка UML и расширяет ее с помощью согласованной группы специализированных стереотипов, например, для бизнес-моделирования" (Fowler, 2003). Для того чтобы отобразить специфику предметной области, стереотипы профиля должны обозначаться новыми графическими пиктограммами. Профили в основном необходимы для *проектного моделирования* и реже — для *аналитического моделирования*. Яркий пример — профиль для создания Web-приложений, опубликованный в работе Коналлена (Conallen, 2000).

Для разработки программного обеспечения изобретены специальные профили. Некоторые из них можно найти по адресу `www.jeckle.de/uml_spec.html#profiles`.

Беглый поиск в Интернете привел к следующими результатам.

- Моделированные данных
  `www.agiledata.org/essays/umlDataModelingProfile.html`
- CORBA (Common Object Request Broker Architecture) — технология создания распределенных объектных приложений, предложенная фирмой IBM.
  `www.omg.org/technology/documents/formal/profile_corba.htm`
- Web-моделирование, схема XSD, моделирование бизнес-процессов
  `www.sparxsystems.com.ua/uml_profiles.htm`
- MOF (Meta-Object Facility — механизм бизнес-объектов) — стандарт группы OMG (Object Management Group) для модельного проектирования
  `mdr.netbeans.org/uml2mof/profile.html`
- Профили для бизнес-моделирования и программирования
  `www-128.ibm.com/developerworks/rational/library/5167.html`
  `www-128.ibm.com/developerworks/rational/library/05/419_soa/`
- Оценка рисков на основе моделирования
  `coras.sourceforge.net/uml_profile.html`
- Встроенные системы реального времени
  `www-omega.imag.fr/profile.php`

B3

Ограничение — это семантическое условие или оговорка, размещенная на элементе моделирования UML. Ограничение может быть изображено графически как текстовая строка, заключенная в фигурные скобки, или как символ заметки (прямоугольник с загнутым правым верхним углом).

Как правило, более сложные ограничения представляются в виде замечания. Поскольку замечание — это отдельный графический элемент, его можно визуально — с помощью отношений — связать с другими элементами моделирования UML.

Заметка является лишь графическим средством выражения ограничения. Символ замечания не имеет самостоятельного семантического значения. Заметку можно использовать для представления информации, не имеющей семантического смысла. Например, замечание может содержать комментарий.

B5

Реализация *видимости* в разных объектно-ориентированных языках может значительно отличаться друг от друга (Fowler, 2004; Page-Jones, 2000). Что касается вопроса, то закрытые свойства данного класса остаются закрытыми и не доступны для объектов производного класса (независимо от того, какой вид наследования используется — закрытый, защищенный или открытый).

Если базовый класс *A* объявлен как закрытый в подклассе *B* (`class B: private A`), то видимость свойств, унаследованных от класса *A*, в классе *B* становится закрытой. Иначе говоря, открытые или закрытые свойства класса *A* становятся закрытыми свойствами класса *B*. В результате дальнейшая специализация класса *B* (например, `class C: public B`) закроет классу *C* доступ к каким-либо свойствам класса *A*. Таким образом, несмотря на то что класс *C* наследует свойства класса *A*, объекты класса *C* не имеют доступа к свойствам класса *A*. Свойства класса *A* становятся невидимыми в классе *C*, хотя определения некоторых свойств в классе *C* могут быть выведены из родительских свойств класса *A*.

Пример, показанный на рис. 5.32, отдаленно напоминает пример из книги Мейерса (Meyers, 1998). Защищенный атрибут `tax_file_number` в классе `Person` становится закрытым атрибутом в классе `Student`. Если `std` — объект класса `Student`, то присваивание `x = tax_file_number` запрещено. Аналогично, вызов `std.sing()` является ошибкой.

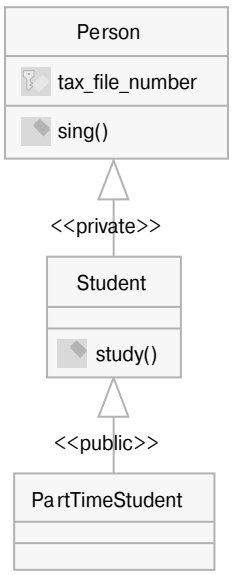


Рис. 5.32. Закрытая видимость класса

Соответствующий код на языке C++ показан ниже.

```
class Person
{
protected:
 int tax_file_number;
public:
 void sing(){};
};
```

```
class Student : private Person
{
};
void main()
{
 Student std;
 int x = std.tax_file_number; // Ошибка компиляции
 std.sing(); // Ошибка компиляции
}
```

Как указано в книге Мейерса (Meyers, 1998): "В противоположность открытому наследованию, компиляторы обычно не конвертируют производный класс (например, `Student`) в базовый класс (например, `Person`), если отношение наследования между классами является закрытым".

К сожалению, *закрытое наследование* является антиподом наследования "разновидность", пропагандируемого в данной книге (например, в разделе 4.2.4.1). Объект класса `Student` больше не является объектом класса `Person`. "Закрытое наследование означает, что класс "реализован посредством" базового класса. Класс *D* объявляется закрытым наследником класса *B*, если программист хочет воспользоваться готовым фрагментом кода, содержащимся в классе *B*, а не потому, что между объектами классов *B* и *D* существует концептуальное отношение" (Meyers, 1998).

Закрытое наследование во многих отношениях является синонимом *удобного наследования*, рассмотренного в разделе 5.2.4.3 и названного неприемлемым вариантом наследования реализации.

B7

Производная информация на самом деле не вносит в модель UML никакой новой информации, а лишь обогащает ее семантику. Она проясняет модель, выявляя информацию, которая в ином случае была бы скрыта от непосредственного наблюдения.

В модели *анализа* производную информацию можно использовать для именования или определения концепции или требования пользователя. В *проектной* модели производную информацию можно использовать для указания на то, что данное значение следует вычислить заново, поскольку величины, от которых она зависит, могут изменяться.

В большинстве практических ситуаций производная информация описывает *ограничение* на существующие свойства, например, что некоторое значение может быть вычислено по существующим величинам. Строго говоря, производная информация может упросить модель, даже если она является излишней.

B9

Принцип заменимости утверждает: "Если переменная или параметр объявлены с типом *X*, любой экземпляр класса, производного от класса *X*, может быть использован как фактический параметр без нарушения семантики его объявления и использования. Иначе говоря, экземпляр производного класса может быть замещен экземпляром базового класса" (Rumbaugh et al., 2005).

Отношение обобщения "является разновидностью" (см. раздел 4.2.4.1 главы 4) и поддерживает *заменимость*. Поскольку отношение "является разновидностью" реализуется с помощью *открытого наследования классов* (см. вопрос B5), принцип заменимости требует использования открытого наследования. Открытое наследование утверждает, что все, что применимо к объектам суперкласса, применимо и к объектам подкласса (подкласс не имеет права отвергать или модифицировать свойства своего суперкласса).

B11

Проблема изменчивости классов состоит в нежелательном влиянии эволюции суперклассов (базовых классов) на всю прикладную программу, содержащую подклассы, производные от суперклассов. Влияние таких изменений практически непредсказуемо, поскольку разработчики суперклассов не знают, как подклассы планируют использовать их свойства.

Если разработчик базового класса не ясновидец, то проблема изменчивости базового класса в объектно-ориентированных приложениях неизбежна. Любые изменения открытых интерфейсов в базовом классе неизбежно повлияют на подклассы. Изменения в реализации операций, наследуемых подклассами, могут иметь значительные последствия и трудно распознаются (особенно в случае реализаций по умолчанию, произвольно переопределяемых в подклассах).

Особый вид проблемы изменчивости класса возникает при *множественном наследовании* (см. раздел 5.2.4.4.3). По существу, любой конфликт, порожденный множественным наследованием, является вариантом проблемы изменчивости базового класса, которая возникает еще до того, как произойдет модификация подкласса.

B13

Наследование — это способ *повторного использования кода* с помощью отношений обобщения. *Делегирование* — это способ *повторного использования кода* с помощью отношений агрегации. В большинстве случаев выбор наследования (обобщения) или делегирования (агрегации) очевиден — семантика "является разновидностью" и требует обобщения, а семантика "содержит" требует агрегации.

Однако, как показано в сложном примере (см. рис. 5.23), обобщение можно реализовать с помощью агрегации. За исключением этой ситуации, наследование следует использовать в сочетании с семантикой "является разновидностью", а делегирование — в сочетании с семантикой "содержит".

Сходство между этими методами заключается в том, что оба они относятся к *повторному использования кода*. Разница между ними состоит в том, что наследование — это способ повторного использования кода в *классах*, а делегирование — способ повторного использования кода в *объектах*. Таким образом, делегирование — более мощный механизм, чем наследование.

Во-первых, делегирование может имитировать наследование, но не наоборот. Во-вторых, делегирование относится к этапу выполнения программы и поддерживает динамическую эволюцию системы, а наследование — статическое наследование, относящееся к этапу компиляции. В-третьих, делегирующий (внешний) объект может использовать как функциональные свойства (реализации операций), так и состояние (значения атрибутов) делегированных (внутренних) объектов, в то время как производный объект не наследует состояние.

B15

Комбинация диаграмм деятельности и последовательностей может повысить выразительность логики более сложных программ. Необходимость такого комбинированного моделирования в языке UML 2.0 была осознана с появлением диаграмм обзора взаимодействий.

Диаграмма обзора взаимодействий разделяет проблемную область на использование взаимодействия и комбинированные фрагменты (из диаграмм последовательностей), а также на конструкции потоков управления (обозначение ветвления и разделения из диаграмм деятельности). Управляющие конструкции позволяют делать обзор потока управления, а более специфичные вычислительные узлы моделируются с помощью использования взаимодействия и комбинированных фрагментов. Использование взаимодействий и фрагменты допускают анализ деталей взаимодействия.

Объяснение упражнений с нечетными номерами

У1

Легкое решение этого вопроса можно получить, наложив *ограничение* на две ассоциации (рис. 5.33). Ограничение `{subset}` означает, что преподавателем, ответственным за курс, должен быть один из преподавателей, читающих лекции на этом курсе.

У3

Решение, представленное на рис. 5.34, получено путем превращения класса ассоциации, показанного на рис. A.16 (см. раздел A.5.4 приложения A), в независимый класс `Assessment`. Замечание к ограничению (про правило “не более трех ассоциаций”) было добавлено в модель и связано с классом `Assessment` и его ассоциациями с двумя другими классами.

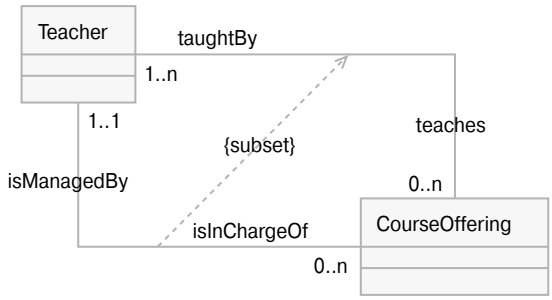


Рис. 5.33. Ограничение, наложенное на ассоциации

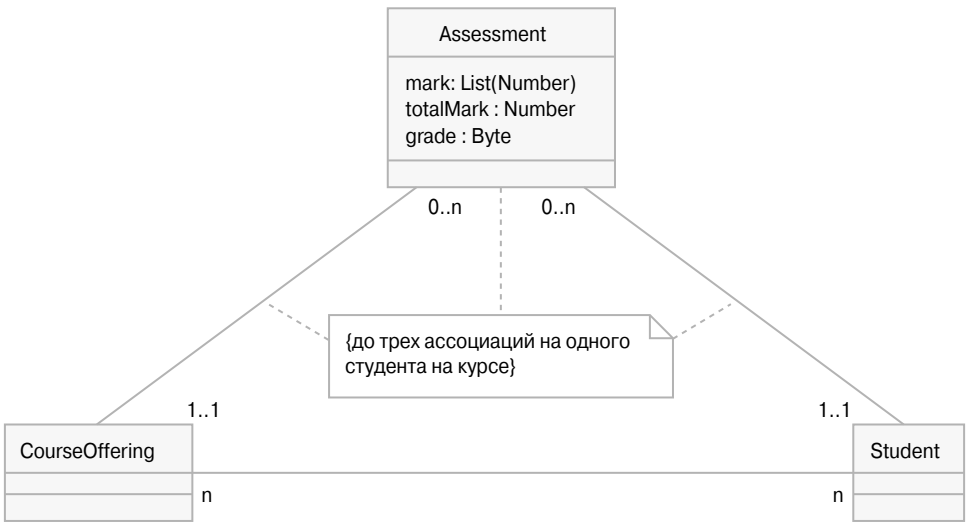


Рис. 5.34. Моделирование ограничения

У5

На рис. 5.35 показана улучшенная диаграмма последовательностей с централизованным решением примера, касающегося системы управления зачислением в университет. Эта модель использует три комбинированных фрагмента — альтернативу, цикл и опцию.

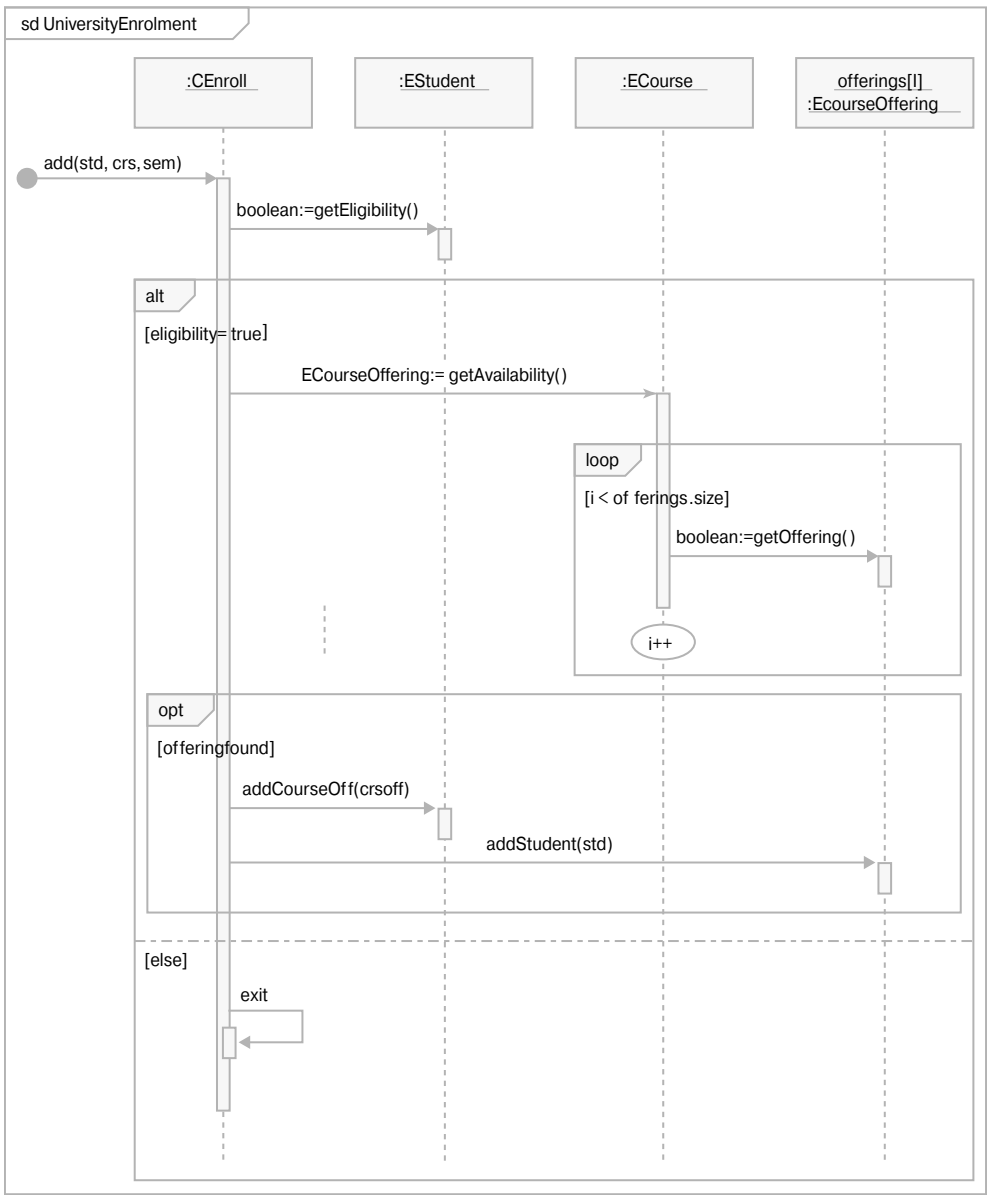


Рис. 5.35. Диаграмма последовательностей с комбинированными фрагментами

Объяснение упражнений. Регистрация времени

PB1

На рис. 5.36 приведена модель коммуникации для подпотока, управляющего секундомером. Выполнение подпотока начинается, когда пользователь активизирует секундомер из меню :CMenuItem. В этот момент открывается диалоговое окно :PStopwatch, предлагающее ввести данные. Для этого оно инициирует экземпляр :CStopwatchInitializer, который должен выполнить операцию refreshView().

Для того чтобы объект сущности :ETimeRecord появился на экране, он должен выполнить операцию getTimeRecord(). Это действие порождает большое количество сообщений другим объектам сущностей, чтобы они выполнили операции getDate(), getTime(), getPerson(), getClients(), getProjects() и getActivates().

Объекты — получатели этих сообщений — обращаются к классу :RReader, чтобы извлечь информацию из базы данных. Наша модель носит упрощенный характер и показывает, что услугами класса :RReader пользуется только класс :ETimeRecord.

Работая в диалоговом окне, пользователь может редактировать многие поля. Для того чтобы редактировать пункты списка, пользователь обращается к объектам :CMouseEvent. Выбранный пункт передается методу pickCurrent() в соответствующем контейнерном объекте (:EClientList, :EProjectList или :EActivityList). Как и ранее, доступ к базе данных инкапсулирован в объекте :RReader.

В заключение запрос stop(), поступающий от класса :CButton к классу :CStopwatchInitializer, порождает передачу сообщения saveTimeRecord() классу :ETimeRecord. Генерирование SQL-запроса для модификации базы данных возложено на класс :RUpdater.

PB2

На рис. 5.37 показана модель классов для подпотока, управляющего секундомером. Эта модель построена на основе решения, полученного в разделе PB1. Диаграмма упрощает модель PCBMER. В центре проекта находится управляющий класс CStopwatchInitializer, но основные задачи, связанные с установкой и определением времени, координируются классом сущности ETimeRecord. Этот класс выполняет множество операций с помощью своих компонентных классов. Для связи с базой данных класс ETimeRecords использует классы RReader и RUpdater.

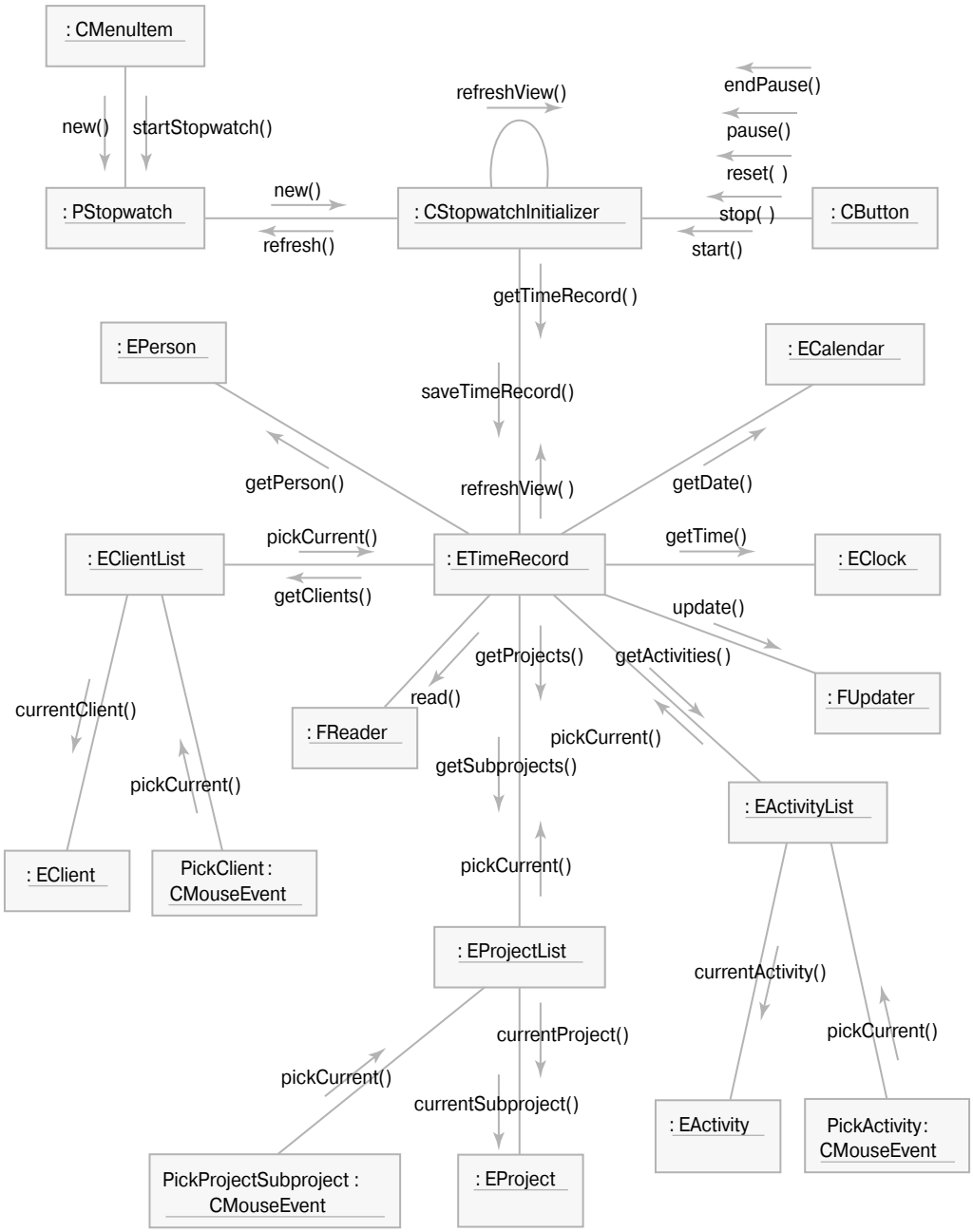


Рис. 5.36. Модель коммуникации для секундомера в системе регистрации времени

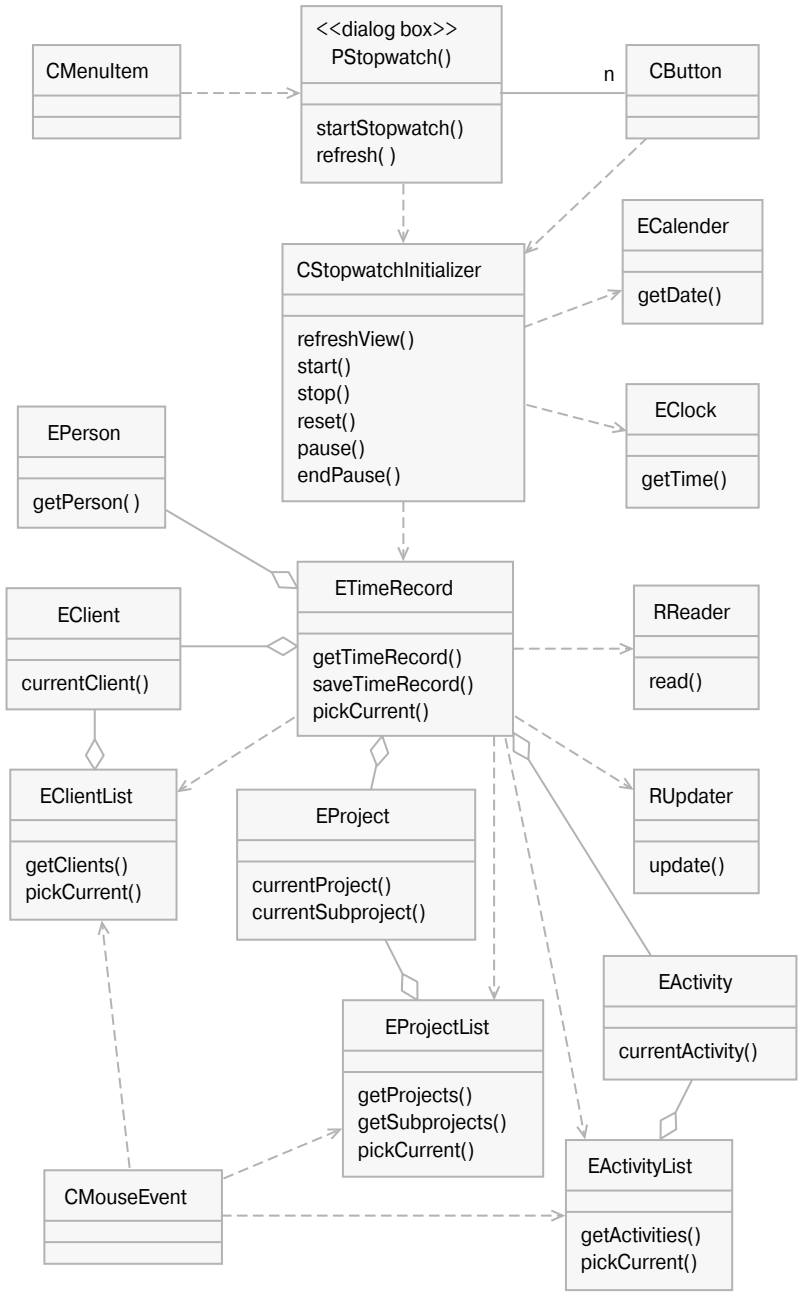


Рис. 5.37. Модель классов для секундомера в системе регистрации времени

ГЛАВА 6

Процесс разработки программного обеспечения

Цели

6.1. Распределенная физическая архитектура
6.2. Многоуровневая логическая структура
6.3. Архитектурное моделирование
6.4. Принципы разработки и повторного использования программ
6.5. Моделирование кооперации
Резюме
Ключевые термины
Многовариантные тесты
Вопросы
Упражнения. Магазин видеокассет
Упражнения. Затраты на рекламу
Ответы на контрольные вопросы
Ответы к многовариантным тестам
Ответы на вопросы с нечетными номерами
Объяснение упражнений. Затраты на рекламу

req_analysis_system design.indb 429 05.06.2008 14:43:16

Цели

В ходе итеративной и поступательной разработки программного обеспечения модели анализа постоянно обогащаются техническими деталями. Поскольку эти технические детали требуют ссылок на программное и аппаратное обеспечение, модель анализа преобразуется в модель проекта. *Системное проектирование* охватывает два основных вопроса: архитектурное проектирование системы и детальное проектирование программ, образующих систему.

Архитектурное проектирование — это описание системы в терминах ее модулей. Оно включает в себя решения, касающиеся стратегий работы клиентских и серверных компонентов. Архитектура определяет иерархическую организацию классов и пакетов, распределение процессов по вычислительным ресурсам, а также стратегию повторного использования и управления компонентами. Архитектурное проектирование решает проблемы, связанные с *многоярусной* физической и *многослойной* логической архитектурой.

Детализированным проектированием называется определение внутренней работы каждого модуля (прецедента использования). В ходе детализированного проектирования разрабатываются полные алгоритмы и структуры данных для каждого модуля. Эти алгоритмы и структуры данных учитывают ограничения, накладываемые базовой платформой реализации. *Детализированное проектирование* решает вопросы, связанные с моделями кооперации, необходимыми для реализации требуемых функциональных свойств, зафиксированных в прецедентах использования.

Прочитав эту главу, читатели будут

- понимать разницу между типичными распределенными физическими архитектурами;
- осознавать высокую важность многослойной логической архитектуры для создания качественных систем;
- уметь оценивать сложность логических архитектур;
- владеть практическими знаниями о прикладных шаблонах в целом и архитектурном проектировании в частности;
- понимать разницу между артефактами, используемыми для архитектурного моделирования;
- знать основные принципы правильного проектирования программ;
- владеть разными стратегиями повторного использования;
- знать методы моделирования кооперации и понимать их связь с моделированием прецедентов использования и взаимодействия.

req_analysis_system design.indb 430 05.06.2008 14:43:16

6.1. Распределенная физическая архитектура

Архитектурное проектирование имеет *физические* и *логические* аспекты. *Физическое архитектурное проектирование* связано с выбором решений, касающихся развертывания системы и ее распределения по многочисленным процессорам. Физическая **архитектура** решает проблемы, касающиеся **клиента** и **сервера**, а также промежуточного программного обеспечения, служащего посредником между клиентом и сервером. Физическая архитектура распределяет **компоненты** по компьютерным **узлам**. С точки зрения моделирования с помощью языка UML физическое архитектурное проектирование использует узлы и *диаграммы развертывания* (см. раздел 3.6.3 главы 3).

Несмотря на то что решение вопросов, касающихся клиента и сервера, описывается физической архитектурой, эти понятия остаются логическими концепциями (Bochenski, 1994). *Клиент* (client) — это вычислительный процесс, посылающий запросы серверному процессу. *Сервер* (server) — это вычислительный процесс, выполняющий запросы клиента. Как правило, клиентский и серверный процессы выполняются на разных компьютерах, но без каких-либо ограничений могут выполняться и на отдельной машине.

В типичном сценарии *клиентский процесс* контролирует вывод информации на дисплей пользователя и обработку событий, генерируемых пользователем. *Серверный процесс* — это любой компьютерный узел с базой данных, предназначенный для удовлетворения запросов, поступающих от клиентского процесса.

С помощью *архитектуры клиент/сервер* (C/S) можно описать любую распределенную систему. Необходимость распределять компоненты возникает по многим причинам.

- Необходимость специализированной обработки на предназначенной машине.
- Необходимость доступа к системе из разных географических точек.
- Экономические причины: использование многочисленных небольших машин может оказаться дешевле, чем использование крупного дорогостоящего компьютера.
- Требование адаптивности, гарантирующей возможность будущего распределения системы.

В *распределенной системе обработки* (distributed processing system) клиент может обращаться к любому количеству серверов. Однако в каждый момент времени клиент может обращаться только к одному серверу. Это значит, что отдельный запрос не может комбинировать данные, полученные от нескольких серверов баз данных. Если же такая ситуация возможна, то говорят, что архитектура создает основу для *системы распределенных баз данных* (distributed database system).

6.1.1. Одноранговая архитектура

Любой компьютерный узел с базой данных может быть клиентом в одной деловой транзакции и сервером в другой. Соединение таких узлов с помощью сети связи порождает архитектурный стиль, известный под названием *одноранговая архитектура* (peer-to-peer), или **P2P** (рис. 6.1).

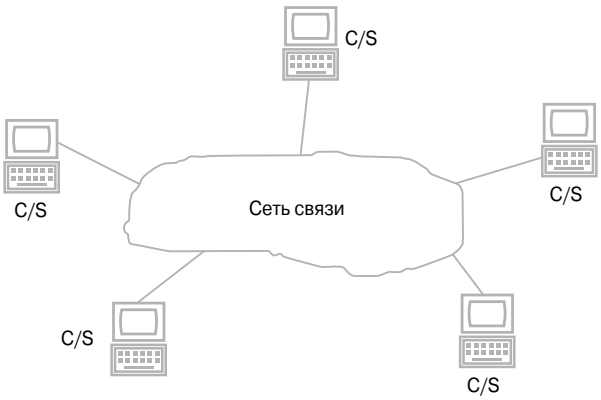


Рис. 6.1. Одноранговая архитектура

В одноранговой архитектуре любой процессор или узел в системе может быть как клиентом, так и сервером. Соответственно, "этот архитектурный стиль определяет единый тип системного элемента (**равноправный узел сети** (peer)) и единый тип сетевого соединения (одноранговое соединение)... Основной организационный принцип этой системы заключается в том, что любой равноправный узел может свободно и непосредственно связываться с любым другим равноправным узлом, не прибегая к помощи центрального сервера. Как правило, равноправные узлы находят друг друга, автоматически обмениваясь списками известных им равноправных узлов (хотя в некоторых случаях могут использоваться централизованные списки равноправных узлов)" (Rozanski and Woods, 2005).

Очевидно, что при разработке одноранговых сетей необходимо оценивать загрузку узлов и линий связи. Минимизация трафика в сети и максимизация ее пропускной способности в рамках одноранговой архитектуры требует особого внимания. Необходимо уделить внимание потенциальным *блокировкам* (deadlock) процессов (когда несколько объектов блокируют друг друга и не могут продолжать работу, поскольку владеют ресурсами, которых ожидают другие процессы). Кроме того, большой проблемой является гарантия разумного времени реакции в одноранговой системе.

К преимуществам одноранговых систем относится их гибкость и стойкость по отношению к сбоям отдельных равноправных узлов (в этом случае возможно автоматическое перенаправление процесса). Более того, благодаря равномерному распределению обработки среди равноправных узлов этот архитектурный стиль является масштабируемым и гибким.

6.1.2. Ярусная архитектура

Большинство корпоративных информационных систем используют многоярусный архитектурный стиль. В противоположность одноранговой архитектуре *ярусная архитектура* (tiered architecture) определяет иерархию ярусов вычислений. Как и в одноранговой архитектуре, каждый **ярус** (tier) в середине иерархии может действовать и как клиент, и как сервер. Однако ярус может быть лишь клиентом по отношению к следующему ярусу в иерархии и лишь как сервер для вызывающего яруса, расположенного выше в иерархии.

Одним из наиболее важных практических способов разработки крупных промышленных систем, основанных на базах данных, является выделение как минимум трех проблем, связанных с реализацией, — представления графического пользовательского интерфейса, корпоративных бизнес-правил и обработки данных. Это разделение проблем порождает *трехъярусную архитектуру* (three-tier architecture), в которой ярус бизнес-логики расположен между клиентом GUI и сервером базы данных.

На практике разделение этих задач может создать намного больше проблем и потребовать включения других сервисов, таких как Web-обработка, сетевая связь или печать. *Ярусная архитектура* отличается от одноранговой тем, что она навязывает иерархические зависимости между ярусами аппаратного и программного обеспечения. Эти зависимости согласовываются с ярусами программного обеспечения, выделенными в рамках логической архитектурной модели, например архитектуры J2EE (см. раздел 4.1.2 главы 4) или PCBMER (раздел 4.1.3).

На рис. 6.2 показано выравнивание между слоями в модели PCBMER и потенциальное развертывание ярусов. Представление, управление и слои компонентов в этой системе реализованы на Web-сервере. Оставшиеся слои PCBMER реализуются на сервере приложения, при этом используется сервер базы данных из подсистемы ресурсов.

При описании ярусов чаще всего упоминается **сервер приложения** (application server). Кроме того, это понятие в многоярусной архитектуре является самым неясным. Это объясняется путаницей между понятиями сервиса приложения и бизнес-сервиса. В среде специалистов по базам данных эти два понятия четко различаются. *Сервис приложения* (application service) — это задача, которую выполняет каждая отдельная программа, работающая с базой данных. *Бизнес-сервис* (business service) — это бизнес-правила, которые база данных устанавливает для всех исполняемых программ. Эти бизнес-правила часто устанавливаются программным путем (с помощью триггеров и хранимых процедур).

В результате трехъярусную архитектуру можно представить с помощью сервисов приложения, бизнес-сервисов и обработки данных. Увы, описанное выше различение сервиса приложения и бизнес-сервиса не является общепринятым. Например, довольно часто термин *сервис приложения* означает ярус, управляющий бизнес-компонентами и следящий за выполнением бизнес-правил (с помощью

отдельного потока управления, идущего от сервиса приложения к каждой программе). В то же время **Web-сервер** обрабатывает события, порождаемые приложением и графическим пользовательским интерфейсом (именно эта ситуация показана на рис. 6.2).

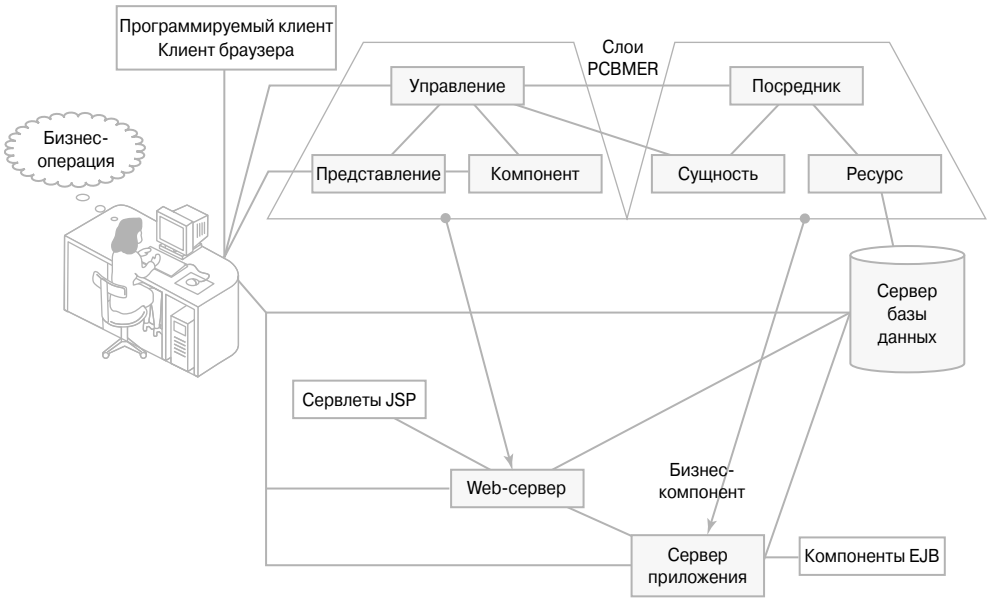


Рис. 6.2. Согласование между слоями модели PCBMER и ярусами развертывания

Иначе говоря, *процесс приложения* (application process) — это логическая концепция, которая может поддерживаться, а может и не поддерживаться аппаратным обеспечением. Логика приложения может быть реализована как на клиентском, так и на серверном узле, т.е. его можно скомпилировать как в виде клиентского, так и в виде серверного процесса, а затем реализовать в виде динамически подгружаемой библиотеки (dynamic link library — DDL), программного интерфейса приложения (application programming interface — API), удаленных вызовов процедур (remote procedure calls — RPC) и т.д.

Если логика приложения компилируется на клиентском узле, то возникает *архитектура "толстого" клиента* (thick client architecture), так называемого "клиента на стероидах". Для взаимодействия с пользовательскими программами "толстый" клиент предполагает использование рабочих станций. Как правило, архитектура "толстого" клиента использует только два яруса ("толстый" клиент и сервер базы данных).

Если логика приложения компилируется на серверном узле, то возникает *архитектура "тонкого" клиента* (thin client architecture), так называемого "худого" клиента. Для взаимодействия с пользовательскими программами "тонкий"

клиент предполагает использование сетевых компьютеров. Кроме того, архитектура "тонкого" клиента предполагает, что клиентом является Web-браузер, обращающийся к HTML-страницам, Java-аплетам, компонентам, сервлетам и т.д.

Помимо описанных выше, возможны также *промежуточные архитектуры* (intermediate architecture), в которых логика приложения частично компилируется на клиентском узле, а частично на сервере.

6.1.3. Архитектура, ориентированная на базы данных

В данной книге рассматривается разработка бизнес-приложений и информационных систем предприятий. В таких системах главную роль играет программное обеспечение, управляющее базами данных (подробнее об этом — в главе 8). Соответственно, базы данных влияют на физическую архитектуру программного обеспечения.

Независимо от того, где расположена логика приложения, программа (клиент) взаимодействует с базой данных (сервером), чтобы получить информацию и предоставить ее для отображения и дальнейших манипуляций пользователя. Однако современные базы данных также можно программировать. Такие базы данных называют *активными*.

Программы баз данных называются *хранимыми процедурами* (stored procedure). Хранимые процедуры хранятся в самой базе данных (они являются **персистентными объектами**). Их можно вызвать из клиентской программы (или из другой хранимой процедуры) с помощью обычного оператора вызова процедуры/функции.

Существуют хранимые процедуры специального вида — **триггеры** (trigger), которые нельзя вызвать явно. Триггер срабатывает автоматически при попытке изменить содержимое базы данных. Триггеры используются для реализации бизнес-правил масштаба предприятия, которые должны быть проведены в жизнь способом, независимым от клиентских программ (или хранимых процедур). Триггеры усиливают целостность и непротиворечивость баз данных. Они не позволяют отдельным приложениям нарушать бизнес-правила, заложенные в базу данных.

Нам необходимо решить, какая часть системы будет запрограммирована в клиенте, а какая в базе данных. При этом рассматриваются следующие программируемые части системы.

- Пользовательский интерфейс.
- Логика представления.
- Функция приложения
- Интегральная логика.
- Функции доступа к данным.

Пользовательский интерфейс (user interface) программы отвечает за отображение информации на конкретный графический пользовательский интерфейс, например Web-браузер, Windows или Macintosh. *Логика представления* (presentation

logic) отвечает за обработку объектов графического пользовательского интерфейса (форм, меню, кнопок действий и т.д.) в соответствии с требованиями функции приложения.

Функция приложения (application function) содержит основную логику программы. Она определяет, что именно делает приложение, и представляет собой связующее звено между клиентом и базой данных. С точки зрения модели PCBMER (см. раздел 4.1.3 главы 4) функция приложения реализуется классами *управляющей* подсистемы.

Интегральная логика (integral logic) отвечает за бизнес-правила масштаба предприятия. Это правила, которые применяются ко всем прикладным программам, т.е. все программы должны функционировать в соответствии с ними. Функции доступа к данным владеют вопросами доступа к постоянным объектам данных на диске.

На рис. 6.3 показан типичный сценарий. Пользовательский интерфейс и логика представления принадлежат *клиенту*. Функции доступа к данным и интегральная логика (триггеры) находятся в ведении *базы данных*. Прикладная функция часто программируется (в виде SQL-запросов) как часть клиента в начале этапа разработки, а затем переносится в базу данных (в виде хранимых процедур) для окончательного развертывания программного продукта.

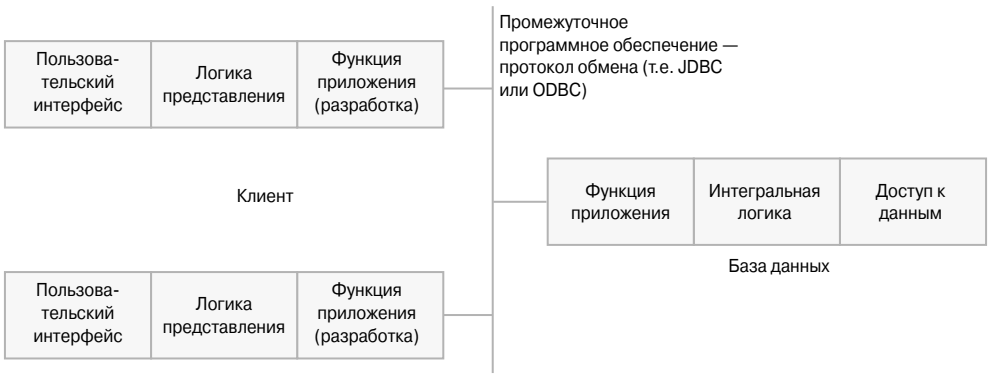


Рис. 6.3. Взаимодействие приложения и базы данных

Контрольные вопросы 6.1

КВ1. Какой архитектурный стиль определяет единственный тип элемента системы?

КВ2. Что является средним ярусом в трехуровневой архитектуре?

КВ3. С помощью каких программных средств реализуются бизнес-правила в базе данных?

6.2. Многоуровневая логическая структура

Разработчики программного обеспечения знают, что трудность разработки небольшой системы не идет ни в какое сравнение с трудностью поставки крупномасштабного решения. Небольшие объектные системы зачастую легче понять, реализовать и развернуть. Для масштабных объектных решений характерны сложные сетевые структуры объектов, реагирующих на случайные события, которые вызывают переплетающиеся цепочки взаимосвязанных операций (методов). В отсутствие четкого архитектурного проекта и строго определенного процесса разработки крупные программные проекты вполне могут завершиться провалом.

Хорошо известный принцип когнитивной психологии — *правило 7 ± 2* — гласит, что кратковременная память человеческого мозга может *одновременно* справиться приблизительно с девятью (7 ± 2) предметами (графическими элементами, идеями, понятиями и т.д.). Нижняя граница, равная пяти (7–2), указывает на то, что работа менее чем с пятью предметами составляет тривиальную проблему.

Правила когнитивной психологии не в состоянии повлиять на тот факт, что нам необходимы большие системы, а большие системы — сложны. Значительная часть этой сложности носит *вторичный характер* (accidental), а не присуща самой системе (см. раздел 1.1 главы 1). Вторичная сложность является ненужным препятствием. Она может превратить существенно сложную систему в излишне *усложненную* (complicated system). Такая система не только сложна, но и не допускает адаптацию.

Причина этого явления заключается в системном моделировании, допускающем неограниченное количество связей между объектами. В подобных системах объекты образуют *сети* (networks) — паутину объектов, образованную перекрестными ссылками. Передача сообщений в таких системах допускается от любого отправителя любому адресату. Возможны также восходящие и нисходящие вызовы. Количество возможных связей между объектами в этих сетях растет экспоненциально с появлением новых объектов. Как заметил Шиперски (Szyperski, 1998): "…объектные ссылки порождают связи над произвольными областями абстракции. В итоге правильное расслоение системной архитектуры становится проблематичным, если вообще возможным".

Удачные системы организованы по иерархическому принципу — любые сетевые подструктуры внутри таких иерархий оказываются ограниченными и тщательно контролируются. *Иерархии* позволяют снизить степень сложности с экспоненциальной до полиномиальной. Они распределяют объект по **уровням** (layers) и ограничивают взаимодействие между ними. Как указывалось в разделе 4.1 главы 4, практически все логические архитектурные модели являются многоуровневыми. Яркий пример, приведенный в разделе 4.1.3, показывает, что иерархии в модели PCBMER являются не строгими, а *ослабленными* (relaxed), поскольку более высокие уровни могут зависеть от нескольких уровней, расположенных ниже (Buschmann et al., 1996).

В последующих подразделах мы определим сложность как понятие и покажем, почему иерархии лучше сетей. Кроме того, мы рассмотрим наиболее важные архитектурные шаблоны, предназначенные для уменьшения сложности.

6.2.1. Архитектурная сложность

Для того чтобы обсуждать сложность объектных систем, необходимо договориться о ее измерении. Как измерить сложность? Сложность имеет много видов и форм. Простой, но очень показательной мерой сложности может выступать количество линий связи между классами. *Линия связи* (communication path) выражает существование постоянной или временной связи между классами (см. раздел A.2.3 приложения A). Другими словами, *сложность* современного предприятия и коммерческой информационной системы выражается "*количеством проводов*".

Это определение сложности согласуется с реальным положением дел, сложившимся в области компьютерных вычислений, прошедшей длинный путь от машины Тьюринга, основанной на *алгоритмах* (неспособных реализовать ничего, кроме вычислимых функций), до открытой *модели взаимодействия* (interaction model). Вегнер (Wegner, 1997) пишет следующее.

> Сдвиг парадигмы от алгоритмов к взаимодействию отражает технологический сдвиг от мэйнфреймов к рабочим станциям и сетям, от перемалывания чисел к встроенным системам и графическим пользовательским интерфейсам, а также от процедурно-ориентированных к объектно-ориентированному и распределенному программированию. Твердое убеждение, что интерактивные системы представляют собой более мощное средство для решения задач, основано на концепции взаимодействия... Результаты работы алгоритмов полностью определяются входными данными, в то время как интерактивные системы, такие как PCS (personal communications services — персональные услуги связи), системы резервирования билетов в авиакомпаниях и роботы, накапливают опыт и могут использовать его для самообучения и адаптации.

Несмотря на очевидность сложности, среди специалистов существует несколько точек зрения на это понятие. Фентон и Пфлигер (Fenton and Pfleeger, 1997) насчитали четыре интерпретации этой концепции.

- *Сложность задачи*, отражающая сложность самой предметной области. Это понятие известно также как вычислительная сложность. Сложность задачи представляет собой вариант основных характеристик программного обсуждения, введенных Бруком (см. раздел 1.1.1 главы 1).
- *Алгоритмическая сложность*, предназначенная для оценки эффективности алгоритмов. Значение этого вида сложности постепенно снижается (по крайней мере, с точки зрения адаптивности программного обеспечения) из-за перехода от алгоритмов к взаимодействиям.

- *Структурная сложность*, предназначенная для установления отношений между структурой программного обеспечения и легкостью его поддержки и усовершенствования. Показатели структурной сложности применяются к зависимостям между объектами программного обеспечения.
- *Когнитивная сложность* позволяет измерить усилия, необходимые для понимания программного обеспечения, оценивая *логический поток* (flow of logic) исполнения программы.

С точки зрения адаптивности показатели когнитивной сложности оценивают степень понятности системы, а показатели структурной сложности — легкость сопровождения и масштабирования. Эти две категории связаны друг с другом. Относительно небольшая когнитивная сложность представляет собой необходимое, но недостаточное условие для низкой структурной сложности. Это вполне соответствует требованию, гласящему, что прежде чем изменять код, его следует сначала понять.

6.2.1.1. Пространственная когнитивная сложность

Вероятно, наиболее удобной мерой когнитивной сложности современных программ является *мера пространственной сложности* (spatial complexity metrics) (Gold et al., 2005). Ее цель — измерить *расстояние*, которое программист должен пройти по коду, чтобы построить его умозрительную модель. Существует несколько показателей пространственной сложности.

В качестве иллюстрации Дус и соавторы (Douce et al., 1999) представили две меры пространственной сложности — *функциональную* (*процедурную*) и *объектно-ориентированную*. Обе они имеют свои слабости, причем объектно-ориентированная мера слабо обоснованна и довольно путаная.

Оценка сложности выводится в два этапа. Сначала вычисляются оценки сложности каждой функции в программе. Затем эти значения суммируются и образуют оценку сложности всей программы.

$$FC = \sum_{i=1}^{totalcalls} dist_i \,, \tag{6.1}$$

где *totalcalls* — количество вызовов функции; *dist* — расстояние, представляющее собой количество строк кода между вызовом функции и ее определением; *FC*— функция пространственной сложности.

$$PC = \sum_{i=1}^{totalfunctions} FC_i \,, \tag{6.2}$$

где *totalfunctions* — количество функций в программе; *PC* — пространственная сложность программы.

Основной недостаток приведенных выше формул заключается в том, что расстояние вычисляется путем подсчета строк кода. Количество строк кода имеет слабое

отношение к современной практике программирования и способам визуализации программ в ходе анализа. Кроме того, строкам кода приписывается некий "пространственный" смысл, поскольку предполагается, что, переходя от вызова функции к ее определению, программист должен "пропускать" строки (возникает вопрос, как это связано с понятностью программы). Несмотря на это, предложенные показатели дают определенное представление о пространственной сложности программы.

6.2.1.2. Структурная сложность

В то время как *когнитивная* сложность концентрируется на логическом потоке исполнения программы, *структурная* сложность описывает зависимости между объектами в программе. В главе 4 указывалось, что *зависимость* между двумя объектами в системе существует, если в результате изменения объекта, предлагающего услуги, может возникнуть необходимость изменить клиентский объект, нуждающийся в этих услугах (Maciaczek and Liong, 2005). Это определение согласуется со стандартом UML (UML, 2005), утверждающим: "Зависимость представляет собой отношение "клиент/сервер" между элементами модели, при котором модификация сервера может повлиять на элементы клиентской модели. Зависимость означает, что семантика клиента без сервера не полная".

Если все зависимости в системе идентифицированы и понятны, система называется *адаптивной* (adaptive), т.е. понятной, поддерживаемой и масштабируемой. Необходимым условием адаптивности является гибкость зависимостей. Следовательно, задача программиста — минимизировать зависимости в системе.

В системах программного обеспечения зависимости можно идентифицировать для объектов, имеющих разную степень детализации, — компонентов, **пакетов**, классов и методов. Зависимости между более специфичными объектами на низком уровне детализации порождают зависимости между объектами на более высоком уровне детализации. Соответственно, управление зависимостями требует тщательного изучения программы и выявления всех зависимостей между структурами данных и вызовами функций в объектах. Далее мы будем рассматривать структурную зависимость между классами. Более подробное изложение этих вопросов можно найти в книге (Maciaszek, 2006) и других публикациях.

6.2.1.2.1. Структурная сложность сетей

Каждая линия связи обычно предполагает двустороннее взаимодействие между классами — от *A* к *B* и от *B* к *A*. На рис. 6.4 проиллюстрирована сложность сети, состоящей из семи классов. Структурная сложность сети выражается формулой

$$_{net}CCD = n(n-1), \tag{6.3}$$

где n — количество объектов (узлов графа); $_{net}CCD$ — кумулятивная зависимость класса в полносвязной сети (в предположении, что объекты являются представителями классов).

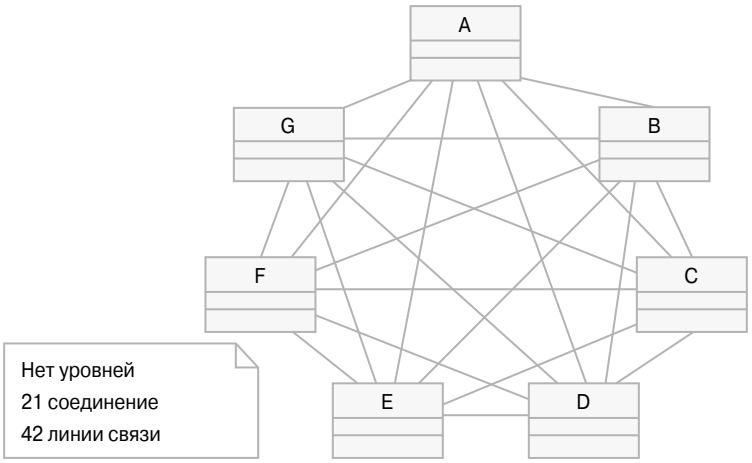


Рис. 6.4. Сложность сети

Применение этой формулы к семи классам позволяет вычислить, что показатель *CCD* равен 42. Очевидно, что рост сложности в сетевых системах происходит по экспоненциальному закону. Однако это относится не к *реальным*, а к *потенциальным зависимостям* между всеми объектами системы, порождаемым плоской структурой сети, показанной на рис. 6.4 (без ограничений на линии связи между объектами). Изменение объекта *может* повлиять на другой объект в системе (т.е. вызвать *волновой эффект* (ripple effect)).

Обратите внимание на то, что показатель сложности зависит от количества классов, а не объектов. В программах объекты — а не классы — посылают сообщения другим объектам того же самого или другого класса. Это усложняет программисту выполнение задач, связанных с реализацией логики приложения и управлением переменными и структурами данных. Однако этой дополнительной сложностью в данном контексте можно пренебречь.

Объект может посылать сообщение другому объекту, только если между ними существует персистентная или временная связь. Временные связи (существующие только на этапе выполнения программы) создаются при вызове отдельной программы. Их можно показывать, а можно и не показывать в структурах программы. Персистентные связи (существующие на этапе компилирования) существуют, только если есть соединение, определенное в модели классов (например, ассоциация).

6.2.1.2.2. Структурная сложность иерархий

Управление сложностью состоит в уменьшении сетевых структур за счет группирования классов в *иерархии классов*, *холархии* (см. раздел 5.2.2 главы 5). При этом подходе классы могут естественным образом образовывать слои, подчеркивая, что взаимодействие происходит *между* уровнями, одновременно допуская взаимодействие *внутри* уровней, характерное для сетевых структур.

Расслоение иерархии дает возможность снизить сложность за счет ограничения количества возможных связей между объектами. Это снижение сложности достигается посредством разделения классов на уровни и разрешения непосредственного взаимодействия классов внутри уровня и между соседними уровнями.

На практике сложность иерархических структур снижается за счет применения архитектурных принципов, описанных в разделе 4.1.3.2. Одним из них является принцип *DDP*, или *принцип нисходящей зависимости* (downward dependency principle), утверждающий, что зависимости между уровнями могут быть направлены только сверху вниз. Иначе говоря, линии связи между слоями являются односторонними, т.е. количество линий связи между уровнями совпадает с количеством соединений. Любая восходящая связь между уровнями реализуется с помощью слабого **связывания** (coupling), не создающего зависимости и поддерживаемого интерфейсами, размещенными на нижнем уровне, но реализованными на верхних, и/или замены передачи сообщений обработкой событий, и/или использования технологий метауровня (таких, как каркасы Struts для программирования на языке Java).

Принцип DDP распространяется и на связь между уровнями. Для связей внутри уровней аналогичного эффекта можно достичь, применив *принцип CEP*, или *принцип исключения циклов* (cycle elimination principle). Циклы внутри уровней можно исключить, используя интерфейсы, а также методы рефакторинга, выделяющие циклически зависимые функциональные свойства в отдельные объекты или компоненты (см. раздел 4.1.3.2 главы 4, а также (Maciaszek and Liong, 2005)).

На рис. 6.5 продемонстрирована сложность иерархии, состоящей из семи классов, образующих четыре уровня, в предположении, что зависимости между уровнями могут быть лишь нисходящими, а зависимости внутри уровней не имеют циклов. По сравнению с сетевой структурой, приведенной на рис. 6.4, сложность удалось снизить: количество линий связи уменьшилось с 42 до 13.

Обозначим уровни символами l_1, l_2, ..., l_n. Пусть для любого уровня l_i выполняются следующие условия:

- size(l_i) — количество объектов на уровне l_i;
- l_i — количество предков уровня l_i;
- $P_j(l_i)$ — j-й предок уровня l_i.

Тогда кумулятивная зависимость между классами в холархии (т.е. в иерархии, допускающей существование нескольких предков уровня в модели PCBMER) вычисляется следующим образом:

$$_{holarchy}CCD = \sum_{i=1}^{n} \frac{size(l_i) \times (size(l_i) - 1)}{2} + \sum_{i=1}^{n} \sum_{j=1}^{l_i} size(l_i) \times (size(p_j(l_i))), \quad (6.4)$$

где n — количество классов на каждом уровне i (т.е. на уровнях 1, 2, 3 и т.д.), а показатель $_{hier}CCD$ — кумулятивная зависимость классов в иерархии.

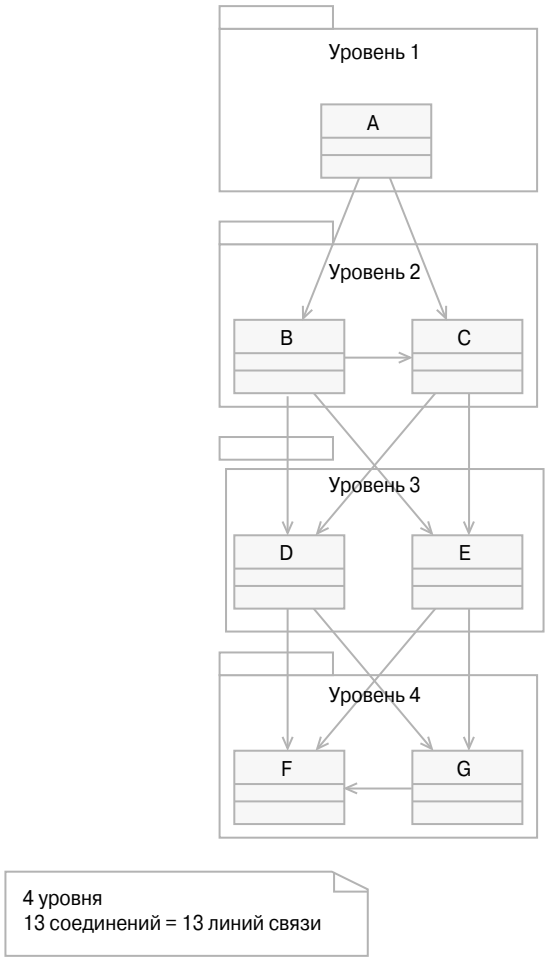


Рис. 6.5. Сложность иерархий

Первая часть формулы (6.4) вычисляет количество потенциальных (всех возможных) односторонних линий связи между всеми классами на каждом уровне. Вторая часть вычисляет количество потенциальных (всех возможных) односторонних линий связи между классами, расположенными на соседних уровнях.

Даже когда количество классов равно всего лишь семи, при переходе от *сети* (см. рис. 6.4) к *холархии* (см. рис. 6.5) достигается значительное сокращение сложности. Сложность сети возрастает по экспоненциальному закону, а сложность холархии — по полиномиальному. Упомянутые формулы позволяют вычислить худший среди возможных показателей CCD. Следовательно, показатель CCD на рис 6.5 равен 13, а не 12, как могло бы показаться (поскольку на диаграмме не показана зависимость между классами `D` и `E`). Вычисление числа 13 приведено ниже.

- Нет соединений на уровне 1.
- По одному соединению на уровнях 2, 3 и 4 (всего 3).
- Два соединения между уровнями 1 и 2.
- Четыре соединения между уровнями 2 и 3.
- Четыре соединения между уровнями 3 и 4.

Формулы для оценки сложности иерархий могут отличаться в зависимости от конкретных ограничений, наложенных на иерархии. Одно из таких ограничений, не использованных на рис. 6.5, возникает при использовании шаблона **Фасад** (Façade), который будет рассмотрен в разделе 6.2.2.1. Шаблон Фасад определяет единственную точку входа (специальный интерфейс или класс) для каждого уровня/подсистемы/пакета в иерархии, сокращая тем самым количество возможных зависимостей между уровнями.

Мацяшек (Maciaszek, 2006) предложил несколько формул для оценки структурной сложности некоторых видов иерархий. Как указывалось выше, при добавлении в систему новых классов все формулы для иерархий приводят лишь к полиномиальному росту сложности.

6.2.2. Архитектурные шаблоны

Описывая архитектурную модель PCBMER (или мета-архитектуру) в главе 4, мы перечислили принципы, которым она должна подчиняться (см. раздел 4.1.3.2). Каждая модель должна иметь соответствующий набор таких принципов. Однако реализация этих принципов в каждом конкретном проекте должна быть согласована с более конкретными **шаблонами проектирования** (design patterns).

"Шаблон проектирования описывает схему уточнения элементов системы программного обеспечения или отношения между ними. Он описывает повторяющуюся общую структуру связанных элементов проектирования, решающую общую задачу проектирования в конкретном контексте" (Rozanski and Woods, 2005). В области архитектурного проектирования такие шаблоны называются *архитектурными* (architectural patterns).

В следующих подразделах описаны шаблоны, имеющие особую важность для архитектурного проектирования. Эти шаблоны известны как *шаблоны GoF*, или *шаблоны "Банды четырех"* (названные так в честь четырех авторов книги (Gamma et al., 1995), в которой излагались принципы шаблонного проектирования и описаны некоторые наиболее популярные в настоящее время шаблоны). Объяснение архитектурных шаблонов, приведенное ниже, представляет собой сокращенный вариант изложения, представленного в книге (Maciaszek and Liong, 2005).

6.2.2.1. Шаблон Фасад

Гамма и его соавторы (Gamma et al., 1995) дают следующее определение шаблона Фасад: "Высокоуровневый интерфейс, облегчающий использование подсистемы. Его цель — минимизировать связи и зависимости между подсистемами".

Высокоуровневый интерфейс не всегда полностью соответствует классическому определению интерфейса (см. раздел А.9 приложения А). Им может быть как *конкретный класс*, который называется *доминантным* (dominant), так и *абстрактный класс* (см. раздел А.8 приложения А). Дело в том, что высокоуровневый интерфейс инкапсулирует основные функциональные свойства уровня (подсистемы, пакета) и обеспечивает главную или вообще единственную точку входа для клиентов уровня. В целом уровень может иметь несколько фасадов, предназначенных для разных клиентов, расположенных на более высоких уровнях.

Как показано на рис. 6.6, клиентские объекты взаимодействуют с уровнем с помощью фасадного объекта — `EFacadeImpl`, реализующего интерфейс `EFacade`. Объект `EFacadeImpl` содержит ссылки (ассоциации) на соответствующие классы уровня (`EClass1` и `EClass2`) и делегирует им запросы клиента. Классы уровня выполняют задания, порученные фасадом, но не ссылаются на него. Фасадный объект не обязан хранить ссылки на все классы уровня, поскольку некоторые классы выполняют лишь вспомогательную работу для остальных классов уровня. При специализации из абстрактного класса модель показывает ссылку на абстрактный класс, но при реализации системы объект `EFacadeImpl` должен хранить ссылки на конкретные подклассы `EClass2` и `EClass3`.

Несмотря на то что фасадный объект может самостоятельно выполнять определенные задания, как правило, он перепоручает работу другим объектам внутри уровня. Благодаря этому достигается сокращение количества линий связи и объектов, с которыми общается клиент. Другим следствием применения фасада является сокрытие классов уровня за фасадным объектом. Однако этот эффект является побочным и не входит в число основных целей фасада. Цель фасада — обеспечить простой интерфейс с подсистемой и сократить количество зависимостей от него, оставив клиенту возможность выполнять сложную работу, непосредственно обращаясь к классам подсистемы (Stelting and Maassen, 2001).

На рис. 6.7, посвященном примеру 6.1, показано, как класс `EFacade` создает единственную точку входа в пакет `Entity` (Сущность). Он определяет четыре операции, которые класс `CEnroll` должен вызывать непосредственно (как показано на диаграмме последовательностей на рис. 4.18). Класс `EFacade` должен делегировать выполнение этих операций соответствующему классу сущностей.

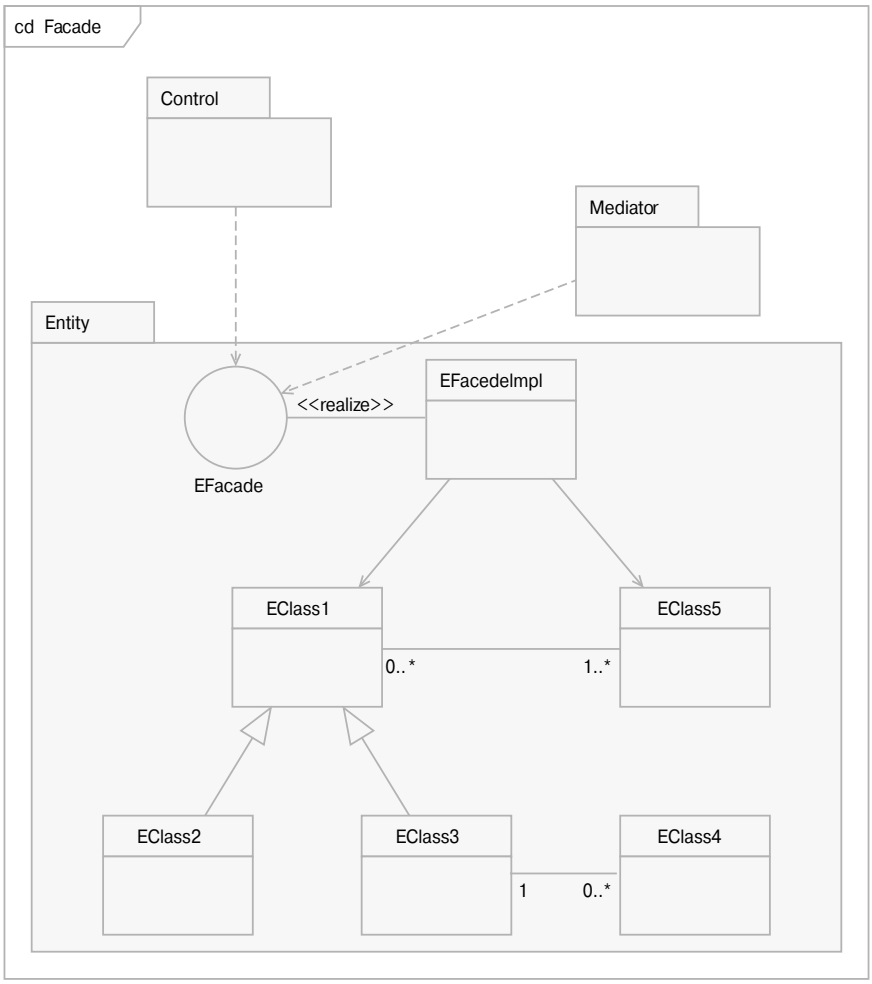


Рис. 6.6. Шаблон Фасад

Пример 6.1. Зачисление в университет

Обратитесь к формулировке задачи 1 для системы, управляющей зачислением в университет (см. раздел 1.6.1 главы 1), и рассмотрите соответствующие примеры, описанные в главе 4. Обратите внимание на "централизованные" взаимодействия из примера 4.18 (см. рис. 4.18) и примера 4.19 (см. рис. 4.20). Примените шаблон `Фасад` к уровню сущностей и постройте диаграмму классов для него. Покажите, какие операции должен определять фасад. Объясните модель. Обсудите возможные варианты соответствующей диаграммы для распределенной модели (см. рис. 4.19 и 4.21).

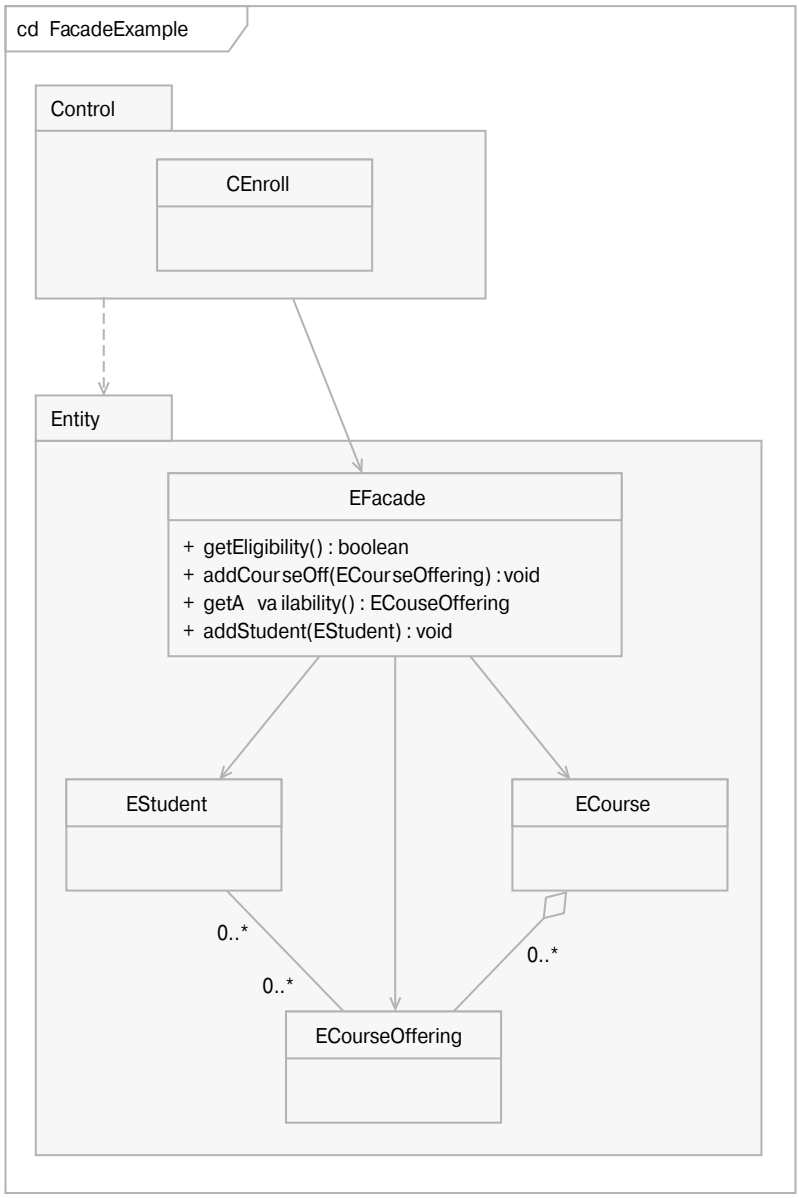


Рис. 6.7. Шаблон Фасад *для системы, управляющей зачислением в университет*

В распределенной модели класс `EFacade` можно упростить еще больше, оставив лишь одну операцию — `addCrs(crs,sem)` (см. рис. 4.19 и 4.21). Выполнение этой операции можно делегировать объекту `EStudent`, который принимает на себя обязанность общаться с классами `ECourse` и `ECourseOffering` и выполнять процедуру зачисления на курс.

6.2.2.2. Абстрактная фабрика

Шаблон Абстрактная фабрика (Abstract Factory) обеспечивает "интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов" (Gamma et al., 1995). В противоположность шаблону Фасад, в котором высокоуровневый интерфейс означал конкретный класс, интерфейс в шаблоне Абстрактная фабрика является либо *истинным интерфейсом* (желательно) (см. раздел А.9 приложения А), либо *абстрактным классом* (см. раздел А.8 приложения А).

Шаблон Абстрактная фабрика позволяет приложению вести себя по-разному, получая доступ к одному из нескольких семейств объектов, скрытых за его интерфейсом. Управление доступом к семействам объектов осуществляется параметрами конфигурации. Типичной для шаблона Абстрактная фабрика является ситуация, когда приложению необходимо использовать разные *ресурсы* в зависимости от обстоятельств, складывающихся в ходе выполнения программы, — например, использовать разные файлы, открывать разные окна графического пользовательского интерфейса или выводить информацию на разных языках.

На рис. 6.8 продемонстрирована визуализация шаблона Абстрактная фабрика. Класс `EAbstractFactory` — это интерфейс (или, может быть, абстрактный класс), определяющий методы создания нескольких объектов (разных ресурсов). Классы `EAbstractProductA` и `EAbstractProductB` — это абстрактные классы (или, может быть, интерфейсы), определяющие общее поведение объектов (ресурсов), используемых в приложении. Шаблон Абстрактная фабрика реализуется с помощью конкретных *фабричных классов* (factory classes). Абстрактная продукция реализуется в виде конкретных *классов продукции* (product classes). Клиентские объекты содержат ссылки на интерфейсы абстрактной фабрики и абстрактной продукции (или абстрактных классов).

"Шаблон Абстрактная фабрика позволяет увеличить общую гибкость приложения, которая проявляется как во время проектирования, так и при выполнении программы. В ходе проектирования разработчик не обязан предвидеть все возможные варианты использования приложения. Вместо этого создается обобщенная среда, а затем разрабатываются реализации, не зависящие от остальной части приложения. В ходе выполнения программы приложение может легко интегрировать новые свойства и ресурсы" (Stelting and Maassen, 2001).

Поскольку шаблон Абстрактная фабрика представляет собой интерфейс, реализуемый в целом семействе классов, его расширение на новые семейства может вызвать волновой эффект и повлиять на существующие конкретные классы. Решение этой проблемы описано в книге (Gamma et al., 1995).

При более детальном анализе шаблон Абстрактная фабрика можно рассматривать как разновидность шаблона Фасад. Интерфейс абстрактной фабрики можно использовать в качестве "высокоуровневого" и с его помощью ограничить связь с пакетом и классами, выполняющими реальную работу в этом пакете.

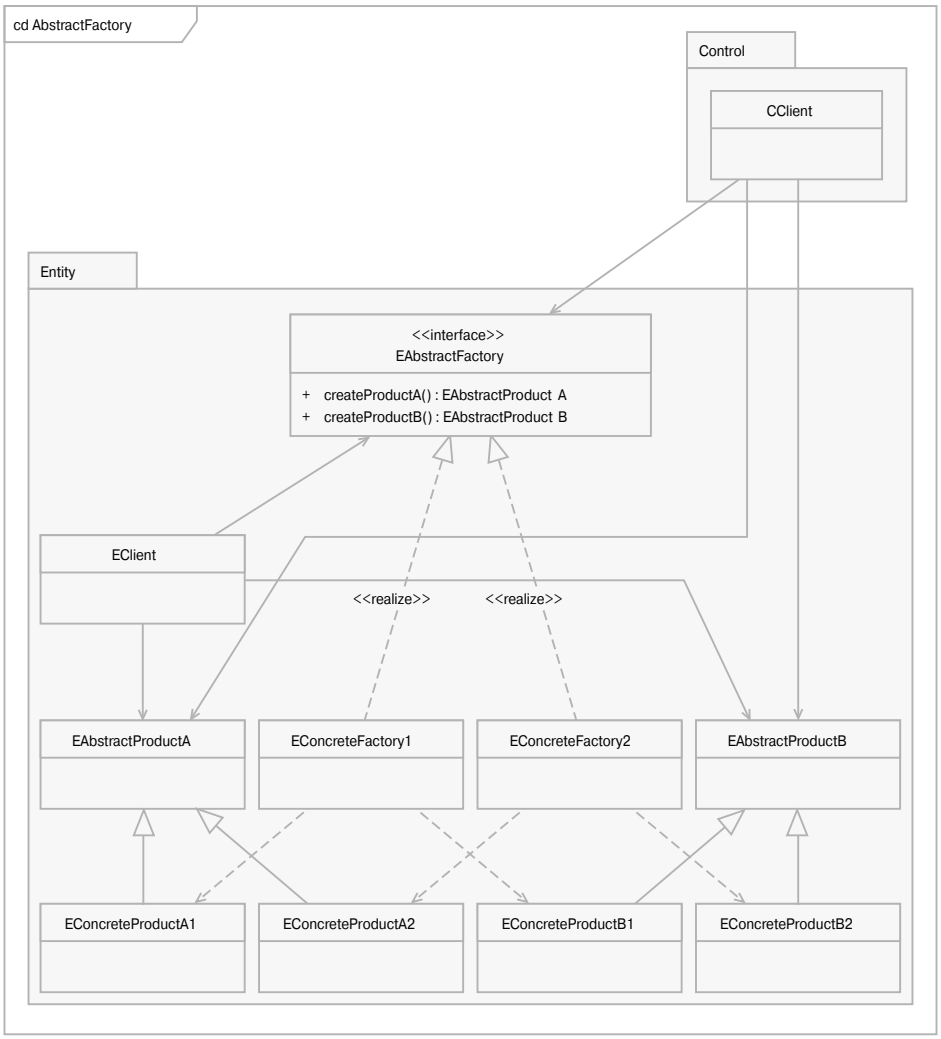


Рис. 6.8. Шаблон Абстрактная фабрика

Пример 6.2. Прямой маркетинг по телефону

Вернитесь к задаче 4, в которой описывается система прямого маркетинга по телефону (см. раздел 1.6.4 главы 1), и рассмотрите примеры, в которых объясняется разница между обычной и бонусной маркетинговыми кампаниями (см. пример 2.5 в разделе 2.5.3 главы 2 и пример 5.9 в разделе 5.2.4.4.2 главы 5).

Для прямого маркетинга по телефону характерно разнообразие кампаний. Кроме того, могут возникать новые разновидности кампаний, которые стартуют и со временем завершаются.

Примените шаблон Абстрактная фабрика к кампаниям на уровне сущностей и создайте диаграмму классов для него. Покажите фабричные операции (операции, выполняемые продуктами фабрики, показывать не обязательно). Кроме того, покажите класс-посредник `MCampaignLauncher`, которому необходим доступ к объектам абстрактной фабрики. Опишите эту модель.

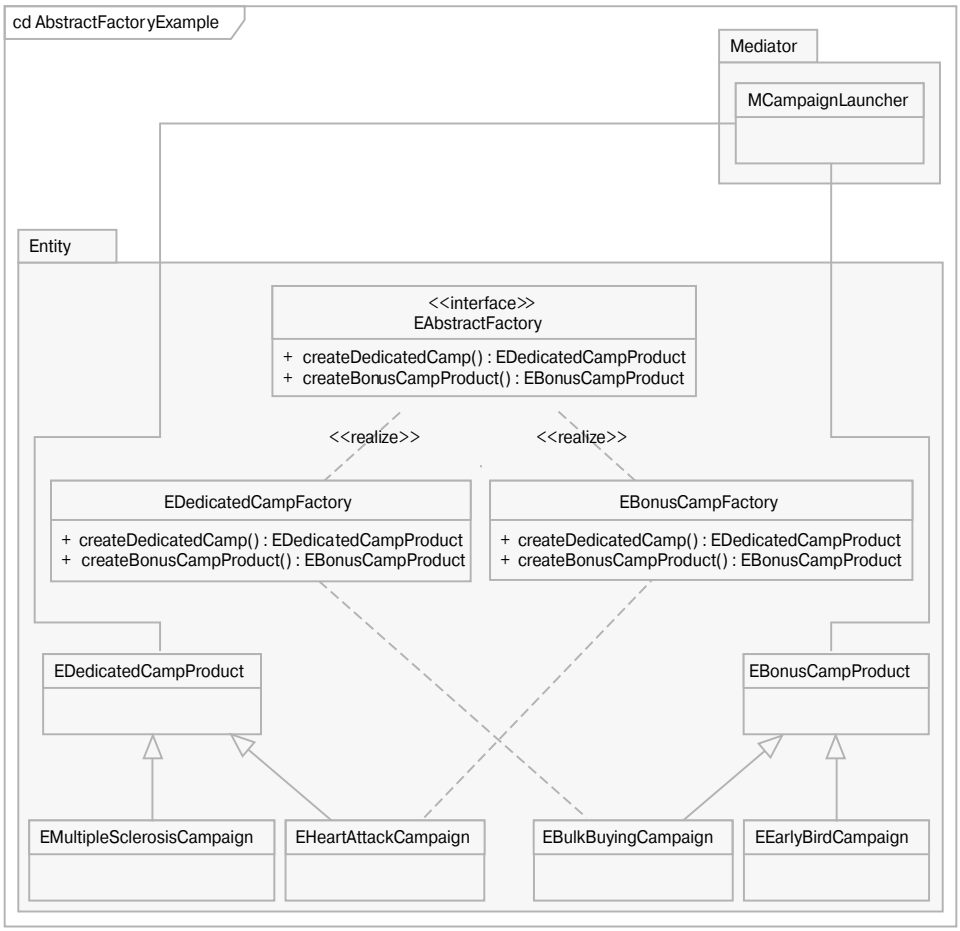


Рис. 6.9. Шаблон Абстрактная фабрика *для системы прямого маркетинга по телефону*

Схема, приведенная на рис. 6.9, учитывает два вида кампаний — обычные и бонусные. На диаграмме показаны два конкретных продукта для каждой из этих абстрактных категорий. Класс `MCampaignLauncher` не зависит от изменений конкретных классов, поскольку он взаимодействует с ними только посредством операций, определенных в шаблонном классе абстрактной фабрики и абстрактных продуктах.

Обобщенная модель абстрактной фабрики позволяет легко включать новые кампании, вызывая лишь операции, создающие новый конкретный класс продукта. Расширить систему для поддержки нового вида кампаний также относительно просто. Для добавления новой кампании достаточно определить дополнительный конкретный фабричный класс, соответствующий интерфейс абстрактного продукта и любые конкретные классы продукта.

6.2.2.3. Цепочка обязанностей

Предназначение шаблона Цепочка обязанностей (Chain of Responcibility) — "избежать связывания отправителя запроса с его адресатом, дав возможность нескольким объектам обработать этот запрос" (Gamma et al., 1995). Шаблон Цепочка обязанностей можно рассматривать как разновидность концепции *делегирования*, интерпретируемой как направленная цепочка сообщений (см. раздел А.6.3 приложения А и раздел 5.3.2.1 главы 5).

Цепочка сообщений начинается на объекте, создающем это сообщение. Если сам объект не может ответить на это сообщение, он делегирует его другому объекту, который передает его дальше. Процесс завершается, когда некий объект отвечает на сообщение или выполняется условие завершения процесса (например, возвращается объект, заданный по умолчанию, или происходит некое событие, свидетельствующее об успешном завершении или неудачном прекращении передачи сообщения по цепочке).

"Шаблон Цепочка обязанностей реализуется в модели "предок-наследник" или "контейнер-содержимое". В рамках этого подхода сообщение, не обработанное наследником, передается предку и, возможно, предку предка, пока на найдется объект, способный его обработать. Шаблон Цепочка обязанностей хорошо подходит для реализации разнообразных функций графического пользовательского интерфейса (GUI). Его могут использовать справочные функции GUI, а также функции расположения компонентов, форматирования и позиционирования. В бизнес-моделях этот шаблон иногда используется в моделях "целое-часть". Например, строка заказа может посылать сообщение самому заказу — сложному заказу, — чтобы он выполнил определенное действие" (Stelting amd Maassen, 2001).

Шаблон Цепочка обязанностей также является предпочтительным в ситуациях, в которых существует набор классов, возможно, принадлежащих разным архитектурным уровням, способных отвечать на первоначальное сообщение. В модели PCBMER шаблон Цепочка обязанностей позволяет применять *принцип NCP* (*neighbor communication principle* — принцип связей между соседями) всякий раз, когда клиентский класс на более высоких архитектурных уровнях требует обработки методов класса, находящегося на удаленных уровнях. Если применяется шаблон Цепочка обязанностей, то клиентский объект, посылающий сообщение, не имеет прямой ссылки на объект, который в конце концов предоставит свой метод.

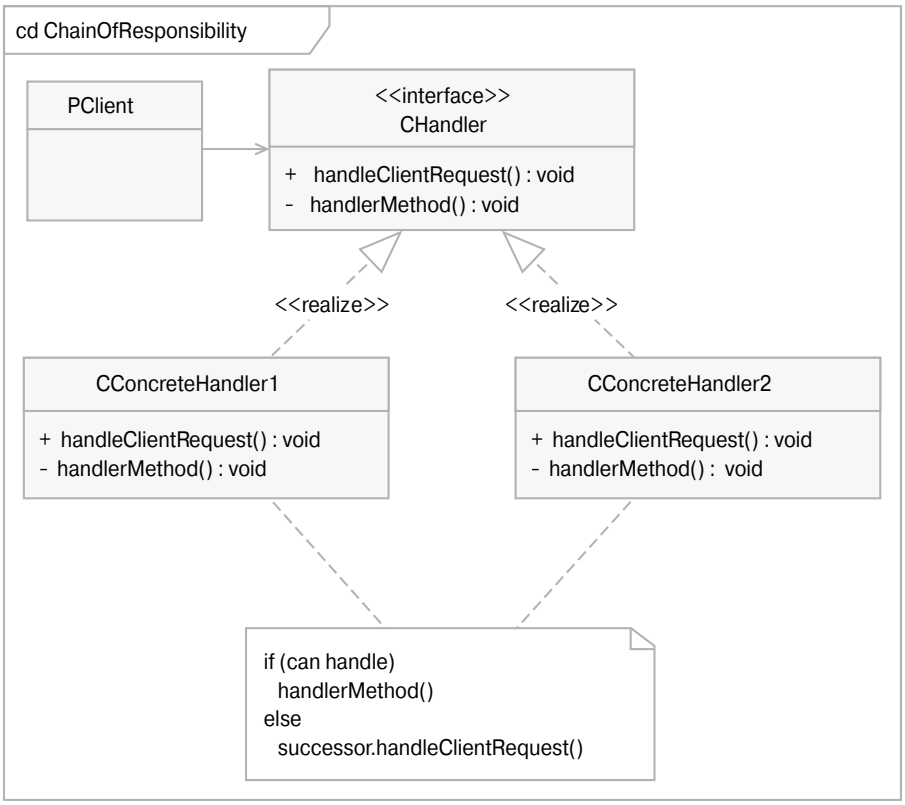


Рис. 6.10. Шаблон Цепочка обязанностей

На рис. 6.10 показана одна из возможных интерпретаций шаблона Цепочка обязанностей. На нем продемонстрирован клиент представления `PClient`, посылающий сообщение `handleClientRequest()` одному из классов, реализующих этот интерфейс, — `CHandler`. Если конкретный обработчик может выполнить запрос, он делает это. Если нет — он пересылает запрос другому объекту. Этот другой объект может быть другим конкретным классом, реализующим абстрактный класс `Chandler`, или классом, расположенным на следующем архитектурном уровне. Если сообщение передается на следующий уровень, интерфейс `CHandler` становится необязательным.

Пример 6.3. Управление взаимоотношениями с заказчиками

Вернитесь к задаче 3, в которой описывается система управления взаимоотношениями с заказчиками (см. раздел 1.6.3 главы 1), и рассмотрите соответствующие примеры из главы 4. Изучите рис. 4.11 (см. пример 4.11 в разделе 4.2.5.3), чтобы понять информационное содержание класса `EContact`.

Рассмотрите сценарий, в котором объект представления (`PWindow`) должен вывести на дисплей информацию о конкретном контакте. Для того чтобы получить данные, объект `PWindow` должен вступить в контакт с управляющим объектом (назовем его `CActioner`). Поскольку объект `CActioner` является управляющим, он не содержит требуемых данных и должен делегировать запрос дальше. Эта цепочка обязанностей может продолжаться до тех пор, пока не достигнет уровня ресурсов, на котором информацию о контакте можно извлечь из базы данных.

Примените шаблон Цепочка обязанностей к описанному выше сценарию. Рассмотрите все уровни модели PCBMER. Как показано на рис. 6.10, использовать интерфейс `CHandler` не обязательно. Опишите эту модель.

На рис. 6.11 показан шаблон Цепочка обязанностей для сценария работы системы управления взаимоотношениями с заказчиками, описанного в примере 6.3. Объект `PWindows` посылает исходное сообщение объекту `CActioner`. Предположим, что объект `CActioner` не знает о существовании объекта `EContact`, содержащего текущие данные о контакте, и передает сообщение `retrieveContact()` объекту `MBroker`, который, в свою очередь, передает его объекту `RReader`, чтобы извлечь данные из базы. После извлечения этих данных и их передачи объекту `MBroker` можно создать экземпляр класса `EContact` и вернуть его объекту `CActioner`. Таким образом, объект `CActioner` может запросить данные у объекта `EContact`, чтобы создать экземпляр класса `BContact` и вернуть ссылку на него объекту `PWindow`. Теперь объект `PWindow` может получить данные, содержащиеся в объекте `Bcontact`, и вывести их на экран.

6.2.2.4. Наблюдатель

Предназначение шаблона Наблюдатель (Observer), известного также под именем Издатель-Подписчик (Publish-Subscribe), — "определить отношение "один ко многим" между объектами так, чтобы при изменении состояния одного объекта происходило оповещение и автоматическое обновление всех зависящих от него объектов" (Gamma et al., 1995).

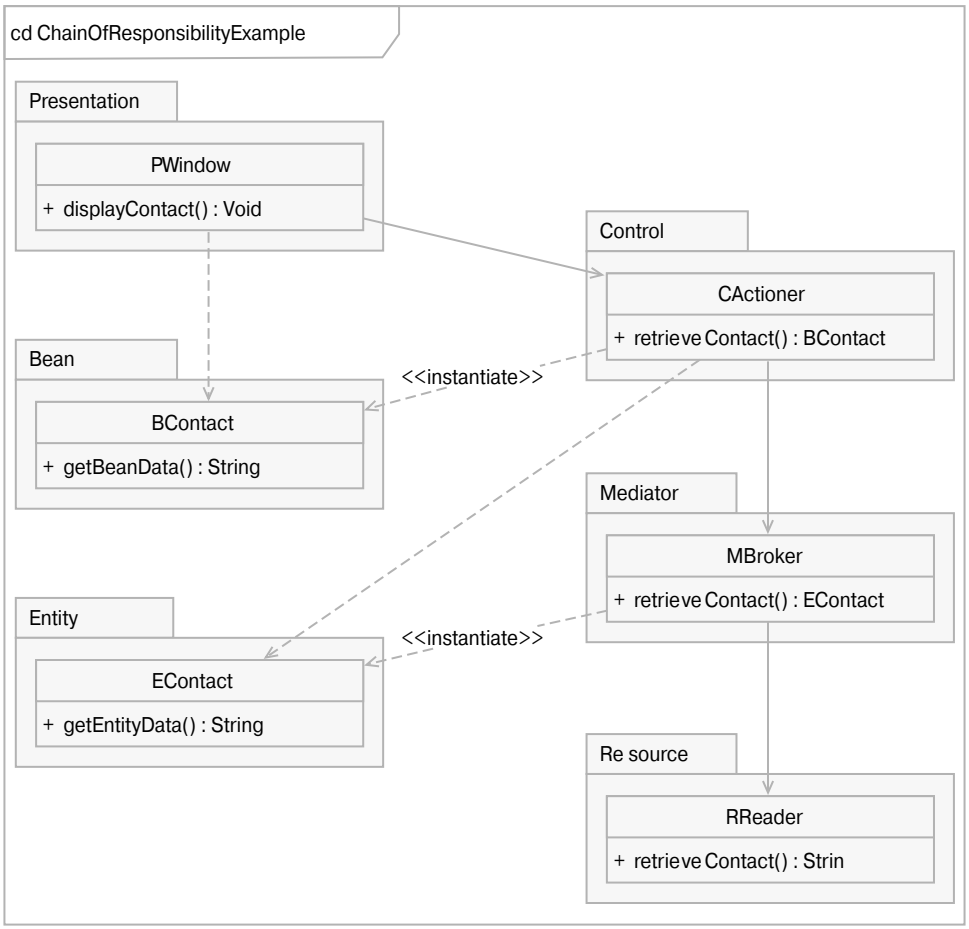


Рис. 6.11. Шаблон Цепочка обязанностей *для системы управления взаимоотношениями с заказчиками*

Шаблон связан с двумя видами объектов.

- Наблюдаемый объект, называемый *субъектом* (subject) или *издателем* (publisher).
- Наблюдающие объекты, называемые *наблюдателями* (observers), *подписчиками* (subscribers) или *слушателями* (listener).

Субъект может иметь много наблюдателей, являющихся его подписчиками. Все подписчики уведомляются об изменениях состояния субъекта и могут выполнять необходимую синхронизацию своих состояний с состоянием субъекта. Наблюдатели не связаны друг с другом и могут по-разному реагировать на уведомления об изменении состояния субъекта.

На рис. 6.12 показан *modus operandi* шаблона Наблюдатель, построенный с помощью мастера шаблонов Case-средства компании Sparx Systems. Абстрактный класс `Subject` (Субъект) хранит ссылки на своих наблюдателей и определяет операции для присоединения, отсоединения и уведомления наблюдающих объектов. Эти операции реализуются в классе `ConcreteSubject`, производном от класса `Subject`. Класс `ConcreteSubject` управляет своим состоянием и уведомляет своих наблюдателей о его изменении. Абстрактный класс (или интерфейс) `Observer` (Наблюдатель) определяет операцию `onUpdate()`, реализованную в классе `ConcreteObserver`. Класс `ConcreteObserver` может хранить ссылку на объект класса `ConcreteSubject` и после получения уведомления об изменении состояния субъекта обновляет свое состояние для поддержания согласованности с субъектом.

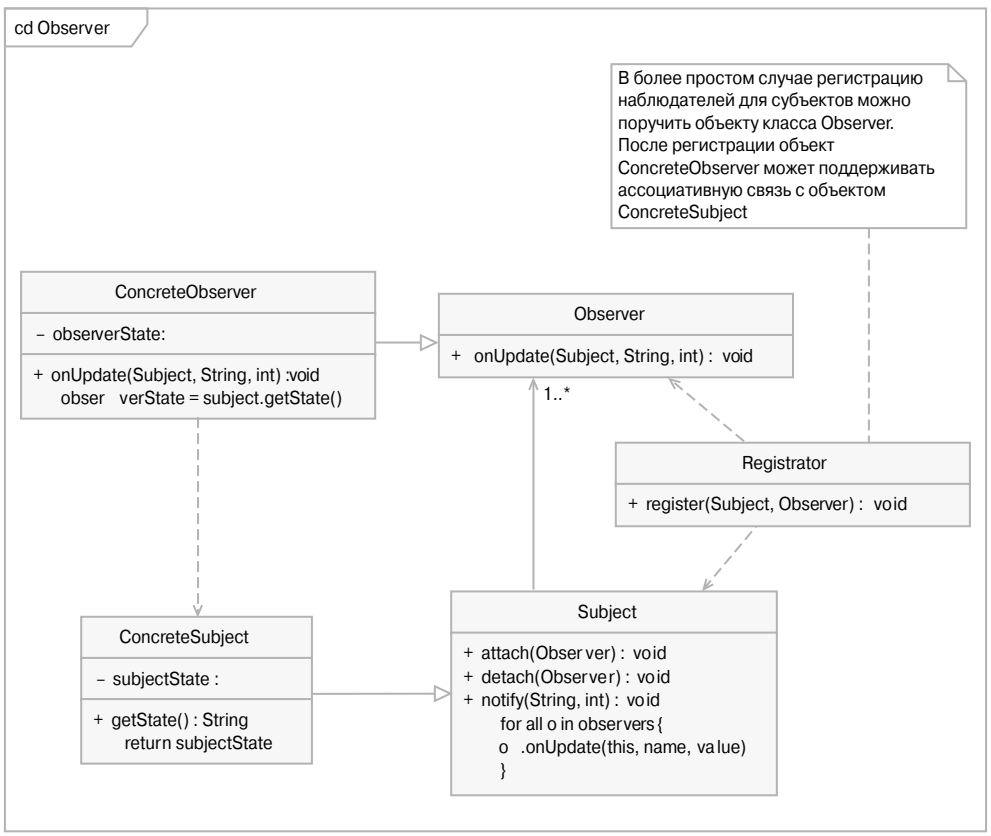


Рис. 6.12. Шаблон Наблюдатель

Несмотря на то что в определении шаблона Наблюдатель, сформулированном "Бандой четырех", упоминаются зависимости, этот шаблон поддерживает слабую связанность между субъектами и наблюдателями. Субъекты косвенно знают

о своих наблюдателях благодаря интерфейсу класса `Observer`. Регистрация и перерегистрация наблюдателей за субъектом осуществляется динамически и может обеспечиваться с помощью отдельного объекта `Registrator`. Уведомления об изменениях состояния субъекта автоматически направляются наблюдателям. Субъекты и объекты выполняют отдельные потоки управления, что еще больше ослабляет связанность между ними.

Шаблон Наблюдатель получил большую популярность благодаря тому, что он широко используется в библиотеках обработки событий, связанных с графическим пользовательским интерфейсом, например в библиотеке Java Swing. Например, класс `JMenuItem` из библиотеки Swing является субъектом, публикующим "уведомление о действии". Любой наблюдатель, желающий знать, какой пункт меню был выбран пользователем, должен подписаться на события класса `JMenuItem`. В реализациях шаблона Наблюдатель на языках программирования Java и C#.NET данные о событиях хранятся как объект класса событий (например, `PMenuEvent`). Затем объект события передается как параметр в сообщении о событии, направляемому наблюдателю (Larman, 2005; Maciaszek and Liong, 2005).

Слабая связанность классов в шаблоне Наблюдатель может (и должна) считаться преимуществом многоуровневых архитектурных моделей, поддерживающим как нисходящие, так и восходящие связи между уровнями, — в частности, когда субъект и наблюдатели не расположены на смежных уровнях. Принцип *UNP* (*upward notification principle* — принцип восходящего уведомления) в модели PCBMER рекомендует использовать шаблон Наблюдатель (см. раздел 4.1.3.2 главы 4). Это в свою очередь приводит к принципу *AVP* (*acquaintance package principle* — принцип осведомленного пакета), определенному в разделе 4.1.3.2, но до сих пор нигде в книге явно не использовавшемуся (Maciaszek and Liong, 2005).

Принцип AVP добавляет к шести уровням модели PCBMER *осведомленный* пакет/подсистему. Этот пакет ортогонален остальным шести, т.е. не расширяет иерархию уровней. Он состоит только из интерфейсов, допускающих *реализацию* на любом уровне модели PCBMER. Другие уровни могут получать доступ к этим реализациям *с помощью* интерфейсов. В зависимости от расположения использующего уровня — выше или ниже уровня реализации — осведомленный пакет допускает нисходящие и восходящие связи, включая прямую связь между удаленными уровнями.

Пример 6.4. Управление взаимоотношениями с заказчиками

Вернитесь к задаче 3, в которой описывается система управления взаимоотношениями с заказчиками (см. раздел 1.6.3 главы 1), и рассмотрите пример 6.3 (см. раздел 6.2.2.3).

Рассмотрите сценарий, в котором класс `EContact` является субъектом, который должен уведомлять наблюдателей об изменениях своего состояния.

Для примера ограничимся одним наблюдателем — `PContact` (окно представления, отображающее текущую информацию о контакте). Класс `EContact` публикует уведомление о событии, посылая классу `PContact` сообщение `onContactChange()`. Класс `PContact` реализует интерфейс наблюдателя, расположенный в пакете `Acquintance` (Уведомление).

Примените шаблон Наблюдатель к описанному выше сценарию. Разработайте модель классов для него и объясните ее.

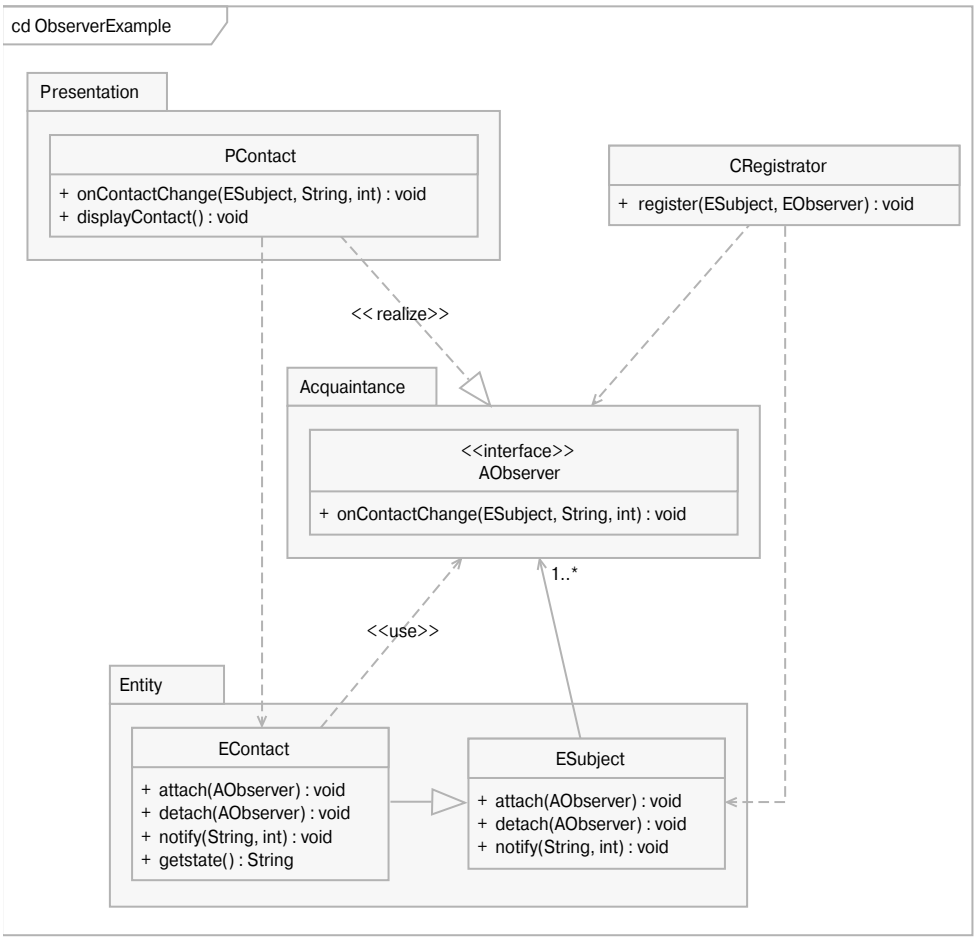


Рис. 6.13. Шаблон Наблюдатель *для системы управления взаимоотношениями с заказчиками*

На рис. 6.13 показана диаграмма классов, построенная для шаблона Наблюдатель, описанного в примере 6.4. Класс `CRegistrator` используется для подписки наблюдателя (класса `PContact`) на сообщения субъекта (класса `EContact`).

Класс AObserver представляет собой интерфейс в пакете Acquintance, реализованный классом PContact и используемый классом EContact для рассылки уведомления об изменениях состояния субъекта. Получив уведомление об изменении состояния субъекта, класс PContact вступает в непосредственный контакт с классом EContact, чтобы выполнить операцию getState(). Для выполнения этой функции методу displayContact() необходимы данные о состоянии субъекта.

6.2.2.5. Посредник

Шаблон Посредник (Mediator) определяет классы, инкапсулирующие взаимодействие между другими классами, как правило, расположенными на разных уровнях. Этот шаблон "обеспечивает слабую связанность, избавляя объекты от необходимости явно ссылаться друг на друга и независимо изменять способы взаимодействия" (Gamma et al., 1995).

Шаблон Посредник позволяет группировать сложные правила обработки данных, включая сложные бизнес-правила в специализированном классе или классах. Тем самым он освобождает другие объекты (*коллег*) от выполнения обработки с помощью потенциально весьма интенсивного обмена сообщениями. В результате объекты-коллеги становятся более связными и простыми. Кроме того, они становятся более независимыми от бизнес-правил и, следовательно, облегчают повторное использование.

Как и в большинстве других шаблонов, архитектор/проектировщик должен соблюдать баланс при использовании шаблона Посредник. Как указывают Гамма и его соавторы (Gamma et al., 1995), шаблон Посредник может противоречить требованиям объектно-ориентированного проектирования, которые касаются равномерного распределения функциональных свойств среди объектов. Однако слишком широкое распределение означает также, что объектная модель содержит большое количество связей и зависимостей между объектами.

Концептуальная модель шаблона Посредник показана на рис. 6.14. Интерфейс Mediator определяет операцию coordinate(), необходимую для выполнения работы, в которую вовлечены объекты класса Colleague. Когда объект Client вызывает метод coordinate(), объект ConcreteMediator использует свои ссылки на объекты класса ConcreteColleague, чтобы скоординировать работу, не вступая в контакты с другими коллегами (таким образом, эта работа выполняется под руководством класса ConcreteMediator). Однако по мере необходимости класс ConcreteMediator может передавать запросы объектам ConcreteColleague.

Обратите внимание на то, что объект ConcreteColleague знает своего посредника, но это знание является косвенным и обеспечивается интерфейсом Mediator. Иначе говоря, в простых системах, требующих только одного посредника, не обязательно определять интерфейс Mediator. В таких ситуациях для

организации связей между коллегами и посредником можно использовать шаблон Наблюдатель (т.е. коллеги могут действовать как субъекты, уведомляющие посредника об изменениях своего состояния, а посредник передает эту информацию другим коллегам). В шаблоне Наблюдатель объект посредника, инкапсулирующего сложную семантику обновления состояния субъекта и наблюдателей, называется *менеджером изменений* (Gamma et al., 1995).

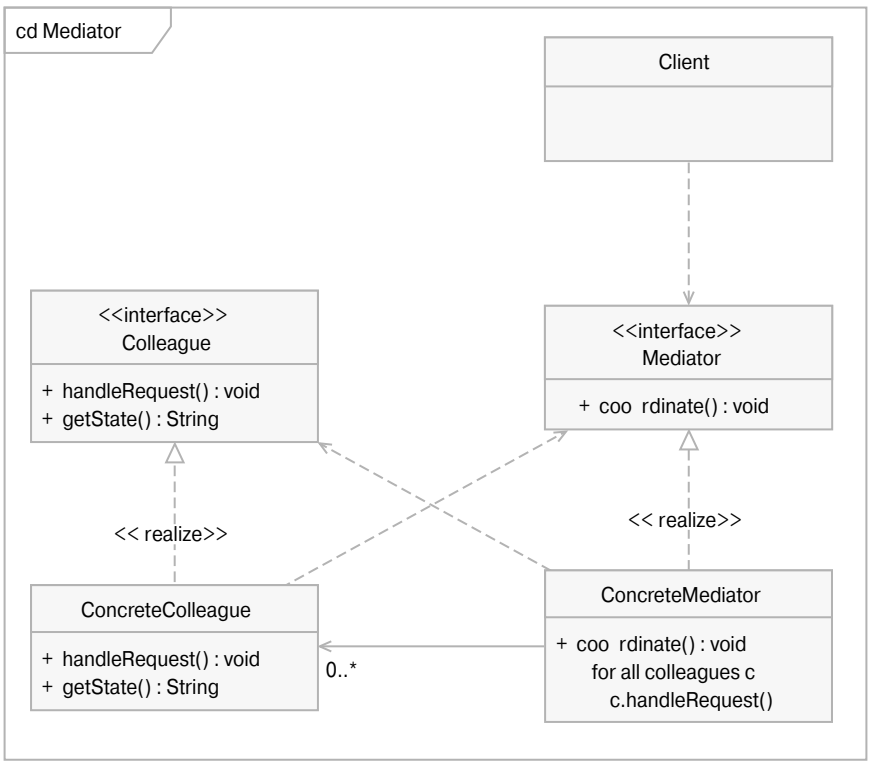


Рис. 6.14. Шаблон Посредник

Пример 6.5. Прямой маркетинг по телефону

Вернитесь к задаче 4, в которой описывается система управления прямым маркетингом по телефону (см. раздел 1.6.4 главы 1), и рассмотрите модель классов (см. пример 4.7 в разделе 4.2.1.2.3 и рис. 4.7 главы 4).

Одной из основных задач системы прямого маркетинга по телефону является создание динамических расписаний следующих звонков для каждого оператора, только что закончившего разговор. Для этого необходимо определять состояние каждого телефонного разговора и создавать новый экземпляр телефонного звонка.

Примените шаблон Посредник к описанному выше сценарию. Постройте диаграмму классов в модели PCBMER, используя принцип CNP (см. раздел 4.1.3.2). Объясните эту модель.

В модели PCBMER отдельный уровень называется так же, как и шаблон. Уровень посредника разделяется на уровни управления, сущностей и ресурсов. Интерфейс или класс `Mediator` также может функционировать как фасад (см. раздел 6.2.2.1) уровня посредника. Более того, уровень управления представляет собой результат применения шаблона Посредник, поскольку он может служить посредником между уровнями представления, компонентов, сущностей и посредника.

Схема решения примера 6.5, приведенная на рис. 6.15, разработана в предположении, что управляющий объект `CActioner` является клиентом, инициирующим создание расписания звонков для операторов системы. Решая эту задачу, он взаимодействует с конкретным посредником `Mediator` посредством интерфейса `IMMediator`. Класс `MMediator` содержит реализацию бизнес-логики, управляющей созданием расписания (с помощью метода `scheduleNextCall()`, и, возможно, других методов, не показанных в модели). Класс `MMediator` содержит набор ссылок на все активные в данный момент объекты класса `ECallScheduled`, чтобы он мог получать информацию об их состоянии, выбирать спонсора и кампанию, в рамках которой планируется следующий звонок, и создавать новый экземпляр класса `ECallScheduled`, связанный с классом `ETelemarketer`.

Контрольные вопросы 6.2

КВ1. Назовите две основные и контрастирующие вычислительные модели.

КВ2. По каким элементам системы вычисляется ее структурная сложность — по объектам или классам?

КВ3. Как называется шаблон, определяющий высокоуровневый интерфейс, облегчающий использование системы?

КВ4. Какой шаблон позволяет применить принцип NCP?

req_analysis_system design.indb 461

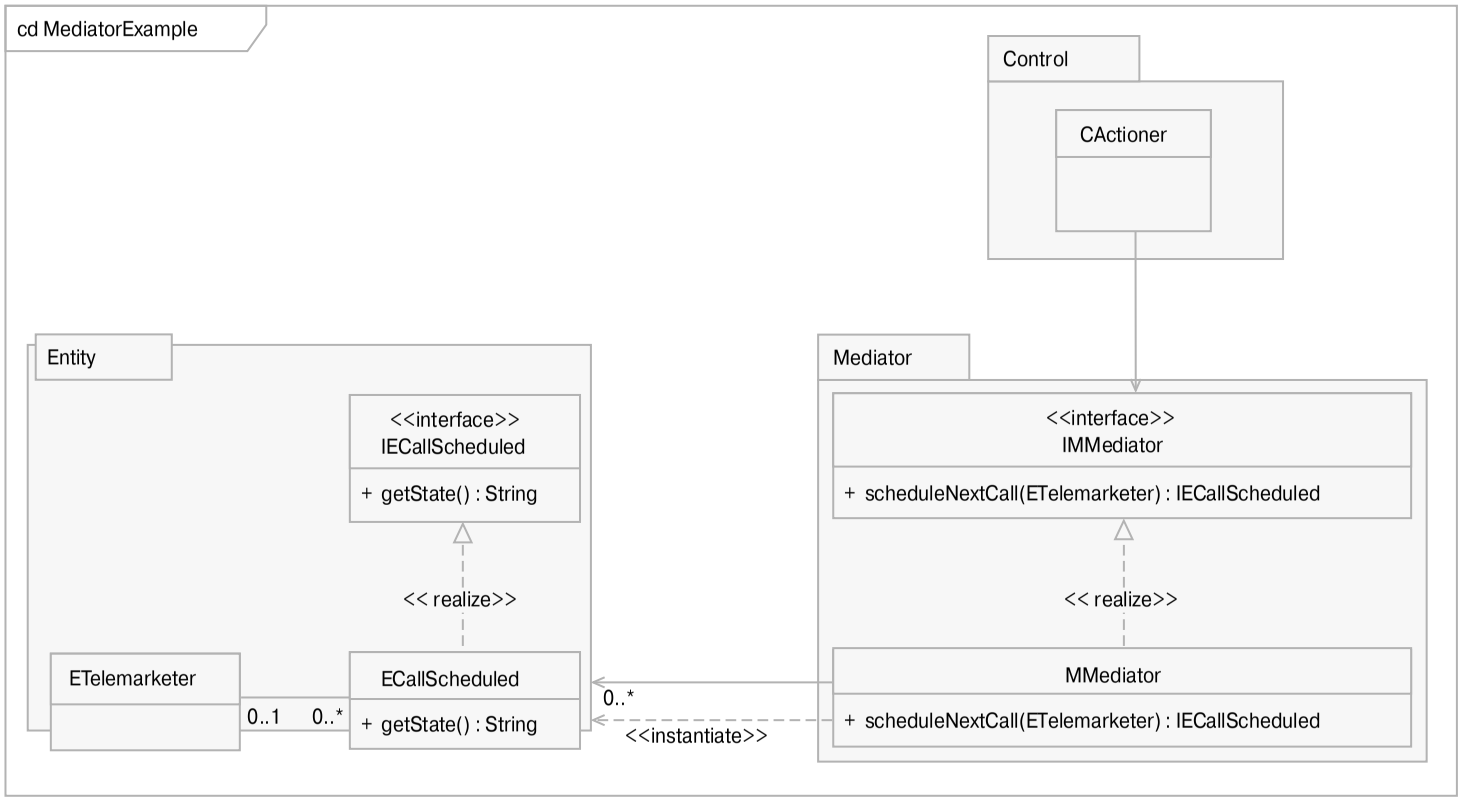


Рис. 6.15. Шаблон Посредник *для системы управления прямым маркетингом по телефону*

05.06.2008 14:43:18

6.3. Архитектурное моделирование

В языке UML *архитектурное моделирование* поддерживается средствами *моделирования реализации* (см. раздел 3.6 главы 3). В основе моделей реализации лежат такие понятия, как узел, компонент, пакет и подсистема. Кроме моделей реализации, язык UML поддерживает архитектурное моделирование с помощью *ограничений*, указанных в диаграммах классов. Ограничения чаще всего выражаются путем визуализации отношений зависимости в моделях. Зависимости представляют собой краеугольный камень архитектурных моделей (см. раздел 4.1 главы 4). Кроме того, они определяют архитектурную сложность системы (раздел 6.2.1).

6.3.1. Пакеты

Для представления групп классов (или других элементов моделирования, например, прецедентов использования) в языке UML предусмотрено понятие *пакета* (package) (см. раздел 3.6 главы 3). Пакеты служат для разделения логической модели прикладной программы. Они представляют собой кластеры классов, сильно связанных между собой, но слабо связанных с другими подобными кластерами (Lakos, 1996).

Пакеты могут быть *вложенными* (nested). Пакет, содержащий вложенные пакеты, имеет доступ к любому классу, непосредственно включенному во вложенные пакеты. Классом может владеть только один пакет. Это не препятствует появлению класса в других пакетах или взаимодействию с классами в других пакетах. Используя объявление видимости класса в пакете как закрытой, защищенной или открытой, можно контролировать взаимодействие и зависимости между классами в разных пакетах (см. раздел 5.1.2.2 главы 5).

Пакет изображается в виде пиктограммы папки (рис. 6.16). Вложенный пакет изображается внутри объемлющего пакета. Для каждого пакета должна быть построена собственная диаграмма классов, определяющая все классы, принадлежащие пакету. На рис. 6.16 показаны разные способы представления распределения классов по пакетам (Fowler, 2004). Из него следует, что программист на языке Java может импортировать класс `Data` либо из библиотечного пакета `java.sql`, либо из пакета `java.util`.

Пакеты могут быть связаны с двумя типами отношений: *обобщением* (generalization) и *зависимостью* (dependency). Зависимость пакета *A* от пакета *B* означает, что изменения в классе *B* могут потребовать изменения в классе *A*. Зависимости между пакетами возникают из-за передачи сообщений, т.е. когда класс из одного пакета посылает сообщение классу из другого класса.

В языке UML определено несколько разных категорий отношения зависимости (например, *по использованию*, *по доступу*, *по видимости*). Однако определение вида зависимости не приносит большой пользы. Истинный характер каждой за-

висимости должен быть зафиксирован как описательное ограничение в CASE-репозитории для использования в проектных моделях.

Обратите внимание на то, что отношение *обобщения* между пакетами предполагает также зависимость. Эта зависимость направлена от пакета подтипа к пакету супертипа. Изменения в пакете супертипа вызывают изменения в пакете подтипа.



Рис. 6.16. Пакеты и классы

На рис. 6.17 показаны пакеты и отношения зависимости между ними, описанные в примере 6.6. Зачисление зависит от классов `Grade` (Оценка) и `Timetable` (Расписание). Могут существовать четыре разных календаря, показанных с помощью отношения обобщения. Класс `Gregorian Calendar` (Григорианский календарь) зависит от класса `java::sql::Date`.

Пример 6.6. Зачисление в университет

Внимательное изучение системы, управляющей зачислением в университет, показывает, что для зачисления студентов на курс она должна "знать" расписание занятий и иметь данные об успеваемости студентов.

Мы не знаем, существуют ли оценки и расписания в виде отдельных модулей, в которые можно было бы встроить нашу систему. Если таких модулей нет, то их необходимо создать.

Задача заключается в следующем: необходимо разработать модель пакетов для системы зачисления в университет, соответствующую описанным выше требованиям. Эта модель должна учитывать требование, что расписание основано на типе `java.sql.Date` для григорианского календаря. Однако в модели следует учитывать возможность использования других календарей.

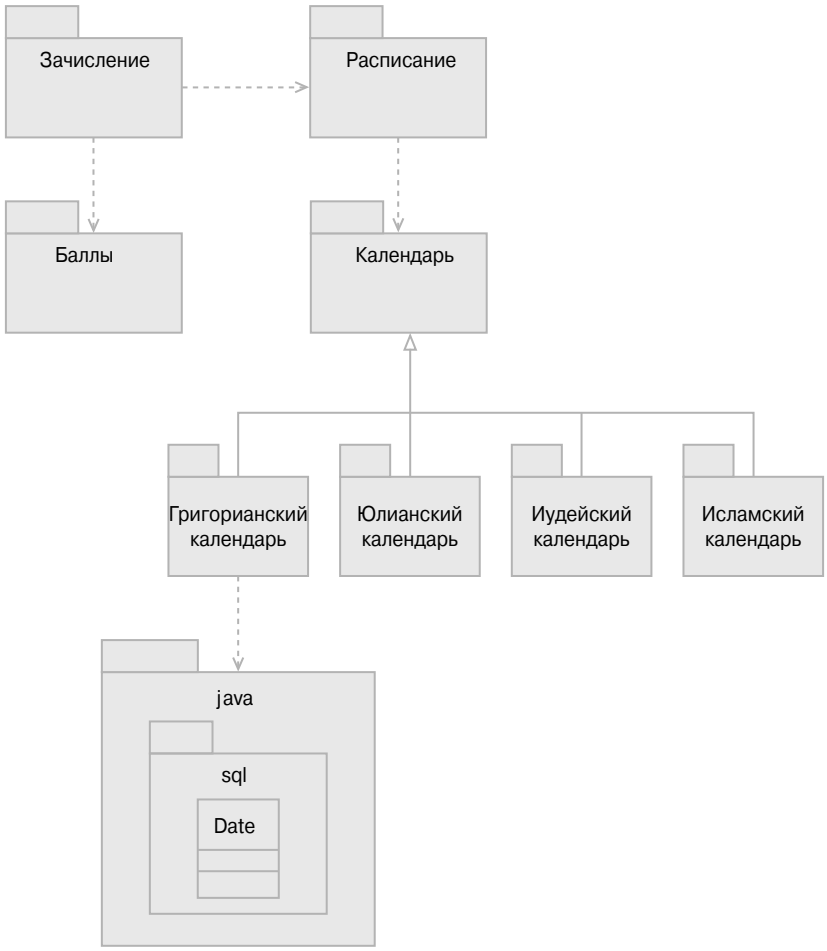


Рис. 6.17. Пакеты для системы управления зачислением в университет

6.3.2. Компоненты

Компонент (component) — это физическая часть системы, фрагмент реализации, программа (Booch et al., 1999; Lakos, 1996; Rumbaugh et al., 2005; Szyperski, 1998). Компоненты зачатую воспринимаются как бинарные исполняемые части системы (EXE-файлы). Однако компонент может быть также частью системы, которая не является непосредственно исполняемым модулем (например, файлом исходного текста программы, файлом данных, динамически компонуемой библиотекой (Dynamic Link Library — DLL) или хранимой процедурой базы данных).

Обозначения компонентов показаны на рис. 6.18 (см. также рис. 3.19 в разделе 3.6.2 главы 3). Язык UML допускает моделирование интерфейсов компонентов с помощью символа “леденца”. Если зависимость между компонентами уста-

навливается в порядке договоренности через интерфейсы, то другой компонент, реализующий тот же набор интерфейсов, может заменить один из компонентов, участвующих в отношении.

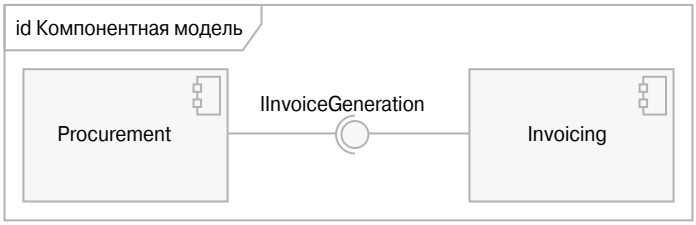


Рис. 6.18. Обозначения пакетов

Ниже приводится перечень характеристик компонента (Rumbaugh et al., 2005; Szyperski, 1998).

- Компонент представляет собой независимо развертываемый программный блок (компонент никогда не развертывается частично).
- Компонент может служить строительным блоком для стороннего разработчика (т.е. компонент в достаточной мере документирован и самодостаточен, чтобы сторонний разработчик мог "встроить" его в другие компоненты).
- Компонент не обладает тупиковым состоянием (т.е. компонент невозможно отличить от его собственных копий; в рамках любого данного приложения присутствует самое большее одна копия конкретного компонента).
- Компонент — заменяемая часть системы, т.е. его можно заменить другим компонентом, который согласуется с тем же интерфейсом.
- Компонент выполняет четкую функцию и с логической, и с физической точки зрения образует единое целое.
- Компонент может быть вложен в другие компоненты.

Диаграмма компонентов показывает компоненты и их взаимосвязь друг с другом. Компоненты могут быть связаны *отношениями зависимости*. Зависимый компонент запрашивает обслуживание у компонентов, на который указывает отношение зависимости. Компоненты также могут быть связаны отношениями композиции, т.е. один компонент может содержать другой.

6.3.2.1. Сравнение компонентов и пакетов

Пакет (package) — это группа элементов модели, имеющая имя (см. раздел 3.6.1 главы 3). На логическом уровне каждый класс принадлежит одному пакету. На физическом уровне каждый класс реализуется по меньшей мере одним компонентом, а компонент, возможно, реализует только один класс. Абстрактные классы, определяющие интерфейсы, зачастую реализуются несколькими компонентами.

Пакеты обычно представляют собой более крупные архитектурные элементы, чем компоненты. Они имеют тенденцию группирования классов *по горизонтали* — за счет статической близости классов, принадлежащих одной проблемной области. Компоненты — это *вертикальные группы* классов с близким поведением. Они могут принадлежать разным проблемным областям, однако вносят вклад в один фрагмент деловой деятельности, возможно, представленный прецедентом.

Описанное выше свойство ортогональности пакетов и компонентов затрудняет установление зависимостей между ними. Зачастую ситуация складывается так, что логический пакет зависит от нескольких физических компонентов.

Пример 6.7. Зачисление в университет

Обратитесь к описанию пакетов, представленных в примере 6.6 (раздел 6.3.1). Рассмотрите пакет `Timetable`. Предположим, что пакет реализуется как программа на языке C#, которая включает логику распределения университетских аудиторий по группам. Программа получает доступ к базе данных по аудиториям и группам. Для предоставления программе этой услуги реализуются две хранимые процедуры.

Постройте диаграмму компонентов, которая показывает зависимости между пакетами и необходимыми компонентами.

На рис. 6.19 показана модель компонентов для примера 6.7. Она включает три компонента: `RoomAllocEXE`, `RoomUSP` и `ClassUSP`. Пакет `Timetable` зависит от этих компонентов.

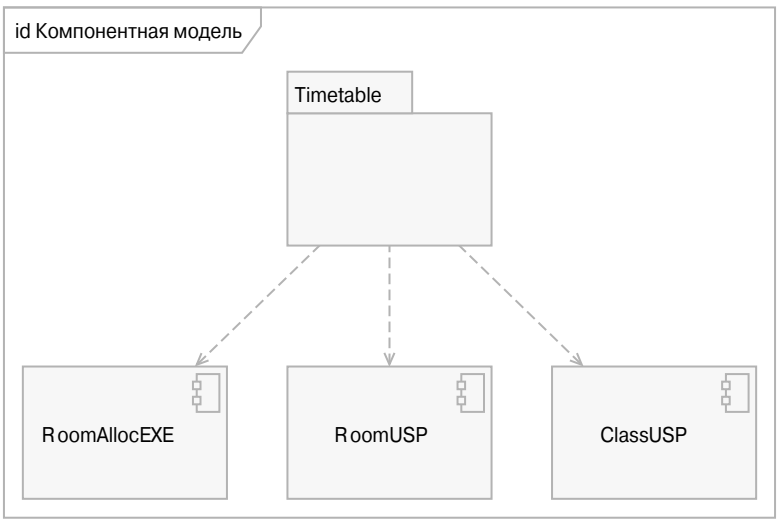


Рис. 6.19. Пакеты для системы управления зачислением в университет

6.3.2.2. Сравнение компонентов с классами и интерфейсами

Подобно классам, компоненты реализуют интерфейсы. Разница между ними имеет двойственный характер. Во-первых, компонент — это физическая абстракция, развертываемая на некотором компьютерном узле. Класс представляет собой логическую сущность, которая, для того чтобы действовать в качестве физической абстракции, должна быть реализована с помощью компонента. Во-вторых, компонент показывает только некоторые интерфейсы содержащихся в ней классов. Многие другие интерфейсы инкапсулированы компонентом — они используются только в кооперирующихся классах и не видимы другим компонентам.

Диаграмма компонентов, соответствующая требованиям, перечисленным в примере 6.8, показана на рис. 6.20.

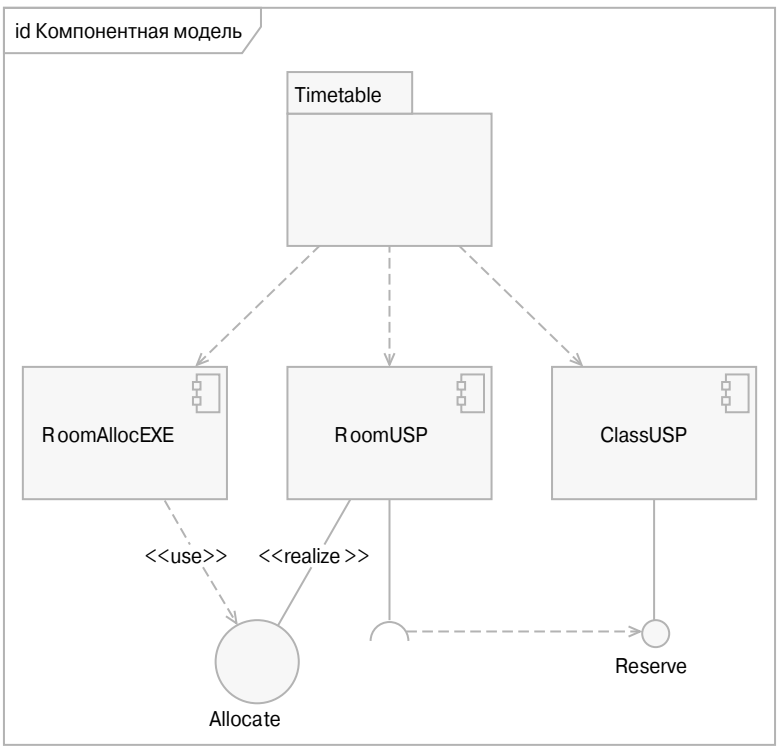


Рис. 6.20. Интерфейсы к диаграмме компонентов системы управления зачислением в университет

Пример 6.8. Зачисление в университет

Обратитесь к трем компонентам, представленным в примере 6.7 (см. раздел 6.3.2.1). Предположим, что компонент `RoomAllocEXE` инициирует процесс распределения аудиторий по курсам за счет обеспечения для

компонента ClassUSP идентификации классов. С этой целью реализует интерфейс Allocate.

Компонент ClassUSP выполняет остальную работу, запрашивая подробную информацию об аудитории у компонента RoomUSP. Для предоставления этой услуги RoomUSP реализует интерфейс Reserve.

6.3.3. Узлы

В языке UML распределенная физическая архитектура (см. раздел 6.1) или любая другая архитектура для системы выражается в виде *диаграммы развертывания* (deployment diagram). Вычислительный ресурс (физический объект, существующий на этапе выполнения программы) на диаграмме называется *узлом* (node). Узел имеет память и некоторые вычислительные возможности. Кроме того, узлом может быть сервер базы данных. Это подчеркивает важность того факта, что современные базы данных являются активными серверами (т.е. программируются).

В языке UML *узел* графически представляется в виде куба, который может быть обозначен как стереотип и содержать ограничения. Стереотипизированный куб может выглядеть как пиктограмма. Каждому узлу присваивается уникальное имя (текстовая строка).

Диаграмма развертывания показывает узлы и отношения между ними. Узлы могут быть связаны *ассоциациями*, которые могут иметь имена, чтобы указать используемый сетевой протокол (если это приемлемо) или описания характера соединения каким-либо другим способом. Ассоциации между узлами можно моделировать с использованием их типичных свойств, таких как порядок, кратность и роли.

На рис. 6.21 показаны четыре узла диаграммы развертывания. Отношения соединения демонстрируют вид связи между узлами.

Узлы представляют собой местоположение выполнения компонентов. Компоненты функционируют и развертываются в узлах. Узлы вместе с их компонентами иногда называют *элементами размещения* (distribution unit) (Booch et al., 1999).

На рис. 6.22 приведены два графических обозначения узла с компонентами, развернутыми (или содержащимися) в нем. Узел Корпоративная база данных, хранящий две хранимые процедуры, представленные компонентами Протокол безопасности клиента и Протокол безопасности счета. Обозначение для компонента, содержащегося в узле, можно расширить, и разместить в диаграмме развертывания всю диаграмму компонентов.

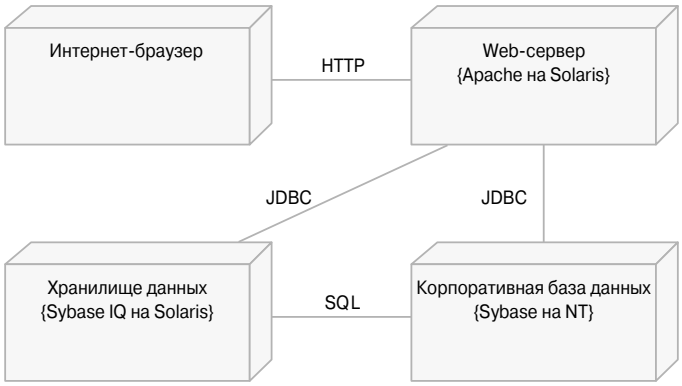


Рис. 6.21. Узлы

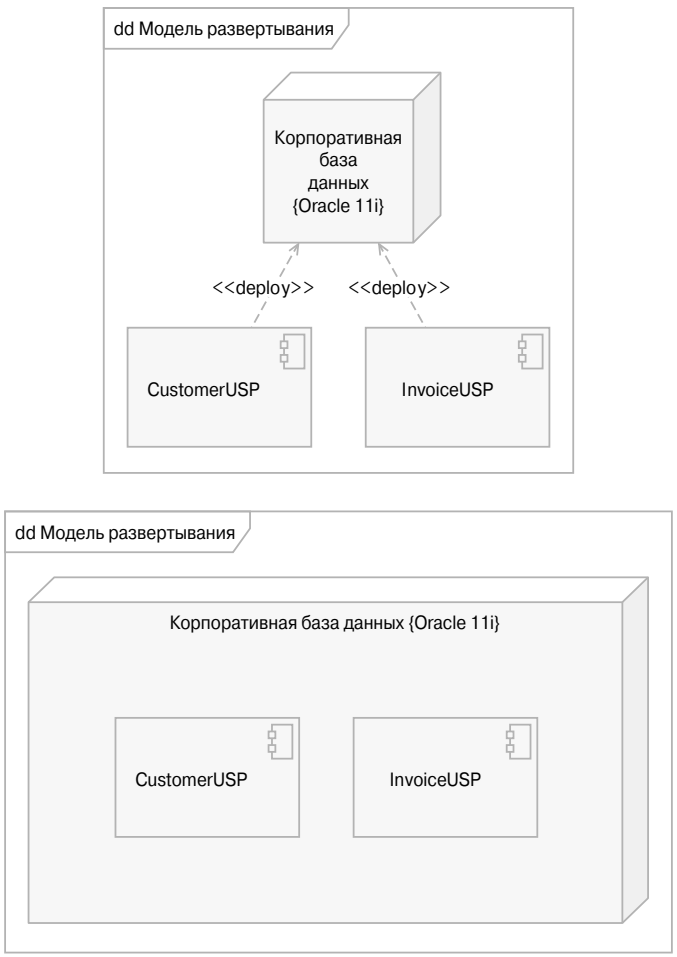


Рис. 6.22. Узел с развернутым/содержащимся компонентом

Контрольные вопросы 6.3

КВ1. Какие виды отношений используются для связи между пакетами?

КВ2. Имеет ли компонент персистентное состояние?

КВ3. Можно ли реализовать класс с помощью нескольких компонентов?

6.4. Принципы разработки и повторного использования программ

Проектирование программ (program design) является внутренней частью всего проектирования систем. *Архитектурное проектирование* (см. разделы 4.1 главы 4 и разделы 6.1–6.3) определяет обобщенную модель выполнения. *Детализированное проектирование* графического пользовательского интерфейса и баз данных определяет внешнее представление и внутреннее устройство этой модели. Проектирование программ заполняет пробелы в этой обобщенной среде и завершается созданием проектной документации, которой должен руководствоваться программист при реализации системы.

В ходе *проектирования программ* в каждый момент времени исследуется только одна прикладная программа. В этом смысле оно представляет собой прямое продолжение *проектирования пользовательского интерфейса*, рассматриваемого в главе 7. Кроме того, проектирование программ является расширением *проектирования баз данных*, обсуждаемого в главе 8. В частности, проектированию программ присущи процедурные аспекты проектирования баз данных, включающих в себя хранимые процедуры и триггеры.

Логика выполнения программы разделяется на клиентские и серверные процедуры. *Клиентский процесс* охватывает большую часть динамического взаимодействия объектов в программе. Правильный баланс между связностью и связанностью объектов может ограничить сложность такого взаимодействия. *Серверный процесс*, помимо прочего, реализует деловые транзакции, инициированные *клиентским процессом*.

6.4.1. Связность и связанность классов

Мимоходом мы определили основные принципы качественного проектирования программ, правда, по большей части в контексте качественного системного и архитектурного проектирования. Краеугольным камнем создания понятных, удобных в сопровождении и масштабируемых программ является *иерархия классов*. Для того чтобы избежать создания объектно-ориентированных программ,

устаревающих на следующий день после развертывания у заказчика, необходимо правильно пользоваться *наследованием* и *делегированием*.

Правильное проектирование программы предполагает сбалансированность между **связностью** (cohesion) и **связанностью** (coupling) классов. Термины связность и связанность были выработаны в теории структурного проектирования. Однако в объектно-ориентированном проектировании эти термины имеют аналогичный смысл (Larman, 2005; Page-Jones, 2000; Schach, 2005).

Связность классов — это степень внутреннего самоопределения класса. Она является мерой независимости класса. Классы, обладающие сильной связностью, выполняют одно действие, или их функционирование подчинено одной цели. Чем выше уровень связности, тем лучше.

Связанность классов измеряет силу связей между классами. Она является мерой взаимозависимости классов. Чем ниже уровень связанности, тем лучше (однако классы должны быть связаны друг с другом, чтобы взаимодействовать между собой!).

Связность и связанность находятся в противоречии друг с другом. Чем лучше связность, тем хуже связанность, и наоборот. Задача разработчика — найти оптимальный баланс между связностью и связанностью. Риль (Riel, 1996) предложил несколько эвристических правил, позволяющих решить эту проблему.

- Два класса должны быть либо независимы друг от друга, либо один из классов должен зависеть только от открытого интерфейса другого класса.
- Атрибуты и связанные методы должны находиться в одном классе (это правило часто нарушается классами, обладающими большим количеством *методов доступа* (get, set), определенных в их открытом интерфейсе).
- Класс должен выражать одну и только одну абстракцию. Информация, не имеющая отношения к данной абстракции, когда подмножество методов работает на соответствующем подмножестве атрибутов, должна быть перенесена в другой класс.
- Системная логика должна быть распределена по возможности равномерно (так, чтобы классы были равномерно загружены совместной работой).

Хороший пример компромисса между связностью и связанностью демонстрирует система управления зачислением в университет, описанная в разделе 4.3.3.3 главы 4. Этот пример содержит две контрастирующие диаграммы последовательностей — для централизованного решения (см. рис. 4.18) и для распределенного решения (см. рис. 4.19).

С точки зрения правильного баланса распределенное решение намного лучше. Оно обеспечивает лучшую (т.е. более слабую) связанность классов `CEnroll` и классов сущностей (не влияя при этом на степень связанности в слое сущностей, как показано на диаграммах классов для обоих решений, представленных на рис. 4.10 и 4.21). Кроме того, это решение обеспечивает лучшую (т.е. более вы-

сокую) связность, поскольку позволяет избежать превращения класса `CEnroll` в так называемый *раздувшийся контроллер* (bloated controller) (Larman, 2005), выполняющий несвойственные ему задачи и перегруженный работой.

6.4.1.1. Виды связанности классов

Для того чтобы два класса могли взаимодействовать, они должны быть связаны друг с другом. Между классами *X* и *Y* существует *связанность*, если класс *X* может *непосредственно* ссылаться на класс *Y*. Ларман (Larman, 2005) приводит шесть общих форм связанности классов.

- Класс *X* содержит класс *Y* или имеет атрибут (член класса или переменную экземпляра), ссылающийся на экземпляр класса *Y*.
- Класс *X* содержит метод, каким-то образом ссылающийся на экземпляр класса *Y*, например, использующий параметр или локальную переменную класса *Y*, или получающий объект класса *Y* в виде сообщения.
- Класс *X* вызывает сервис класса *Y* (т.е. посылает ему сообщения).
- Класс *X* является прямым или косвенным подклассом класса *Y*.
- Класс *X* содержит метод, входным аргументом которого является объект класса *Y*.
- Класс *Y* является интерфейсом, и класс *X* реализует этот интерфейс.

6.4.1.2. Закон Деметера

Связанность классов необходима для взаимодействия объектов, однако она должна быть как можно более сильно ограничена *уровнями класса*, т.е. представлять собой *связанность уровней* (intralayer coupling). Межуровневая связанность должна быть минимизирована и тщательно сориентирована. Дополнительные руководящие принципы для ограничения произвольного взаимодействия классов сформулированы в *законе Деметера*, известного в популярных формулировках — "не разговаривайте с незнакомцами" и "разговаривайте только со своими друзьями" (Leiberherr and Holland, 1989).

Закон Деметера определяет, на что могут быть нацелены сообщения в методах класса. Он гласит, что целью сообщений может быть только один из перечисленных ниже объектов (Larman, 2005; Page-Jones, 2000).

- Объект, в котором определен метод (т.е. в C++ — это объект, на который ссылается указатель `this`, в Java — экземпляр `self`, а в Smalltalk — экземпляр `super`).
- Объект, являющийся аргументом сигнатуры метода.
- Объект, на который ссылается атрибут объекта (включая объект, на который ссылается один из атрибутов коллекции).
- Объект, созданный методом.
- Объект, на который ссылается глобальная переменная.

Для ограничения связанности, порожденной наследованием, третье правило можно отнести лишь к атрибутам, определенным в самом классе. Атрибут, унаследованный классом, не может использоваться для обозначения целевого объекта для сообщения. Это ограничение известно как *усиленный закон Деметера* (Page-Jones, 2000).

С точки зрения практики закон Деметера предлагает лишь рекомендации для правильного архитектурного проектирования (см. раздел 6.2) и принципы архитектурного моделирования (см. раздел 4.1.3.2 главы 4). В частности, принцип NCP (взаимодействия соседей) реализуется путем применения закона Деметера к классам, находящимся на соседних уровнях, — "друзьям". "Незнакомцами" в этом случае являются классы, расположенные на более удаленных уровнях. Однако следует помнить, что во многих архитектурных моделях (включая модель PCBMER) соседними могут оказаться сразу несколько уровней.

6.4.1.3. Методы доступа и бессмысленные классы

Как упоминалось в разделе 6.4.1, атрибуты и связанные с ними методы должны находиться в одном классе (Reil, 1996). Класс должен сам "управлять своей судьбой". Он может ограничить доступ других классов к своему состоянию за счет запрещения методов доступа в своем интерфейсе. **Методы доступа** (accessor methods) определяют операции **наблюдателя** (операция get) или **модификатора** (операция set).

Методы доступа "открывают" класс для внутренних манипуляций со стороны другого класса. Несмотря на то что связанность подразумевает доступ к другим классам, чрезмерная распространенность методов доступа может привести к неравномерному распределению "интеллектуальной" нагрузки на классы. Класс, содержащий слишком много методов, рискует оказаться *бессмысленным* — другие классы определяют, что для него "хорошо".

Существуют ситуации, когда класс следует открыть для других классов. Это случается всякий раз, когда необходимо реализовать некоторую стратегию с помощью нескольких классов (Riel, 1996). Существует большое количество таких примеров.

Представим себе два класса, `Integer` (Целое число) и `Real` (Действительное число). Предположим, что нам необходимо реализовать стратегию преобразования целых чисел в вещественные и наоборот. В каком из двух классов должна быть реализована стратегия? Следует ли создать специальный класс `Converter` (Преобразователь) для реализации этой стратегии? Или, может быть, один из классов должен допустить использование методов доступа и вследствие этого может стать "бессмысленным" по отношению к этой стратегии.

Уместно вспомнить знаменитое высказывание Пейдж-Джонса на конференции OOPSLA'87.

> Представим себе, что на объектно-ориентированной ферме производят объектно-ориентированное молоко. Должна ли объектно-ориентированная корова отправлять объектно-ориентированному молоку сообщение "вытечь из коровы", или объектно-ориентированное молоко должно отправлять объектно-ориентированной корове сообщение "выдать молоко"?

Пример 6.9. Зачисление в университет

Предположим, что нам необходимо добавить студента в список слушателей курса. Для этого следует осуществить две проверки. Во-первых, необходимо выявить, изучение каких курсов является обязательным условием для прослушивания данного курса. Во-вторых, необходимо проверить учебное личное дело студента, чтобы выяснить, удовлетворяет ли студент этим обязательным условиям. Зная это, мы можем принять решение о том, можно ли внести студента в число слушателей курса.

Предположим, что сообщение `enrol()` (записать [на предлагаемый курс]) должно отправляться управляющему объекту `CEnroll`. Рассмотрим три класса — `ECourseOffering`, `ECourse`, и `EStudent` (Предлагаемый курс, Курс и Студент), *сотрудничающие* для решения этой задачи. Класс `EStudent` знает, как получить запись об академической успеваемости студента, а класс `ECourse` — как найти его реквизиты.

Задача заключается в том, чтобы выработать круг возможных диаграмм взаимодействия для решения проблемы. Рассмотрите аргументы за и против различных вариантов решения.

На рис. 6.23 показан первый вариант решения задачи, поставленной в примере 6.9 с помощью диаграммы последовательностей и диаграммы коммуникации. Управляющий объект класса `CEnroll` инициирует транзакцию с помощью отправки сообщения `enrol()` объекту класса `ECourse`. Объект класса `ECourse` запрашивает у объекта класса `EStudent` данные относительно учебного досье и сравнивает их с обязательными условиями. Объект класса `ECourse` решает, может ли объект класса `EStudent` быть внесен в список, и дает указание объекту класса `ECourseOffering` добавить объект класса `EStudent` в список студентов.

Сценарий, показанный на рис. 6.23, дает слишком много полномочий объекту класса `ECourse`, на которого возлагается реализация стратегии, а класс `EStudent` лишается смысла. Очевидно, что решение не сбалансировано, однако ясного выхода из ситуации не видно.

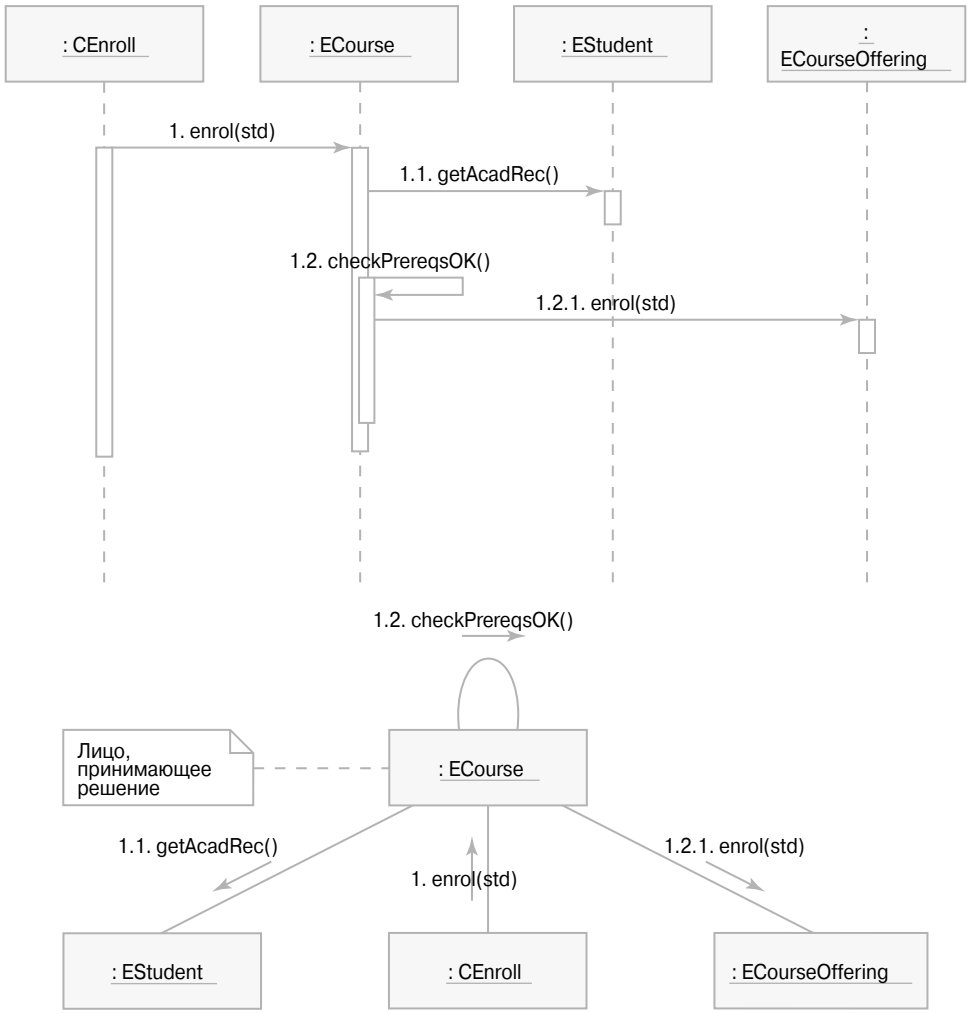


Рис. 6.23. Класс `ECourse` *как средство реализации стратегии, лежащей в основе зачисления в университет*

Можно перенести акценты с класса `ECourse` на класс `EStudent` и получить решение, показанное на рис. 6.24. Теперь объект класса `CEnroll` просит объект класса `EStudent` выполнить основную работу. Объект класса `EStudent` вызывает метод `getPrereq()` объекта класса `ECourse`. Именно объект класса `EStudent` решает, возможна ли запись на курс, и дает указание объекту `ECourseOffering` записать студента.

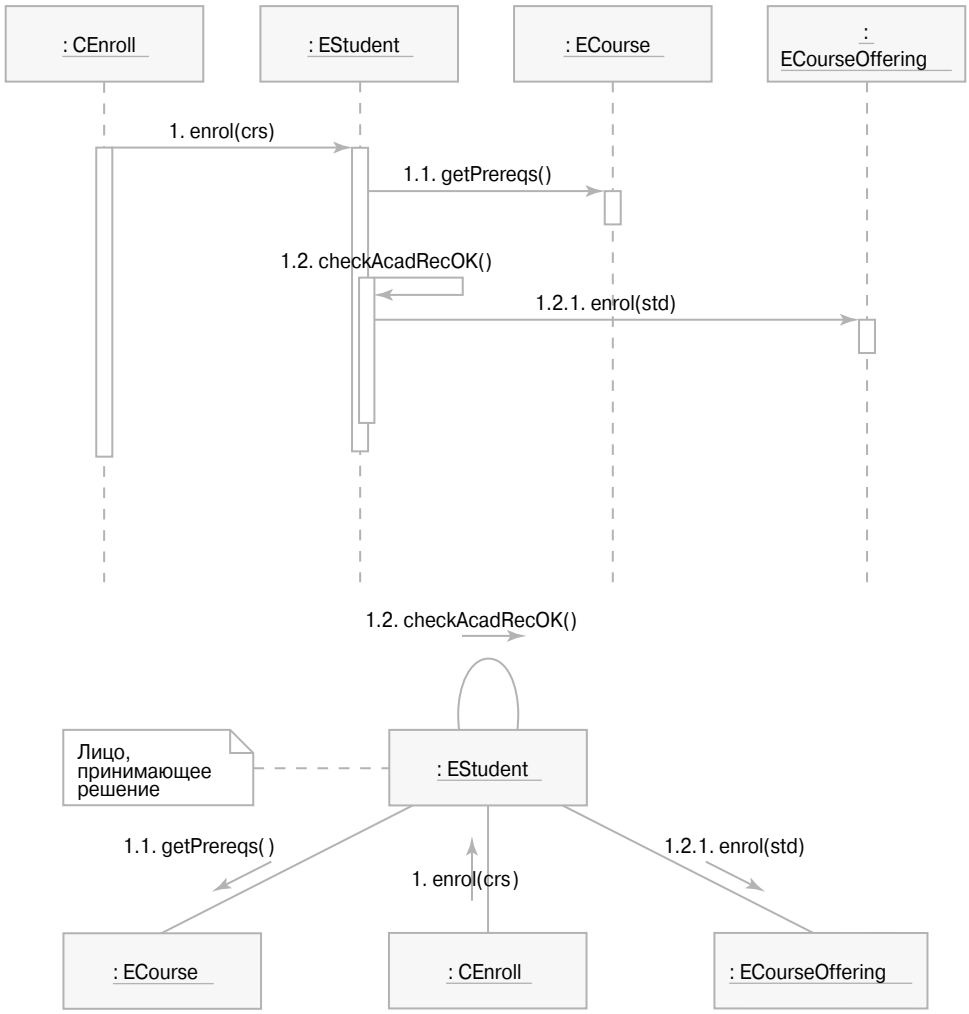


Рис. 6.24. Класс `EStudent` *как средство реализации стратегии, лежащей в основе зачисления в университет*

На рис. 6.25 показано более сбалансированное решение, при котором реализация политики возлагается на объект класса `ECourseOffering`. Это решение является объективным по отношению к объектам класса `ECourse` *и* `EStudent`, но оно делает существование этих двух объектов бесполезным и бессмысленным. Объект класса `ECourseOffering` действует подобно "главной программе" (по выражению Риеля (1996), играет роль *класса* "*Бог*").

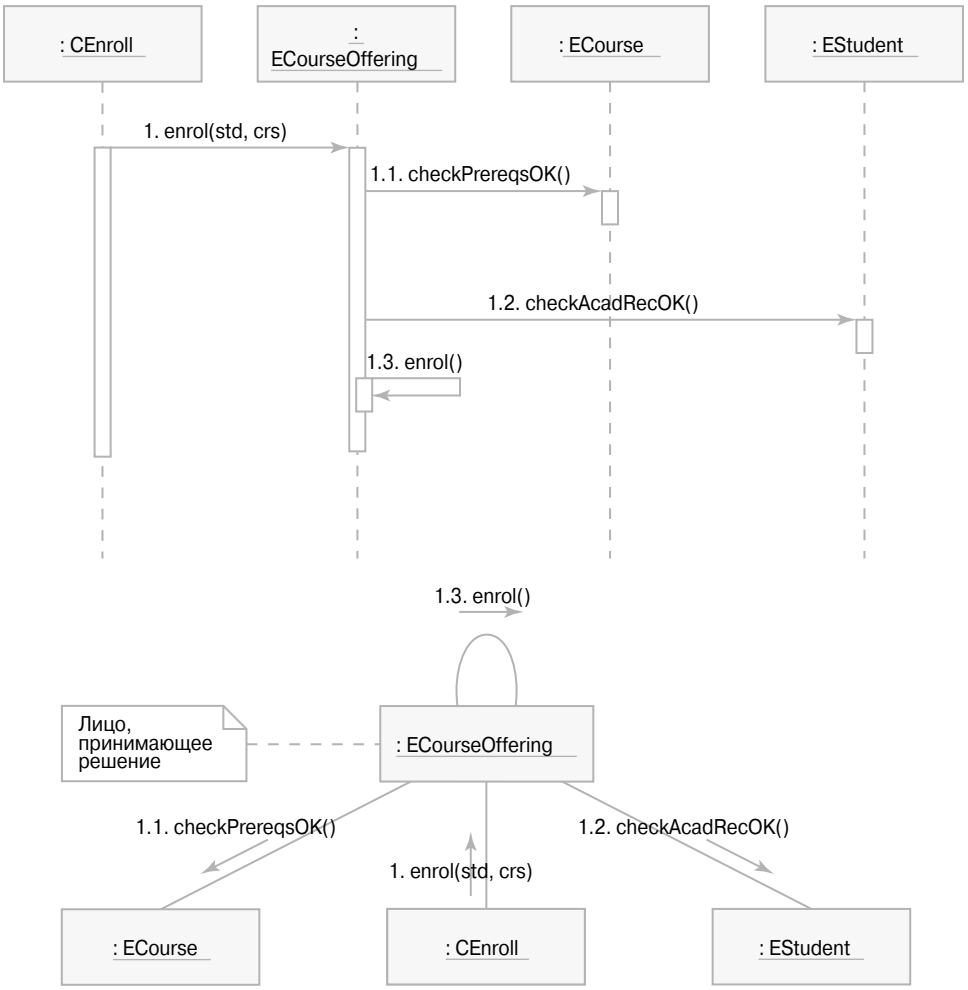


Рис. 6.25. Класс `ECourseOffering` *как средство реализации стратегии, лежащей в основе зачисления в университет*

Все решения, описанные выше, являются *распределенными* по своей природе. Их обоснование приведено в разделе 4.3.3.3 главы 4. Возможно также *централизованное* решение. Такое решение должно возлагать реализацию стратегии на класс `CEnroll`. Тем не менее лучше создать отдельный класс, который можно было бы расположить на *уровне посредников* модели PCBMER и назвать `MEnrollmentPolicy`.

Класс-посредник `MEnrolmentPolicy` на рис. 6.26 устраняет связь между тремя классами сущностей при реализации стратегии записи на курсы. Это дает определенные преимущества, так как любые изменения политики инкапсулированы в рамках одного класса-посредника. Однако существует риск, что класс `MEnrolmentPolicy` может превратиться в класс "Бог".

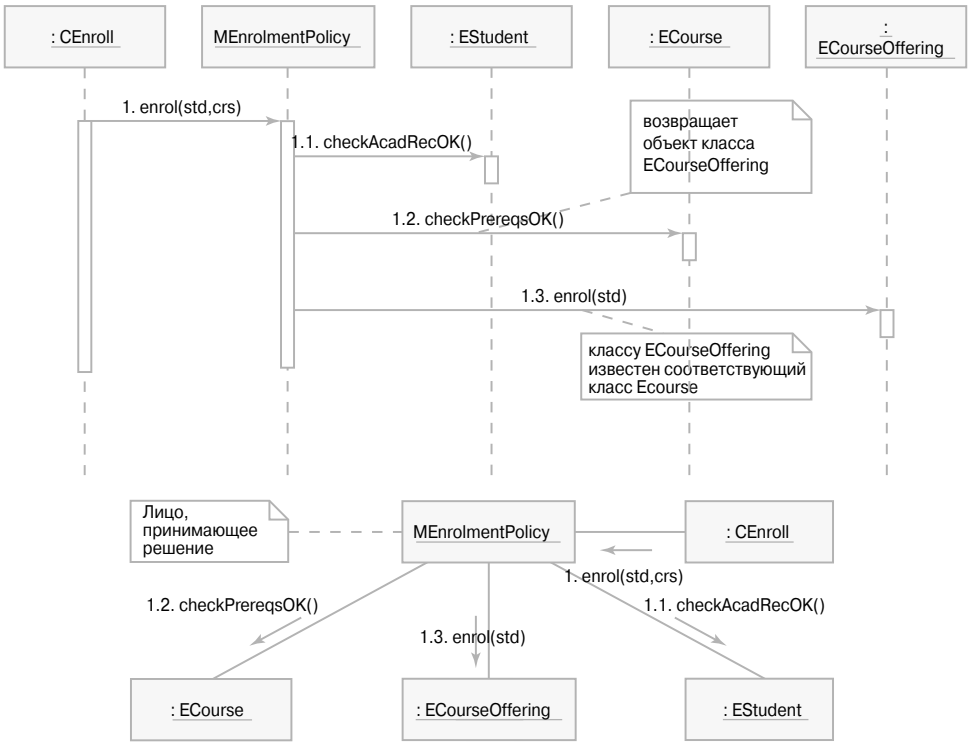


Рис. 6.26. Класс `MEnrolmentPolicy` *как средство реализации стратегии, лежащей в основе зачисления в университет*

6.4.1.4. Динамическая классификация и связность классов со смешанными экземплярами

В разделе A.7.5 приложения A рассматривается проблема *динамической классификации* и показано, что наиболее распространенные объектно-ориентированные среды программирования ее не решают. Цена, которую приходится платить за отсутствие подобной поддержки, зачастую выражается в проектировании классов, которые отличаются *связностью смешанных экземпляров* (mixed-instance cohesion).

Пейдж-Джонс (2000) дает следующее определение подобного класса: "Класс со *связностью смешанных экземпляров* обладает некоторыми свойствами, которые не определены для некоторых объектов класса". Некоторые методы класса применимы только к подмножеству объектов этого класса, и некоторые атрибуты имеют смысл только для подмножества объектов. Например, класс `Employee` (Сотрудник) может определять объекты, которые являются как рядовыми работниками, так и менеджерами. Менеджерам выплачивается надбавка. Отправка сообщения о выплате надбавки `payAllowance()` объекту класса `Employee` не имеет смысла, если этот объект не принадлежит к подмножеству менеджеров.

Для того чтобы избавиться от связности смешанных объектов, необходимо расширить иерархию обобщения, чтобы идентифицировать подклассы класса `Employee`, например `OrdinaryEmployee` (Рядовой сотрудник) и `Manager` (Менеджер). Однако объект класса `Employee` может в какой-то момент времени быть разновидностью класса `OrdinaryEmployee`, а затем разновидностью класса `Manager`, и наоборот. Для того чтобы исключить связность смешанных объектов, необходимо разрешить объектам динамически изменять принадлежность классу во время выполнения программы, но это не помогает, если динамическая классификация не поддерживается.

Пример 6.10. Зачисление в университет

Рассмотрите следующие варианты примера 6.9.

- Посещение вечернего курса возможно только для студентов-вечерников.
- Студенты стационарного обучения могут записываться только на дневные курсы.
- Если студенты-вечерники желают записаться на вечерние курсы, они должны внести небольшую дополнительную плату.
- Студенты-вечерники автоматически рассматриваются как студенты стационара, если они записываются на курсы, позволяющие получить более чем на шесть кредитных пунктов (т.е. обычно более чем на два предложенных курса) в данном семестре, и наоборот.

Предложите сильно связную структурную модель кооперации без связности смешанных экземпляров. Предложенную модель следует подвергнуть критической оценке, а также выдвинуть и обсудить альтернативное решение, позволяющее избежать проблем с динамической классификацией.

Для того чтобы исключить связность смешанных экземпляров, упомянутую в примере 6.10, требуется произвести специализацию класса `EStudent` в виде двух подклассов `PartTimeStudent` (Студент вечернего отделения) и `FullTimeStudent` (Студент стационара) (рис. 6.27). Если каждый студент должен быть отнесен к вечерней *или* дневной форме обучения, то класс `Student` является абстрактным. Сообщение о необходимости внести дополнительную плату `payExtraFee(crsoff)` никогда не отправляется объекту класса `FullTimeStudent`, поскольку он не содержит методов его обработки.

К сожалению, осталась одна проблема. Студент вечернего отделения может отдать предпочтение изучению дневного курса (т.е. `evening_preference = 'False'`), не внеся никакой дополнительной платы. Другими словами, мы по-прежнему сталкиваемся с проблемой связности смешанных экземпляров

подкласса `PartTimeStudent`. Отправка сообщения `payExtraFee(crsoff)` объекту класса `PartTimeStudent` не имеет смысла, если студент записывается на дневной предлагаемый курс.

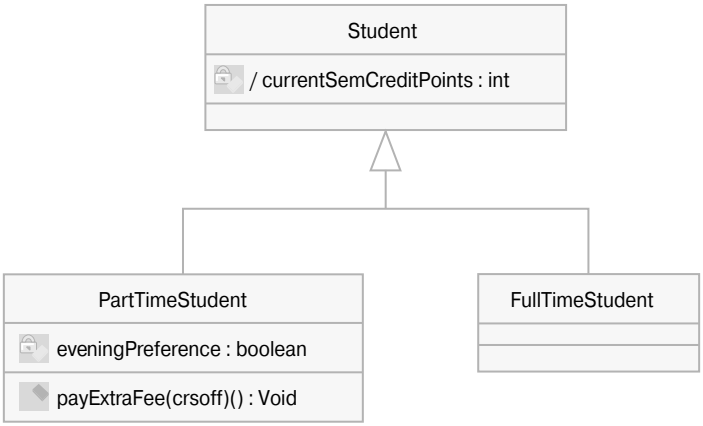


Рис. 6.27. Моделирование классов, позволяющее исключить связность смешанного элемента в системе управления зачислением в университет

На рис. 6.28 показано расширение схемы, позволяющее исключить второй аспект связности смешанных экземпляров. Класс `DayPrefPartTimeStudent` (Студент вечернего отделения, предпочитающий дневные курсы) не обладает методом `payExtraFee(crsoff)`. Но что делать, если объект класса `DayPrefPartTimeStudent` *вынужден* записываться на вечерний курс, поскольку на дневном курсе обучения нет свободных мест? Наверное, в этом случае можно применить другой вид дополнительной оплаты. Следует ли продолжать специализацию классов, чтобы дойти до производного класса `UnluckyDayPrefPartTimeStudent` (Студенты вечернего отделения, предпочитающие дневные курсы обучения, но не попавшие на них).

Для того чтобы не оказаться в нелепом положении, следует отказаться от идеи все дальше и дальше продолжать процесс исключения связности смешанных экземпляров. Мы даже больше не станем упоминать динамическую классификацию. В действительности форма обучения студента определяется текущим значением атрибута `currentSemCreditPoints`.

Аналогично, студент может в любой момент изменить свое предпочтение в отношении вечернего или дневного времени посещения предложенного курса. Поскольку среды программирования не поддерживают динамическую классификацию, изменение принадлежности объекта разным классам относится к компетенции программиста. Для постоянных объектов, значения OID которых содержат идентификаторы класса, это слишком грубый метод.

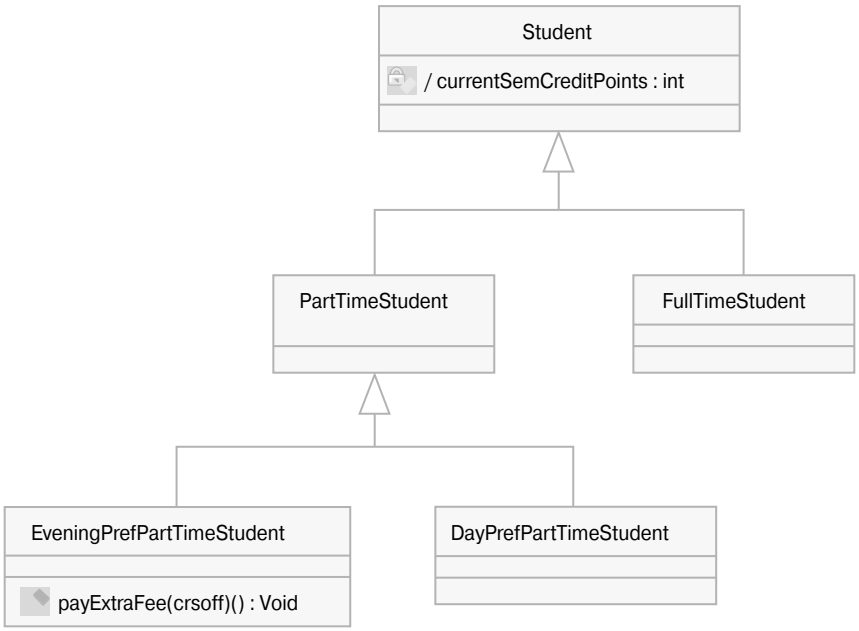


Рис. 6.28. Моделирование классов, позволяющее исключить другой аспект связности смешанного элемента в системе управления зачислением в университет

Альтернативное решение сводится к ограничению глубины иерархии наследования, исключению динамической классификации и разрешению определенного уровня связности смешанных экземпляров. Например, можно вернуться к модели классов, приведенной на рис. 6.27, и решить проблему, связанную с предпочтениями студента, позволив объекту по-разному реагировать на сообщение `payExtraFee(crsoff)` в зависимости от значения атрибута `eveningPreference`.

Подобное решение можно запрограммировать с помощью оператора `if`, как показано в приведенном ниже примере псевдокода.

```
method payExtraFee(crs_off) for the class PartTimeStudent
if evening_preference = 'False'
     return
else
     do it
end method
```

Несмотря на то что использование оператора `if` в объектно-ориентированной программе означает отказ от наследования и полиморфизма, это может оказаться неизбежным по чисто прагматическим соображениям. Вместо того чтобы бороться с *динамической классификацией*, программист вводит в класс *динамическую семантику*. Объект по-разному реагирует на одно и то же сообщение в зависимости

от его локального текущего состояния. Для проектирования динамической семантики для класса можно использовать диаграммы состояний. В ходе проектирования, вполне вероятно, пострадает *связность* класса.

6.4.2. Стратегия повторного использования

Язык UML определяет повторное использование (reuse) как "использование ранее существовавших артефактов" (Rumbaugh et al., 2005). В ходе изложения мы уже обсуждали объектно-ориентированные методы для повторного использования программного обеспечения, такие как наследование и делегирование (см. разделы 5.2 и 5.3 главы 5). Как указывалось ранее, называть реализацию наследования неотъемлемой чертой объектной ориентации и основным способом повторного использования кода небезопасно. Снова процитируем работу Румбо и соавторов (Rumbough et al., 2005): "Следует помнить, что повторное использование кода может осуществляться разными средствами, включая копирование кода, а не только с помощью наследования. Одной из основных ошибок при моделировании является навязывание неприемлемого обобщения при попытке повторного использования кода, которая часто приводит к проблемам".

В данном разделе рассматриваются *стратегии* повторного использования программного обеспечения. Оказывается, эти стратегии также влияют на степень детализации, с которой осуществляется повторное использование. Могут применяться следующие степени детализации повторного использования:

- класс;
- компонент;
- идея решения.

В связи со степенью детализации существуют три соответствующие стратегии повторного использования (Coad et al., 1995; Gamma et al., 1995), в основе которых лежат следующие программные сущности:

- инструментальные средства (библиотеки классов);
- каркасы;
- шаблоны анализа и проектирования.

6.4.2.1. Повторное использование инструментальных средств

Стратегия повторного использования *инструментальных средств* придает особое значение многократному использованию программного кода на уровне *классов*. При этой стратегии повторного использования программист "заполняет пробелы" в программе за счет осуществления вызовов *конкретных классов* из некоторых библиотек классов. Основной текст программы не подлежит повторному использованию — его пишет программист.

Существуют два типа (уровня) инструментальных средств (Page-Jones, 2000):

- базовые инструментальные средства;
- архитектурные инструментальные средства.

Базовые классы (foundation classes) широко представлены объектными программными средами. Они включают классы для реализации элементарных типов данных (такие, как `String`), структурных типов данных (таких, как `Date`) и коллекций (таких, как `Set`, `List` или `Index`).

Архитектурные классы (architecture classes) обычно представлены как части системного программного обеспечения, такого как операционные системы, системы управления базами данных (СУБД) или программное обеспечение, реализующее графический пользовательский интерфейс. Например, когда мы приобретаем объектную СУБД, то фактически получаем архитектурные инструментальные средства, реализующие ожидаемые функциональные возможности системы, такие как поддержка хранения персистентных объектов, выполнение транзакций и обеспечение параллельности.

6.4.2.2. Повторное использование каркасов

Стратегия повторного использования *каркасов* (framework) придает особое значение многократному использованию на уровне компонентов (см. разделы 3.6.2 главы 3 и раздел 6.3.2). В противоположность повторному использованию инструментальных средств каркас предоставляет в распоряжение разработчиков скелет программы. Затем программист "заполняет пробелы" этого скелета (настраивает его), создавая программный код, вызовы которого встроены в каркас. Помимо конкретных классов (для самого каркаса), каркас предоставляет в распоряжение программиста массу абстрактных классов, которые он должен реализовать (настроить).

Каркас — это настраиваемое прикладное ПО. Лучшими примерами каркасов служат системы ERP (enterprise resource planning systems — системы планирования ресурсов предприятий), такие как SAP, PeopleSoft, Baan или J.D. Edwards. Однако повторное использование этих систем не основано на чистых объектно-ориентированных методах.

Объектно-ориентированные каркасы для разработки информационных систем предлагаются в рамках распределенных компонентных технологий, таких как J2EE/EJB и .NET. Они известны как *бизнес-объекты*, т.е. "поставляемые" программные продукты, призванные удовлетворить специфические деловые или прикладные потребности. Например, бизнес-объект может быть каркасом системы бухгалтерского учета с настраиваемыми классами, такими как `Invoice` (Счет-фактура) или `Customer` (Клиент).

Несмотря на то что каркасы привлекательны с точки зрения повторного использования, они обладают рядом недостатков. Пожалуй, самым значительным из них является то, что исходное решение на основе "наименьшего общего знамена-

теля", которое они предлагают, является неоптимальным, а то и вовсе устарелым. В результате каркасы не дают конкурентного преимущества своим пользователям и могут оказаться весьма обременительными в стремлении внедрить более современные решения.

6.4.2.3. Повторное использование шаблонов

Стратегия повторного использования кода в ходе разработки (см. раздел 6.2.2) еще более усиливается благодаря *шаблонам* (patterns). Они предоставляют в распоряжение разработчиков апробированные идеи и примеры взаимодействия объектов, приводящие к понятным и масштабируемым решениям. Шаблоны могут применяться на этапе анализа или проектирования программного обеспечения. По этой причине существуют *шаблоны анализа* (analysis patterns), *архитектурные шаблоны* (architectural patterns) и *шаблоны проектирования* (design patterns).

Шаблон — это документально оформленное решение, доказавшее свою работоспособность в ряде ситуаций. Эти ситуации четко обозначены и могут использоваться в качестве индекса для поиска приемлемого решения в ходе разработки. Кроме того, для того чтобы дать возможность разработчику принять обоснованное решение, необходимо привести перечень всех известных недостатков и побочных эффектов шаблона.

Повторное применение шаблонов во многом носит концептуальный характер, хотя многие проектные шаблоны содержат примеры программного кода для повторного использования программистами. *Диапазон применения шаблона проектирования* (например, Gamma et al., 1995) определяется последовательностью взаимодействий — обычно он охватывают область, более широкую, чем класс, но более узкую, чем компонент. *Диапазон применения шаблона анализа* (например, Fowler, 1997) зависит от уровня абстракции моделирования, к которой применяется шаблон.

Контрольные вопросы 6.4

КВ1. Какой термин используется для определения степени внутреннего самоопределения класса?

КВ2. Связность смешанных объектов возникает из-за относительной слабости сред объектно-ориентированного программирования. В чем заключается эта слабость?

КВ3. Как называется повторное использование на уровне компонентов?

6.5. Моделирование кооперации

Архитектурное проектирование (architectural design) оказывает влияние на *детализированное проектирование* (detailed design) в том смысле, что оно определяет целевую программно-аппаратную платформу, с которой должен быть согласован детализированный проект. Помимо этого, детализированное проектирование является прямым продолжением анализа. Задача заключается в том, чтобы превратить модели анализа в документы детализированного проекта, на основе которых программисты могут реализовать систему.

В ходе *анализа* мы упрощаем модели, абстрагируясь от деталей, мешающих ясно представить определенную точку зрения на систему. При *проектировании* мы стремимся действовать наоборот. В каждый момент времени мы рассматриваем только одну из архитектурных частей системы и добавляем в модель технические детали или создаем качественно новую проектную модель на более низком уровне абстракции.

В ходе изложения мы довольно свободно использовали термин **кооперация** (collaboration), иногда заменяя его словом *взаимодействие*, говоря о множествах объектов, которые кооперируются для выполнения задания. По историческим причинам в языке UML смысл термина "взаимодействие" был не вполне точным и постоянно изменялся. Начиная с версии UML 2.0 понятие "взаимодействие" стало точнее и было погружено в контекст так называемых **композитных структур** (composite structures). Термин *композитная структура* означает "композицию взаимосвязанных элементов, образующих экземпляры на этапе выполнения программы и взаимодействующих по линиям связи для достижения общих целей" (UML, 2005). Композитные структуры можно моделировать с помощью отдельных *диаграмм композитных структур* (composite structure diagrams).

6.5.1. Кооперация

> *Кооперация* описывает структуру, состоящую из кооперирующихся элементов (ролей), каждый из которых выполняет специализированную функцию, а вместе с другими обеспечивает некоторое желательное функциональное свойство. Основная цель кооперации — объяснить, как работает система, поэтому она обычно учитывает лишь те аспекты реальности, которые важны для этого объяснения... Кооперация представляет собой разновидность классификатора и определяет набор ролей, которые играют взаимодействующие экземпляры, а также набор соединительных звеньев, образующих линии связей между участвующими экземплярами (UML, 2005).

Как показано на рис. 6.29, кооперация обозначается эллипсом с пунктирными границами, который имеет *имя* и сопровождается прямоугольными пиктограммами, представляющими кооперирующиеся сущности (*роли*). Эти роли связываются друг с другом *соединительными звеньями* (connectors). Несмотря на то что кооперация сама по себе является классификатором, как правило, она описывает струк-

туру и поведение другого классификатора, например прецедента использования, класса, фрагмента взаимодействия или деятельности. Имя этого классификатора можно указать после имени кооперации (рис. 6.29).

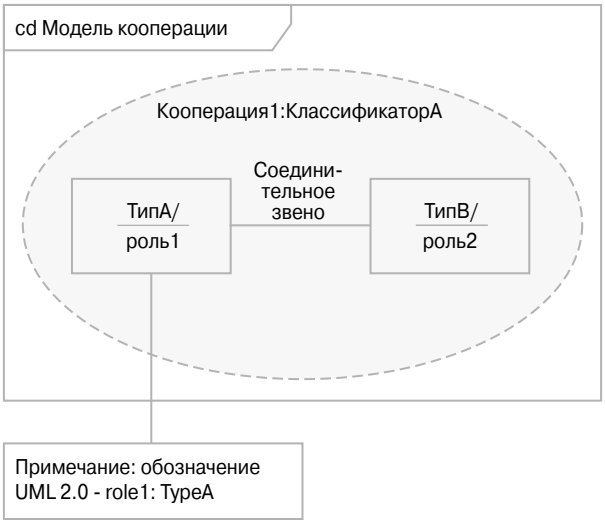


Рис. 6.29. Обозначение кооперации

Роль может иметь *тип*, с которым связан ее экземпляр. Тип — это классификатор (как правило, класс). Указывать тип роли не обязательно. Роли имеют смысл только в рамках кооперации, причем в рамках разных кооперацией один и тот же объект/экземпляр может играть разные роли.

Аналогично, *соединительное звено* — отношение между двумя ролями — имеет смысл только в рамках кооперации. "Соединительное звено может быть экземпляром ассоциации или представлять собой возможность взаимодействия между экземплярами, поскольку их имена известны (передаются как параметры либо хранятся в переменных или слотах) или в кооперации участвует один и тот же экземпляр. Связь может быть реализована с помощью простых средств, например указателя, или сложных инструментов, таких как сетевые соединения. В противоположность ассоциациям, определяющим связи между любыми экземплярами ассоциированных классификаторов, соединительные звенья определяют связи между экземплярами, играющими исключительно роль участников соединения" (UML, 2005).

На рис. 6.30 показана кооперация `Enrolment` (Зачисление), представляющая собой решение примера 6.11. Эта кооперация состоит из трех ролей — `/студент`, `/целевой курс` и `/посещаемый класс`. На диаграмме показаны типы двух последних ролей. Каждая роль соединена с двумя остальными.

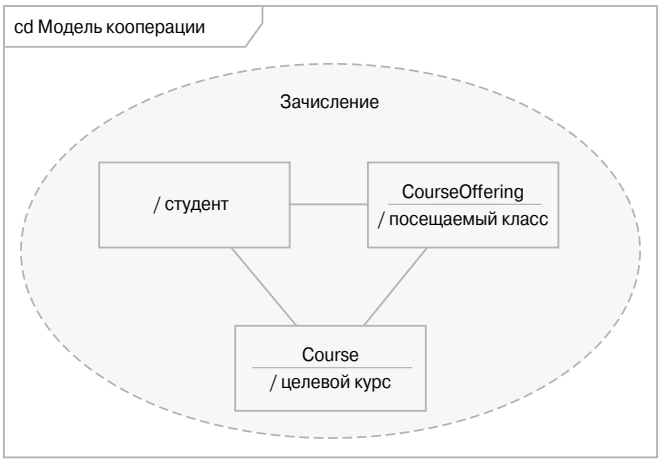


Рис. 6.30. Обозначение кооперации для системы, управляющей зачислением в университет

Пример 6.11. Зачисление в университет

Рассмотрите пример 6.9 из раздела 6.4.1.3, описывающий процедуру зачисления студента на курс путем добавления его имени в список слушателей предлагаемого курса. Рассмотрим бизнес-объекты (сущности), идентифицированные в примере 6.9. Создайте модель кооперации для зачисления студентов на курс, указав роли, которые играют бизнес-объекты, и соединительные звенья между этими ролями.

6.5.2. Композитная структура

Язык UML 2.0 предлагает альтернативные обозначения для моделирования кооперации в ситуациях, в которых типы (классификаторы) ролей заданы явно. Эти обозначения показаны на рис. 6.31. Несмотря на то что эти обозначения являются лишь альтернативой, явное представление классов позволяет создать новый вид диаграммы классов, которая в языке UML 2.0 называется *диаграммой композитной структуры* (composite structure diagram).

Преобразовать кооперацию в композитную структуру при решении примера 6.12 можно просто, выполнив перенос всех ролей из эллипса, символизирующего кооперацию, в классы (тем самым определив все недостающие типы ролей), соединив кооперацию с классами и определив имена ролей для всех соединительных звеньев. Поскольку композитные структуры часто моделируются наряду с диаграммами взаимодействия, свойствам (в частности, методам) в классах/интерфейсах можно приписать имена. Результат преобразования диаграммы, представленной на рис. 6.30, показан на рис. 6.32.



Рис. 6.31. Обозначения композитной структуры

Пример 6.12. Зачисление в университет

Рассмотрите пример 6.11 из раздела 6.5.1, описывающий процедуру зачисления студента на курс. Постройте соответствующую диаграмму композитной структуры.

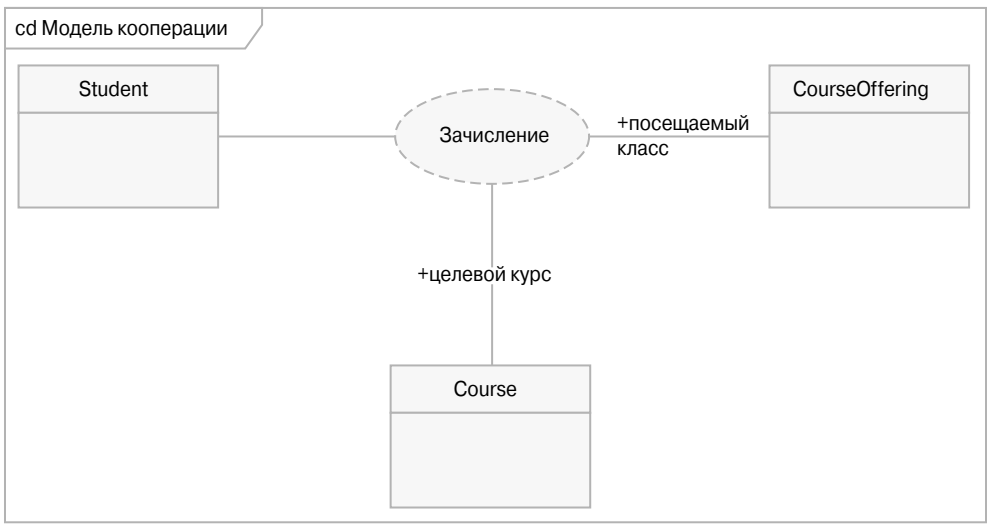


Рис. 6.32. Композитная структура для системы, управляющей зачислением в университет

Композитная структура особенно полезна для представления кооперации в виде повторно используемого *шаблона* (см. разделы 6.2.2 и 6.4.2.3). Иначе говоря, с помощью композитных структур можно документировать шаблоны. В этом случае кооперация становится именем шаблона. Структуру шаблона определяют классы/интерфейсы вместе со своими свойствами. Любые дополнительные правила, описывающие поведение шаблона, можно представить в виде ограничений (возможно, в примечаниях).

6.5.3. Переход от прецедента использования к композитной кооперации

Кооперации можно определять для классификаторов, находящихся на разных уровнях абстракции. Следовательно, с их помощью можно представлять *и* модели анализа, *и* модели проектирования. Кроме того, можно моделировать кооперации в виде композиции других кооперацией. *Композитная кооперация* (composite collaboration) состоит из *субординированных кооперацией* (subordinate collaborations). Каждая субординированная кооперация в свою очередь может быть композитной. Визуальное представление композитной кооперации показано на рис. 6.33.

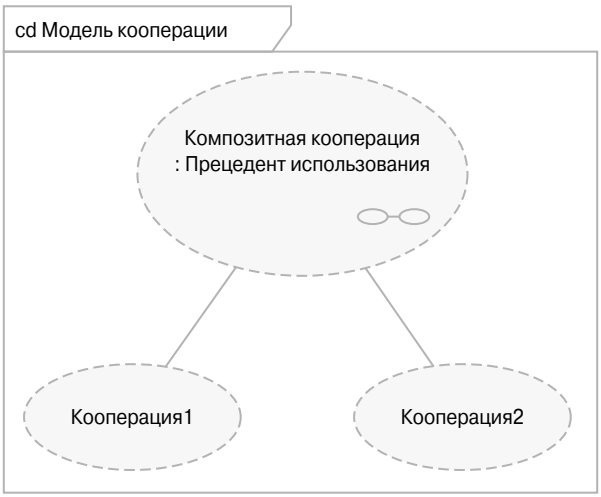


Рис. 6.33. Обозначение композитной кооперации

В одном из своих наиболее полезных применений композитная кооперация представляет прецедент использования, а субординированные кооперации выражают требования прецедента использования. Как было показано в разделе 3.1 главы 3 и в разделе 4.3.1 главы 4, прецедент использования проявляет свою силу в текстуальных описаниях потоков событий, а не в графических представлениях в виде диаграмм. Не отрицая этого факта, модели композитной кооперации могут демонстрировать вложенную структуру подпотоков и требований, перечисленных в прецеденте использования. Каждая субординированная кооперация может быть впоследствии уточнена и представлена в виде новой композитной кооперации.

Описание прецедента использования допускает много форматов. Один из них был показан в разделе 3.1.4 главы 3. Независимо от формата представления, прецедент использования необходимо уточнять настолько, насколько это потребуется, чтобы ответить на большинство (если не на все) вопросов программиста. Для того чтобы облегчить эту работу, описание прецедента использования иногда дополняется набросками графического пользовательского интерфейса.

Пример 6.13. Затраты на рекламу

Вернитесь к упражнениям, сформулированным в конце глав 2 и 5, в которых описывается система оценки затрат на рекламу. В частности, контекст этого примера поможет помочь рис. 5.31, на котором показано окно `Maintain Category-Product Links` (Поддержка связи категория–товар) (см. упражнение "Затраты на рекламу" в конце главы 5).

Система оценки затрат на рекламу позволяет измерить расходы на рекламирование разных товаров в разных средствах массовой информации. Для того чтобы правильно оценить эти затраты, система AE должна поддерживать согласованный список товаров, классифицировать их по категориям и распознавать их товарные марки.

В табл. 6.1 описан прецедент использования для учета товаров в системе оценки затрат на рекламу. Поскольку обычно операции с записями сводятся к созданию (create), чтению (read), обновлению (update) и удалению (delete), этот прецедент использования называется *Операции CRUD*. Как и многие прецеденты использования, этот документ сопровождается набросками графического прецедента использования.

Разработайте композитную кооперацию для прецедента использования *Операции CRUD*. Достаточно показать только один уровень вложения кооперации.

Таблица 6.1. Описание прецедента использования *Операции CRUD* в системе оценки затрат на рекламу

Прецедент использования	Операции CRUD
Краткое описание	Этот прецедент использования позволяет системе оценки затрат на рекламу осуществлять учет товаров. Он включает в себя возможности для просмотра (чтения) списка товаров, создания записи о новом товаре, удаления записи о товаре и обновления информации о товаре

Действующие лица	`Сотрудник, собирающий информацию`
	`Сотрудник, проводящий верификацию`
	`Сотрудник, проводящий валоризацию`
	`Сотрудник, создающий отчеты`

Предусловия	Действующее лицо имеет право вести учет товаров. Просматривать список товаров может любой `Сотрудник`. Создавать, модифицировать и удалять записи о товарах могут лишь `Сотрудник, собирающий информацию`, `Сотрудник, проводящий верификацию`, `Сотрудник, проводящий валоризацию`, и `Сотрудник, создающий отчеты`

Продолжение табл. 6.1

Прецедент использования	Операции CRUD
Основной поток	*1. Основной поток* Прецедент использования начинается, когда Сотрудник выбирает вид работы с окном Products (Товары), выбрав опцию Maintain Product (Учет товаров). Система извлекает информацию обо всех товарах и выводит ее в окне просмотра. Подпоток Чтение записей выполнен. Сотрудник, собирающий информацию, или Сотрудник, проводящий верификацию, может выбрать операцию чтения, обновления или удаления записи о товаре. На экран выводится диалоговое окно, содержащее группу полей редактирования. Поля, не допускающие редактирование, закрашиваются серым цветом, и курсор в них не отображается. Большинство полей имеют имена (приглашения). Поля содержат следующую информацию: product_id (идентификатор товара), product_name (название товара), category_name (название категории), brand_name (название товарной марки), product_status (статус товара), created_by (сотрудник, создавший запись), last_modified_by (сотрудник, последним обновивший запись), created_on (дата создания записи), last_modified_on (дата последней модификации записи), notes (примечания). Диалоговое окно не содержит меню — события генерируются щелчками на кнопках OK и Cancel (Отмена). Щелчок на кнопке OK означает запись чисел, указанных в поле, в базу данных и закрытие окна. Щелчок на кнопке Cancel означает игнорирование всех изменений, отмену всех операций и закрытие окна. Диалоговое окно имеет три режима: Insert product (Вставка записи), Update Product (Изменение записи) и Delete product (Удаление записи). Основным способом открытия диалогового окна является выбор пункта Record (Запись) в строке главного меню. Соответствующие кнопки на линейке меню обеспечивают ускоренное открытие окна. Диалоговое окно является модальным, т.е. перед выходом из окна пользователь должен завершить операции и закрыть его. Навигация между полями осуществляется с помощью клавиш Tab (Следующее поле) и Shift+Tab (Предыдущее поле). Нажатие клавиши Enter означает щелчок на кнопке OK, заданной по умолчанию. Если Сотрудник, собирающий информацию, или Сотрудник, проводящий верификацию, решает создать новую запись, выполняется подпоток Создание записи. Если Сотрудник, собирающий информацию, или Сотрудник, проводящий верификацию, решает изменить запись, выполняется подпоток Изменение записи.

Продолжение табл. 6.1

Прецедент использования	Операции CRUD
	Если Сотрудник, собирающий информацию, или Сотрудник, проводящий верификацию, решает удалить новую запись, выполняется подпоток Удаление записи. Если Сотрудник, контролирующий качество, решает выйти, прецедент использования заканчивается. *2. Подпотоки* *Прочитать запись* В окне просмотра система выводит следующую информацию: product_id (идентификатор товара), product_name (название товара), category_name (название категории), brand_name (название товарной марки), product_status (статус товара), created_by (сотрудник, создавший запись), last_modified_by (сотрудник, последним обновивший запись), created_on (дата создания записи), last_modified_on (дата последней модификации записи). Информация выводится в виде таблицы, а окно дополняется линейками вертикальной и горизонтальной прокрутки экрана. Окно называется Products (Товары), а все столбцы имеют имена. Порядок следования столбцов на экране можно изменять (используя операцию "перетащить и опустить"). Пользователь может добавлять столбцы в окно просмотра (используя выпадающее меню, открывающееся после щелчка правой кнопкой мыши). В окно можно добавить следующие столбцы: product_id (идентификатор товара), product_name (название товара), category_name (название категории), brand_name (название товарной марки), product_status (статус товара). Пользователь может удалить из окна любой столбец, кроме столбца product_name (название товара), используя выпадающее меню, открывающееся после щелчка правой кнопкой мыши. Величины, указанные в строках, редактировать нельзя. Двойной щелчок мыши приводит к открытию окна Update product (Изменение записи). Строки можно упорядочивать только в двух столбцах: product_name (название товара) и product_id (идентификатор товара). Столбцы, допускающие сортировку строк, визуально отличаются от других. Текущий столбец, допускающий сортировку, также отличается от других. *2.2. Создать запись* Система открывает диалоговое окно Create product (Создание записи). Поля product_id (идентификатор товара), created_by (сотрудник, создавший запись), last_modified_by (сотрудник, последним обновивший запись), created_on (дата создания записи), last_modified_on (дата последней модификации записи) не допускают редактирования. Когда создается новая запись о товаре, значение поля product_id автоматически заносится в базу данных.

Окончание табл. 6.1

Прецедент использования	Операции CRUD
	Поля `product_name`, `category_name`, `brand_name`, `product_status`, `notes` допускают редактирование. Поля `product_name` и `notes` также допускают ввод данных. Поля `category_name`, `brand_name` и `product_status` допускают выбор данных из базы данных с помощью списка, открывающегося щелчком на кнопке со стрелкой, направленной вниз. Альтернативные потоки: `AF1`, `AF2`. *2.3. Обновить запись* Система выводит диалоговое окно Update product (Изменение записи) и отображает на его панели название товара. Поля `product_id`, `created_by`, `last_modified_by`, `created_on`, `last_modified_on` не допускают редактирования. Поля `product_name`, `category_name`, `brand_name`, `product_status`, `notes` допускают редактирование. Поля `product_name` и `notes` также допускают ввод данных. Поля `category_name`, `brand_name` и `product_status` допускают выбор данных из базы данных с помощью списка, открывающегося щелчком на кнопке со стрелкой, направленной вниз. Альтернативные потоки: `AF1`, `AF2`, `AF4`. *2.4. Удалить запись* Система выводит диалоговое окно Delete Product (Удаление записи) и отображает на его панели название товара. Ни одно поле не допускает редактирования. Альтернативные потоки: `AF3`, `AF4` `AF1` Система не позволяет создавать или модифицировать записи о товарах, название которых совпадает с величиной поля `product_id`, уже существующим в базе данных `AF2` Система не позволяет создавать или модифицировать записи о товарах, без указания данных в полях `category_name` и `brand_name`, уже существующих в базе данных `AF3` Система не допускает связывание товаров с рекламой, удаленной из базы данных `AF4` Система не допускает одновременного открытия диалогового окна, содержащего информацию об одном и том же товаре
Постусловия	После успешного создания или изменения записи окно просмотра подсвечивает строку с информацией о товаре. После успешного удаления записи окно просмотра обновляется и подсвечивает первую строку. После выхода из прецедента использования окно Products закрывается

Создание композитной модели кооперации для такого подробного прецедента использования представляет собой довольно простую задачу, если интерпретировать прецедент как композитную кооперацию и выделить из него подпотоки в виде субординированных кооперации. Эта ситуация продемонстрирована на рис. 6.34.



Рис. 6.34. Модель композитной кооперации в системе оценки затрат на рекламу

6.5.4. Переход от кооперации к взаимодействию

Кооперация определяет соединительные звенья, по которым роли обмениваются сообщениями, но не идентифицирует сами сообщения. Спецификация потоков сообщений в кооперации образует модель взаимодействия. Следовательно, кооперацию можно использовать как промежуточный этап на пути к созданию диаграмм последовательностей и коммуникации. Для этого роли, участвующие в кооперации, необходимо изобразить в виде *линий жизни* на диаграммах последовательностей, а соединительные звенья заменить *сообщениями* в рамках взаимодействия.

Пример 6.14. Затраты на рекламу

Вернитесь к примеру 6.13 (раздел 6.5.3) и внимательно рассмотрите окно Update product (Изменить запись), представленное на рис. 6.35. Проанализируйте подпоток *Изменить запись* в описании прецедента использования, представленном в табл. 6.1. Предположим, что сценарий для этого подпотока ограничен следующими операциями: 1) в окне просмотра (первичном окне, содержащем список товаров) открывается новое окно для редактирования записи; 2) в этом окне активизируются редактируемые поля (поля, не допускающие редактирования, игнорируются); 3) изменения касаются только поля `category_name` и 4) для сохранения изменений пользователь щелкает на кнопке OK.

Создайте модель кооперации для описанного выше сценария. Используя стереотипы ролей, идентифицируйте уровни модели PCBMER, которым они принадлежат. Будем считать, что кооперация требует использования только трех уровней — представления, управления и сущностей. Используя стрелки на соединительных звеньях, продемонстрируйте применение принципа нисходящих связей (DCP) в архитектуре PCBMER.

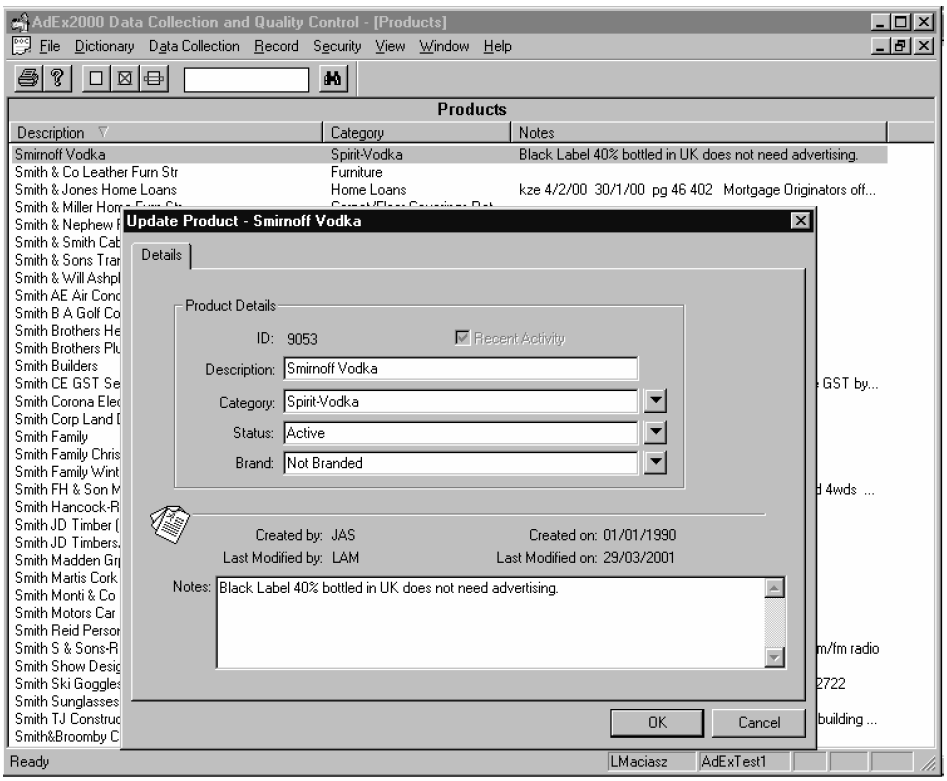


Рис. 6.35. Окно Update Product *для системы оценки затрат на рекламу. Источник: компания Nielsen Media Research, Сидней, Австралия.*

На рис. 6.36 показана модель кооперации для примера 6.14. В модели существует семь ролей: две на уровне представления, две — на уровне управления и три — на уровне сущностей. Кооперация показывает, что роль `/браузер товаров` (окно просмотра товаров) создает роль `/отдельный товар` (окно для изменения записи). Соединительное звено, направленное от роли `/отдельный товар` к роли `/получатель данных`, используется для передачи окна, предназначенного для изменения записей, бизнес-объектам, имеющим типы `Product` (Товар), `Brand` (Товарная марка) и `Category` (Категория).

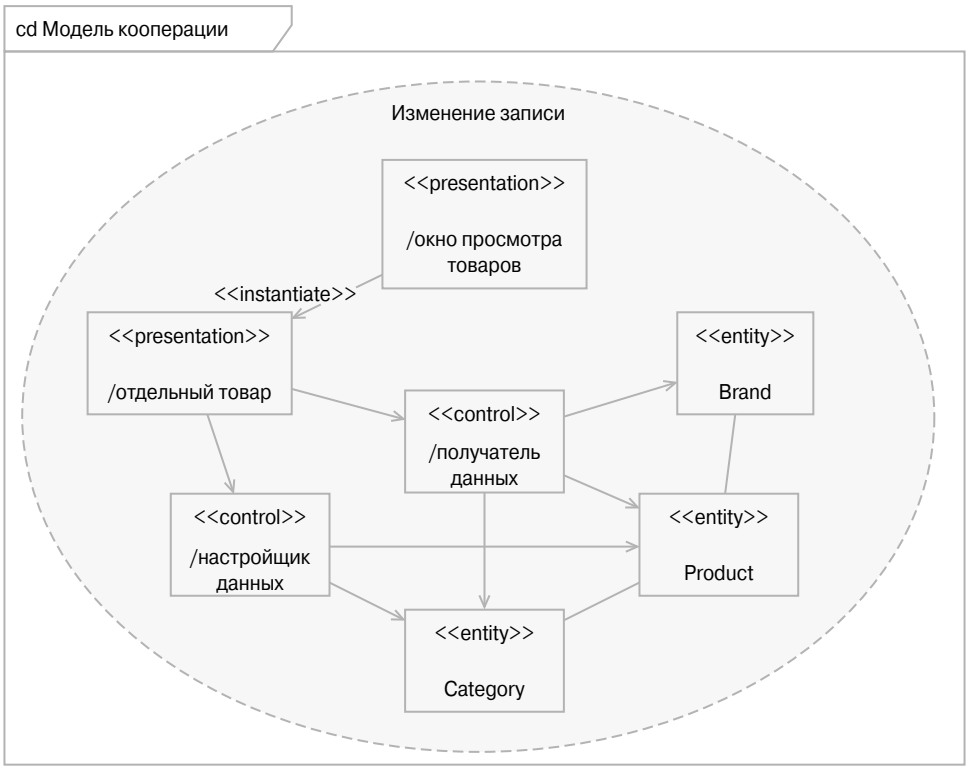


Рис. 6.36. Модель композитной кооперации в системе оценки затрат на рекламу

Соединительное звено, направленное от роли `/отдельный товар` к роли `/получатель данных`, используется для изменения бизнес-объектов типа `Product` и `Category` после модификации значения `categoryName` в соответствующем окне. Поскольку переменная `categoryName` идентифицирует новый объект типа `Category`, с которым связан объект типа `Product`, изменение переменной `categoryName` означает обновление связей между объектами, имеющими типы `Product` и `Category`, т.е. объект типа `Product` должен быть связан с новым объектом типа `Category`, и наоборот, объект типа `Category` должен иметь связь с объектом типа `Product`.

Диаграмма последовательностей для сценария модификации записей, описанного в примере 6.15, показана на рис. 6.37. Роли в модели кооперации теперь представляют собой линии жизни на диаграмме последовательностей. Эта диаграмма использует пиктограммы, изображающие объекты представления, управления и сущностей.

Пример 6.15. Затраты на рекламу

Вернитесь к примеру 6.14 и рассмотрите окно Update product (Изменить запись), представленное на рис. 6.35 (на поля Status и Notes можно не обращать внимания). Рассмотрите модель кооперации для сценария модификации записи, продемонстрированного на рис. 6.36 и описанного в примере 6.14.

Используя модель кооперации, постройте диаграмму последовательностей для этого сценария. Поясните построенную модель взаимодействия и прокомментируйте, как она изменится, если допустить использование компонентов.

Линия жизни `/окно просмотра товаров` порождает линию жизни `/отдельный товар`, т.е. обновляет пустое окно для модификации записей. Однако конструктор передает объект типа `Product` в сообщении `new()`. Линия жизни `/отдельный товар` должна получить данные для вывода на экран. Посылая три отдельных сообщения, она запрашивает у линии жизни `/лицо, получающее данные`, переменные `ProductName` (Название товара), `categoryName` (Название категории) и `BrandName` (Название торговой марки) для заполнения полей Description (Описание), Category (Категория) и Brand (Торговая марка). Линия жизни `/лицо, получающее данные`, отвечает, обращаясь за данными к объекту-сущности. Прежде чем получить переменные `categoryName` и `BrandName`, `/лицо, получающее данные`, извлекает из объекта типа `Product` ссылки на объекты типов `Category` и `Brand`.

Пересылка трех разных сообщений от линии жизни `/отдельный товар` к линии жизни `/лицо, получающее данные`, — почти стандартное решение. Для того чтобы улучшить его, необходимо ввести компонентный объект, в котором хранились бы все данные, необходимые окну модификации записей для обновления экрана. Линия жизни `/отдельный товар` могла бы создать пустой компонентный объект и передать его линии жизни `/лицо, получающее данные`, для заполнения.

Сообщение `populateCategoryList()`, посылаемое линией жизни `/лицо, получающее данные`, самому себе, активизируется, когда пользователь щелкает на пиктограмме со стрелкой, расположенной рядом с полем `Category` (см. рис. 6.35). Это позволяет пользователю выбрать другую категорию товара из выпадающего списка категорий. Линия жизни `/отдельный товар` возвращает этот список линии жизни `/лицо, получающее данные`. После этого пользователь может выбрать из списка требуемую категорию (переменную `CategoryName`), которая выводится на экран в виде сообщения `showSelected()`.

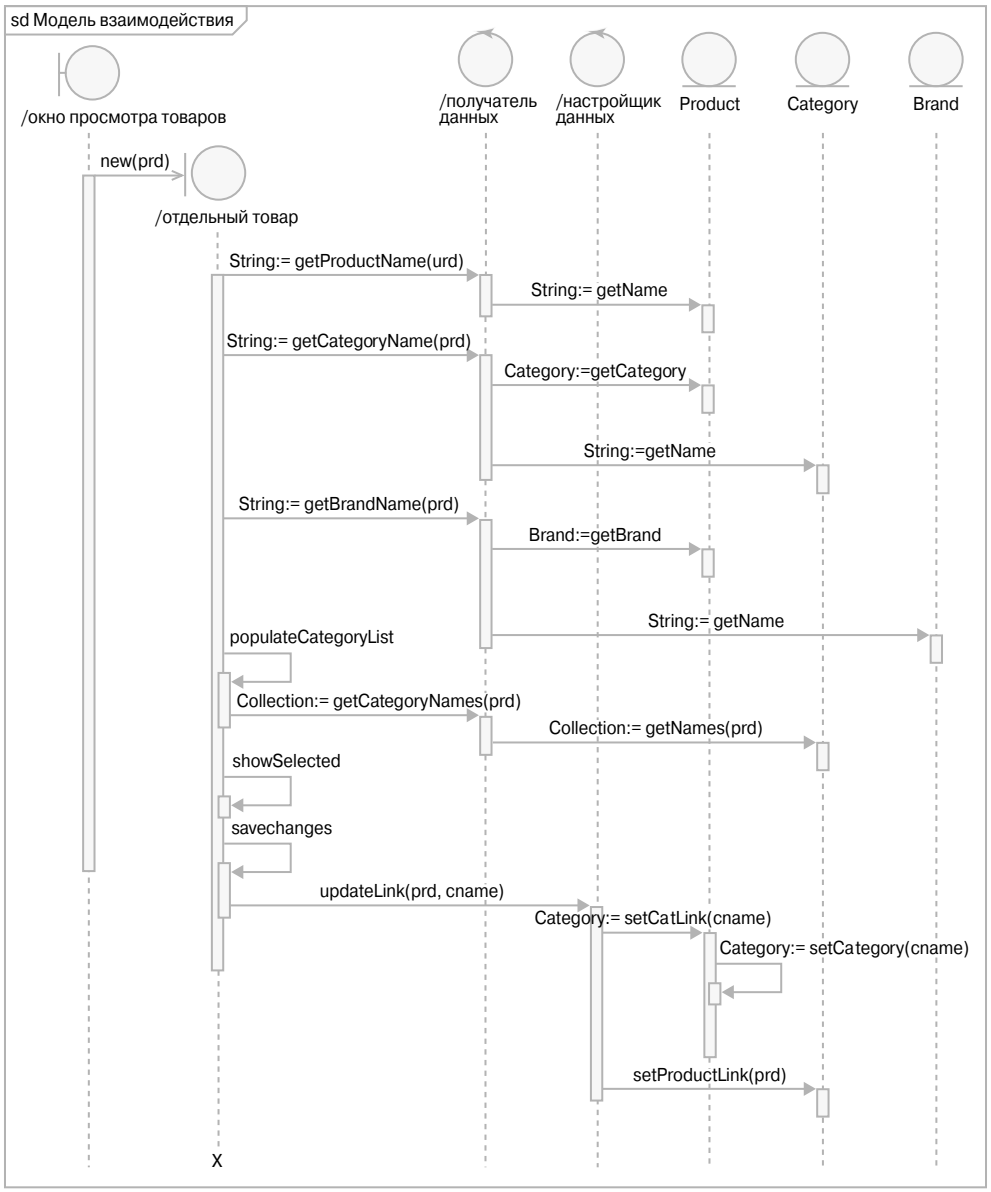


Рис. 6.37. Диаграмма последовательностей для системы оценки затрат на рекламу

Щелкнув на кнопке OK, пользователь активизирует сообщение `saveChanges()`, содержащее вызов метода `updateLink()` из линии жизни `/лицо, устанавливающее данные`. Этот вызов передает адресату объект типа `Product` и значение переменной `categoryValue`. Владея переменной `categoryName`,

объект типа `Product` может получить ссылку на объект типа `Category` и использовать ее для установления связи с классом `Category` (как именно это происходит, на диаграмме не указано). Установка связи между объектами типов `Category` и `Product` происходит довольно просто, поскольку объект типа `Product` передается объекту типа `Category` с помощью сообщения `setProductLink()`.

6.5.5. Переход от взаимодействия к композитной структуре

Кооперация имеет функциональную и структурную части. Структурная часть описывает статический аспект кооперации. Ее можно представить с помощью *диаграммы композитной структуры*, представляющей собой диаграмму классов, участвующих в кооперации. Как правило, на диаграмме композитной структуры не указываются отношения между классами. Однако эти диаграммы обычно обогащаются деталями реализации. В частности, на них могут демонстрироваться сигнатуры операций класса/интерфейса. Операции можно естественным образом получить из модели взаимодействия, разработанной для кооперации.

Пример 6.16. Затраты на рекламу

Вернитесь к примеру 6.15 и проанализируйте диаграмму последовательности, показанную на рис. 6.37. Используя диаграмму последовательностей, постройте соответствующую диаграмму композитной структуры.

На рис. 6.38 показана диаграмма композитной структуры, полученная путем соответствующего именования классов и демонстрации сигнатур их операций. Эти операции были идентифицированы по диаграмме последовательностей, показанной на рис. 6.37. Иначе говоря, эта композитная структура является уточненным вариантом кооперации, представленной на рис. 6.36, но без демонстрации отношений.

Контрольные вопросы 6.5

КВ1. В какой модели кооперации роли всегда имеют явно указанные типы?

КВ2. Можно ли идентифицировать сообщения с помощью моделей кооперации?

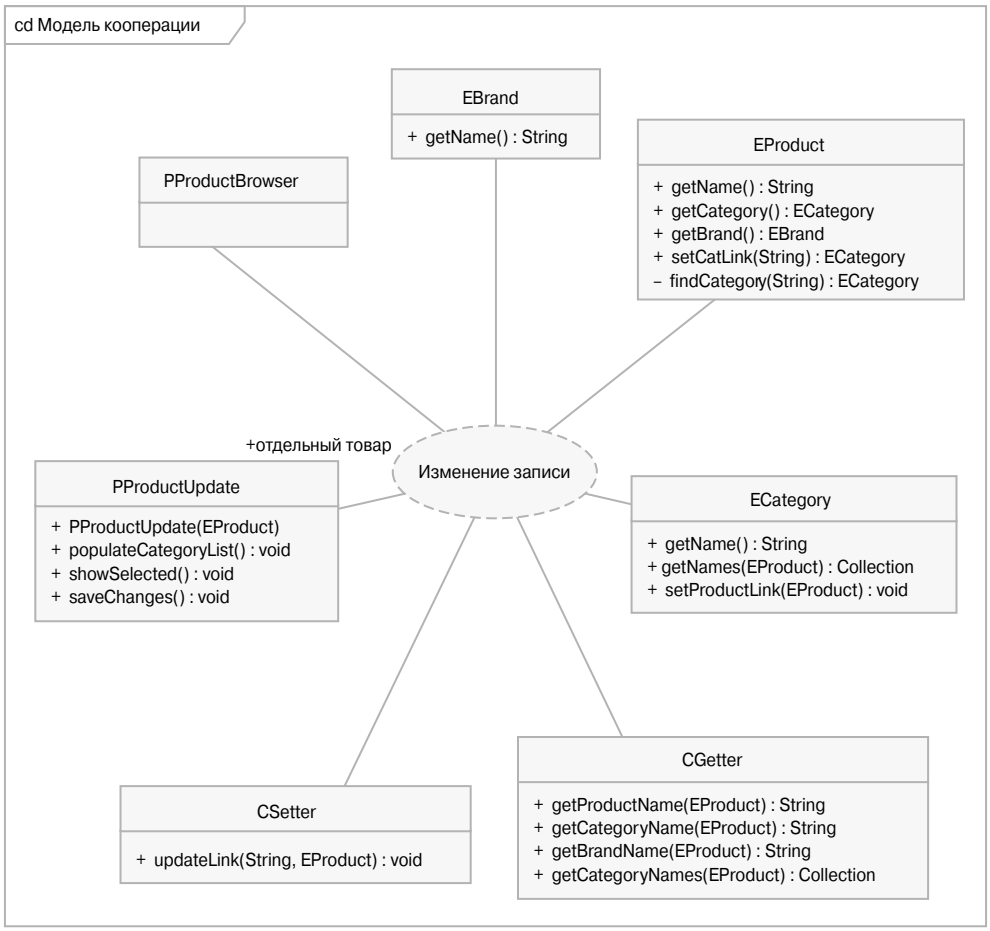


Рис. 6.38. Диаграмма композитной структуры для системы оценки затрат на рекламу

Резюме

В предыдущей главе мы перешли от анализа к проектированию. В данной главе стало ясно, что проектирование тесно связано с реализацией системы. Эта глава посвящена двум основным (и разным) аспектам проектирования — архитектурному проектированию системы и детализированному проектированию программ, образующих систему.

Типичные информационные системы основаны на архитектуре *клиент/сервер*. Несмотря на то что принцип C/S часто называют устаревшим, в реальности он является концепцией, лежащей в основе практически всех физически распределенных архитектур, в том числе *одноранговых*. *Многоярусные системы* расширяют

основы архитектуры C/S, размещая логику приложения и Web-сервисы на разных ярусах. Технологии *баз данных* также вносят важный вклад в современные архитектурные модели.

Современные системы программного обеспечения довольно сложные. Следует помнить, что решения, связанные с моделированием, должны как можно сильнее уменьшать присущую им сложность. Наиболее важным механизмом уменьшения *сложности* программного обеспечения является *иерархическое расслоение* архитектуры системы. Соответствующее распределение классов по *уровням* (пакетам, подсистемам), организованное в соответствии с моделью PCBMER или аналогичными моделями, представляет собой важную цель архитектурного проектирования. Иерархическая архитектура позволяет сократить рост *структурной сложности* программного обеспечения, снизив его с экспоненциального до полиномиального. Существенную роль в реализации архитектурных принципов при разработке программного обеспечения играют *архитектурные шаблоны*.

Архитектурное моделирование описывает распределение элементов программного обеспечения (классов, интерфейсов и т.д.) по пакетам, компонентам и узлам. Между этими концепциями существуют запутанные зависимости и отношения, в основном потому, что они пересекаются с логическими и физическими моделями программ и структур данных.

Хорошо разработанная программа максимально увеличивает *связность класса* и минимизирует *связанность классов*. Принципы связности и связанности выполняются, если проект подчиняется *закону Деметера*, определяющему допустимых адресатов для сообщений, посылаемых методами класса. Излишне широкое использование *методов доступа* может привести к появлению *бессмысленных классов*. *Связность смешанных классов* — нежелательное свойство, но иногда оно может стать необходимым, поскольку программные среды не поддерживают *динамическую классификацию*.

Повторное использование является основным фактором, влияющим на архитектурное и детализированное проектирование. Оно предоставляет разработчику выбор между *повторным использованием инструментальных средств*, *моделей* и *шаблонов*. Этот выбор может быть неоднозначным, так как допускаются и смешанные стратегии.

Детализированное проектирование делает акцент на *взаимодействиях*. *Модель кооперации* определяет кооперирующиеся элементы (*роли*) и линии связи (*соединительные звенья*) между ролями, необходимыми для обеспечения требуемого функционального свойства. Для визуализации кооперации можно использовать *диаграммы композитных структур*, которые представляют собой отображения между прецедентами использования и кооперациями, кооперациями и взаимодействиями, а также между взаимодействиями и композитными структурами.

Ключевые термины

"Банда четырех" (GoF). Шаблоны, предложенные Э. Гамма и его соавторами (Gang of Four).

P2P. Одноранговая архитектура (peer-to-peer).

Web-сервер (Web-server). Архитектурный *ярус*, ответственный за обработку управляющих событий и работу графического пользовательского интерфейса.

Архитектура (architecture). См. статью *Архитектура* в разделе "Ключевые термины" главы 1. "Организационная структура системы, включая ее декомпозицию на части, связность, механизмы взаимодействия и принципы, влияющие на проектирование системы" (Rumbough et al., 2005).

Зависимость (dependency). См. статью *Зависимость* в разделе "Ключевые термины" главы 4.

Закон Деметера (Law of Demeter). Принцип проектирования, определяющий разрешенные цели для пересылки сообщений в методах класса. Его цель — ограничить разрешенные связи между объектами.

Клиент (client). См. статью *Связность* в разделе "Ключевые термины" главы 4. Степень внутренней самоопределенности класса.

Композитная структура (composite structure). Структура, "описывающая взаимодействие объектов в контексте для формирования сущности, имеющей общую цель" (Rumbough et al., 2005).

Компонент (component). См. статью *Компонент* в разделе "Ключевые термины" главы 1 и 3. "Модульная часть проекта системы, скрывающая реализацию за набором внешних интерфейсов" (Rumbough et al., 2005).

Кооперация (collaboration). "Спецификация контекстуального отношения между экземплярами, взаимодействующими в контексте для реализации требуемой функциональности" (Rumbough et al., 2005).

Метод доступа (accessor method). Метод класса, открывающий доступ к состоянию объекта для просмотра (метод *наблюдателя*) или модификации (*модифицирующий* метод).

Модифицирующий метод (mutator). Метод класса, изменяющий состояние вызывающего объекта. Называется также механизмом установки (setter).

Наблюдатель (observer). Метод класса, имеющий доступ к состоянию объекта, но не имеющий право его изменять. Называется также механизмом получения (getter).

Пакет (package). См. статью *Пакет* в разделе "Ключевые термины" главы 3. "Универсальный механизм для организации элементов в виде групп, установки прав владения этими элементами и присваивания им уникальных имен" (Rumbough et al., 2005).

Персистентый объект (persistent object). Объект, продолжающий существовать после завершения выполнения программы, поскольку он хранился в базе данных еще до запуска программы. Любой объект, существующий в базе данных.

Повторное использование (reuse). См. статью *Возможность повторного использования* в разделе "Ключевые термины" главы 2 (Rumbough et al., 2005).

Развертывание (deployment). "Распределение артефактов программного обеспечения по физическим узлам во время выполнения" (Rumbough et al., 2005).

Связанность (coupling). См. статью *Связанность* в разделе "Ключевые термины" главы 4.

Сервер (сервер). Вычислительный процесс, обслуживающий запросы *клиента*.

Сервер приложения (application server). Архитектурный *ярус*, обрабатывающий бизнес-компоненты и следующий за соблюдением бизнес-правил.

Сложность (complexity). "Свойство программного обеспечения, описывающее сложность понимания и управления программами из-за большого количества разных, но взаимосвязанных частей".

Триггер (trigger). Процедурный код, автоматически вызываемый и выполняемый в ответ на модификацию таблицы в базе данных.

Узел (node). См. статью *Узел* в разделе "Ключевые термины" главы 3. "Физический объект, существующий на этапе выполнения программы и представляющий собой вычислительный ресурс, а также имеющий память и некоторые возможности для обработки данных" (Rumbough et al., 2005).

Уровень (layer). Один из нескольких уровней в архитектурной иерархии. Аналогичен ярусу, но обычно представляет собой логическое понятие.

Шаблон (pattern). "Параметризованная кооперация, представляющая набор ролей для параметризованных классификаторов, отношений и функциональных свойств, которые можно применить ко многим ситуациям с помощью связывания элементов модели (как правило, классов) с ролями шаблона" (Rumbough et al., 2005).

Шаблон Абстрактная фабрика (Abstract Factory). Шаблон, определяющий "интерфейс для создания семейств, связанных или взаимозависимых объектов без указания конкретного класса" (Gamma et al., 1995).

Шаблон Наблюдатель (Observer). Шаблон, "определяющий отношение "один ко многим" между объектами так, чтобы при изменении состояния одного объекта происходило оповещение и автоматическое обновление всех зависящих от него объектов" (Gamma et al., 1995).

Шаблон Посредник (Mediator). Шаблон, "обеспечивающий слабую связанность и избавляющий объекты от необходимости явно ссылаться друг на друга и независимо изменять способы взаимодействия" (Gamma et al., 1995).

Шаблон Фасад (Façade). Шаблон, определяющий "высокоуровневый интерфейс, облегчающий использование подсистемы". Его цель — "минимизировать связи и зависимости между подсистемами" (Gamma et al., 1995).

Шаблон Цепочка обязанностей (chain of responsibility). Шаблон, "позволяющий избежать связывания отправителя запроса с его адресатом, дав возможность нескольким объектам обработать этот запрос" (Gamma et al., 1995).

Ярус (tier). Один из нескольких уровней в архитектурной иерархии. Аналогичен *уровню*, но обычно представляет собой физическое понятие.

Многовариантные тесты

МТ1. Какие из перечисленных ниже архитектурных стилей допускают понятия клиентского и серверного процессов?

а. Одноранговая архитектура.

б. Ярусная архитектура.

в. Архитектура, ориентированная на базы данных.

МТ2. Что измеряется при оценке усилий, требуемых для понимания программного обеспечения?

а. Структурная сложность.

б. Когнитивная сложность.

в. Алгоритмическая сложность.

г. Сложность задачи.

МТ3. Шаблон, "обеспечивающий слабую связанность и избавляющий объекты от необходимости явно ссылаться друг на друга и независимо изменять способы взаимодействия".

а. Наблюдатель.

б. Фасад.

в. Абстрактная фабрика.

г. Ни один из перечисленных.

МТ4. Какое их перечисленных ниже имен является синонимом шаблона Наблюдатель?

а. Слушатель.

б. Зритель.

в. Сторож.

г. Созерцатель.

MT5. Какой из перечисленных ниже объектов не может быть адресатом сообщения внутри класса в соответствии с законом Деметера?

а. Объект, на который ссылается глобальная переменная.

б. Объект, являющийся аргументом сигнатуры метода.

в. Объект, на который ссылается атрибут объекта ассоциированного класса.

г. Объект, созданный методом класса.

MT6. Что из перечисленного ниже не является артефактом, разрешенным для повторного использования?

а. Объект.

б. Класс.

в. Идея о решении.

г. Компонент.

MT7. Что является средством связи между ролями?

а. Ассоциации.

б. Соединительные звенья.

в. Связи.

г. Отношения.

Вопросы

В1. Объясните, в чем состоит различие между распределенной системой обработки и распределенной системой баз данных?

В2. Что такое трехуровневая архитектура? В чем ее преимущества и недостатки?

В3. Что подразумевается под активной базой данных?

В4. Какова структурная сложность сети, состоящей из пяти классов? Изобразите иерархию, образованную четырьмя слоями, на которых размещены девять классов. Предположим, что допускаются лишь нисходящие зависимости между слоями. Как уменьшится сложность благодаря четырехслойной иерархии?

В5. Предположим, что модель классов для банковского приложения содержит класс `InterestCalculation` (Вычисление процентного дохода). Какому уровню модели PCBMER принадлежит этот класс? Обоснуйте свой ответ.

В6. Какие преимущества дает шаблон Фасад? В каких ситуациях следует применять этот шаблон?

В7. Назовите преимущества и недостатки шаблона Посредник.

В8. Объясните, как можно использовать шаблон Цепочка ответственности при реализации справочной системы для типичного каскадного выпадающего меню (так, чтобы можно было получить справочную информацию для каждого пункта меню)?

В9. Каким образом компоненты и пакеты соотносятся между собой?

В10. Какое влияние оказывают на разработку принципы связности и связанности классов?

В11. Что такое "бессмысленный класс"?

В12. Объясните корреляцию между динамической классификацией и связностью смешанного экземпляра.

В13. Как можно классифицировать типы сообщения/метода (отдельно от конструкторов и деструкторов)?

В14. Отправитель сообщения может иметь право, а может и не иметь права посылать себя (свой идентификатор объекта) целевому объекту? Можно ли применить это утверждение к асинхронным сообщениям? Обоснуйте свой ответ.

В15. В чем заключается различие между замещением и перегрузкой?

В16. Сравните повторное использование инструментальных средств и каркасов.

В17. Какие диаграммы языка UML используются для проектирования структурного аспекта кооперации?

В18. Объектные системы реализуют кратность ассоциативных связей с помощью определенных коллекций ссылок (множеств или списков). В языке Java эта реализация обычно использует интерфейс `Collection` из библиотеки `java.util.Collection`. В языке C++ эта реализация обычно использует понятие параметризованного типа, т.е. шаблонного класса. Объясните, как можно реализовать ассоциации в моделях класса, приведенных на рис. 5.12 и 5.13 (см. раздел 5.1.5.1 главы 5) с помощью языков Java и C++?

Упражнения. Магазин видеокассет

Дополнительные требования

Рассмотрите следующие дополнительные требования к системе управления работой магазина видеокассет (они перечислены в конце главы 4 и приводятся здесь для удобства).

- За кассеты и диски, возвращенные позже срока, взимается дополнительная плата за период, превышающий срок проката. Каждый носитель обладает уникальным идентификационным номером.
- Фильмы заказываются у поставщика, который, в общем случае, может поставить кассеты и диски в течение одной недели. Обычно один заказ делается на несколько фильмов.
- Зарезервировать можно те фильмы, которые заказаны у поставщика и/или все копии которых находятся в прокате. Можно также зарезервировать те фильмы, которых нет в запасе и которые не заказаны у поставщика; при этом с клиента требуется задаток за один период проката.
- Клиент может также сделать несколько предварительных заказов, однако для каждого зарезервированного фильма готовится отдельный запрос на резервирование. Резервирование может быть отменено из-за отсутствия реакции со стороны клиента, более точно, в течение одной недели с того момента, когда клиенту было сообщено о возможности взять фильм напрокат. Если за фильм был внесен задаток, он записывается на счет клиента.
- В базе данных хранится обычная информация о поставщиках и клиентах, т.е. адреса, телефонные номера и т.д. В каждом заказе поставщику указываются заказываемые фильмы, их количество, форматы кассеты/диска, а также дата ожидаемой доставки, отпускная цена, возможные скидки и т.д.
- Когда кассета возвращается клиентом или поступает от поставщика, вначале удовлетворяются предварительные заказы. Работники магазина устанавливают контакт с клиентами, сделавшими предварительный заказ. Для правильной обработки резервирования фильмов информация, связанная с резервированием, обновляется дважды: после установления контакта с клиентом, когда ему сообщают, что "зарезервированный фильм пришел", и после сдачи фильма клиенту напрокат. Эти шаги гарантируют правильное проведение операции резервирования.

- Клиент может взять несколько кассет или дисков, однако каждому видеоносителю, взятому напрокат, соответствует отдельная запись. Для каждого выдаваемого напрокат фильма фиксируются дата и время выдачи, установленный и фактический срок возврата. Позже запись о прокате обновляется, чтобы отразить факт возврата видеофильма и факт окончательного платежа (или возврата денег). Кроме того, в записи хранится информация о продавце, отвечающем за прокат фильма. Детальная информация о клиенте и о прокате хранится в течение года, чтобы можно было легко определить уровень доверия к клиенту. Старая информация о прокате сохраняется в течение года в целях проведения аудита.
- Все операции выполняются наличными, по электронному переводу денег или по кредитным карточкам. От клиентов требуется внести плату за прокат при выдаче кассет/дисков.
- Если кассета или диск возвращены позже установленного срока (или не могут быть возвращены по каким-либо причинам), плата взимается либо со счета клиента, либо принимается непосредственно от клиента.
- Если кассета или диск задержаны более чем на два дня, клиенту отправляется уведомление о задержке. После отправки двух уведомлений о задержке одной и той же кассеты или диска, клиента предупреждают о том, что он является "нарушителем", и при следующем его обращении в магазин руководство рассматривает вопрос о снятии с него статуса "нарушителя".

MB1. Обратитесь к приведенным выше дополнительным требованиям и к примерам из глав 3 и 4, касающимся этой системы, в частности к модели прецедентов использования, показанной на рис. 4.15 (см. раздел 4.3.1.2 главы 4).

MB1.1. Постройте диаграмму композитного взаимодействия для прецедента использования "Резервирование видеокассет".

MB1.2. Постройте диаграмму взаимодействия для прецедента использования "Резервирование видеокассет". Рассмотрите только бизнес-объекты (классы сущностей).

MB1.3. Постройте диаграмму взаимодействия для прецедента использования "Резервирование видеокассет". Предположим, что бизнес-объекты были заранее загружены в подсистему сущностей, а взаимодействие связано лишь уровнями представления, управления и сущностей. По мере необходимости используйте архитектурные шаблоны.

MB2. Обратитесь к приведенным выше дополнительным требованиям и к примерам из глав 3 и 4, касающимся этой системы, в частности к модели прецедентов использования, показанной на рис. 4.15 (см. раздел 4.3.1.2 главы 4).

MB2.1. Постройте диаграмму композитного взаимодействия для прецедента использования "Возврат видеокассет".

MB2.2. Постройте диаграмму взаимодействия для прецедента использования "Возврат видеокассет". Рассмотрите только бизнес-объекты (классы сущностей).

MB2.3. Постройте диаграмму взаимодействия для прецедента использования "Возврат видеокассет". Предположим, что бизнес-объекты были заранее загружены в подсистему сущностей, а взаимодействие связано лишь уровнями представления, управления и сущностей. По мере необходимости используйте архитектурные шаблоны.

MB3. Обратитесь к приведенным выше дополнительным требованиям и к примерам из глав 3 и 4, касающимся этой системы, в частности к модели прецедентов использования, показанной на рис. 4.15 (см. раздел 4.3.1.2 главы 4).

MB3.1. Постройте диаграмму композитного взаимодействия для прецедента использования "Заказ видеокассет".

MB3.2. Постройте диаграмму взаимодействия для прецедента использования "Заказ видеокассет". Рассмотрите только бизнес-объекты (классы сущностей).

MB3.3. Постройте диаграмму взаимодействия для прецедента использования "Заказ видеокассет". Предположим, что бизнес-объекты были заранее загружены в подсистему сущностей, а взаимодействие связано лишь уровнями представления, управления и сущностей. По мере необходимости используйте архитектурные шаблоны.

MB4. Обратитесь к приведенным выше дополнительным требованиям и к примерам из глав 3 и 4, касающимся этой системы, в частности к модели прецедентов использования, показанной на рис. 4.15 (см. раздел 4.3.1.2 главы 4).

MB4.1. Постройте диаграмму композитного взаимодействия для прецедента использования "Поддержка клиента".

MB4.2. Постройте диаграмму взаимодействия для прецедента использования "Поддержка клиента". Рассмотрите только бизнес-объекты (классы сущностей).

MB1.3. Постройте диаграмму взаимодействия для прецедента использования "Поддержка клиента". Предположим, что бизнес-объекты были заранее загружены в подсистему сущностей, а взаимодействие связано лишь уровнями представления, управления и сущностей. По мере необходимости используйте архитектурные шаблоны.

Упражнения. Затраты на рекламу

Дополнительная информация

Рассмотрите дополнительные требования к системе оценки затрат на рекламу. Для дальнейшего анализа обратите внимание на рис. 6.39, на котором схематически изображены требования к системе и показан их визуальный контекст.

- Система оценки затрат на рекламу поддерживает ассоциации между рекламой, рекламируемым товаром, рекламным агентством, которое оплачивает рекламу, и агентством, регистрирующим затраты. Эти ассоциации называют *рекламными связями* (ad links). Средства массовой информации, в которых была размещена реклама, можно определить по описаниям экземпляров рекламного объявления.
- Система оценки затрат на рекламу позволяет изменять деловые отношения, касающиеся *рекламных связей*, т.е. сменить рекламодателя, агентство и рекламируемый товар. Это можно сделать, просто внеся изменения в записи (например, для того, чтобы исправить ошибку). Кроме того, это можно сделать, изменив условия ведения бизнеса (например, если учет рекламы поручается другому агентству). В последнем случае сохраняются исторические деловые отношения (исторические *рекламные связи*).
- Прецедент использования "Обновление рекламной связи путем исправления" позволяет пользователю модифицировать рекламную связь, изменив рекламодателя, агентство или товар ("обновление рекламной связи" или "модификация рекламной связи", по сути, являются одной и той же функцией). Прецедент использования начинается, когда пользователь выбирает рекламную связь в окне просмотра Ad link (Рекламные связи), щелкает правой кнопкой мыши, чтобы открыть выпадающее меню, и активизирует опцию Modify ad link (Модификация рекламной связи).
- Диалоговое окно Modify ad link имеет двойное предназначение: обновлять рекламные связи путем коррекции записей и путем изменения условия ведения бизнеса. В каждый момент времени можно активизировать только одну из этих опций, щелкнув на одной из кнопок, расположенных в диалоговом окне.
- Окно Modufy ad link отображает информацию о рекламной связи (скопированную из выбранной рекламной связи в окно просмотра) и позволяет пользователю изменять рекламное агентство и/или рекламодателя и/или товар. При модификации рекламной связи можно использовать только агентства, рекламодатели и товар, хранящиеся в базе данных.

Любые изменения агентства, рекламодателя или товара немедленно отображаются в окне Modify ad link, но изменения записей в базе данных вступят в силу только после того, как пользователь щелкнет на кнопке Save (Сохранить). До этого момента любые изменения можно отменить.

- В большинстве случаев модификация рекламной связи с помощью исправления создает новую ассоциацию между рекламодателем, агентством и товаром, в которой связь "рекламодатель–агентство" определяется заранее (в существующей рекламной связи). Создавать новые отношения между рекламодателем и агентством могут только пользователи системы, которые имеют право выполнять это задание (за исключением ситуации, когда новая связь "рекламодатель–агентство" для рекламодателя является первой, т.е. появился новый рекламодатель, до сих пор не имевший связей с агентством).
- Окно Modify ad link (рис. 6.39) показано на фоне окна просмотра рекламных связей. Анализ обоих окон показывает, что реклама несколько раз была связана с одним и тем же рекламодателем/агентством/товаром. Для этого существует несколько причин, которые выходят за рамки данного упражнения (кроме необходимости сохранять исторические рекламные связи в базе данных). Следовательно, в поставленной задаче не требуется обеспечивать возможность иметь несколько связей между рекламой и одним и тем же рекламодателем/агентством/товаром.
- После открытия окна Modify ad links все командные кнопки (за исключением кнопки Cancel (Отмена)) и поля для ввода становятся недоступными, пока пользователь не установит на панели Change Mode (Режим изменения) радиокнопки Correction (Изменение) или Business Change (Изменения условий бизнеса). Панель с рекламными связями содержит строку, в которой описываются детали изменяемой рекламной связи. После выбора варианта изменений пользователь может щелкнуть на кнопке Select Agency (Выбрать агентство), Select Advertiser (Выбрать рекламодателя) или Select Product (Выбрать товар). Щелчок на каждой из этих кнопок открывает список выбора агентства, рекламодатели или товара.
- После щелчка на кнопке Save (Сохранить) происходит модификация рекламной связи и установка связи с новым агентством и/или рекламодателем, и/или товаром. Затем диалоговое окно закрывается, а в исходном окне подсвечивается рекламная связь и отображается исправленная информация. Кнопка Cancel (Отмена) позволяет закрыть диалоговое окно без модификации рекламной связи.

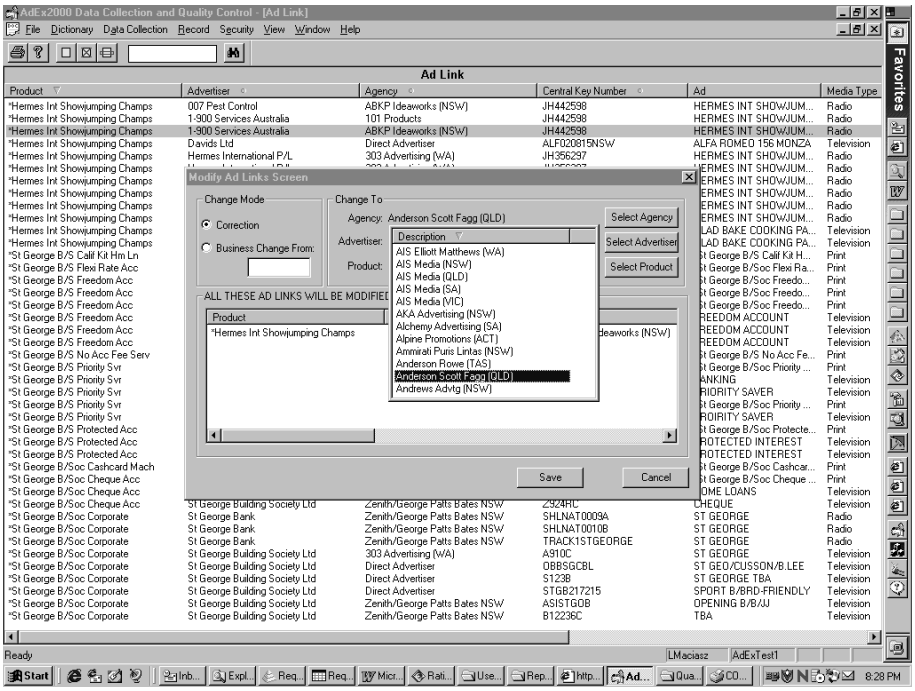


Рис. 6.39. Окно просмотра строк Ad link и окна Modify Ad Links. Источник: компания Nielsen Media Research, Сидней, Австралия

ЗР1. Разработайте композитную модель кооперации для части системы, описанной выше. Поясните ее.

ЗР2. Идентифицируйте бизнес-объекты (сущности), вовлеченные в прецедент использования "Обновление рекламной связи путем исправления". Разработайте модель кооперации для этого прецедента использования, указав роли, которые исполняют бизнес-объекты, и соединительные звенья между этими ролями. Поясните свой ответ.

ЗР3. Идентифицируйте объекты PCBE (presentation (представление), control (управление), bean (компонент), entity (сущность)), участвующие в прецеденте использования "Обновление рекламной связи путем исправления", считая, что бизнес-объекты на уровне сущностей являются точным образом персистентного состояния базы и что нет необходимости использовать уровни посредника и ресурсов. Разработайте модель кооперации для этого прецедента использования, указав роли, которые исполняют объекты PCBE, и соединительные звенья между этими ролями. Используя стереотипы ролей, идентифицируйте уровни PCBE, которым принадлежат эти роли. Используя стрелочки зависимостей на соединительных

звеньях, продемонстрируйте применение принципов нисходящих связей (DCP) в архитектуре PCBMER и покажите создание объектов. Объясните созданную модель кооперации.

ЗР4. На основе модели кооперации из упражнения ЗР3 (рис. 6.42) постройте диаграмму последовательностей для описанного выше сценария. Для простоты будем считать, что для рекламной связи можно выбирать только других рекламодателей, но не агентства или товары. Объясните свою модель взаимодействия.

Ответы на контрольные вопросы

Контрольные вопросы 6.1

КВ1. Одноранговая архитектура. Единственным типом элемента системы является равноправный элемент.

КВ2. Бизнес-логика и бизнес-правила предприятия.

КВ3. С помощью триггеров.

Контрольные вопросы 6.2

КВ1. Машина Тьюринга, основанная на алгоритмах и открытой модели взаимодействия.

КВ2. Классы

КВ3. Шаблон Фасад.

КВ4. Шаблон Цепочка обязанностей.

Контрольные вопросы 6.3

КВ1. Обобщение и отношения зависимости.

КВ2. Нет. Компонент нельзя отличить от копий, которыми он владеет, и в любом приложении существует по крайней мере одна копия компонента.

КВ3. Да, может.

Контрольные вопросы 6.4

КВ1. Связность класса.

КВ2. Отсутствие поддержки динамической классификации.

КВ3. Повторное использование модели.

Контрольные вопросы 6.5

КВ1. Диаграмма композитной структуры.

КВ2. Нет. Для этого используются модели взаимодействия.

Ответы к многовариантным тестам

МТ1. г

МТ2. б

МТ3. г (шаблон Посредник)

МТ4. а

МТ5. в

МТ6. а

МТ7. б

Ответы на вопросы с нечетными номерами

В1

Система управления распределенными базами данных является супермножеством *распределенной системы обработки данных*. В распределенной системе обработки данных клиентская программа может быть связана со многими базами данных, но каждый оператор доступа/модификации в этой системе может быть направлен только одной из этих баз данных.

Система управления распределенными базами данных ослабляет указанное выше ограничение. Это значит, что клиентская программа может содержать операторы доступа, в которых сочетаются данные из разных баз данных, и операторы модификации, изменяющие данные во многих базах данных (Date, 2000). В идеальном случае операторы доступа/модификации должны действовать на данные "прозрачно", т.е. пользователь не должен знать, где именно находятся данные. Базы данных могут быть неоднородными, т.е. управляться разными системами.

Системы управления распределенными базами данных трудны для реализации. Эти трудности возникают в основном из-за жестких требований к управлению транзакциями и обработке запросов.

B3

Активная база данных может хранить не только данные, но и программы. Такие хранимые программы могут быть вызваны по именам или запускаться операторами, пытающимися изменить базу данных.

Современные системы управления коммерческими базами данных являются активными. Они обеспечивают программное расширение языка SQL, например PL/SQL в системе Oracle или Transact SQL в системах Sybase или Microsoft SQL. Это позволяет писать программы и хранить их в словаре базы данных. Эти программы можно вызывать из прикладных программы или запускать при модификации базы данных.

Программы в активных базах данных являются "полными с вычислительной точки зрения", но не "полными с точки зрения ресурсов". Иначе говоря, они могут выполнить любые вычисления, но не имеют доступа к внешним ресурсам, т.е. к окну графического пользовательского интерфейса или Интернет-браузеру, не могут посылать электронные письма или SMS, а также осуществлять мониторинг технических устройств.

B5

Такой класс естественным образом принадлежит уровню *управления*. Этот класс реализует реакцию на пользовательские события (что является основной целью уровня управления). Класс `InterestCalculation` обеспечивает уровень взаимодействия между пользовательскими событиями (уровень представления) и содержимым базы данных (представленным на этапе выполнения классами уровня сущностей). Изменение способа вычислений (программной логики) локализуется в классе `InterestCalculation` и не влияет на классы представления или сущностей.

B7

Персистентные объекты сущностей, как правило, имеют сложные ассоциации и изощренную бизнес-логику. Деловые операции с этими множествами объектов сущностей очень сложно программировать. Шаблон Посредник обеспечивает упрощенное решение этой дилеммы. Его основное преимущество заключается в том, что он берет на себя обязательство обрабатывать логику приложения, связанную с коммуникациями между объектами сущностей, для выполнения деловых транзакций. Вторичным преимуществом является то, что любые изменения деловых транзакций и стратегии коммуникации между объектами сущностей локализуются в посреднике. Централизация прикладной специфики в посреднике повышает адаптивность системы благодаря разделению транзакций между бизнес-объектами.

Очевидным недостатком шаблона Посредник является значительный рост его собственной сложности при увеличении количества объектов сущностей и усложнении деловых транзакций. Это нарушает объектно-ориентированный принцип, требующий, чтобы сложность была равномерно распределена между классами системы. Для

того чтобы устранить этот недостаток, необходимо создать несколько посредников, сосредоточенных на группах деловых транзакций и/или группах классов сущностей.

B9

Как правило, компоненты и пакеты представляют собой наборы классов. Однако эти представления имеют разный уровень абстракции и часто ортогональны друг другу.

Пакет — это логическое понятие, не имеющее конкретной реализации и не представляющее собой элемент программы, допускающий повторного использования (хотя отдельные классы в пакете могут быть предназначены для повторного использования).

Компонент — это физическое понятие, как правило, предназначенное для *повторного использования* в ходе выполнения программы. Компонент "публикует" свой внешний интерфейс (свои сервисы) в форме контракта. Внутренняя реализация операций в интерфейсе компонента не публикуется. Компоненты могут быть связаны друг с другом с помощью своих интерфейсов, даже если они реализуются с помощью разных языков программирования.

Компонент — это единица компиляции. Компонент может состоять из многих классов, но только один из этих классов может быть *открытым* (т.е. иметь открытую видимость). Имя единицы компиляции компонента (например, `Invoice.java`) должно соответствовать имени его открытого класса (`Invoice`). Некоторые инструменты визуального моделирования (например, Rational Rose) автоматически создают компоненты в ходе генерирования кода на языке Java для открытых классов (по одному компоненту для одного открытого класса).

Пакет — это группа классов в архитектурной модели. Группирование классов осуществляется по принципу их статической близости в предметной области. *Компонент* предлагает набор операций в среде функционирования. Этот набор операций определяет поведенческую близость классов и может использоваться в нескольких предметных областях.

B11

Хорошо спроектированный класс выполняет услуги для других классов, а не просто показывает свои атрибуты. Кроме того, он принимает на себя обязанности по управлению своим пространством состояний и не позволяет другим классам устанавливать значения своих атрибутов. Класс, спроектированный неправильно (т.е. нарушающий указанные выше правила), является *бессмысленным*.

Класс является бессмысленным, если его открытый интерфейс состоит в основном из методов доступа, т.е. методов наблюдателя (`get`) и модифицирующих методов (`set`). На практике большинство классов содержат несколько методов доступа, для того чтобы обеспечить связанность между классами, которые в противном случае были бы связными.

B13

Сообщения можно классифицировать по-разному. Основная классификация была предложена Пейдж-Джонсом (2000).

- *Сообщения для чтения* (read messages). Описательные сообщения, ориентированные на представление.
- *Сообщения для модификации* (update message). Информативные сообщения, ориентированные на прошлое.
- *Сообщения для кооперации* (collaborative message). Императивные сообщения, ориентированные на будущее.

Отправитель *сообщения для чтения* запрашивает информацию у адресата (целевого объекта). Метод, который выполняется в результате получения сообщения для чтения, иногда называют методом *наблюдателя* (`get`).

Отправитель *сообщения для модификации* предоставляет адресату информацию, которая должна помочь ему модифицировать свое состояние. Метод, который вызывается в результате получения сообщения о модификации, иногда называют *модифицирующим методом* (`set`).

Сообщения для чтения и модификации называют также *сообщениями методов доступа* (accessor messages). Методами доступа являются связанные друг с другом методы наблюдателя и модификации.

Сообщение для кооперации посылается объекту, на который возложена обязанность выполнить крупное задание, требующее кооперации многих объектов. Сообщение о сотрудничестве является частью цепочки сообщений. Каждое сообщение для сотрудничества может быть сообщением для чтения или модификации.

Кстати, в языке UML сообщения разделяются на сигналы и вызовы (Rumbaugh et al., 2005). *Сигнал* — это односторонняя асинхронная связь, направленная от отправителя к адресату, так что получение сигнала является событием по отношению к адресату. *Вызов* — это двусторонняя синхронная связь, в которой отправитель вызывает операцию, применяемую к адресату. В этой классификации сообщения для чтения, модификации и кооперации являются вызовами.

B15

Замещение и перегрузка подразумевают, что существует несколько методов с одинаковыми именами. Разница между ними заключается в следующем.

- *Замещение* связано с наследованием и полиморфизмом. Метод в суперклассе можно заместить методом в подклассе. Не только имена, но и сигнатуры обоих методов в суперклассе и подклассе должны быть одинаковыми. (Абстрактная операция должна быть полиморфной, а ее реализация должна размещаться в подклассе. Это явление не считается замещением, поскольку абстрактная операция не имеет метода в суперклассе.)

- *Перегрузка* связана с существованием нескольких методов, имеющих одинаковые имена в одном и том же классе. Очевидно, что сигнатуры перегруженных методов должны отличаться друг от друга.

B17

Разработка функционального аспекта кооперации связана с описанием динамики вызовов методов или событий, приводящих к реализации поведения системы (как правило, к реализации прецедента использования). Кооперация моделируется с помощью *диаграмм кооперации*, в том числе *диаграмм композитной структуры*. Несмотря на то что соединительные звенья между ролями представляют собой линии связи, по которым реализуется поведение системы, диаграммы кооперации в принципе похожи на *диаграммы классов* (и диаграммы классов прежде всего отражают структуру системы, одновременно демонстрируя ее поведение).

Соответственно, для того чтобы отразить детали кооперации, не учтенные в диаграммах кооперации, при проектировании функциональных свойств необходимы дополнительные модели. Этими дополнительными моделями являются, как правило, *диаграммы взаимодействия*, т.е. *диаграммы последовательностей и диаграммы кооперации*.

Делая выбор между диаграммой последовательностей и диаграммой кооперации, необходимо учесть, что на *диаграммах последовательностей* не показываются линии статических отношений (как правило, ассоциации), используемые при передаче сообщений. В то же время диаграммы кооперации выражают семантику отношений с помощью соединительных звеньев. Соответственно, диаграммы последовательностей можно легко построить раньше диаграмм кооперации. С другой стороны, линии отношений демонстрируются на *диаграммах коммуникации*. Следовательно, их можно использовать вместо диаграмм кооперации.

Другим важным инструментом проектирования функциональных свойств являются *диаграммы конечных автоматов*. Конечные автоматы могут дополнять другие поведенческие модели, чтобы демонстрировать состояния, в которые может перейти объект, или взаимодействие, которое может быть результатом операций (событий), выполняемых над объектом или взаимодействием.

Решения упражнений. Затраты на рекламу

ЗР1

Модель композитной кооперации показана на рис. 6.40. Эта модель различает две кооперации, соответствующие двум разным прецедентам использования, которые, в свою очередь, представлены первичным и вторичным окнами, показанными на рис. 6.39. Композитная кооперация `Modify ad link (Модификация`

рекламных связей) состоит из двух альтернативных, но аналогичных кооперaций — Modify ad link as correction (Модификация рекламных связей с помощью коррекции) и Modify ad link as business change (Модификация рекламных связей с помощью изменения условий бизнеса). Эти две кооперации являются альтернативными в том смысле, что пользователь должен выбрать одну из них. Они являются аналогичными, потому что выполняют аналогичные функции, за исключением того, что изменение условий бизнеса не удаляет старую рекламную связь и записывает модифицированные данные.



Рис. 6.40. Модель композитной кооперации для системы оценки затрат на рекламу

ЗР2

На рис. 6.41 показана только модель кооперации для бизнес-объектов. Она представляет собой обобщенную и статическую модель, в которой роли прямо соответствуют персистентным структурам данных (таблицам или классов) в базе данных. В этой модели не указана кратность, но имена ролей свидетельствуют о том, что любая реклама может иметь много рекламных связей.



Рис. 6.41. Модель кооперации для бизнес-объектов в системе оценки затрат на рекламу

ЗРЗ

Модель кооперации для объектов PCBE показана на рис. 6.42. Основой этой модели являются элементы, описанные ниже.

- `PADLinkBrowser` создает экземпляр класса `Modify ad link screen` (Экран модификации рекламных связей (см. рис. 6.39)), используя данные, которые он уже имеет (иначе говоря, нам не нужно проверять правильность этих данных и заново извлекать их из базы данных, если они неверны).
- Когда пользователь выбирает любую из трех пиктограмм. открывающих списки, открывается роль `/список выбора`. Для передачи данных роль `/список выбора` вызывает роль `/получатель данных`, имеющую доступ к ролям `/агентства`, `/рекламодатели` и `/товары`. Это позволяет получателю данных создать экземпляр роли `/поставщики списка выборов` и передать ему имена ролей `/агентства`, `/рекламодатели` и `/товары` для создания соответствующих компонентов. Эти компоненты используют роль `/список выбора` для демонстрации списков выбора.

- После того как пользователь выберет желательный пункт в списке выбора, роль `/рекламная связь после коррекции` готова к сохранению. Для этого она вызывает роль `/настройщик данных`, чтобы создать экземпляр роли `/добавить новую рекламную связь к рекламе` с соответствующими связями с агентством, рекламодателем и товаром.

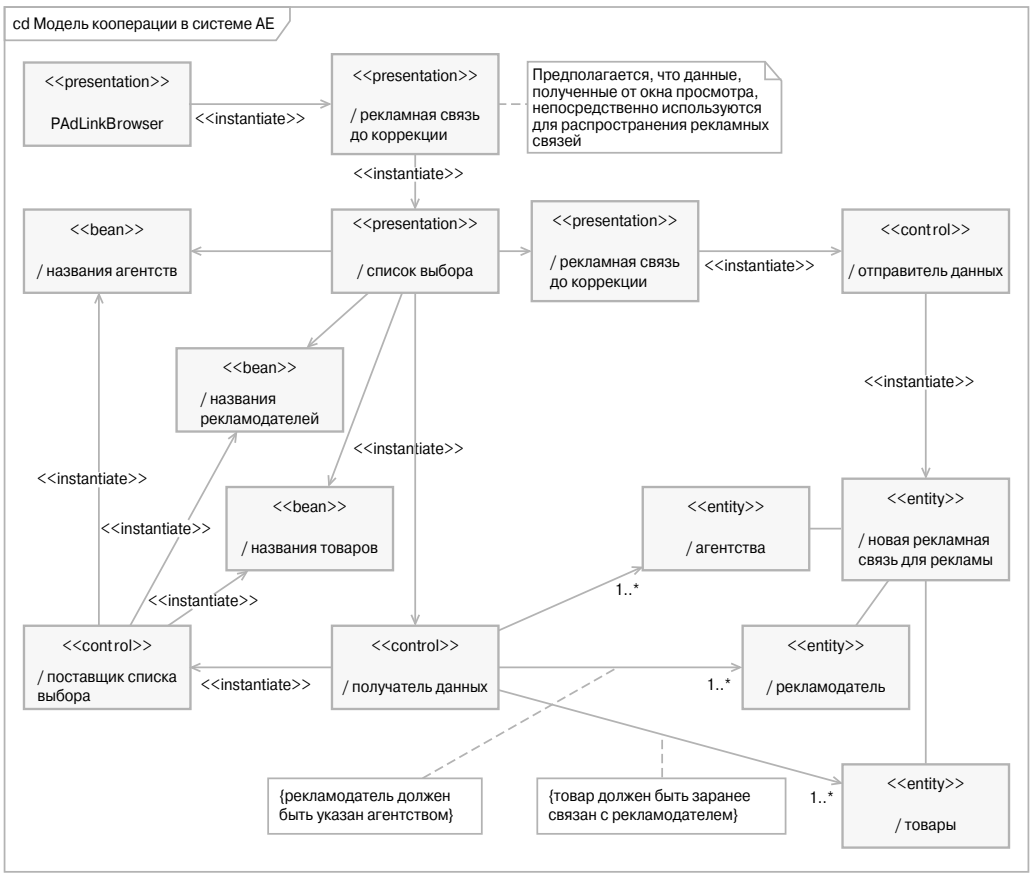


Рис. 6.42. Модель кооперации в системе оценки затрат на рекламу, учитывающая уровни PCBE

ЗР4

Диаграмма последовательностей, соответствующая модели кооперации, показанной на рис. 6.42, приведена на рис. 6.43. В сочетании с моделью кооперации эта диаграмма является самоочевидной. В принципе она лишь определяет сообщения, проходящие по соединительным звеньям между ролями.

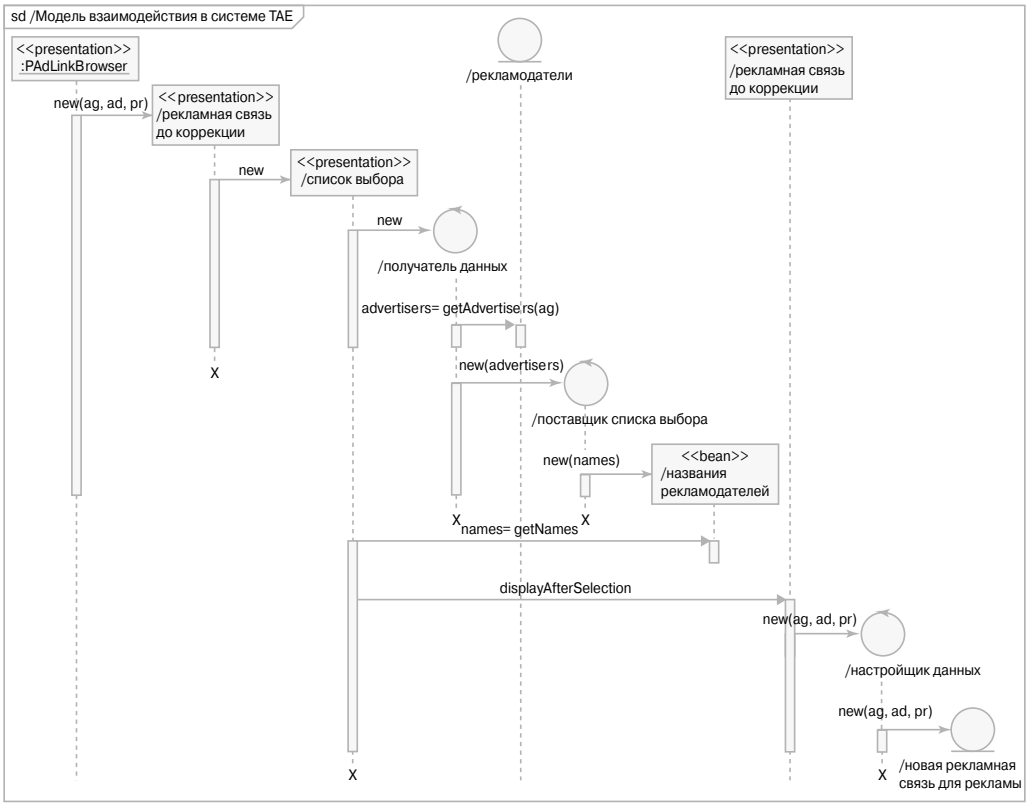


Рис. 6.43. Диаграмма последовательностей в системе оценки затрат на рекламу, учитывающая уровни PCBE

ГЛАВА 7

Проектирование графического пользовательского интерфейса

Цели

7.1. Принципы проектирования графического пользовательского интерфейса
7.2. Проектирование оконного интерфейса
7.3. Проектирование Web-интерфейса
7.4. Моделирование навигации в графическом пользовательском интерфейсе
Резюме
Ключевые термины
Многовариантные тесты
Вопросы
Упражнения. Управление взаимоотношениями с заказчиками
Упражнения. Прямой маркетинг по телефону
Ответы на контрольные вопросы
Ответы к многовариантным тестам
Ответы на вопросы с нечетными номерами
Объяснение упражнений. Управление взаимоотношениями с заказчиками

Цели

Времена, когда пользовательский интерфейс представлял собой зеленый текст на черном экране, а клавиатура служила единственным устройством для ввода информации, давно прошли. Современный графический пользовательский интерфейс (GUI — графический пользовательский интерфейс) не только красив, но и интерактивен. Теперь пользователь может управлять работой программы с помощью мыши (не говоря уже о голосовых и тактильных устройствах). Несмотря на то что программы по-прежнему проектируются так, чтобы предотвратить запрещенные или несанкционированные события, перенос акцентов управления с *алгоритма* на *взаимодействие* с пользователем изменил способы проектирования и реализации систем GUI.

Разработка графического пользовательского интерфейса представляет собой сложную задачу, требующую знаний в разных областях: от графики до программирования. Об аспектах проектирования графического пользовательского интерфейса написано много книг (например, Constantine and Lockwood, 1999; Fowler, 1998; Fowler and Stanwick, 2004; Galitz, 1996; Olsen, 1998; Ruble, 1997; Sklar, 2006; Windows, 2000). В данной главе основное внимание сосредоточено на фактах, которые должны знать проектировщики систем, чтобы успешно разрабатывать графические пользовательские интерфейсы и описывать навигацию между интерфейсными окнами или Web-страницами.

- Прочитав главу, читатели будут
- знать принципы разработки качественного графического пользовательского интерфейса;
- понимать сходство и различия между графическими пользовательскими интерфейсами настольных систем и Web-страниц;
- владеть технологией проектирования графического пользовательского интерфейса;
- разбираться в визуальных элементах графического пользовательского интерфейса — контейнерах, меню, средствах управления и т.д.;
- уметь применять разные способы навигации по интерфейсу;
- владеть навыками моделирования навигации по графическому пользовательскому интерфейсу.

7.1. Принципы проектирования графического пользовательского интерфейса

Разработка пользовательских интерфейсов начинается с предварительных набросков диалоговых окон на этапе анализа требований. Эти наброски используются в процессе сбора требований, при разборе возможных сценариев работы системы с заказчиками, для создания прототипов и разработки описания прецедентов использования. В процессе проектирования осуществляется дальнейшая разработка окон графического пользовательского интерфейса для приложения в соответствии с основными возможностями специализированного программного обеспечения, а также особенностями и ограничениями выбранной программной среды.

В главе 6 и других главах подчеркивается, что информационные системы предприятий неизбежно являются клиент-серверными. Если можно так выразиться, серверное решение *образует* программное решение, а клиентское решение его *продает*.

Клиентов графического пользовательского интерфейса можно разделить на программируемых клиентов, работающих на *настольных платформах* (desctop platforms), и браузерных клиентов на *Web-платформах* (Maciaszek and Liong, 2005; Singh et al., 2002). **Программируемый клиент** (programmable client) — это, как правило, *"толстый" клиент* (см. раздел 6.1.2 главы 6). В этом случае программа хранится и выполняется на клиентском компьютере и имеет доступ к ресурсам клиента. **Браузерный клиент** (browser client) представляет собой графический интерфейс пользователя, основанный на технологиях Web, и нуждается в помощи сервера для получения данных и выполнения программ. Такой клиент называется *"тонким"*, или Web-клиентом.

Независимо от вида клиента, проектирование графического пользовательского интерфейса должно подчиняться определенным универсальным принципам и использовать преимущества новейших технологий взаимодействия человека и компьютера. Проектирование графического интерфейса пользователя представляет собой *междисциплинарную деятельность*. Оно требует усилий *коллектива* — один человек, как правило, не обладает знаниями, необходимыми для реализации многоаспектного подхода к проектированию. Правильное проектирование графического пользовательского интерфейса требует объединения навыков художника, специалиста по анализу требований, системного проектировщика, программиста, эксперта по технологии, специалиста в области социальной психологии, а также, возможно, некоторых других специалистов, в зависимости от природы системы.

7.1.1. Переход от прототипа графического пользовательского интерфейса к его реализации

Типичный процесс проектирования графического пользовательского интерфейса для приложений информационных систем начинается с *прецедентов использования*. Аналитик, занимающийся описанием потока событий для прецедента использования, обладает некоторым зрительным образом интерфейса для поддержки взаимодействия человека и компьютера. В некоторых случаях аналитик может вставлять наброски графического интерфейса в описание прецедента использования. Сложные взаимодействия человека и компьютера невозможно адекватно описать, ограничиваясь лишь с помощью текста. Иногда процесс сбора и согласования требований заказчика невозможен без эскизов графического интерфейса пользователя.

Проектировщик, участвующий в определении взаимодействий для реализации прецедентов, должен иметь ясное зрительное представление о том, как выглядят экраны графического пользовательского интерфейса. Если до него этого уже не сделал аналитик, проектировщик становится первым человеком, который вырабатывает графические представления пользовательского интерфейса. Схемы интерфейса, предлагаемые проектировщиком, должны соответствовать технологии, на которой базируется интерфейс, — средствам организации окон и элементам управления окнами, Интернет-браузерам и т.д. Для того чтобы успешно воспользоваться технологическими возможностями, может потребоваться консультация *эксперта в области технологии*.

Прежде чем проект кооперации (см. раздел 6.5 главы 6) попадет к программистам для реализации, необходимо сконструировать прототип экранов, дружественный по отношению к пользователю. Для решения этой задачи привлекаются *художники-графики*, а также *специалисты по социальной психологии* и *бихевиоризму*. Совместными усилиями они могут предложить привлекательный и удобный графический пользовательский интерфейс.

Программист не должен слепо следовать указаниям проектировщиков при реализации экранов. Он обязан предложить изменения, обусловленные средой программирования. В некоторых случаях эти изменения могут улучшить интерфейс, а в других случаях — ухудшить из-за ограничений, связанных с программированием или производительностью труда.

Основной момент в проектировании графического пользовательского интерфейса заключается в том, что *интерфейс управляется пользователем* (тем не менее, системную целостность, надежность и безопасность контролирует сама система, а не пользователь). Современные объектно-ориентированные программы *управляются событиями* (event-driven). Объекты реагируют на события и сообщения. Внутренние взаимодействия между объектами запускаются внешними событиями, активизированными пользователем.

Основным фактором, который оценивается заказчиками при покупке программного обеспечения, является впечатление и ощущение от использования графического пользовательского интерфейса. Прототипы интерфейса могут служить как для оценки его внешнего вида, так и для реализации его функций. Однако реальный вид графического пользовательского интерфейса выясняется лишь на этапе реализации.

Пример 7.1. Управление взаимоотношениями с заказчиками

Обратитесь к задаче 3, в которой описывается системы управления взаимоотношениями с заказчиками (см. раздел 1.6.3 главы 1), а также к последующим примерам в главе 4. В частности, рассмотрите проект класса `Organization` (Организация) в примере 4.6 (см. раздел 4.2.1.2.3).

Цель этого примера — продемонстрировать, как может измениться экран графического пользовательского интерфейса для класса `Organization` от первоначального прототипа до конечной реализации. Предполагается, что в качестве базовой технологии создания интерфейса используется технология Microsoft Windows.

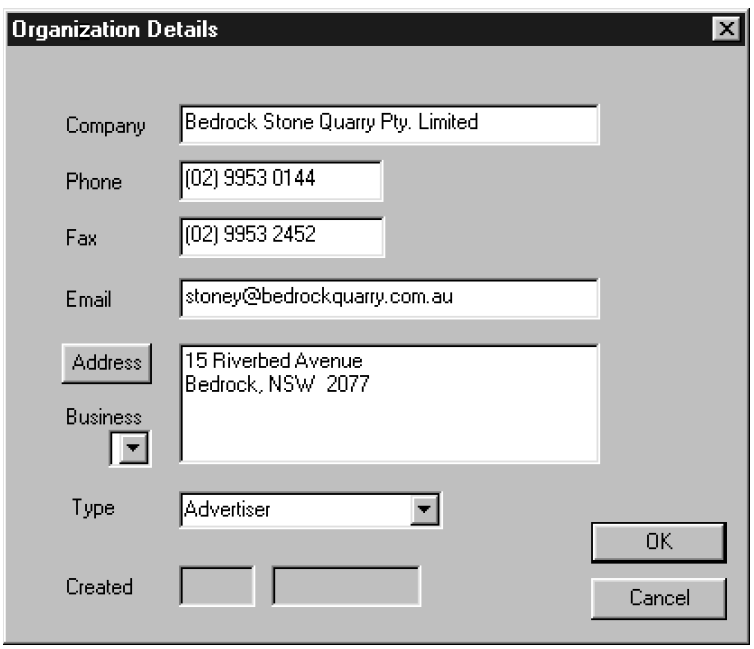


Рис. 7.1. Прототип окна для системы управления взаимоотношениями с клиентами на этапе анализа. Источник: компания Nielsen Media Research, Сидней, Австралия

Прототип графического пользовательского интерфейса для класса `Organization` представлен на рис. 7.1. Его основной целью является визуализация данных и контроль объектов, расположенных в окне. Обратная связь с пользователями, получаемая в виде их откликов, а также идеи коллектива разработчиков интерфейса изменяют не только внешний вид окна, но и его содержание.

На рис. 7.2 показано, как окно `Maintain Organizations` (Учет организаций) может измениться на этапе проектирования. Как видим, проектировщик предпочел диалоговое окно с закладками, а также внес много других изменений. Эти изменения лучше соответствуют принципам проектирования графического пользовательского интерфейса для Microsoft Windows и учитывают функциональные требования, пропущенные в первоначальном прототипе.

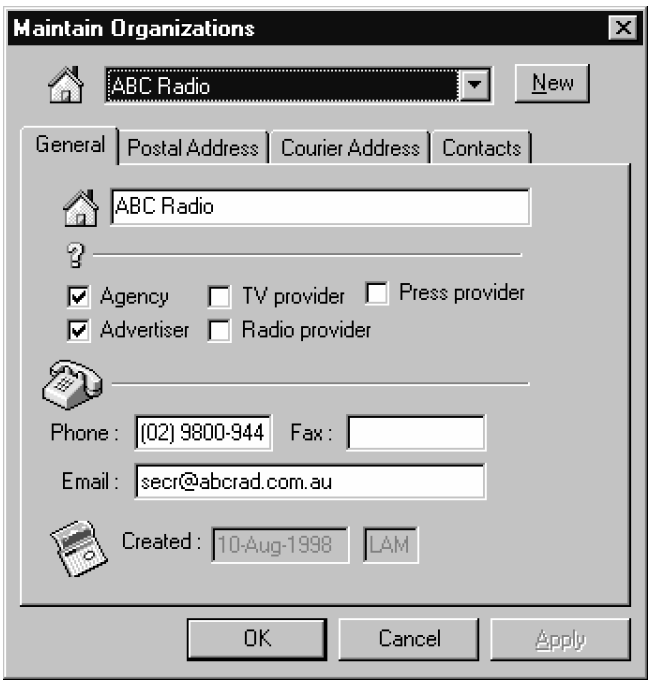


Рис. 7.2. Прототип окна для системы управления взаимоотношениями с клиентами на этапе проектирования. Источник: компания Nielsen Media Research, Сидней, Австралия

Дизайн окна на рис. 7.2 по-прежнему проблематичен с точки зрения внешнего вида и функциональных требований. В результате окно снова подвергается изменениям. Его окончательная реализация показана на рис. 7.3.

Окончательная реализация окна, показанная на рис. 7.3, запрещает использование выпадающего списка для выбора названий организаций (для обновления данных об организации предлагается открывать вторичное диалоговое окно).

Панель `Classification` (Классификация) теперь не допускает редактирования (поскольку пользователь системы не имеет права изменять классификацию организации). Кроме того, в окончательной организации увеличены поля редактирования и введена группа полей Hystory (История) и т.д.

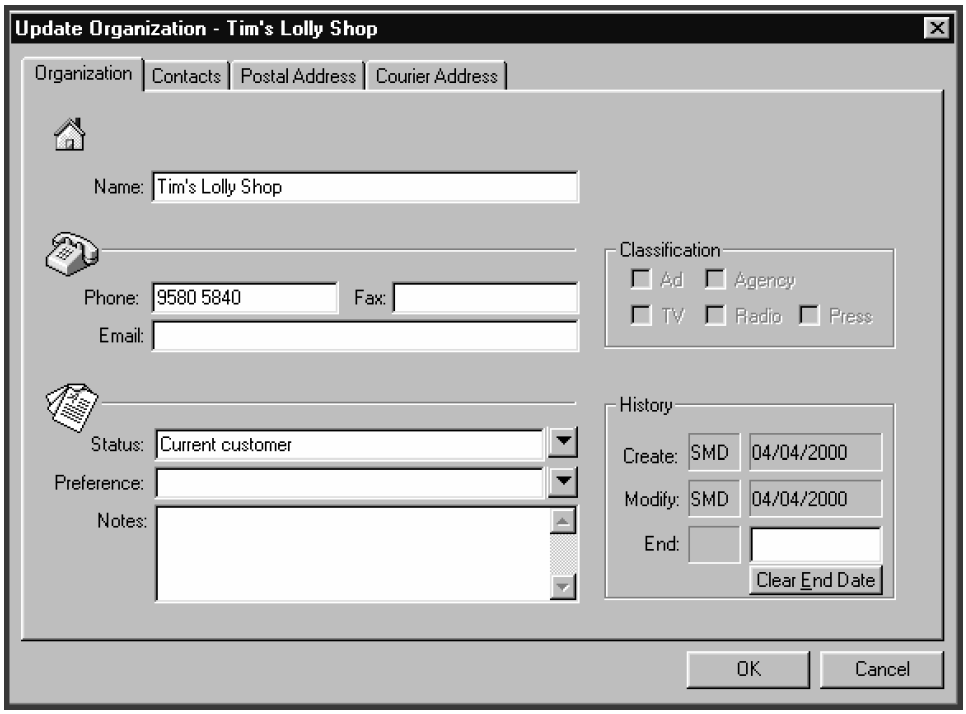


Рис. 7.3. Реализованное окно для системы управления взаимоотношениями с клиентами на этапе проектирования. Источник: компания Nielsen Media Research, Сидней, Австралия

7.1.2. Руководящие принципы проектирования интерфейса, ориентированного на пользователя

Проектирование графического интерфейса ориентируется на пользователя. Этот принцип стал основанием для ряда рекомендаций (например, Windows, 2000), которыми руководствуются разработчики программного обеспечения. Эти принципы обсуждаются также во многих книгах (например, Galitz, 1996; Ruble, 1997).

Руководящие принципы служат разработчикам основой, на которой они создают интерфейсы. Принимая любые проектные решения, касающиеся графического пользовательского интерфейса, разработчики должны использовать их на подсознательном уровне. Некоторые из этих руководящих принципов выглядят как хорошо известные старые истины, другие основаны на современной технологии создания GUI-интерфейсов.

7.1.2.1. Ориентация на пользователя

Интерфейс управляется пользователем (user in control) — вот главный принцип построения графического пользовательского интерфейса. Лучше было бы назвать этот принцип *ориентацией на пользовательское восприятие контроля* (user's perception of control). Некоторые называют его принципом *отсутствия опеки* (no mothering), так как программа не должна действовать как заботливая мать, делая многие вещи за пользователя (Triesman, 1994). Основной смысл этого принципа заключается в том, что пользователь самостоятельно инициирует действия, и если в результате этого контроль переходит к программе, то она поддерживает с пользователем необходимую обратную связь (в виде курсора в форме песочных часов, индикатора ожидания или чего-то подобного).

На рис. 7.4 показан типичный поток управления при взаимодействии человека и компьютера. Событие, инициированное пользователем (выбор пункта меню, щелчок мышью, перемещение указателя мыши по экрану и т.д.), может привести к открытию окна графического пользовательского интерфейса или вызову программы — как правило, программы доступа к базе данных на языке SQL. Эта программа временно перехватывает контроль у пользователя.



Рис. 7.4. Поток управления в программе графического пользовательского интерфейса

Процесс выполнения программы имеет возможность вернуть управление назад тому же или другому окну. В другом случае он может вызвать другой модуль на языке SQL или вызвать внешнюю программу. В некоторых случаях программа может выполнять задания пользователя. Это возможно, к примеру, если программе требуется выполнить вычисления, которые обычно связаны с явным пользовательским событием, или если программа перемещает указатель на другое поле экрана, а для события перемещения с исходного поля предусмотрен обработчик события выхода, связанный с ним.

7.1.2.2. Согласованность

Согласованность (consistency) несомненно является вторым основным принципом разработки качественного интерфейса. Фактически согласованность означает соблюдение стандартов и следование некоторым общепринятым правилам работы с графическим пользовательским интерфейсом. Согласованность может рассматриваться, по меньшей мере, в двух аспектах.

- Соответствие стандартам поставщика графического пользовательского интерфейса.
- Соответствие стандартам в области именования, программирования и другим, разработанным в организации стандартам, связанным с графическим пользовательским интерфейсом.

Оба аспекта одинаково важны, причем второй принцип (на который оказывают влияние разработчики) не должен противоречить первому. Если приложение разрабатывается для среды Windows, то следует обеспечить "впечатление и ощущение", свойственные работе в системе Windows. Для системы Macintosh замена знаменитого "яблочного" меню вложенным меню (типа "кенгуру"), которую однажды попытался сделать автор, — вообще никудышная идея!

Разработчик графического пользовательского интерфейса не должен слишком увлекаться творчеством и предлагать необычные новшества. Это может плохо сказаться на уверенности и умении пользователей. Пользователям следует представлять знакомую среду, поведение которой предсказуемо. Как заметил Трайсман (Treisman, 1994), можно представить, какова была бы реакция автомобилистов, если бы производитель автомобилей выпустил на рынок новую машину, у которой поменяли местами педали скорости и тормоза!

Не следует также недооценивать соответствие внутренним стандартам в области именования, программирования, аббревиатур и т.п. Сюда относятся именование и программирование меню, командных кнопок, полей экранов, а также стандарты по расположению объектов на экране и последовательному использованию элементов графического интерфейса в рамках всех приложений, разрабатываемых собственными силами.

7.1.2.3. Индивидуализация и настройка

Индивидуализация и настройка (personalization and customization) — два взаимосвязанных принципа разработки графического пользовательского интерфейса, в совокупности обеспечивающие свойство *адаптивности*. Индивидуализация интерфейса — это простая настройка под персональные потребности, в то время как настройка (так, как мы понимаем ее здесь) — административная задача приспособления программного обеспечения к требованиям различных групп пользователей.

Примером индивидуализации является изменение пользователем порядка и размеров колонки в программе просмотра строк (сетки) с последующим сохранением этих изменений. Эти предпочтения учитываются при последующих обращениях к этой программе.

Примером настройки является различие в функционировании программы по отношению к опытным пользователям и новичкам. Например, новичку программа может предложить помощь и дополнительные предупреждающие сообщения, указывающие на потенциальную опасность некоторых событий, инициируемых пользователем.

Во многих случаях различия между индивидуализацией и настройкой весьма размыты и малозаметны. Изменение пунктов меню, создание новых меню и тому подобное попадают в обе категории. Если они предназначены для личного использования — это индивидуализация. Если они осуществляются системным администратором в интересах всего коллектива пользователей — это настройка.

С индивидуализацией и настройкой связана такая особенность Интернета, как *локализация* (Lethbridge and Laganiére, 2001). Приложение можно адаптировать к местным особенностям (выбрав язык, набор символов, обозначения валют и формат представления дат), послав запрос операционной системе, в которой выполняется программа.

Другим аспектом адаптивности является настройка к потребностям людей с ограниченными возможностями. Например, незрячие люди могут потребовать, чтобы приложение использовало шрифт Брайля или голосовую связь. Глухие люди могут нуждаться в замене визуального вывода на звуковую информацию. Для пользователей с другими физическими ограничениями могут потребоваться особые настройки.

7.1.2.4. Толерантность

Хорошо спроектированный интерфейс должен позволять пользователям экспериментировать и совершать ошибки, проявляя терпимость к последним. Подобная *толерантность* (forgiveness) стимулирует исследовательскую активность пользователя, поскольку позволяет ему выполнять ошибочные последовательности действий с возможностью в любой момент совершить при необходимости “откат” в начало. Толерантность подразумевает многоуровневую систему отмены операций.

Об этом принципе легко говорить, однако он трудно поддается реализации. Реализация толерантности интерфейса многопользовательских баз данных отличается особой сложностью. Пользователь, снявший (и потративший) деньги с банковского счета, не имеет возможности отменить эту операцию! Единственное, что он в состоянии сделать, — это положить деньги назад на счет, осуществив другую транзакцию. Должен или не должен терпимый к ошибкам интерфейс предупреждать пользователя о последствиях снятия денег со счета — спорный вопрос (и относится он к принципу индивидуализации интерфейса).

7.1.2.5. Обратная связь

Принцип обратной связи (feedback) дополняет первый принцип, утверждающий, что интерфейс должен управляться пользователем. Это значит, что пользователь должен точно знать, что именно происходит, когда контроль временно передается программе. Разработчик должен встроить в систему визуальные и/или звуковые подсказки для каждого события, инициируемого пользователем.

В большинстве случаев указатель в виде песочных часов или индикатор ожидания представляет достаточный уровень обратной связи, чтобы понять, что программа выполняет некую операцию. Для тех компонентов приложения, которые иногда могут замедлять работу программы, может потребоваться более ясная обратная связь (например, в виде отображения поясняющего сообщения).

В любом случае разработчик никогда не должен предполагать, что приложение выполняется настолько быстро, что обратная связь не потребуется. Любые отклонения в рабочей нагрузке приложения докажут, что разработчик, к сожалению, ошибался.

7.1.2.6. Эстетичность и удобство

Эстетичность (aestethics) интерфейса влияет на зрительное восприятие системы. *Удобство* (usability) касается легкости, простоты, эффективности, надежности и продуктивности в использовании интерфейса. Безусловно, оба принципа касаются удовлетворенности пользователя. Именно в этом вопросе разработчик графического пользовательского интерфейса нуждается в помощи художника и эксперта по социальной психологии и бихевиоризму.

Существует много "золотых правил", касающихся создания эстетичного и удобного интерфейса (Constantine and Lockwood, 1999; Galitz, 1996). При этом следует учитывать такие вопросы, как фиксация и движение человеческого глаза, использование цветов, чувство уравновешенности и симметрии, выравнивание и отступ между элементами, чувство пропорции, группирование связанных элементов и т.д.

Принцип эстетичности и удобства превращает разработчика графического пользовательского интерфейса в художника. В этом смысле неплохо помнить о том, что простота красива. В действительности *простоту* (simplicity) часто рассматривают как еще один принцип создания пользовательского интерфейса, прочно связанного

с принципами эстетичности и удобства. В сложных приложениях простота лучше всего достигается с помощью подхода "разделяй и властвуй" за счет последовательного раскрытия информации таким образом, что она отображается только тогда, когда в ней возникает необходимость, возможно, в различных окнах.

Контрольные вопросы 7.1

КВ1. Как классифицируется графический пользовательский интерфейс клиента?

КВ2. Назовите основной принцип разработки графического пользовательского интерфейса.

КВ3. Какой принцип наиболее тесно связан с принципом "интерфейс должен управляться пользователем"?

7.2. Проектирование оконного интерфейса

Проектирование *программируемых настольных приложений* имеет два основных аспекта: проектирование окон и проектирование элементов ввода и редактирования информации в окна. Оба аспекта зависят от базовой среды GUI. В дальнейшем рассмотрение концентрируется на среде Microsoft Windows (Maciaszek and Liong, 2005; Windows, 2000).

Типичное приложение для среды Windows состоит из единственного **главного окна** (primary window). Главное окно поддерживается набором *всплывающих окон* (pop-up window), которые являются **вторичными** (secondary windows). *Вторичные окна* поддерживают действия пользователя с главным окном. Многие действия, поддерживаемые вторичными окнами, представляют собой набор основных операций над базой данных — так называемый набор CRUD-операций (create, read, update, delete — создать, читать, обновить, удалить).

С точки зрения программирования окна являются **контейнерами графического пользовательского интерфейса** (GUI container). Они представляют собой прямоугольные области на экране интерфейса и содержат другие контейнеры, меню и средства управления (например, командные кнопки). Контейнеры могут быть как главными, так и диалоговыми (вторичными) окнами, а также **панелями** (panes) и **полосами**, или **линейками** (panels).

Контейнеры, меню и средства управления являются *компонентами* графического пользовательского интерфейса. В языке Java предусмотрен набор компонентов графического пользовательского интерфейса для создания приложений и аплетов, помещенных в библиотеке классов, и интерфейсов **Swing**. Имена *конкретных классов* из библиотеки Swing начинаются с буквы J — `JDialog`, `JButton` и т.д.

7.2.1. Главные окна

Главное окно имеет границу (**рамку**). Рамка (frame) содержит строку заголовка (caption bar) окна, строку меню, панели инструментов, строку состояния, а также отображаемое и модифицируемое содержимое окна. Горизонтальные и вертикальные полосы прокрутки используются для прокрутки содержимого окна.

Отображаемое и модифицируемое содержимое может быть организовано в виде *панелей* (panes). Панели позволяют осуществлять просмотр различных, но связанных между собой, информационных элементов содержания и манипулировать ими. На рис. 7.5 показано главное окно, которое отображается после успешного входа в систему и открытия приложения. Панель в левой части окна содержит схему приложения в стиле программы Windows Explorer. Окно (а значит, и приложение) может быть закрыто из меню File (Файл) или с помощью кнопки Close (Закрыть), расположенной в правом верхнем углу панели. Комментарии соответствуют хорошо известной терминологии Windows.

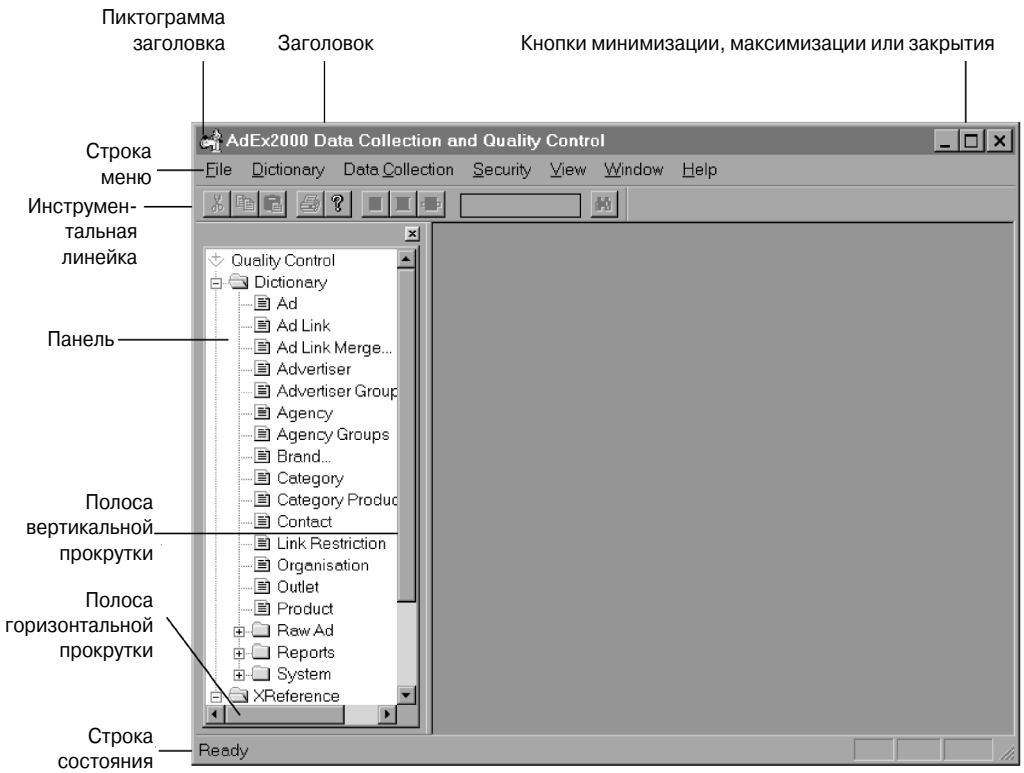


Рис. 7.5. Главное окно системы оценки затрат на рекламу. Источник: компания Nielsen Media Research, Сидней, Австралия

Типичными отличительными чертами главного окна являются наличие строки меню и панели инструментов. Панель инструментов содержит командные кнопки для наиболее часто используемых пунктов меню. Пиктограммы панели инструментов дублируют эти пункты меню. Они предназначены для быстрого доступа к наиболее часто используемым командам.

Пример 7.2. Управление взаимоотношениями с заказчиками

Обратитесь к задаче 5, в которой описывается система управления контактами с клиентами (см. раздел 1.6.5 главы 1), а также к примерам 6.13–6.15 (см. раздел 6.5 главы 6). В частности, рассмотрите рис. 6.35 (см. раздел 6.5.4) и покажите схему главного окна, в котором открывается диалоговое окно модификации данных. Включите в проект панель, аналогичную панели, показанной на рис. 7.5, *слева*.

На рис. 7.6 показано главное окно системы из примера 7.2. В левой части расположено *окно просмотра дерева* (tree browser) (см. раздел 7.2.1.2), позволяющее выбирать конкретные "контейнеры данных", которые будут изображаться в правой части окна, где открывается *окно просмотра строк* (row browser) (см. раздел 7.2.1.1).

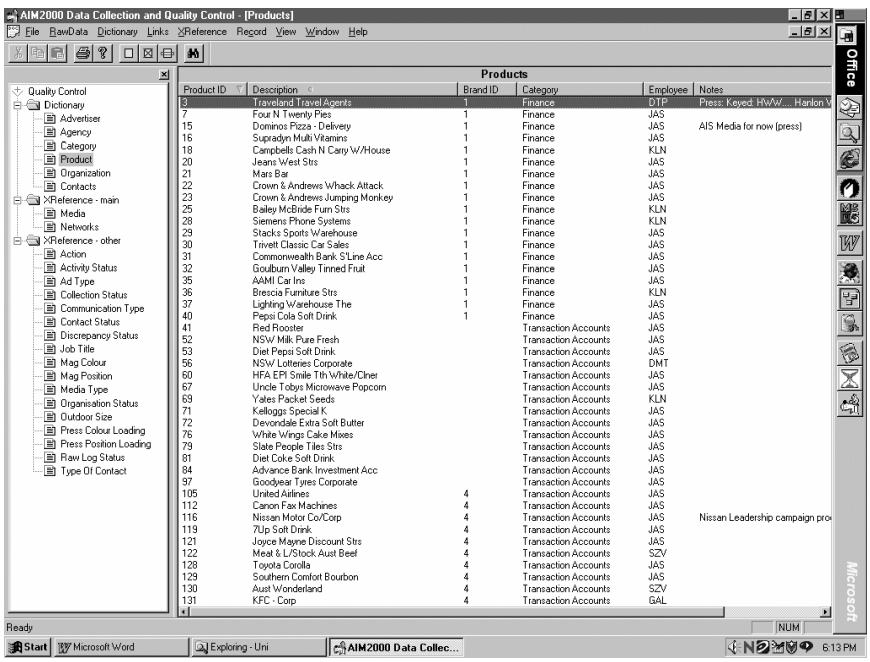


Рис. 7.6. Главное окно, в котором система оценки затрат на рекламу выводит список товаров. Источник: компания Nielsen Media Research, Сидней, Австралия

7.2.1.1. Окно просмотра строк

Довольно часто главное окно информационной системы используется для просмотра записей базы данных. Такое окно иногда называют *окном просмотра строк* (row browser). Пользователь имеет возможность "прокручивать" строки вверх и вниз с помощью вертикальной полосы прокрутки или соответствующих клавиш (<PageUp>, <PageDown>, <Home>, <End>, а также со стрелками вверх и вниз).

Пример окна просмотра строк показан на рис. 7.7. Документ, помеченный как Ad Link (Рекламные связи), в главном окне является *дочерним окном* (child window). Это окно обладает собственным набором *кнопок управления окном* (window button) для минимизации, восстановления и закрытия окна, расположенных в правом углу строки меню. Ширину колонок сетки окна просмотра можно *изменять*, так же как и *порядок их расположения*. Округлые *зазубрины* рядом с именами колонок указывают на то, что колонки можно сортировать, — после щелчка на колонке записи будут отсортированы в порядке возрастания или убывания значений в этой колонке.

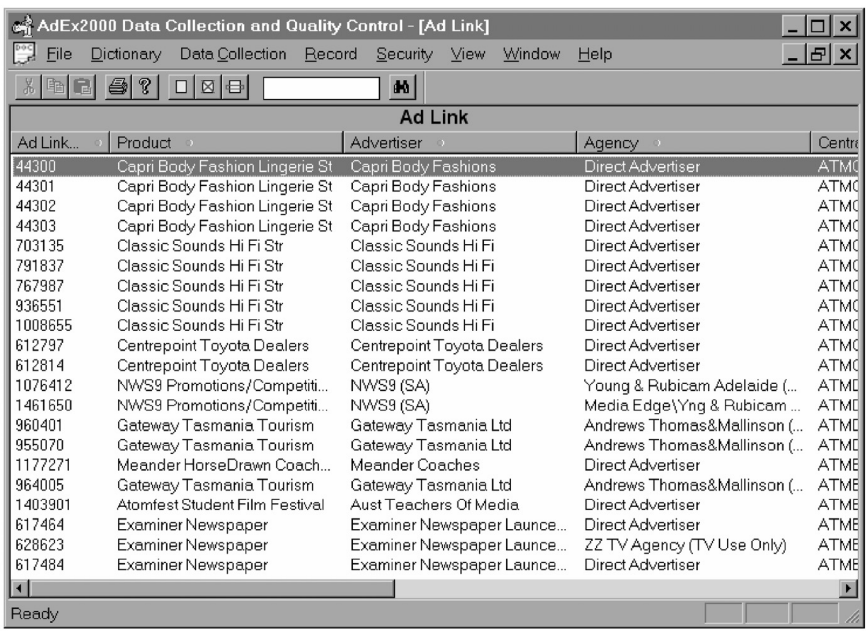


Рис. 7.7. Окно просмотра строк в системе оценки затрат на рекламу. Источник: компания Nielsen Media Research, Сидней, Австралия

В любой момент времени в окне просмотра строк активизирована только одна строка (запись). Двойной щелчок мышью на этой строке обычно приводит к открытию *окна редактирования* (edit window), которое содержит детали, относящиеся к этой записи. Окно редактирования позволяет модифицировать содержимое записи.

Панели могут использоваться для разделения окна по вертикали, или по горизонтали, или в обоих направлениях. На рис. 7.8 показано разделение окна по горизонтали. Как ясно из заголовка окна, для отображения товаров по рекламодателям и рекламным агентствам используются три панели. Средняя панель содержит перечень рекламодателей, связанных с текущим выбранным рекламным агентством, показанным в верхней панели. На нижней панели отображаются товары, рекламируемые выбранным рекламодателем.

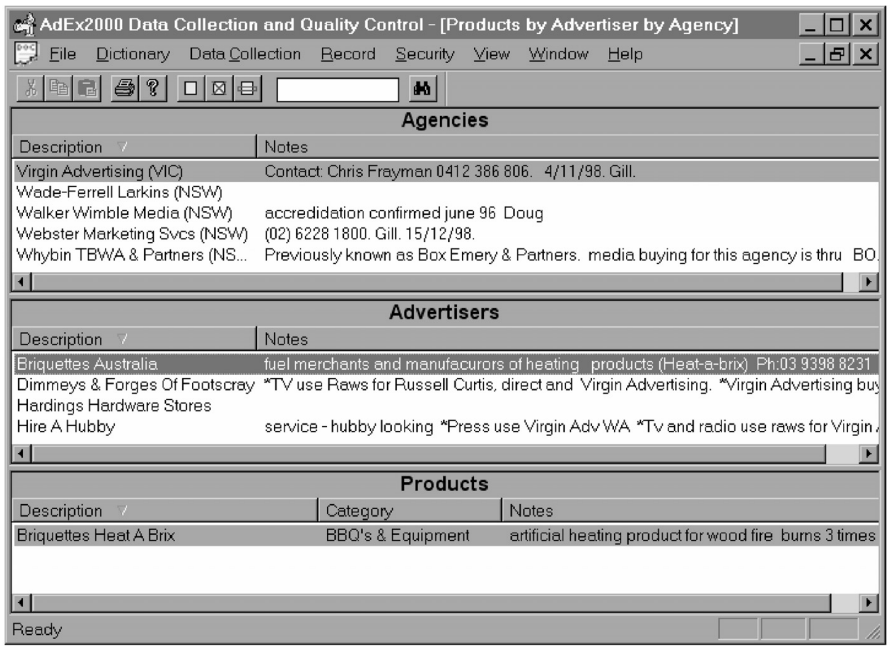


Рис. 7.8. Окно просмотра строк с несколькими панелями в системе оценки затрат на рекламу. Источник: компания Nielsen Media Research, Сидней, Австралия

7.2.1.2. Окно просмотра деревьев

Еще один популярный вариант главного окна — *окно просмотра деревьев* (tree browser). Программа просмотра деревьев отображает связанные записи в виде схемы с отступами. Схема содержит элементы управления, которые позволяют сворачивать и разворачивать дерево. Хорошо известный пример окна просмотра деревьев — отображение каталогов файлов компьютера с помощью программы Windows Explorer.

В отличие от окна просмотра строк, окно просмотра деревьев должно допускать модификацию на месте, т.е. модификацию содержимого окна без активизации окна редактирования. Модификации в окне просмотра деревьев осуществляются с помощью операции “перетащить и опустить” (“drag and drop”).

На рис. 7.9 показано окно просмотра деревьев, распложенное в левой панели главного окна. Правая панель представляет собой окно просмотра строк. Выбор записи для группы агентств приводит к отображению агентств, входящих в эту группу в окне просмотра строк.

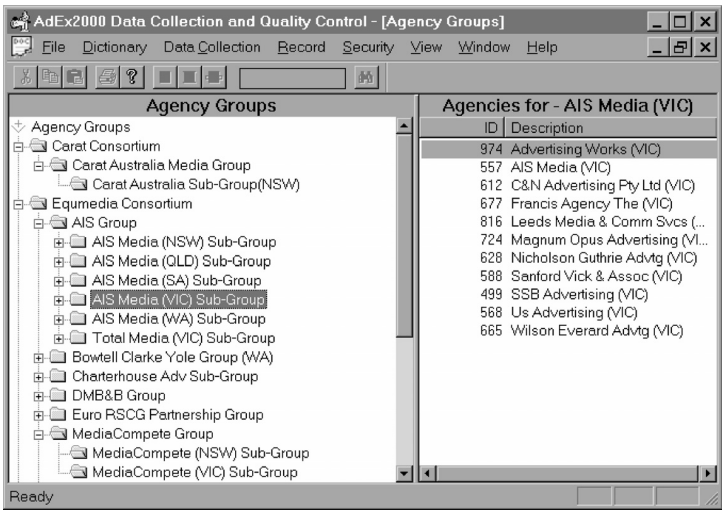


Рис. 7.9. Окно просмотра деревьев в системе оценки затрат на рекламу. Источник: компания Nielsen Media Research, Сидней, Австралия

7.2.2. Вторичное окно

За исключением простейших приложений информационных систем главное окно всегда сопровождается *вторичным окном* (second window). Оно расширяет функциональные возможности главного окна, в частности, в отношении операций, которые модифицируют базу данных (т.е. за счет операций вставки, удаления или обновления).

Вторичное окно обычно является *модальным* (modal) по отношению к его главному окну. Прежде чем приступить к работе с любым другим окном приложения, пользователь должен ответить и закрыть вторичное окно. *Немодальные* (modeless) вторичные окна допускаются, однако, их использование не рекомендуется.

С технической точки зрения *окно входа* в систему (logon window) является вторичным, хотя теоретически оно никогда не открывается в контексте главного окна. Фактически главное окно открывается лишь в случае успешной авторизации. Простые приложения могут состоять из одного или нескольких вторичных окон.

Основные визуальные различия между главным и вторичным окном показаны на примере экрана входа в систему (рис. 7.10). У вторичного окна отсутствуют: строка меню, панель инструментов, полосы прокрутки и строка состояния. События инициируются пользователем посредством *командных кнопок*, таких как OK, Cancel (Отмена), Help (Помощь).

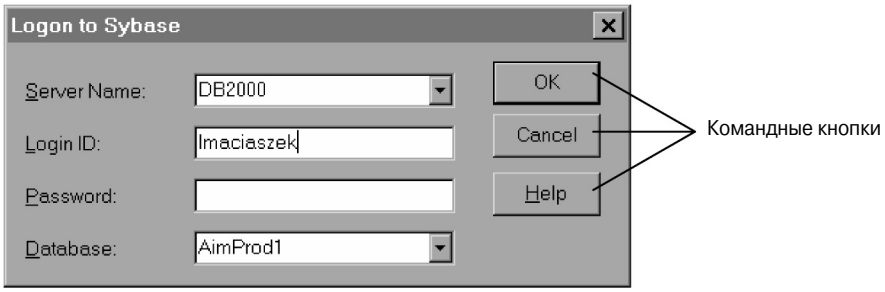


Рис. 7.10. Простой пример вторичного окна — окно входа в систему оценки затрат на рекламу. Источник: компания Nielsen Media Research, Сидней, Австралия

Вторичные окна выступают в виде различных экранных форм и масок. Ниже перечислены основные типы вторичных окон.

- Диалоговое окно.
- Папка с вкладками.
- Выпадающий список.
- Окно сообщений.

7.2.2.1. Диалоговые окна

Диалоговое окно (dialog box) является почти синонимом понятия вторичного окна и предоставляет наиболее часто требуемые свойства вторичных окон. Оно поддерживает диалог между пользователем и приложением. Диалог предполагает ввод пользователем некоторой информации, которая обрабатывается приложением.

Пример диалогового окна приведен на рис. 7.11. В этом окне, предназначенном для вставки, отображается рекламная продукция, соответствующая текущему товару, выбранному в подокне просмотра строк главного окна. Пользователь может модифицировать любые значения редактируемых полей, ограниченных рамками с белым заполнением.

Пример 7.3. Конвертация валют

Вернитесь к задаче 7, в которой описана система конвертации валют (см. раздел 1.6.7 главы 1), и к примерам, изложенным в главе 5. В частности, изучите спецификацию системы в примере 5.12 (см. раздел 5.4.2 главы 5) и обратите внимание не следующий фрагмент.

Рассмотрите реализацию оконного интерфейса для системы конвертации валют в обоих направлениях (например, австралийских долларов в американские и наоборот). Окно должно содержать три поля — для суммы в австралийских долларах, в американских долларах и для обменного курса.

Кроме того, в окне должны быть три кнопки: USD to AUD, AUD to USD и Close (для выходы из приложения).

Разработайте проект окна для конвертера валют в соответствии с указанной выше спецификацией.

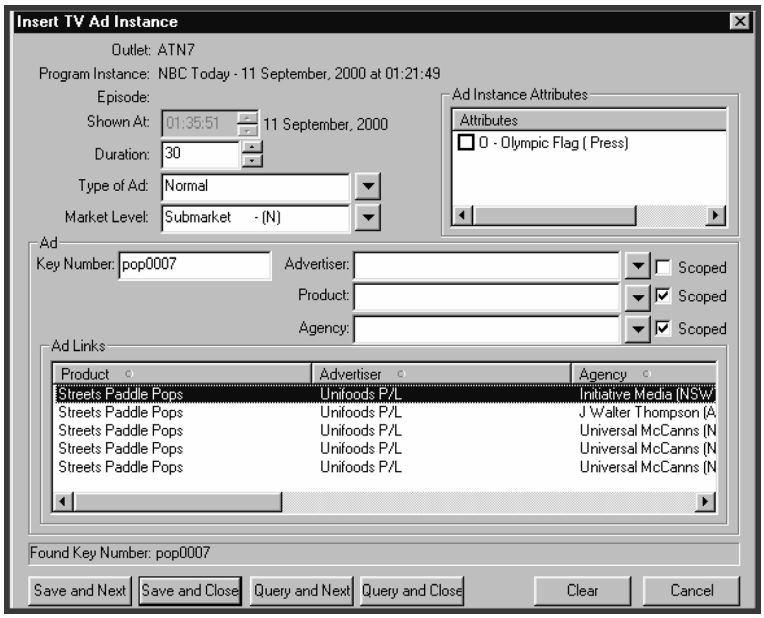


Рис. 7.11. Диалоговое окно в системе оценки затрат на рекламу. Источник: компания Nielsen Media Research, Сидней, Австралия

Диалоговое окно для конвертера валют, описанного в примере 7.3, показано на рис. 7.12.

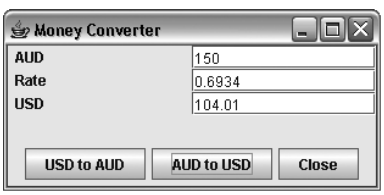


Рис. 7.12. Диалоговое окно в системе конвертации валют

7.2.2.2. Папка с вкладками

В тех случаях, когда объем информации, которую необходимо отобразить во вторичном окне, превышает "полезную площадь", а предмет рассмотрения можно разделить на несколько логических групп, можно использовать *папку с вкладками*

(tab folder). В каждый момент времени видимой является только одна из вкладок, ярлык которой расположен на вершине вкладок. (В системе Microsoft Windows папка с вкладками называется *ведомостью свойств со вкладками* (tabbed property sheet), а каждая из вкладок — *страницей свойств* (property page).)

На рис. 7.13 показана папка с вкладками для ввода информации о новых контактах. Три вкладки разделяют большой объем информации, вводимой пользователем, на три группы. Командные кнопки в нижней части экрана применимы ко всему окну, а не только к текущей видимой странице вкладки.

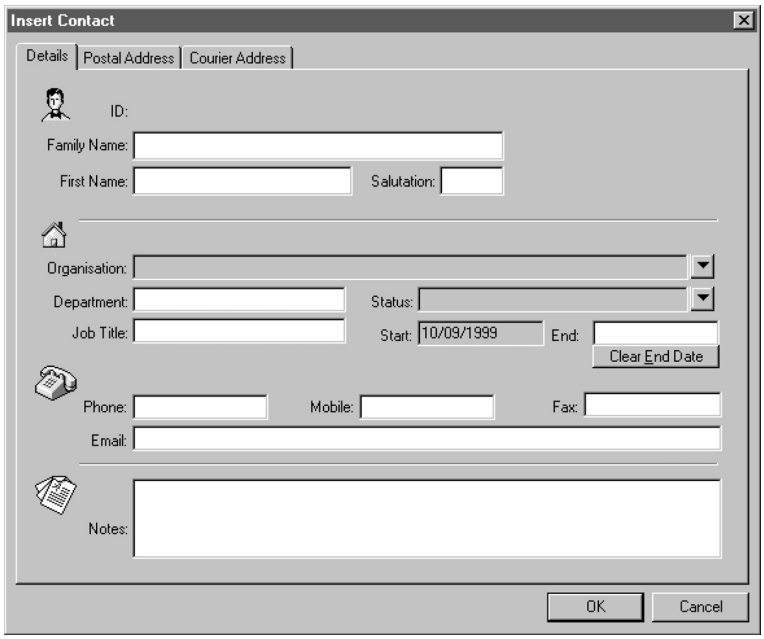


Рис. 7.13. Папка с вкладками в системе управления взаимоотношениями с заказчиками. Источник: компания Nielsen Media Research, Сидней, Австралия

7.2.2.3. Выпадающий список

В некоторых случаях вкладку удобно заменить *выпадающим списком* (drop-down list). Выпадающий список — это *список выбора элементов* (picklist), из которого пользователь может выделить подходящий. Для того чтобы вставить новый элемент в список, пользователь может ввести новое значение, которое добавляется в список и отображается при открытии списка в следующий раз.

Как видно из рис. 7.14, выпадающий список не обязательно должен быть простым перечислением значений, он может представлять собой окно просмотра деревьев.

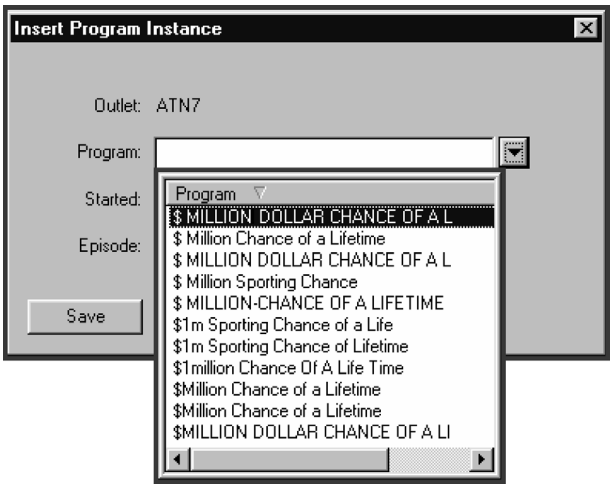


Рис. 7.14. Выпадающий список в системе оценки затрат на рекламу. Источник: компания Nielsen Media Research, Сидней, Австралия

7.2.2.4. Окна сообщения

Окно сообщения — это вторичное окно, в котором выводится сообщение для пользователя. Это сообщение может быть предупреждением, объяснением, исключительным условием и т.д. Командные кнопки в окне сообщения предлагают пользователю сделать выбор.

На рис. 7.15 показано сообщение, требующее подтверждения пользователя (с помощью кнопки OK).



Рис. 7.15. Окно сообщения в системе оценки затрат на рекламу. Источник: компания Nielsen Media Research, Сидней, Австралия

7.2.3. Меню и панели инструментов

Компоненты графического пользовательского интерфейса включают в себя меню и панели инструментов. В библиотеке Java Swing имена классов отражают их функциональное предназначение — например, `JMenuBar`, `JMenu`, `JMenuItem`, `JCheckBoxMenuItem`, `JRadioButtonMenuItem`.

Пункты меню объединяются в *выпадающие* списки (pull-down), *открывающиеся каскадом* (cascading) или *всплывающие* (pop-up). (Последние открываются после щелчка правой кнопкой мыши.)

Пункты меню реагируют на события, генерируемые пользователем, запуская определенные процессы. Как правило, действия, соответствующие пунктам меню, активизируются *щелчком мыши*. Часто используемые пункты меню могут иметь *"быстрые" клавиши* (accelerator key). Эти клавиши позволяют выполнять действия, соответствующие пунктам меню, не открывая само меню. Кроме того, ускоренный доступ к пункту открытого меню обеспечивается нажатием клавиши с первой (подчеркнутой) буквой в его названии.

На рис. 7.16 показаны разные варианты меню (Maciaszek and Liong, 2005), а также имена классов из библиотеки Java Swing, с помощью которых реализованы эти меню.

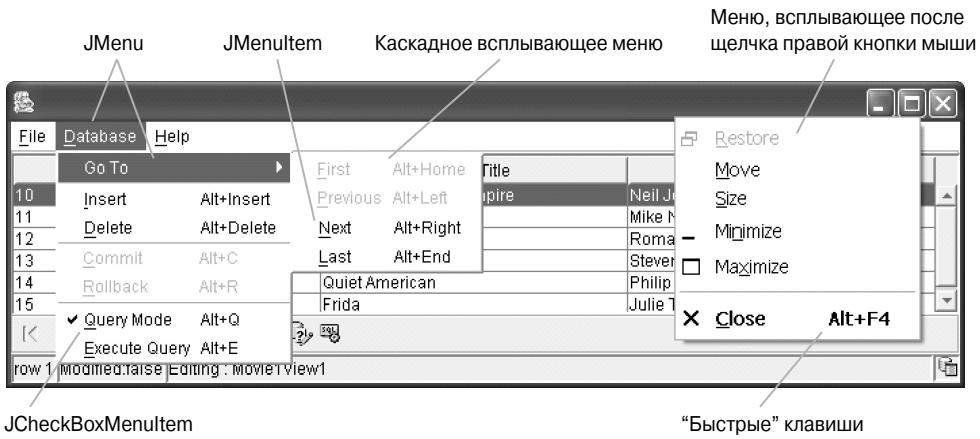


Рис. 7.16. Меню. Источник: Maciaszek and Liong (2005). Воспроизводится с разрешения компании Pearson Education Ltd.

Выбор пункта меню и использование "быстрых" клавиш — не самые эффективные способы активизации действий. Для наиболее часто используемых пунктов меню разработчики графического пользовательского интерфейса создают панели кнопок. *Панель инструментов* (toolbar) содержит пиктограммы, дублирующие функциональные свойства основных пунктов меню.

Панель инструментов можно заменить фиксированной или *плавающей* в рамках окна, т.е. ее можно отделить от окна в отдельное небольшое окно и разместить в любой позиции (Maciaszek and Liong, 2005). Библиотека Swing реализует панели инструментов с помощью класса `JToolBar`. Пример панели инструментов приведен на рис. 7.17.

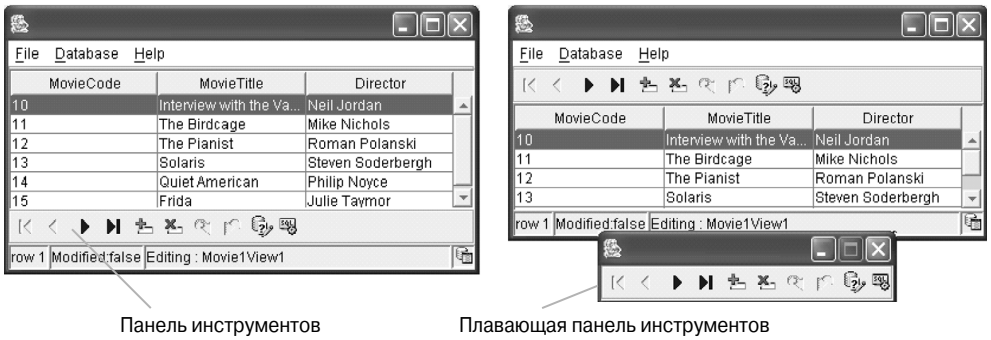


Рис. 7.17. Панели инструментов. Источник: Maciaszek and Liong (2005). Воспроизводится с разрешения компании Pearson Education Ltd.

7.2.4. Кнопки и другие средства управления

Меню и панели инструментов предназначены для визуализации *обработки событий*, происходящих в пользовательском интерфейсе. Аналогичное представление обеспечивается с помощью средств управления графического пользовательского интерфейса. *Средства управления* предназначены для перехватывания, анализа и реализации событий, генерируемых пользователями. Средства управления можно разделить на следующие группы.

- Командные кнопки. Классы библиотеки Swing, производные от абстрактного класса `AbstractButton`.
- Другие средства управления. Классы, производные от абстрактного класса `JComponent`.

На рис. 7.18 показана визуализация средств контроля, обеспечиваемых библиотекой Swing, при этом имена компонентов повторяют имена соответствующих классов (Maciaszek and Liong, 2005).

Отличия между разными кнопками иногда не заметны, поэтому необходимо дать некоторые объяснения. После щелчка мышью кнопка JButton запускает событие и всплывает снова (если результаты события не выводятся в отдельном окне и не закрывают собой кнопку). В противоположность этому кнопка JToggleButton после щелчка мышью остается нажатой, пока пользователь не щелкнет на ней снова.

Кнопки JRadioButton и JCheckBox представляют собой две разновидности кнопки JToggleButton. Кнопка JRadioButton (Радиокнопка) используется для реализации группы кнопок, в которой может быть выбрана только одна кнопка. Кнопка JCheckBox (Флажок) представляет собой независимое средство контроля, которое может находиться в состоянии “нажата” или “отжата”.

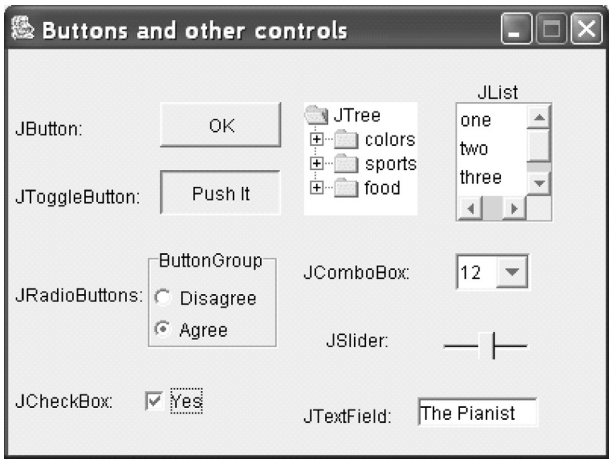


Рис. 7.18. Кнопки и другие средства управления. Источник: Maciaszek and Liong (2005). Воспроизводится с разрешения компании Pearson Education Ltd.

Средства управления JList, JTree и JComboBox непосредственно связаны с контейнерами, упомянутыми ранее. Класс JList используется для реализации окна просмотра строк (см. раздел 7.2.1.1), класс JTree — для реализации окна просмотра деревьев (см. раздел 7.2.1.2), а класс JComboBox — для реализации выпадающих списков (см. раздел 7.2.2.3).

Контрольные вопросы 7.2

КВ1. Назовите основное отличие главного окна от вторичного.

КВ2. Какой компонент графического пользовательского интерфейса имеет страницы?

КВ3. Какой компонент графического пользовательского интерфейса имеет "быстрые" клавиши?

7.3. Проектирование Web-интерфейса

"Web-приложение — это Web-система, позволяющая пользователям реализовать бизнес-логику с помощью Web-браузера" (Conallen, 2000). *Бизнес-логика* может быть реализована на сервере и/или клиентском компьютере. Следовательно, **Web-приложение** представляет собой разновидность клиент-серверной системы (см. раздел 6.1 главы 6), представленной в Интернет-браузере.

Клиентский Интернет-браузер отражает Web-страницы на экране компьютера. *Web-сервер* доставляет Web-страницы браузеру. Документы Web-страницы могут быть *статичными* (т.е. не допускать изменений) и *динамичными*. Кроме того, документами Web-страницы могут быть *формы*, заполняемые пользователями. Для разделения Web-страниц на части используются *фреймы* (frames), так что пользователь может просматривать несколько Web-страниц одновременно.

Web-страница, используемая в качестве входной точки Web-приложения, может рассматриваться как разновидность главного окна. В отличие от настольных информационных систем, строки меню и панели инструментов на Web-страницах не выполняют специальных задач. Они используются лишь для реализации перехода с одной страницы на другую, а также для выполнения общих задач, связанных с содержанием Web-страницы (например, печать или копирование). Пользовательские события в Web-приложении программируются с помощью *пунктов меню*, *командных кнопок* и *активных ссылок* (гиперссылок).

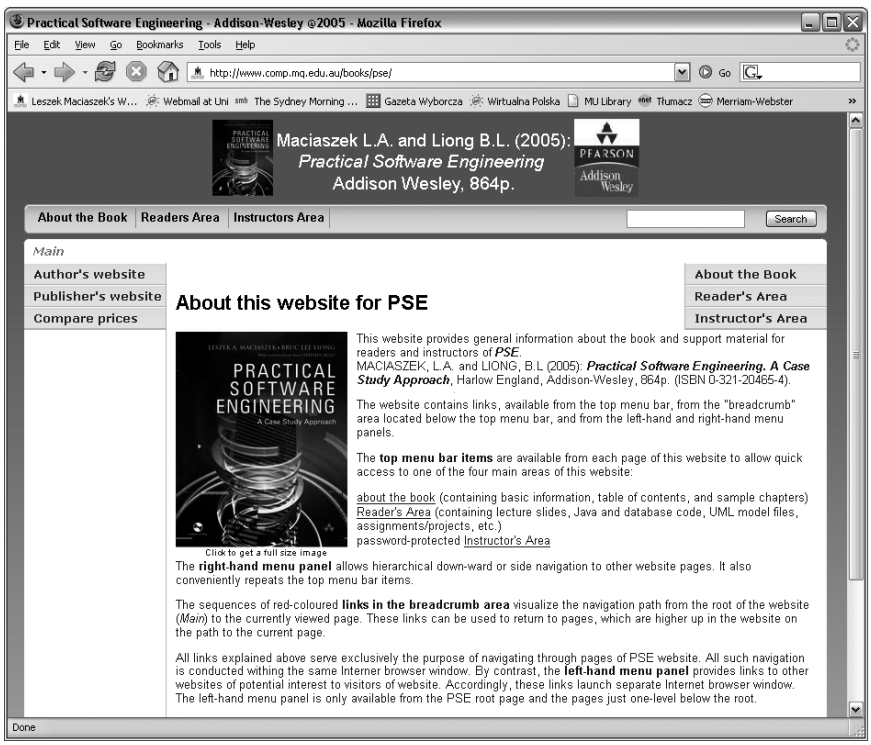


Рис. 7.19. Web-страница на Web-сайте

На рис. 7.19 показан пример Web-страницы на Web-сайте. Она не является частью отдельного Web-приложения. Даже в этом простом случае Web-страница предоставляет множество разных способов вызова других страниц Web-сайта

и выполнения некоторых функций (например, кнопка Search (Кнопка) в правом верхнем углу позволяет искать ключевые слова на Web-странице). Связи с другими Web-страницами реализуются с помощью следующих средств управления.

- *Строка меню*, расположенная под заголовком Web-страницы (About the Book, Readers Area, Instructors Area).
- Ниже строки меню расположена *навигационная цепочка* (breadcrumbs) (на данной странице этот элемент навигации имеет только один пункт — Main).
- Слева и справа расположены *панели меню*, которые могут иметь средства навигации по дополнительным спискам меню.

7.3.1. Технология реализации Web-приложений

Основной технологией реализации Web-приложений является *Web-сервер*, представляющий Web-страницы в браузере. Однако основная обработка обычно выполняется на *сервере приложения* (см. раздел 6.1.2 главы 6). Сервер приложения управляет логикой приложения (деловыми транзакциями и бизнес-правилами). Важность сервера приложения такова, что часто он выполняет функции Web-сервера, т.е. Web-сервер становится частью сервера приложения.

Сервер приложения поддерживает *состояние приложения*, чтобы отслеживать действия, выполняемые пользователем в оперативном режиме (online). Простым способом мониторинга этого состояния являются файлы *cookie*, содержащие короткие строчки символов, описывающие действия пользователя в оперативном режиме. Поскольку количество пользователей, работающих в оперативном режиме, может быть произвольно большим, для ограничения действия пользователей может использоваться *блокировка сессии по времени* (session timeout). Если пользователь не проявит активности в течение 15 минут (типичный интервал простоя), то сервер отсоединится от клиента. Файлы cookie могут удаляться с компьютера клиента, а могут оставаться на нем.

Сценарии и аплеты позволяют создавать динамические **клиентские страницы** (client page). *Сценарий* (script), например, написанный на языке JavaScript, — это программа, интерпретируемая браузером. **Аплет** (applet) — это компилируемый компонент, выполняемый в контексте браузера, но имеющий ограниченный доступ к другим ресурсам компьютера клиента (по соображениям безопасности). Образно говоря, аплеты выполняются в "*песочнице*" (sandbox).

Web-страница может содержать сценарии, выполняемые на сервере. Такие страницы называют **серверными** (server page). Серверная страница имеет доступ ко всем ресурсам сервера баз данных. Серверные страницы управляют сессиями клиента, размещают файлы cookie в браузере и создают клиентские страницы (создают страницы документов из бизнес-объектов, расположенных на сервере, и посылают его клиенту).

Стандартные *библиотеки доступа к данным* (data access libraries) используются для того, чтобы открыть сценариям, расположенным на серверных страницах, доступ к базе данных. Как правило, для этого используются технологии Open Database Connectivity (ODBC), Java Database Connectivity (JDBC), Remote Data Object (RDO) и ActiveX Data Objects (ADO). В ситуациях, когда организация выбирает в качестве стандарта конкретную систему управления базами данных, низкоуровневая функция вызывает библиотеку Database Library (DBLib), открывающую прямой доступ к базе данных.

Для реализации *Web-серверов* используются технологии HyperText Markup Language (HTML), Active Server Pages (ASP) и Java Server Pages (JSP). Для создания *Web-страниц* применяются клиентские сценарии на языках JavaScript или VBScript, документы на языке eXetsible Markup Language (XML), аплеты Java, компоненты JavaBeans или средства управления ActiveX.

Для получения Web-страниц от Web-сервера клиенты используют протокол HyperText Transfer Protocol (HTTP). Страницы могут выполнять сценарии или содержать компилируемые и непосредственно выполняемые модули DLL (Dynamic Link Library), например Internet Server Application Programming Interface (ISAPI), Netscape Server Application Programming Interface (NSAPI), Common Gateway Interface (CGI) или **сервлеты** Java (Conallen, 2000).

Файлы *cookie* представляют собой простой механизм для поддержки связи между клиентом и сервером. Без них система была бы *бесконтактной* (connectionless Internet system). Более сложный механизм поддержания связи между клиентами и серверами превращает Интернет в *систему распределенных объектов*. В такой системе объекты имеют уникальные идентификаторы OID (см. раздел А.2.3 приложения А) и взаимодействуют, обмениваясь этими идентификаторами. Основными механизмами работы в системе распределенных объектов являются технологии CORBA, DCOM и EJB. В рамках этих технологий объекты могут взаимодействовать с помощью протокола HTTP или посредством Web-сервера (Conallen, 2000).

Архитектура развертывания, поддерживающая более сложные Web-приложения, состоит из четырех ярусов вычислительных узлов (см. раздел 6.1.1 главы 6).

1. Клиент с браузером.
2. Web-сервер.
3. Сервер приложения.
4. Сервер базы данных.

С помощью браузера *клиентского узла* можно отображать как статические, так и динамические страницы. Кроме того, браузер может загружать и выполнять сценарии и аплеты. Дополнительные функциональные свойства клиентского браузера можно реализовать с помощью объектов ActiveX или JabaBeans. Код, выполняемый на клиентском компьютере, но за пределами браузера, может удовлетворять другие требования, предъявляемые к графическому пользовательскому интерфейсу.

Web-сервер обрабатывает запросы, поступающие от браузера, и динамически обновляет страницы и код для исполнения и отражения на экране клиентского компьютера. Кроме того, Web-сервер выполняет настройку и параметризацию сессии связи с пользователем.

Если в реализацию системы вовлечены распределенные объекты, то *сервер приложения* является ее обязательным элементом. Именно он управляет бизнес-логикой. Бизнес-компоненты публикуют свои интерфейсы для других узлов с помощью интерфейсов компонентов, например CORBA, DCOM или EJB.

Бизнес-компоненты инкапсулируют персистентные данные, хранящиеся в базе данных, возможно, реляционной. Они взаимодействуют с *сервером базы данных* с помощью протоколов связи с базами данных, например JDBC или ODBC. Узел базы данных обеспечивает масштабируемое хранение данных и многопользовательский доступ к ним.

7.3.2. Проектирование содержания

Проектирование содержания Web-сайтов привело к появлению новой специальности — *проектировщик Web-содержания* (Web content designer). *Проектирование содержания* определяет, каким образом пользователю должно быть представлено визуальное содержание Web-сайта или Web-приложения в Web-браузере. Аналогично проектированию оконного графического интерфейса пользователя, проектирование Web-содержания требует сочетания навыков художника, с одной стороны, и программиста, с другой. Кроме того, принципы правильного проектирования графического пользовательского интерфейса в равной степени применимы и к проектированию Web-содержания (см. раздел 7.1.2).

Различие между этими двумя видами проектирования проявляется в том, что при разработке Web-сайта или Web-приложения разработчик может не знать целевую аудиторию. Соответственно, при разработке Web-содержания необходимо обеспечивать более высокую гибкость приложения и учитывать разные потребности, интересы, степень подготовленности и предпочтения пользователей. Это еще больше подчеркивает важность принципов проектирования, изложенных в разделе 7.1.2.

Содержание должно соответствовать природе и цели Web-сайта или Web-приложения. Склар (Sklar, 2006) выделяет следующие разновидности Web-сайтов и Web-приложений в зависимости от их предназначения.

- *Доска объявлений.* Сайт, демонстрирующий присутствие организации в сети Web.
- *Издательский сайт.* Публикует газеты и периодические издания.
- *Портал.* Сайт, который публикует собственное информационное содержание и действует как шлюз доступа к Web-сервисам и ресурсам, например магазинам, поисковым машинам, серверам электронной почты (основным источником дохода является реклама).

- *Сайт организации, имеющей особые или общественные интересы, а также некоммерческой организации.* Содержит новости, контактную информацию, ссылки, загружаемые файлы и т.д. в соответствии со своим предназначением.
- *Блог* (сокращение от слова weblog). Сайт, содержащий частную информацию, которая отражает уникальные интересы автора и предлагает сотрудничество другим блоггерам.
- *Виртуальная галерея.* Сайт, содержащий литературные, визуальные и музыкальные произведения писателей, художников, фотографов, музыкантов и других авторов (как правило, эти произведения защищены авторскими правами с помощью цифровой технологии).
- *Сайты электронной коммерции, каталоги и Интернет-магазины.* Приложение (но, несомненно, не просто сайт), обеспечивающее ведение бизнеса в Интернете.
- *Сайт для сопровождения продукции.* Распространяет информацию, инструкции, обновления, советы, документацию, справочные и другие материалы, предназначенные для пользователей и клиентов.
- *Сайты сетей интра- и экстранет* (intranet and extranet). Позволяют сотрудникам обращаться к программному обеспечению организации, а также получать документы, инструкции, электронные письма и другие услуги с помощью закрытых локальных компьютерных сетей (в случае экстранет возможен доступ из Интернета).

7.3.2.1. Web-сайт для континуума Web-приложений

Эта книга посвящена моделированию прикладного программного обеспечения, но в случае Web-приложений (как станет ясно чуть позднее) разделительная линия между сайтом и приложением размыта. В этом случае вместо разделения проекта на две эти части, мы говорим о континууме, в пределах которого сайт плавно переходит в приложение. Рассмотрим, например, рис. 7.19. Превращает ли кнопка Search, расположенная в правом верхнем углу страницы, сайт в приложение? А что сказать о сайте, позволяющем проводить интерактивное тестирование студентов?

С континуумом сайт–приложение тесно связан континуум настольного и Web-приложения. Помимо всего прочего, многие приложения сначала разрабатываются как настольные системы и лишь потом переносятся в среду Интернета. Приложения, связанные с Web-транзакциями, требуют не менее тщательного моделирования и проектирования, чем соответствующая настольная система. Простота доступа и неизвестность пользовательской базы Web-приложений подразумевает очень высокую строгость разработки (например, в области безопасности).

К сожалению, с точки зрения разработки программного обеспечения эти два континуума немного противоречат друг другу. Преобразование Web-сайта в Web-приложение сопряжено с определенным риском того, что надежное проектирование на найдет своего воплощения в окончательном результате. Самая боль-

шая опасность заключается в том, что приложение не будет основано на продуманной архитектурной опоре и потеряет гибкость (см. раздел 4.1 главы 4 и раздел 6.2 главы 6).

Несмотря на указанные выше трудности, определение Web-приложения довольно очевидно. Одно из таких определений было дано в самом начале раздела 7.3: "Web-приложение — это Web-система, позволяющая пользователям реализовать бизнес-логику с помощью Web-браузера" (Conallen, 2000). В одном из наиболее известных средств разработки Web-приложений — системе Macromedia Dreamweaver — дается следующее определение Web-приложения: "Это коллекция Web-страниц, взаимодействующих с посетителями, друг с другом и разнообразными ресурсами Web–сервера, включая базы данных".

Из приведенных определений Web-приложения совершенно ясно, что момент, в который Web-сайт превращается в Web-приложение, определяется природой средств, обеспечивающих взаимодействие пользователя и Web-браузера. С точки зрения программирования разница между этими понятиями выражается в противоположности между *статическими* и *динамическими* страницами. Система Macromedia Dreamweaver разъясняет эту мысль так.

> Web-приложение — это Web-сайт, содержащий страницы с частично или полностью неопределенным содержанием. Окончательное содержание страницы определяется только после того, как посетитель запросит ее у Web-сервера. Поскольку окончательное содержание страниц изменяется от запроса к запросу в зависимости от действий посетителя, такие страницы называются динамическими.

Кстати, различие между статическими и динамическими Web-страницами также выражается в различиях между Web-сервером и сервером приложения. *Web-сервер* управляет статическими страницами. Он находит их по запросу браузера и посылает их ему. *Сервер приложения* управляет динамическими страницами. Он получает неполную страницу от Web-сервера, ищет на ней инструкции, обращается к базе данных за требуемой информацией, заносит новую информацию на страницу и передает ее Web-серверу.

7.3.2.2. Формы

Web-приложение выполняется целиком во *фрейме* браузера, в который входят строка заголовка фрейма, строка меню, панель кнопок и строка URL-адреса (рис. 7.20). Область содержания приложения состоит из собственных фреймов, включая фреймы навигации, командные кнопки и **формы**.

Форма (form) выводит на экран информацию, предназначенную для пользователя, а также дает ему возможность вводить данные и посылать собранную информацию серверу для дальнейшей обработки и вывода ее результатов на страницу. Как следует из рис. 7.20, форма состоит из *полей* (fields), в которые вводится информация. Ввод данных можно облегчить с помощью разных средств управления,

характерных для настольных приложений, например выпадающих или открытых списков, флажков и радиокнопок. Кроме того, формы можно создавать в виде папок с закладками, они могут содержать таблицы, выводить сообщения и т.д.

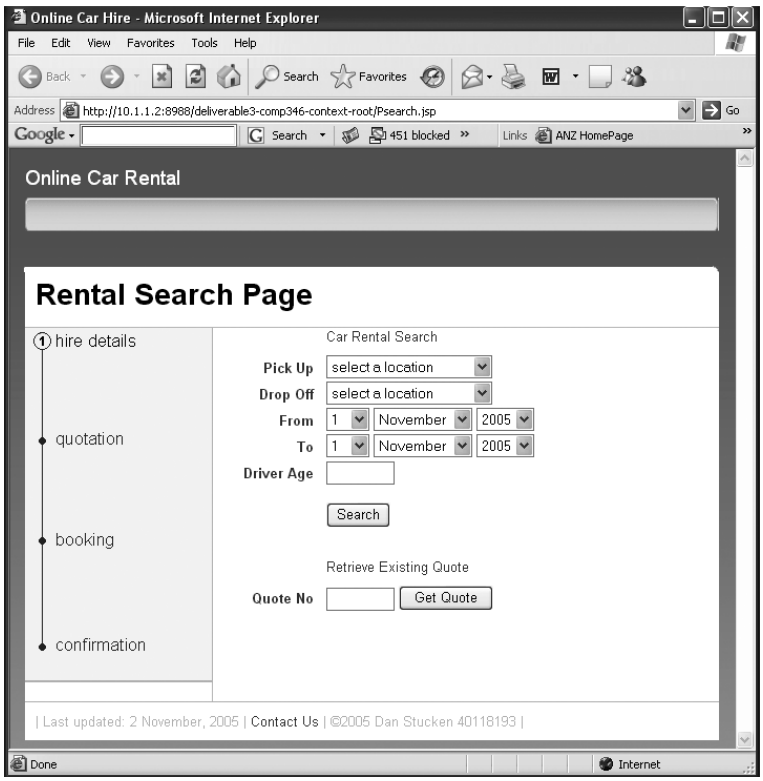


Рис. 7.20. Пример формы

Принципы проектирования полей в Web-формах аналогичны принципам разработки настольных приложений. Фоулер и Стэнвик (Fowler and Stanwik, 2004) сформулировали следующие рекомендации по разработке полей.

- Выберите тип поля. (Должно ли оно содержать произвольный текст? Должно ли оно содержать значение, заданное по умолчанию? Если да, то какое именно значение *следует* задать по умолчанию?)
- Выберите правильный размер поля. Как правило, размер рамки или окна, в котором размещено поле, должно совпадать с размером самого поля, но следует помнить, что поле можно прокручивать в горизонтальном направлении, поэтому его размер может быть меньше, чем длина строки.
- Выберите формат представления величин в полях, если это возможно, — текст выровняйте по левому краю, числа — по правому краю и т.д.

- Предусмотрите возможность управлять курсором в поле как с помощью клавиатуры, так и с помощью мыши (но помните, что пользователи, вводящие данные, предпочитают делать это с помощью клавиатуры и ненавидят, когда их вынуждают переключаться на работу с мышью, поэтому не делайте этого).
- Если логика приложения позволяет, предусмотрите возможность вырезания, копирования и вставки текста в полях.
- Присвойте полям соответствующие метки и выровняйте их слева или справа от полей (если поля имеют разную длину, а пользователи предпочитают переходить с поля на поле с помощью клавиши <Tab>, рекомендуется выравнивать метки слева (рис. 7.21)).

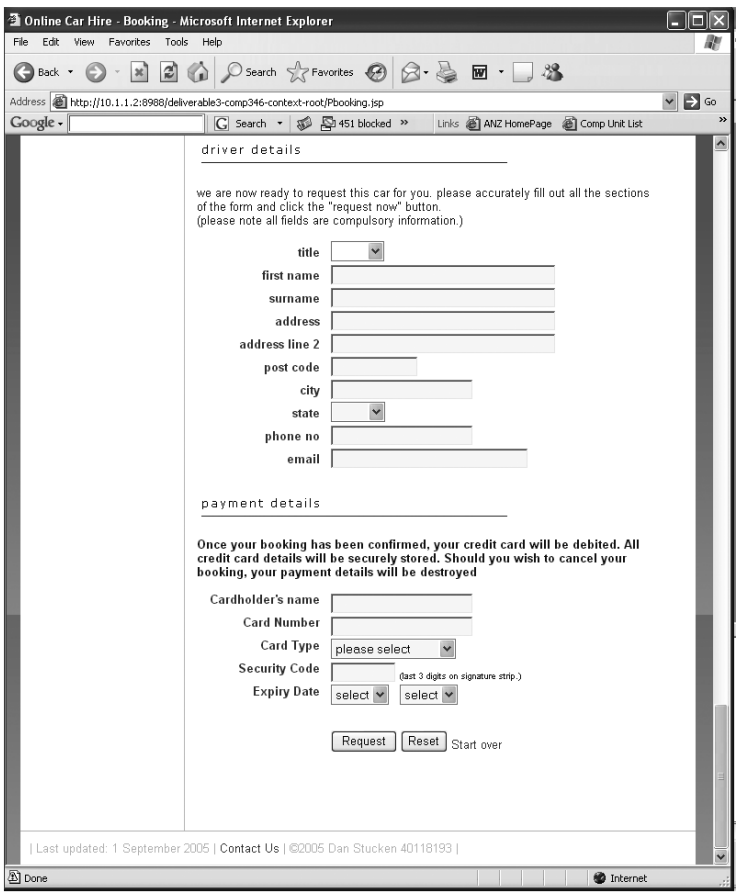


Рис. 7.21. Поля формы

- Разделяйте длинные формы на несколько более коротких или (если это невозможно) убедитесь, что они не содержат “ложных кнопок”, т.е. визуальных элементов между полями, которые можно принять за конец формы.

- Группируйте формы визуально понятным способом (не обязательно в отдельных рамках или окнах).
- По возможности используйте выпадающие списки, но заменяйте их открытыми всплывающими списками с полосами прокрутки, если количество пунктов слишком велико.
- По возможности используйте флажки и радиокнопки (это также позволит избежать ошибок при вводе данных).

Пример 7.4. Конвертация валют

Вернитесь к задаче 7, в которой описана система конвертации валют (см. раздел 1.6.7 главы 1), и к примерам, изложенным в главе 5. Предположим, что конвертер валют должен пересчитывать суммы, выраженные только в австралийских долларах, но в любую другую конвертируемую валюту. Кроме того, рассмотрите возможность конвертации разных видов денег, включая наличные и безналичные. Предположим, что результат конвертации должен быть выведен на отдельной Web-странице.

Разработайте Web-форму для Web-конвертера валют, руководствуясь перечисленными выше требованиями.

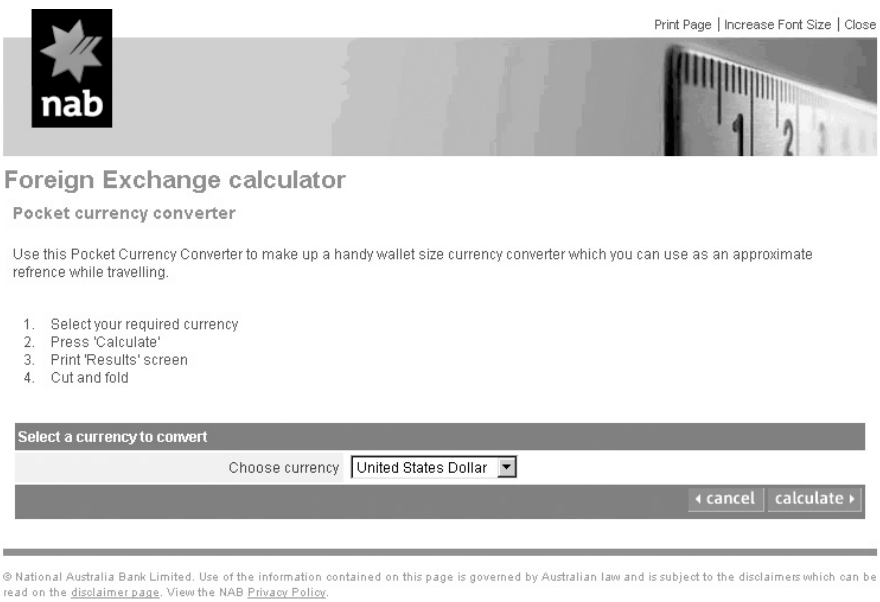


Рис. 7.22. Проект формы для ввода данных в систему конвертации валют

На рис. 7.22 показана Web-форма для конвертации валют, предоставляемая пользователю как услуга Национального австралийского банка (National Australia Bank — NAB) в соответствии с требованиями, перечисленными в примере 7.4. Эта форма состоит из трех полей и двух командных кнопок. Два поля из этих полей дополнены выпадающим списком для выбора соответствующих опций.

Пример 7.5. Конвертация валют

Вернитесь к примеру 7.4. Разработайте Web-форму для вывода результатов пересчета. Эта форма должна содержать обменный курс, сумму денег для конвертации и дату обмена. Кроме того, она должна позволять пользователю вернуться к форме для ввода данных, чтобы выполнить следующий обмен.

Форма для вывода результатов пересчета конвертером валют банка NAB в соответствии с требованиями, перечисленными в примере 7.5, показана на рис. 7.23.



Рис. 7.23. Проект формы для вывода результатов работы системы конвертации валют

Web-форма работает в браузере, и некоторые стандартные пункты меню и функции его панели инструментов могут оказывать на нее влияние, если не ограничить их средствами Web-приложения. Приложение должно ограничить следующие функции (Fowler and Stanwick, 2004).

- Кнопки для перехода вперед и назад по Web-страницам, если этот переход не контролируется приложением (такой переход может привести к потере данных, введенных пользователем, или прервать бизнес-операцию).
- Кнопки и пункты меню, закрывающие окно просмотра и приводящие к неконтролируемому выходу из приложения.

- Редактирование, загрузку и печать приложений, имеющих доступ к информации, доступной для ограниченного круга пользователей.

7.3.3. Проектирование навигации

Навигация по экранам графического пользовательского интерфейса (окнам, Web-страницам) происходит в результате либо действий пользователя, либо выполнения программы. В настольных приложениях навигация пользователя между окнами осуществляется с помощью пунктов меню, панелей инструментов, командных кнопок и нажатий клавиш. В Web-приложениях существуют аналогичные возможности, хотя визуальное представление может изменяться, в частности, пункты меню и панели инструментов, характерные для настольных приложений, в Web-приложениях, как правило, не используются. С другой стороны, активных ссылок (гиперссылок) для перехода в Web-приложении в настольных приложениях нет.

В любом случае навигация в рамках Web-приложения более дружественна по отношению к пользователю, чем в настольных системах. В них не существует главных и вторичных окон, — каждая Web-страница может предоставлять любую комбинацию возможностей для навигации: *меню* могут сосуществовать с *кнопками*, *ссылками* и *навигационными цепочками*. На рис. 7.19 показана Web-страница, содержащая большое количество меню (сопровождающихся объяснениями на панели содержания страницы) и несколько активных ссылок (текст, подчеркнутый на панели содержания). На рис. 7.20 показана Web-страница с активизированными кнопками и меню в левой части экрана. Эта страница иллюстрирует текущий шаг процедуры при оформлении проката автомобиля.

Навигация между Web-страницами приложения должна быть тщательно спланирована. Логика навигации должна быть ясной и понятной либо интуитивно, либо с помощью объяснений, доступных на панели навигации. Пользователь никогда не должен теряться в "гиперпространстве" Web-страниц.

Способы навигации весьма разнообразны и зависят от характера и сложности Web-приложения. Деловые приложения, связанные с транзакциями (см. рис. 7.20), порождают поток действий, связанных с последовательностью страниц. Они дают пользователю возможность исследовать Web-страницы на начальных стадиях процесса, когда он ищет услуги или товары, а позднее позволяют внести оплату в интерактивном режиме. Ввод данных в таких приложениях осуществляется на небольшом количестве довольно длинных страниц и сопровождается средствами навигации для быстрого ввода без перехода между страницами. Приложения, связанные с извлечением данных из баз данных, например библиотечные системы, предоставляют возможности поиска в соответствии с разными критериями, последовательного просмотра единиц хранения, сканирования содержимого выбранных единиц хранения и т.п. Некоторые приложения могут также содержать карту сайта, чтобы обеспечить соединение со всеми страницами.

7.3.3.1. Меню и ссылки

Меню и ссылки представляют собой два основных средства навигации между страницами. На рис. 7.24 показан фрагмент страницы, продемонстрированной на рис. 7.19, с указанием меню и ссылок. Меню и ссылки имеют одинаковый *аффорданс* (affordance). Этот неологизм используется для описания поведения, которое пользователь ожидает от элемента графического пользовательского интерфейса (Fowler and Stanwick, 2004). *Аффорданс ссылки* проявляется в том, что он переводит на другую страницу. *Аффорданс меню* также означает переход на другую страницу, но иногда пункт меню может подразумевать выполнение процесса, приписанного другой странице. Более того, меню может быть иерархическим, поэтому некоторые его пункты приводят не на следующую страницу, а к открытию *подменю* (рис. 7.25).

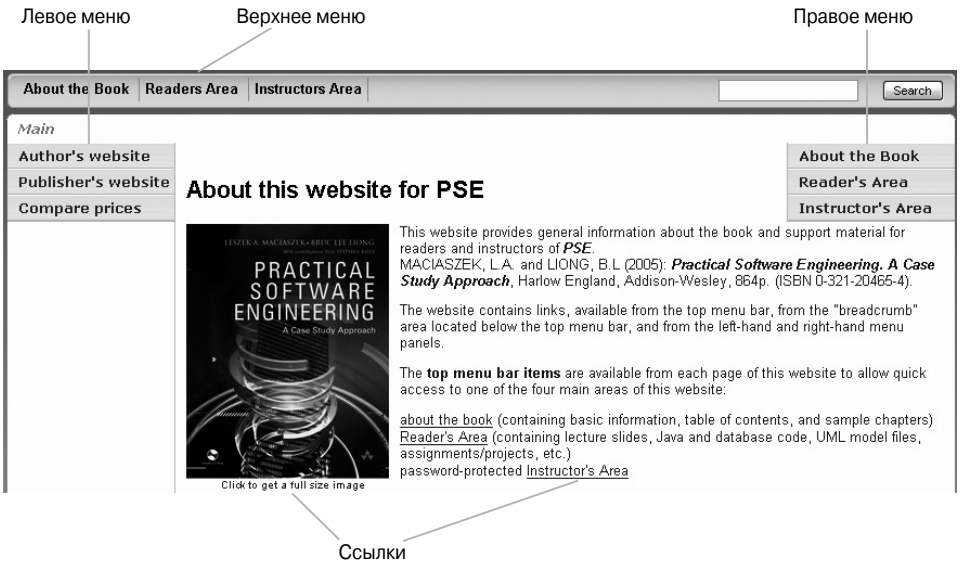


Рис. 7.24. Меню и ссылки

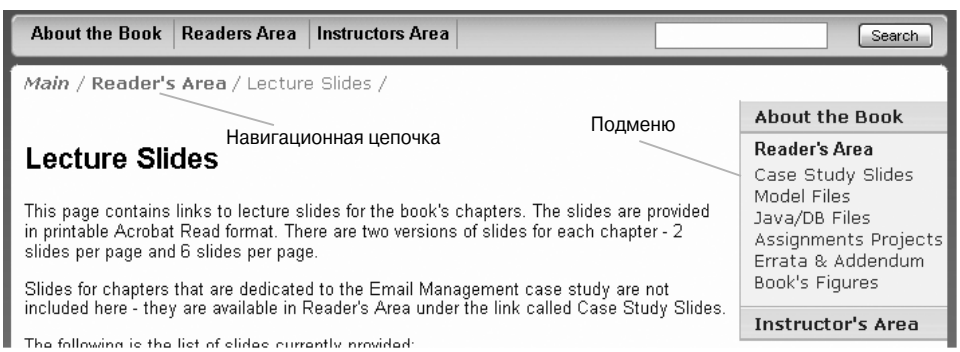


Рис. 7.25. Навигационная цепочка и подменю

Следует различать следующие элемента Web-страницы:

- верхнее меню;
- левое меню;
- правое меню.

Верхнее меню (top menu) обычно используется для навигации по Web-сайту. Правила проектирования левого и правого меню более гибкие. На рис. 7.24 *левое меню* (left-hand menu) используется для навигации по страницам, находящимся вне пределов Web-сайта, а *правое меню* (right-hand menu) выводит на экран страницы, находящиеся в пределах Web-сайта. Может возникнуть вопрос: "А почему бы не поменять эти меню местами?" Такой вопрос можно подкрепить как минимум двумя соображениями. Многие приложения (в противоположность Web-сайтам) должны иметь единственный выход и не позволять переходы назад, или на внешние Web-сайты, или в другие приложения. Более того, исследование того, как люди сканируют страницу, показало, что обычный пользователь просматривает страницу слева направо, затем смотрит наверх, в центр и вправо (Fowler and Stanwick, 2004). Следовательно, меню, выражающее предназначение Web-сайта или приложения, следует размещать слева, куда пользователь посмотрит в первую очередь.

7.3.3.2. Навигационные цепочки и панели

Существует особый вид меню, основное предназначение которых — информировать, а не обеспечивать переход между страницами. Это — *навигационные цепочки* (breadcrumbs) и *навигационные панели* (navigation panels), сообщающие о местоположении текущей страницы в последовательности страниц и использующиеся для перехода на нее.

Область навигационных цепочек (breadcrumb area) обычно размещается в верхней части страницы, непосредственно под заголовком меню (см. рис. 7.25). Эта область состоит из набора связанных меток, сообщающих пользователям, на какой странице они в данный момент находятся. По мере перехода пользователя со страницы на страницу пункты навигационной цепочки изменяются. Как правило, они позволяют пользователю возвращаться назад на страницы, которые он посетил до этого.

Навигационные панели (рис. 7.26) похожи на навигационные цепочки, но обычно они используются в приложениях, связанных с выполнением транзакций, показывая все этапы процедуры и не допуская возвращения на предыдущие стадии, пока не будет отменено выполнение операции. Часто переход на следующую страницу осуществляется щелчком на кнопке предыдущей страницы, а не на кнопке навигационной панели. Кроме того, расположение навигационной панели на странице зависит от предпочтений проектировщика, так что она не обязательно размещается в верхней части экрана.

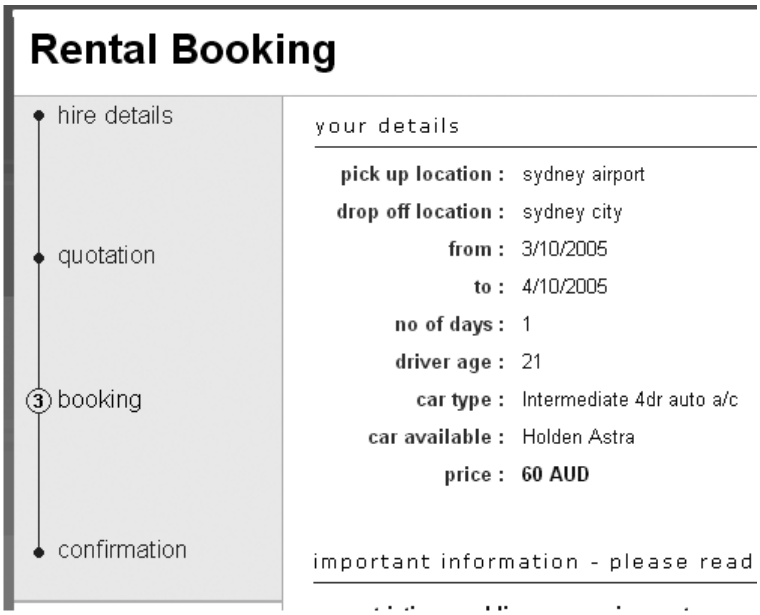


Рис. 7.26. Навигационная панель

7.3.3.3. Кнопки

В настольных приложениях вычислительные действия активизируются с помощью пунктов меню или кнопок, а иногда и тех и других. В Web-приложениях основным средством для вызова действия являются *кнопки* (рис. 7.27), а пункты меню используются лишь для навигации. *Аффорданс кнопки* — делать нечто после щелчка мышью (Fowler and Stanwick, 2004).

Принципы разработки кнопок для настольных приложений распространяются и на Web-страницы. Эти принципы перечислены ниже (Fowler and Stanwick, 2004).

- Если кнопки образуют группу или содержат примерно одинаковое количество текста, их размеры должны быть одинаковыми.
- Кнопки должны быть сгруппированы в отдельной области страницы, отдельно от полей для ввода данных.
- Если страница больше, чем окно, то кнопки в верхней и нижней части экрана должны повторяться.
- Кнопки в окнах с вкладками (tab frames) должны разделяться на кнопки, относящиеся к отдельной вкладке, и кнопки, относящиеся ко всему окну.
- Кнопки должны программироваться так, чтобы не реагировать на многократные щелчки нетерпеливых пользователей.

- Имена кнопок должны точно отражать действие, которые они выполняют (в частности, указывать, когда данные будут сохранены в базе данных, а когда они хранятся временно и могут быть аннулированы пользователем).

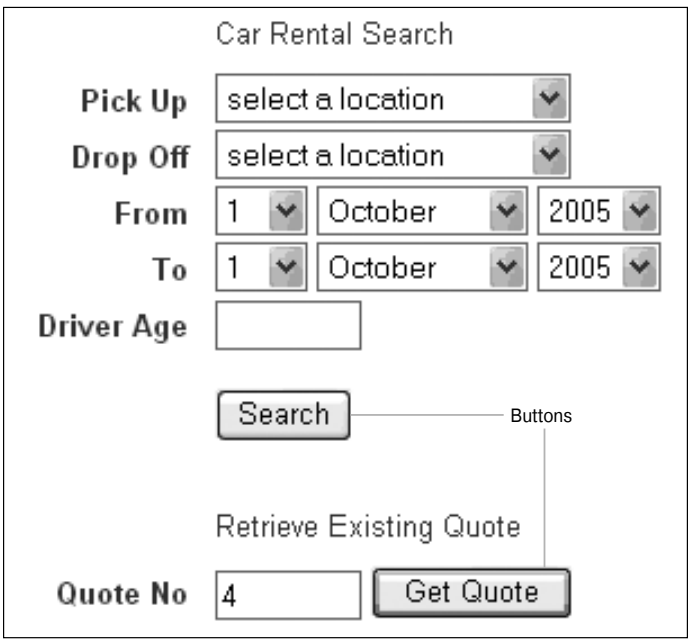


Рис. 7.27. Кнопки

7.3.4. Использование моделей графических пользовательских интерфейсов для Web-проектирования

Для создания прикладных программ разработчики используют системное программное обеспечение. В качестве классического примера укажем, что для хранения персистентных данных разработчики обычно используют системы управления базами данных. За исключением "экзотических" ситуаций создание специфического механизма для хранения персистентных данных даже не рассматривается. Аналогично, системы планирования ресурсов предприятия (enterprise resource planning — ERP) обеспечивают стандартное решение для ведения бухгалтерских книг, учета кадров и производственных процессов.

Существует множество программных сред, предназначенных для разработки приложений с графическим пользовательским интерфейсом. Интегрированные среды разработки (integrated development environment — IDE), используемые программистами, представляют собой именно *окружение* (environment), а не язык программирования. Современное программирование в большей мере представ-

ляет собой деятельность, связанную с повторным использованием готовых компонентов, чем с разработкой нового исходного кода. Это порождает особые ожидания и выдвигает новые требования не только к программистам, но и к проектировщикам системы, архитекторам и даже аналитикам.

Под *средой графического пользовательского интерфейса* (GUI framework) мы подразумеваем любую технологию, библиотеку программного обеспечения и другие программы, которые разработчики могут использовать при разработке GUI-интерфейса. Обычными средами в этой категории являются библиотеки Swing, Java Server Faces, Struts, Spring и др. Они принимают на себя часть обязанностей по реализации приложения, тем самым уменьшая сложность прикладного программного обеспечения. Конкретной целью разработки должно быть использование технологий, не только облегчающих программирование, но и позволяющих создавать системы в соответствии с выбранным архитектурным стилем. Оказывается, для того чтобы гарантировать достижение наиболее важных архитектурных целей и выполнение принципов разработки, некоторые технологии могут быть обязательными.

7.3.4.1. Дилемма MVC

Одной из наиболее сложных архитектурных проблем является разделение и исключение циклов между объектами `Presentation` (Представление) и `Control` (Управление) в программируемом клиенте и Web-приложении. Этот вопрос сложен, поскольку интерактивный стиль коммуникации между пользователем и системой создает естественный цикл обработки — объект `Presentation` посылает запрос на услуги объекта `Control`, а объект `Control` выбирает объект `Presentation`, который должен получить ответ.

Возможно, нет ничего удивительного в том, что соответствующие технологии разделяются на централизованные (известные как *Model 1*) и децентрализованные (*Model 2*). В рамках технологий *Model 1* каждый объект `Presentation` сочетается с объектом `Control` (как в технологиях Java Swing или Microsoft Foundation Class (MFC)). В рамках технологий *Model 2*, представленных моделью Core PCBMER, существует физическое разделение объектов `Presentation` и `Control`. Следовательно, в Web-приложениях страницы JSP размещаются на уровне `Presentation`, а сервлеты — на уровне `Control`. Однако сам по себе разрыв связей не дает большого выигрыша, если не исключить циклические зависимости, сделав страницы JSP независимыми от сервлетов. Поскольку страницы JSP являются HTML-страницами, они не могут реализовать интерфейс или подписаться на события. Следовательно, перед нами возникает задача, как решить эту проблему в рамках подхода MVC (Model–View–Controller) (см. раздел 4.1.1 главы 4).

Как показано на рис. 7.28, в решении примера 7.6 обработка начинается, когда Web-страница Request принимает ввод пользователя, создает экземпляр класса `Calculator` и запрашивает операцию `calculate()` для вычисления обмени-

ваемой суммы. Для того чтобы выполнить вычисления, объект `Calculator` должен вызвать процедуру `getRate()` из базы данных. Эта задача выполняется объектом `Query`. Получив обменный курс, объект `Calculator` создает экземпляр `Bean` и вычисляет его содержимое, предназначенное для вывода на Web-страницу Result. Затем объект `Calculator` создает объект `Result` и передает ему ссылку на объект `Bean`. Это дает объекту `Result` возможность получить данные у объекта `Bean` и вывести их на экран. Запрос пользователя на выполнение операции `startOver()` приводит к возврату управления Web-странице Request.

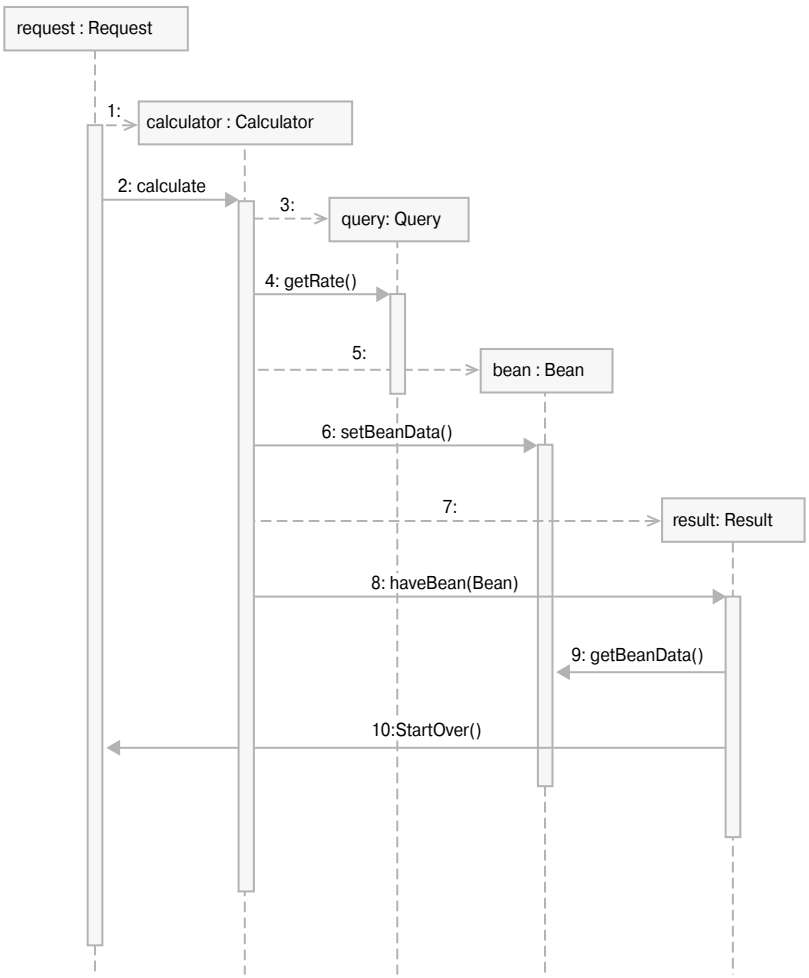


Рис. 7.28. Диаграмма последовательностей для исходного проекта валютного конвертера

Пример 7.6. Конвертация валют

Вернитесь к примерам 7.4 и 7.5 из раздела 7.3.2.2. Предположим, что для реализации функциональных свойств конвертера валют нам необходимы Web-страницы Request (Запрос) и Result (Результат) (см. рис. 7.22 и 7.23). Конвертацию должен делать объект `Calculator`, обменный курс должен получать объект `Query`, а содержать результат для вывода на Web-страницу Result — объект `Bean`. Постройте диаграмму последовательностей для описанного выше сценария и объясните ее.

На рис. 7.29 показана диаграмма классов, построенная для диаграммы последовательностей, приведенной на рис. 7.28. На этой диаграмме существует опасная циклическая зависимость между классами `Request`, `Calculator` и `Result`. Для конкретной ситуации, представленной на рис. 7.28, фактическое количество соединительных связей равно 6. Однако этот показатель нельзя использовать в качестве индикатора структурной сложности (см. раздел 6.2.1.2 главы 6). Проект представляет собой сетевую структуру, показатель *кумулятивной зависимости классов* (CCD) которой вычисляется по формуле 7.1 (вытекающей из уравнения 6.3 раздела 6.2.1.2.1):

$$_{net}CCD = n(n-1) = 5 \times 4 = 20. \quad (7.1)$$

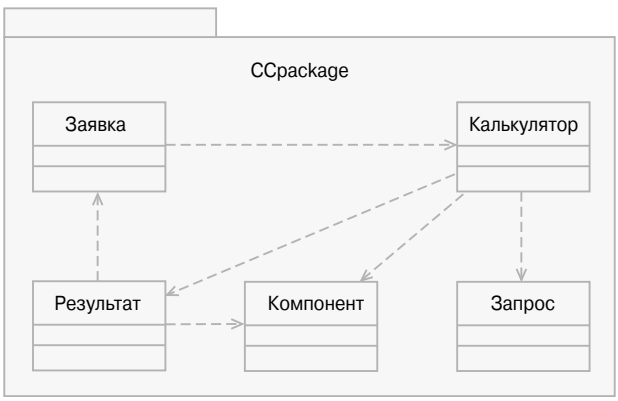


Рис. 7.29. Диаграмма классов для исходного проекта валютного конвертера

Пример 7.7. Конвертация валют

Вернитесь к примеру 7.6. Постройте диаграмму зависимости между классами для диаграммы последовательностей, показанной на рис. 7.28. Что вы видите? Какова структурная сложность этого проекта?

7.3.4.2. Технология Struts

Проект валютного конвертера, представленный выше, совершенно неудовлетворителен. Для того чтобы улучшить его, необходимо воспользоваться специальной средой для разработки графического пользовательского интерфейса, например Jakarta Struts (Sam-Bodden and Judd, 2004).

На рис. 7.30 показано, как с помощью технологии **Struts** можно создать простое Web-приложение. Работа этого приложения начинается с JSP-страницы PX. В файле конфигурации `web.xml` среда Struts определяет класс `ActionServlet`, загружаемый при запуске приложения. Этот класс сконфигурирован так, чтобы принимать запросы. Сервлет работает как маршрутизатор для пакета `Controller` (Контроллер) и делегирует специфические функциональные свойства классам пакета `Action`. Среда Struts инкапсулирует запросы в объекте `ActionForm`, реализованном в виде компонента JavaBean. Соответствие между JSP-страницами, действиями и формами поддерживается главным конфигурационным файлом `struts-config.xml`.

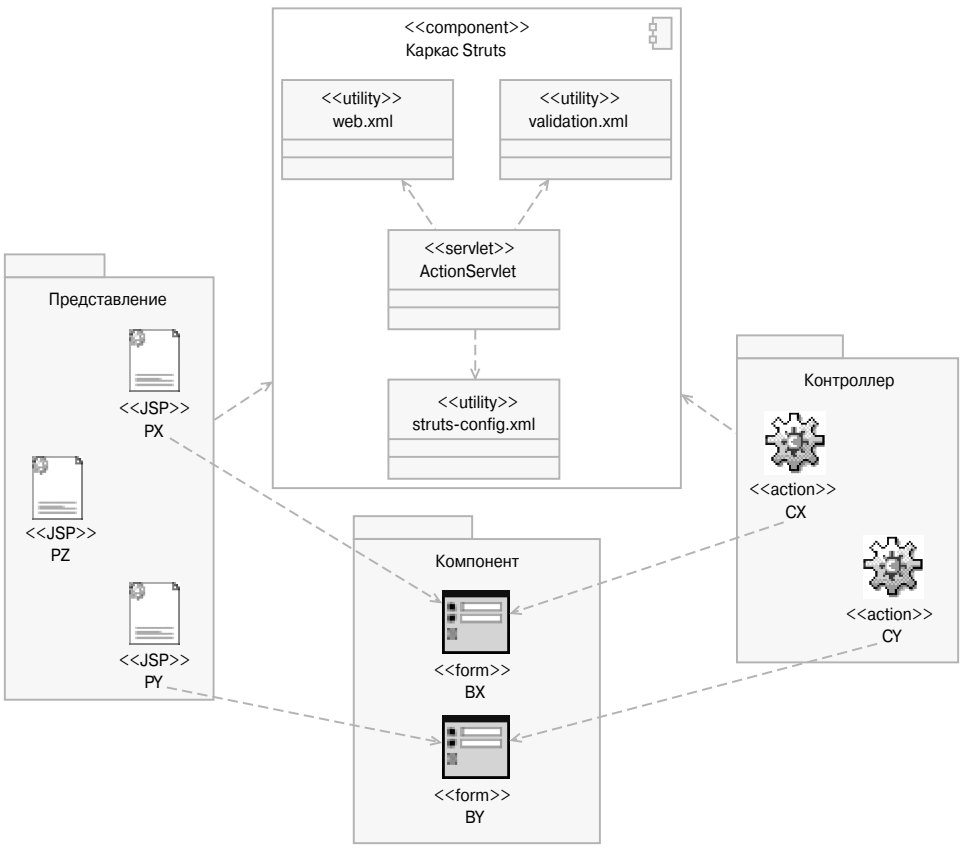


Рис. 7.30. Применение технологии Struts

Кроме того, среда Struts обеспечивает проверку корректности входных данных. Код для проверки можно разместить в методе `validate()`, который вызывается до вызова метода `execute()` одного из классов пакета `Action`. Файл конфигурации `validation.xml` информирует среду Struts о том, какому объекту передать управление, чтобы вывести на экран сообщение об ошибке (например, объекту `PZ`), если метод `validate()` обнаруживает неправильные данные, заданные при вводе.

Если проверка входных данных прошла успешно, объект `ActionServlet` передает управление одному из объектов пакета `Action`. Файл `struts-config.xml` связывает классы пакета `Action` с классами пакета `ActionForm`. Например, объект класса `BY` может содержать данные, установленные объектом класса `CY` и подготовленные для отображения на JSP-странице. После завершения работы объект класса `CY` передает сообщение, используемое для определения JSP-страницы, которой следует передать контроль (например, `PY`), или JSP-страницы, которая должна вывести сообщение об ошибке (например, `PX`). (Это сообщение возвращается объекту `ActionServlet`, который с помощью файла `struts_config.xml` определяет компонент представления, являющийся адресатом этого сообщения.)

Кроме элементов, представленных на рис. 7.30, среда Struts содержит много других функциональных свойств, облегчающих разработку Web-приложений, которые лучше соответствуют модели MVC. Для замены Java-кода на JSP-страницах и реализации функциональных свойств сервлетов используются библиотеки дескрипторов. Для упрощения общих визуальных элементов и эскизов страниц среда Struts Tiles предоставляет отдельные схемы JSP-страниц.

В целом среда Struts представляет собой мощную технологию, позволяющую решить проблему зависимостей в модели MVC и в архитектурных моделях, построенных в рамках подхода MVC, например PCBMER. Рис. 7.31 обобщает рис. 7.30 и показывает, что технология Struts позволяет практически полностью исключить зависимости между пакетами `Presentation` и `Controller`. Точнее говоря, эти зависимости существуют, но управляются мета-уровнем среды Struts, обеспечивающим декларативные средства для определения зависимостей и берущим на себя ответственность за управление потоком логики приложения. Соответственно, как показано на рис. 7.31, пакеты `Presentation` и `Controller` зависят от среды Struts, а не друг от друга.

Пример 7.8. Конвертация валют

Вернитесь к примерам 7.6 и 7.7. Используя технологию Struts, разработайте проект конвертера валют, соответствующий принципам модели PCBMER, и постройте соответствующую диаграмму зависимости между классами. Компоненты среды Struts можно использовать в обобщенном смысле, как показано на рис. 7.31. Распределите классы по уровням PCBMER (уровень `Entity` использовать не обязательно).

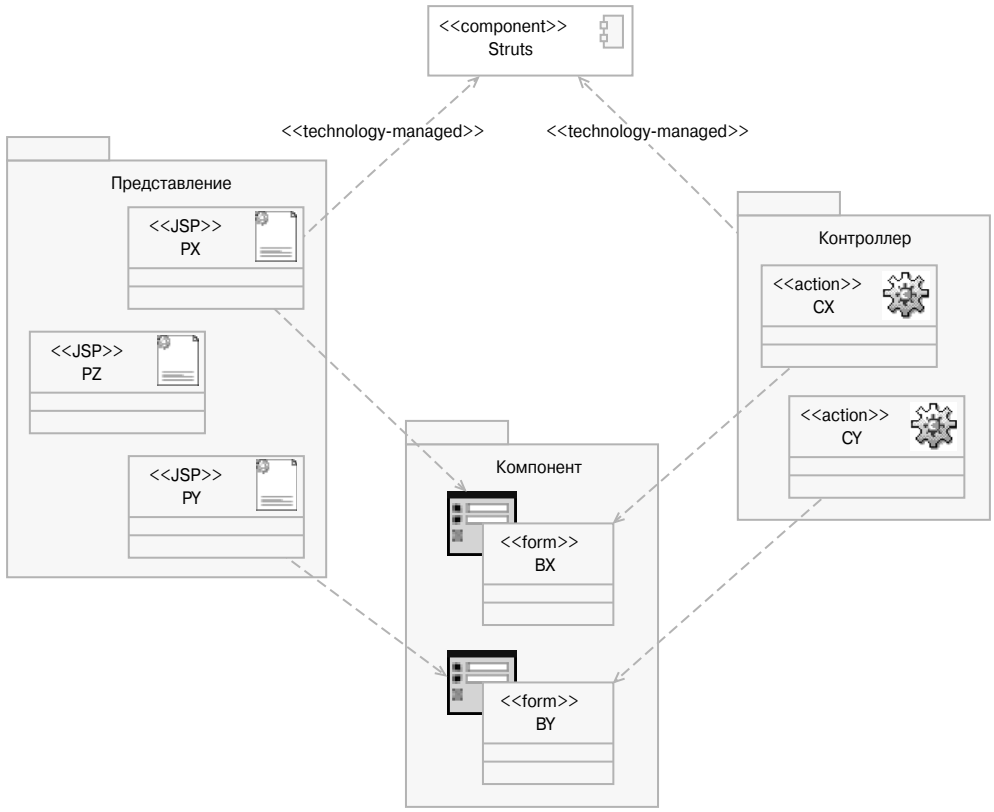


Рис. 7.31. Исключение восходящих и циклических зависимостей с помощью технологии Struts

В качестве решения примера 7.8 на рис. 7.32 показан проект конвертера валют, соответствующий принципам модели PCBMER. Технология Struts предоставляет услуги посредника между пакетами `Presentation` и `Controller`, эффективно заменяя *архитектурно-управляемые зависимости* зависимостями, *управляемыми технологией*.

Итоговый показатель CCD для проекта, созданного с помощью технологии Struts, вычисляется по следующей формуле (вытекающей из уравнения (6.4), приведенного в разделе 6.2.1.2.2 главы 6):

$$_{holarchy}CCD = \sum_{i=1}^{n} \frac{size(l_i) \times (size(l_i) - 1)}{2} + \sum_{i=1}^{n} \sum_{i=1}^{n} (size(l_i) \times (size(p_j(l_i)))) = 2 + 8 = 10. \quad (7.2)$$

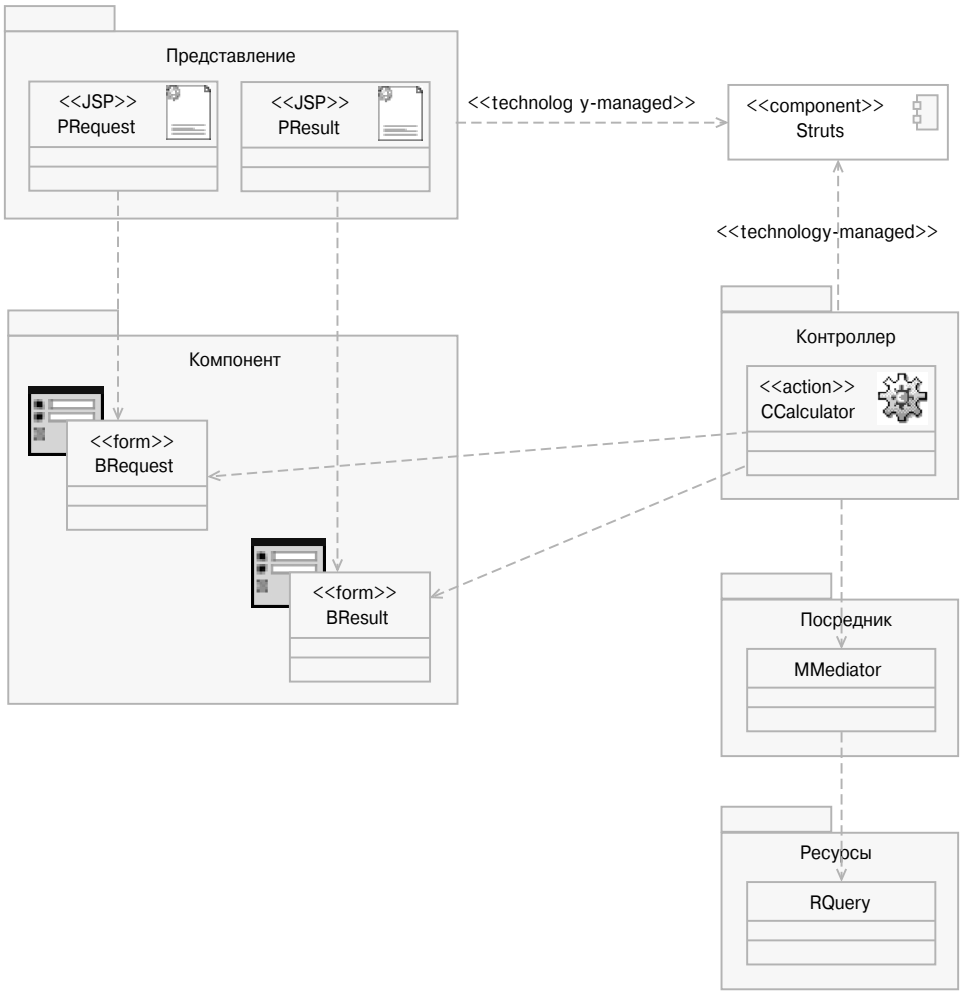


Рис. 7.32. Диаграмма зависимостей между классами в проекте конвертера валют, созданном с помощью технологии Struts

Контрольные вопросы 7.3

КВ1. Какие компоненты графического пользовательского интерфейса предназначены для программирования пользовательских событий в Web-приложениях?

КВ2. Какой механизм поддержки связи между Web-клиентом и сервером является самым простым?

КВ3. Назовите аффорданс ссылки.

7.4. Моделирование навигации в графическом пользовательском интерфейсе

В целом проблемы, связанные с моделированием навигации в графическом пользовательском интерфейсе, аналогичны проблемам, возникающим при создании настольных систем и Web-приложений. Разница заключается лишь в деталях. Следовательно, в последующем обсуждении можно использовать терминологию настольных приложений.

С точки зрения пользователя, приложение представляет собой набор взаимодействующих экранов (окон или Web-страниц). Задача проектировщика графического пользовательского интерфейса заключается в организации зависимостей между экранами в рамках связной и понятной структуры. Пользователь никогда не должен чувствовать себя потерянным среди открытых экранов.

В идеале ссылка из главного окна на верхнее вторичное окно, открытое в текущий момент, должна быть маршрутом, а не иерархией. Для этого вторичное окно делают *модальным* по отношению к предыдущему окну.

Проектирование графического пользовательского интерфейса направлено на помощь пользователю. В то же время основным средством, позволяющим объяснить возможности приложения, остаются *меню* и *панели инструментов*. Выпадающие меню дают пользователям возможность выполнять команды, а всплывающие меню косвенно объясняют зависимости между окнами.

Графическое представление окон графического пользовательского интерфейса — с помощью прототипов или других инструментов для подготовки эскизов — ничего не говорит о том, как на самом деле пользователь *перемещается* по окнам. По этой причине нам по-прежнему необходимо проектировать *навигацию по окнам* (window navigation). Модель навигации по окнам должна состоять из диаграмм, визуализирующих контейнеры и компоненты экранов и показывающих, каким образом пользователь может переходить из одного окна в другое.

7.4.1. Раскадровка работы пользователя

Описание структуры и логики, лежащих в основе визуального представления приложения и навигации, иногда называют *раскадровкой* (storyboarding) (Sklar, 2006). Язык UML имеет мало возможностей для непосредственной раскадровки, но его профили устраняют этот недостаток. Один из таких профилей называется *раскадровкой работы пользователя* (UX storyboards) (Heumann, 2003).

Если бы не было раскадровок UX или аналогичных моделей навигации по окнам, разработчик должен был бы расширить прецеденты использования, включив в них прототипы окон. Именно так мы делали в предыдущих главах, когда чувствовали, что описание прецедента использования остается неточным без представления прототипов экранов.

Моделирование с помощью раскадровок UX состоит из пяти этапов (Heumann, 2003).

1. *Добавление нового действующего лица в прецедент использования.* Этот этап включает в себя определение, насколько грамотно действующее лицо (пользователь) использует компьютер (см. раздел 3.1.1 главы 3), насколько велики его знания в предметной области и как часто он использует систему.
2. *Добавление характеристик удобства в прецедент использования.* Удобство (см. раздел 7.1.2.6) представляет собой нефункциональное требование, обычно выражающееся в виде системных ограничений, указанных в дополнительных спецификациях (см. раздел 2.6.4 главы 2). Характеристики удобства включают в себя полезные подсказки (облегчающие работу с графическим пользовательским интерфейсом или его реализацию) и любые строгие требования, которым должна соответствовать система (время реакции, приемлемый уровень ошибок, время обучения и т.д.).
3. *Идентификация элементов UX.* Этот этап связан с идентификацией контейнеров и компонентов графического пользовательского интерфейса. Для элементов UX используется модель классов со специальными стереотипами.
4. *Моделирование потоков в прецеденте использования с помощью элементов UX.* Этот этап подразумевает моделирование взаимодействия элементов интерфейса с помощью элементов UX (см. раздел 7.4.3). Для изображения взаимодействия пользователя с экранами графического пользовательского интерфейса или между экранами GUI используются диаграммы последовательностей и классов языка UML.
5. *Моделирование навигации по экранам в прецеденте использования.* Этот этап связан с моделированием структурного взаимодействия на основе элементов UX (подробнее об этом — в разделе 7.4.4). Для изображения ассоциаций, по которым происходит навигация по элементам UX, используются стереотипизированные диаграммы классов языка UML.

В целом раскадровка UX предназначена для того, чтобы представить проектирование графического пользовательского интерфейса как неотъемлемую часть разработки системы. Она описывает аспекты моделирования, характерные для проектирования графического пользовательского интерфейса, и связана со следующими элементами (Kozaczenski and Thario, 2003).

- Экраны презентации.
- Экранные события, на которые должна реагировать система.
- Данные, которые система должны выводить на экран.
- Данные, которые пользователь должен вводить на экране для дальнейшей обработки.

- Декомпозиция экрана на более мелкие области, которые должны управляться по отдельности.
- Переход (навигация) между экранами.

Профиль UX раскадровок предусматривает несколько стереотипов для классов. Основными стереотипами являются ключевые слова «`screen`», «`input form`» и «`compartment`». Эти стереотипы находятся на относительно высоком уровне абстракции. Более полный список стереотипов включает в себя другие элементы UX, классифицированные по видам структурного и функционального взаимодействия. В частности, возможен такой список.

1. Структурные элементы UX.
   - Главное окно.
   - Панель в главном окне.
   - Окно просмотра строк.
   - Окно просмотра деревьев.
2. Вторичное окно.
   - Диалоговое окно.
   - Окно сообщения.
   - Папка с закладками.
3. Оконные данные.
   - Текстовое окно.
   - Комбинированное окно.
   - Окно счетчика.
   - Столбец.
   - Строка.
   - Группа полей.
4. Функциональные элементы UX.
   - Пункт выпадающего меню.
   - Пункт всплывающего меню.
   - Кнопка панели инструментов
   - Командная кнопка.
   - Двойной щелчок.
   - Опция списка.
   - Клавиша.
   - Функциональная клавиша.
   - “Быстрая” клавиша.
   - Кнопка прокрутки.
   - Кнопка закрытия окна.

7.4.2. Моделирование элементов UX

Профиль UX предусматривает лишь несколько стереотипов, представляющих собой простейшие элементы UX. Более сложным является стереотип *пакета* «storyboard». Этот стереотип определяет пакет, содержащий раскадровку UX.

Классы UML можно стереотипизировать с помощью ключевых слов, приведенных ниже.

- «screen». Абстракция экрана определяет окно, выводимое на экран.
- «input form». Стереотип, представляющий контейнер формы, с помощью которого пользователь может взаимодействовать с системой, вводя данные или выполняя определенные действия. Этот контейнер может быть классом, производным от одного из классов библиотеки Java Swing, например JInternalFrame, JTabberFrame, JDialog или JApplet.
- «compartment». Стереотип, представляющий любую область экрана, которую можно использовать в многооконном режиме. Ею может быть, например, панель инструментов.

Классы UX содержат динамическое содержание графического пользовательского интерфейса (поля на экране) и любые действия, связанные с экранами, формами для ввода данных и отделениями экрана. В профиле UX определяются дескрипторы (см. раздел 5.1.1.3 главы 5), связанные с *полями* (Kozaczynski and Thario, 2003). Другие дескрипторы могут добавить разработчики раскадровок. Среди этих дескрипторов наиболее интересными являются следующие:

- *editable* (редактируемое) — указывает, может ли пользователь редактировать поле или нет;
- *visible* (видимое) — указывает, может ли поле выводиться на экран или оно должно быть скрыто от пользователя (но оставаться доступны для программы);
- *selectable* (выбираемое) — указывает, можно ли выбрать поле (в зависимости от этого поле либо подсвечивается, либо изображается активным).

Например, поле может иметь следующие значения дескрипторов: {editable = true, visibility = visible} или {editable = false, visibility = hidden}. И наоборот, тот факт, что поле является видимым, можно указать, пометив его с помощью пиктограммы открытой видимости (знак "плюс" (+) перед именем поля). Для того чтобы показать, что поле является скрытым, перед его именем указывается маркер закрытой видимости (знак "минус" (–)).

Существуют две категории *действий*, перечисленных в классах UX, — действия пользователя и внешние действия (Heumann, 2003). *Пользовательскими* (user actions) называются любые действия, совершаемые пользователем по отношению к графическому интерфейсу. *Внешними* (environmental actions) называются любые действия, совершаемые системой по отношению к графическому интерфейсу.

Одним из основных внешних действий является переход на новый экран. Профиль UX рекомендует отмечать внешние действия символом доллара, указанным перед именем.

Пример 7.9. Затраты на рекламу

Вернитесь к рис. 7.5 (см. раздел 7.2.1). Идентифицируйте основные элементы моделирования UX (стереотипы на уровне классов), использованные на этом рисунке, для представления содержания экрана.

На рис. 7.33 показана модель классов UX, представляющая содержание экрана на рис. 7.5 для примера 7.9. Этот экран состоит из четырех отделений и одной формы для ввода данных.

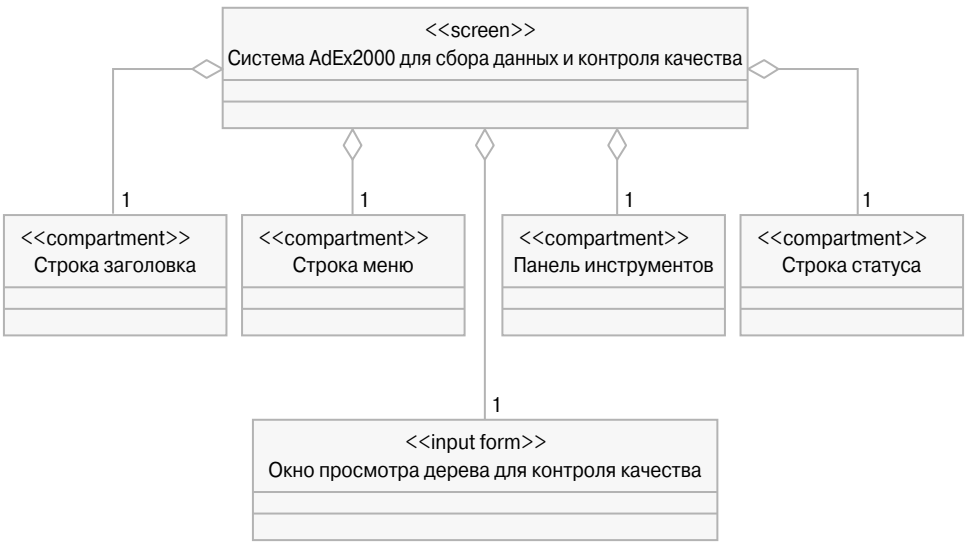


Рис. 7.33. Элементы UX для главного окна системы для оценки затрат на рекламу

7.4.3. Функциональная кооперация UX

Зная элементы UX на уровне классов, можно начинать моделирование потока событий UX, происходящих с этими элементами. Поток событий UX отражает функциональный аспект коммуникации UX. Соответственно, для представления потока событий UX используются диаграммы взаимодействий UML (диаграммы последовательностей и/или кооперации).

Пример 7.10. Управление взаимоотношениями с заказчиками

Вернитесь к рис. 7.3 (см. раздел 7.1.1) и 7.6 (см. раздел 7.2.1). На рис. 7.3 показано окно Update Organization, содержащее поле Status с выпадающим списком. Это поле определяет статус организации с точки зрения управления взаимоотношениями с заказчиками (т.е. является ли организация потенциальным, бывшим или текущим клиентом).

Список возможных вариантов статуса организации короток и фиксирован (рис. 7.34), но иногда возникает необходимость модифицировать его (вставить, изменить или удалить пункт). В таком случае пользователь может вывести на экран этот список (возможно, для изменения или удаления), дважды щелкнув на опции Organization Status в окне просмотра деревьев в левой части рис. 7.6. Однако, для того чтобы ввести новый статус организации, пользователь должен выбрать (подсветить) опцию Organization Status в окне просмотра деревьев, а затем выполнить действие Insert (Вставить), выбрав соответствующий пункт меню (пункт Record (Запись) на рис. 7.6) или щелкнув на кнопке панели инструментов (она представлена небольшой пиктограммой на рис. 7.6).

Идентифицируйте элементы UX и разработайте функциональную кооперацию UX для процесса вставки нового статуса организации. Обычно моделирование с помощью раскадровок UX позволяет избежать создания прототипов окон, но для облегчения решения этого примера на рис. 7.35 показано окно Insert Organization Status.

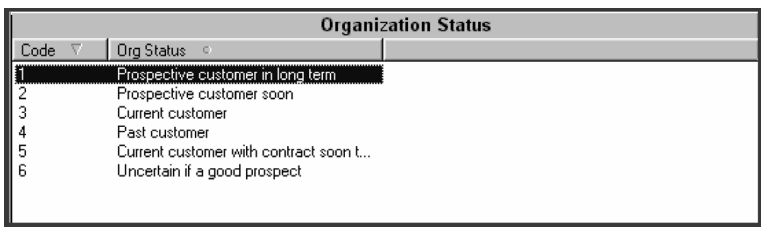


Рис. 7.34. Статус организации в системе управления взаимоотношениями с заказчиками. Источник: компания Nielsen Research, Сидней, Австралия

Элементы UX для примера 7.10 включают в себя три класса, показанных на рис. 7.33: `Quality Control Tree Browser` (Окно просмотра деревьев для контроля качества), `Menu Bar` (Строка меню) и `Toolbar` (Панель инструментов). Остальные классы, необходимые для решения примера 7.10, представлены в модели классов UX на рис. 7.36. Эти классы связаны с окном, предназначенным для вставки данных о новом статусе организации (см. рис. 7.35).

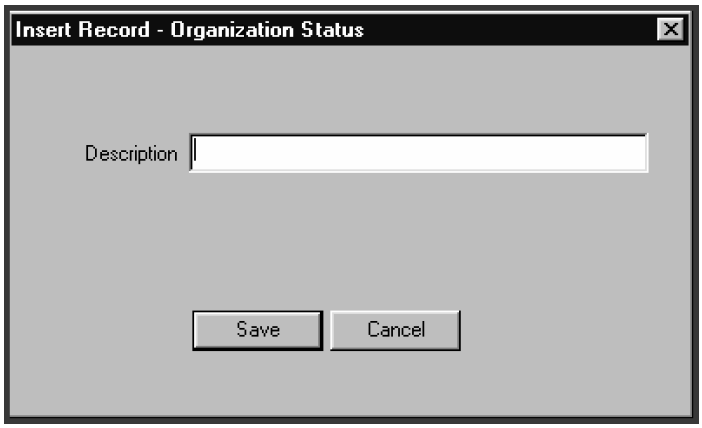


Рис. 7.35. Окно для вставки статуса организации в системе управления взаимоотношениями с заказчиками. Источник: компания Nielsen Research, Сидней, Австралия

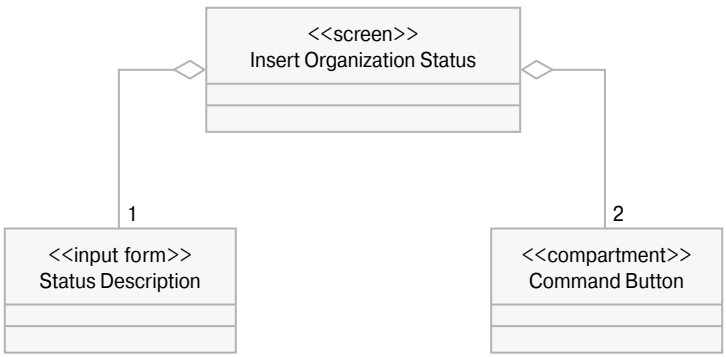


Рис. 7.36. Элементы UX для диалогового окна в системе управления взаимоотношениями с заказчиками

На рис. 7.37 показана диаграмма последовательностей для *функциональной кооперации UX*, необходимой для удовлетворения требований, предъявленных в примере 7.10. Эта модель показывает ветвление процесса из-за действий, предпринятых пользователем. Например, на ней показана возможность открытия окна Insert Organization Status либо с помощью класса `Menu Bar` (событие 2), либо с помощью класса `Toolbar` (событие 3). Потоки разветвленных событий могут воссоединиться в определенной точке (например, в точках, соответствующих событиям 4 и 5). Обратите внимание на то, что после щелчка на кнопке Save (Сохранить) или Cancel (Отмена) откроется окно просмотра деревьев Quality Control Tree Browser.

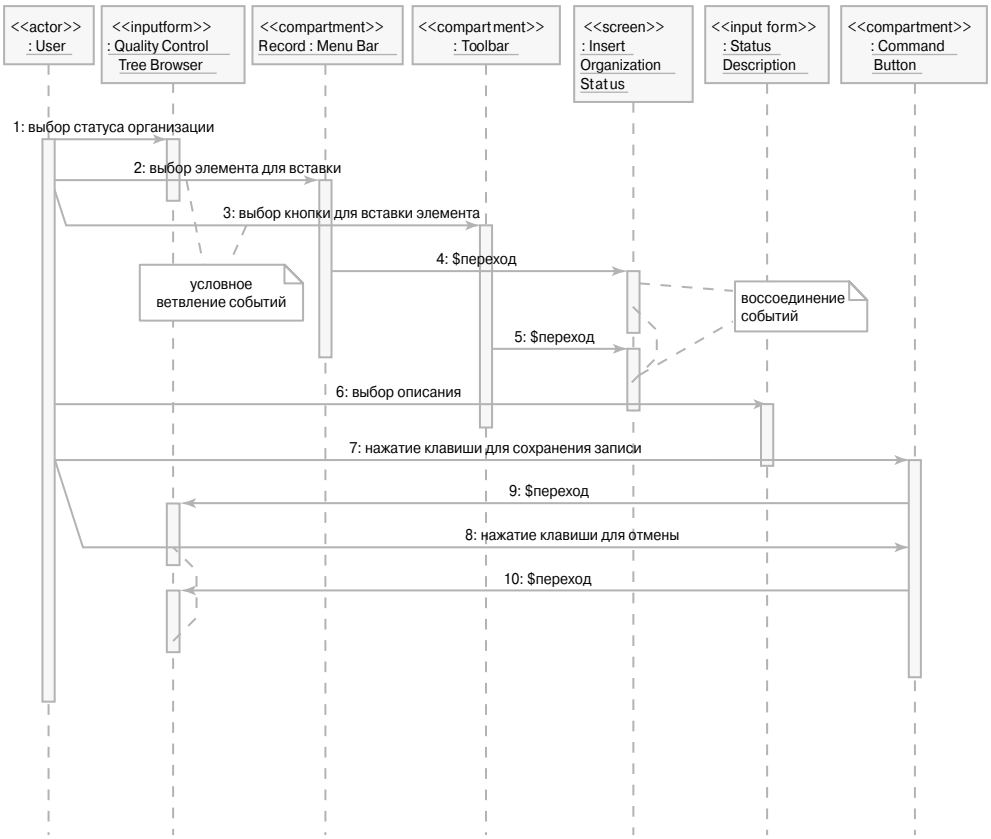


Рис. 7.37. Диаграмма последовательностей для модели кооперации UX в системе управления взаимоотношениями с заказчиками

Модели кооперации UX ограничиваются моделированием взаимодействия человека и компьютера. С точки зрения архитектуры PCBMER это значит, что в эту модель включаются только классы, относящиеся к уровню `Presentation` (Представление). По этой причине, например, в модели, представленной на рис. 7.37, не показаны процессы, связанные с сохранением статуса организации в базе данных и демонстрацией нового статуса организации в окне просмотра строк Organization Status (см. рис. 7.34).

Аналогично, эта модель не содержит никаких ссылок на тот факт, что при сохранении статуса организации база данных генерирует класс `Code` (см. рис. 7.34). Необходимость отобразить этот код в окне просмотра строк Organization Status обусловлена другой моделью кооперации UX (однако эта модель выходит за рамки рассмотрения в примере 7.10).

7.4.4. Структурная кооперация UX

Структурный аспект кооперации UX является следствием функциональной кооперации UX. *Структурная кооперация UX* порождает диаграмму классов со стереотипами UX. Окно атрибутов каждого класса представляет динамическое содержание экрана, формы для ввода данных и отделений экрана. Операционную часть каждого класса представляют пользовательские и внешние события.

Структурная модель кооперации UX представляет собой *навигационную диаграмму* (navigation diagram) для прецедента использования, построенную на базе раскадровки UX. С этой точки зрения структурная модель UX демонстрирует направленные отношения между классами, чтобы показать возможные переходы между экранами, формами для ввода данных и разделами экрана.

Пример 7.11. Управление взаимоотношениями с заказчиками

Вернитесь к примеру 7.10 (см. раздел 7.4.3). Разработайте модель структурной кооперации UX, соответствующую модели функциональной кооперации UX, показанной на рис. 7.37.

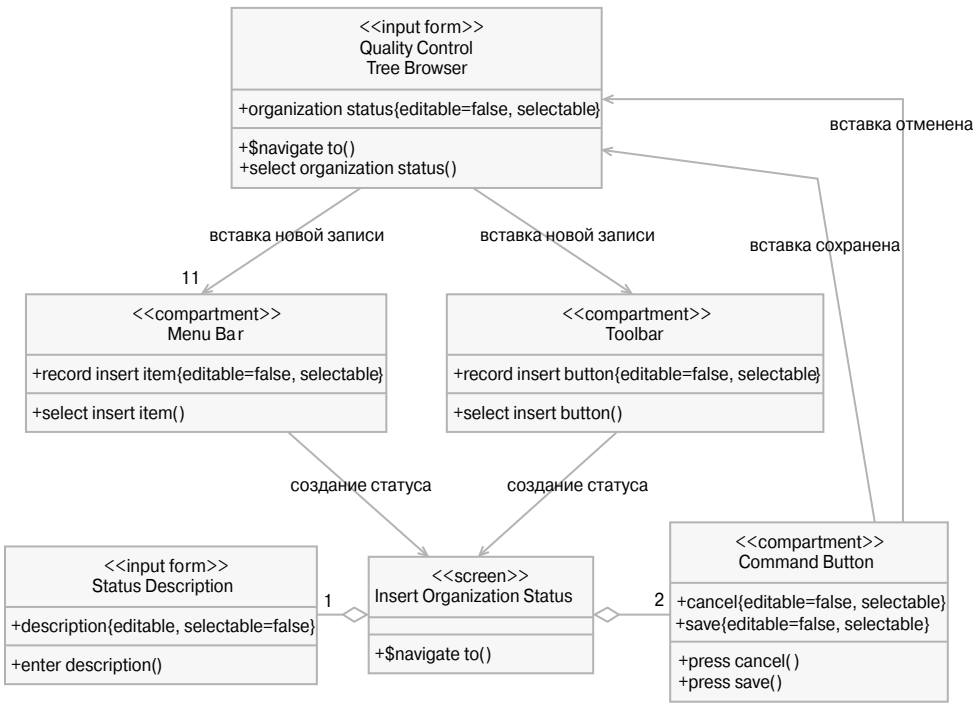


Рис. 7.38. Диаграмма классов для модели структурной кооперации UX в системе управления взаимоотношениями с заказчиками

Модель, описанная в примере 7.11, продемонстрирована на рис. 7.38. Класс и события в классах были выведены непосредственно из диаграммы последовательностей, показанной на рис. 7.37. Динамическое содержание этих классов было определено в соответствии с окнами, показанными на рис. 7.6 и 7.35. (Однако следует подчеркнуть, что в большинстве ситуаций раскадровки UX устраняют необходимость в разработке эскизов или прототипов окон.)

Динамическое содержание классов UX, показанное на рис. 7.38, сопровождается значениями дескрипторов, описывающими возможность редактирования и выбора полей. Видимость поля обозначается знаками "плюс" перед именами полей.

Навигационные диаграммы, показанные на рис. 7.38, необходимо определять для всех прецедентов использования системы. Кроме того, можно создавать навигационные диаграммы для действий, показанных на диаграмме действий. Желательны также высокоуровневые навигационные диаграммы, демонстрирующие основные переходы между окнами в системе.

Контрольные вопросы 7.4

КВ1. Как называется стереотип UX, представляющий части экрана?

КВ2. Как скрываются поля, моделируемые в раскадровках UX?

КВ3. Какие диаграммы UX используются для построения моделей функциональной кооперации UX?

Резюме

Разработка графического пользовательского интерфейса охватывает весь жизненный цикл производства программного продукта — она начинается на этапе анализа и продолжается до его реализации. Данная глава обращена к проектированию графического пользовательского интерфейса, в особенности для настольных систем в среде Microsoft Windows. В ней также рассматриваются визуальные элементы интерфейса и навигация по окнам и Web-страницам. Кроме того, введены профили UML, называемые раскадровками UX (UX storyboard) и предлагающие графические обозначения для проектирования окон и навигации между ними.

Проектирование графического пользовательского интерфейса представляет собой *междисциплинарную деятельность*, требующую объединения усилий различных специалистов. Основными принципами проектирования графического пользовательского интерфейса являются принцип "интерфейс управляется пользователем", а также принципы согласования, индивидуализации, насройки, толерантности, обратной связи, эстетичности и удобства.

Проект графического пользовательского интерфейса состоит из многочисленных контейнеров и компонентов, предназначенных для конкретной платформы. Контейнеры состоят из многих окон и Web-страниц, используемых в приложении. Контейнеры рассматриваются как *компоненты* графического пользовательского интерфейса. К компонентам относятся меню, панели инструментов, связи, кнопки и другие элементы. Примерами технологий для проектирования Web-страниц являются библиотека Swing языка Java и среда Jakarta Struts, предлагающие настраиваемые решения.

Визуальное проектирование отдельных окон представляет собой лишь один из аспектов разработки графического пользовательского интерфейса. Второй основополагающий аспект разработки связан со схемами *навигации по окнам*, фиксирующими возможные пути навигации пользователя между окнами приложения. Для решения этой проблемы в главе введены профили UML — *раскадровки UX*.

Ключевые термины

UX (User eXperience). Пользовательский опыт.

Web-приложение (Web-application). Web-сайт, содержащий динамические страницы, содержание которых может изменяться между запросами пользователей.

Web-страница (Web-page). Окно Web-приложения.

Аплет (applet). Программа, выполняющаяся на клиентской машине в контексте другой программы (Web-браузера) и владеющая ограниченными возможностями обработки.

Библиотека Swing. Библиотека классов, поддерживающая реализацию графического пользовательского интерфейса настольных приложений.

Браузерный клиент (browser client). "Тонкий" клиент, представляющий графический пользовательский интерфейс в сети Web и нуждающийся в сервере для получения данных и программ.

Вторичное окно (secondary window). Всплывающее окно, поддерживающее деятельность пользователя в *главном окне* или другом вторичном окне.

Главное окно (primary window). Основное окно настольного приложения.

Клиентская страница (client page). Web-страница, отображаемая в Web-браузере и способная иметь программную логику, интерпретируемую браузером.

Контейнер (container). Компонент оконного интерфейса — окно, *панель* или *линейка*.

Панель (panel). *Контейнер*, в который можно вставить компоненты графического пользовательского интерфейса.

Полоса (pane). *Контейнер*, т.е. часть окна, например, полоса прокрутки или полоса вкладок.

Программируемый клиент (programmable client). "Толстый" клиент с программой, размещаемой и выполняемой на его компьютере, имеющий доступ к ресурсам клиентской машины.

Серверная страница (server page). Web-страница, имеющая программируемую логику, реализуемую сервером.

Сервлет (servlet). Программа, выполняемая на серверной машине, способная получать запросы от клиентского Web-браузера и генерировать ответы.

Технология Struts. Технология, поддерживающая реализацию графического пользовательского интерфейса Web-приложений.

Форма (form). Часть Web-страницы, содержащая набор полей для ввода данных.

Фрейм (frame). Прямоугольная область просмотра, содержащая Web-страницу или другой фрейм.

Многовариантные тесты

MT1. Какой принцип проектирования графического пользовательского интерфейса можно связать с локализацией?

а. Индивидуализация.
б. Адаптивность.
в. Настройка.
г. Все перечисленные выше.

MT2. На какие части можно разделить окно графического пользовательского интерфейса?

а. Фреймы
б. Формы.
в. Полосы.
г. Панели.

MT3. На какие части можно разделить окно графического пользовательского интерфейса Web-приложения?

а. Web-страницы.
б. Формы.
в. Фреймы.
г. Полосы.

MT4. С каким понятием связано понятие "песочницы"?

а. Web-приложения.
б. Настольные приложения.
в. Аплеты.
г. Страницы сервера.

МТ5. Каким уровням архитектуры PCBMER принадлежат страницы JSP?

а. Компонентов.

б. Представления.

в. Управления.

г. Сущностей.

МТ6. Какому уровню архитектуры PCBMER принадлежат действия форм Struts?

а. Компонентов.

б. Представления.

в. Управления.

г. Сущностей.

МТ7. Что из перечисленного ниже не является стереотипом UX?

а. Окно.

б. Форма для ввода данных.

в. Экран.

г. Отделения.

Вопросы

В1. Рассмотрите рис. 7.4 (см. раздел 7.1.2.1) и 7.15 (см. раздел 7.2.2.4). На рис. 7.15 показано окно сообщения, информирующее пользователя о том, что он нарушил бизнес-правило, пытаясь удалить информацию о программе (программой считается программа на радио или телевидении). На рис. 7.4 показано, что обработка бизнес-правила вызывается из SQL-процедуры, а не из окна графического пользовательского интерфейса. Почему? Может ли бизнес-правило, представленное на рис. 7.15, активизироваться непосредственно как событие окна интерфейса?

В2. Какие принципы проектирования графического пользовательского интерфейса вы считаете основными при переходе от процедурного программирования к объектно-ориентированному программированию? Аргументируйте свою точку зрения.

В3. Как программируемый клиент, так и браузерный клиент могут представлять собой приложение графического пользовательского интерфейса. Что является этими клиентами? Какие возможности для развертывания они предоставляют?

В4. В чем отличие главного окна от вторичного?

В5. Понятие интерфейса в программировании используется в разных контекстах и с разным смыслом. Назовите наиболее распространенный смысл этого слова в контексте проектирования программного обеспечения.

B6. Что такое панель? Чем она может быть полезна при проектировании графического пользовательского интерфейса?

B7. Библиотека Java Swing состоит в основном из так называемых легковесных компонентов, но некоторые компоненты являются тяжеловесными. Найдите в литературе или в Интернете упоминания об этих видах компонентов и укажите разницу между ними.

B8. JavaServer Faces (JSF) — это новая технология, как и библиотека Swing, предназначенная для создания Web-ориентированных интерфейсов. Найдите в Интернете свежую информацию об этой технологии.

B9. Web-приложения, написанные на языке Java, сочетают использование сервлета и технологии JSP. Как связаны эти две технологии?

B10. Объясните, почему профили UX используют дескрипторы для описания свойств динамического содержания классов UX?

Упражнения. Управление взаимоотношениями с заказчиками

УВ31. Обратитесь к задаче 3, посвященной системе управления взаимоотношениями с заказчиками (см. раздел 1.6.3 главы 1), и последующим примерам из главы 4. Представьте себе, что пользователи требуют, чтобы эта система поддерживала работу календаря в программе Microsoft Outlook (рис. 7.39).

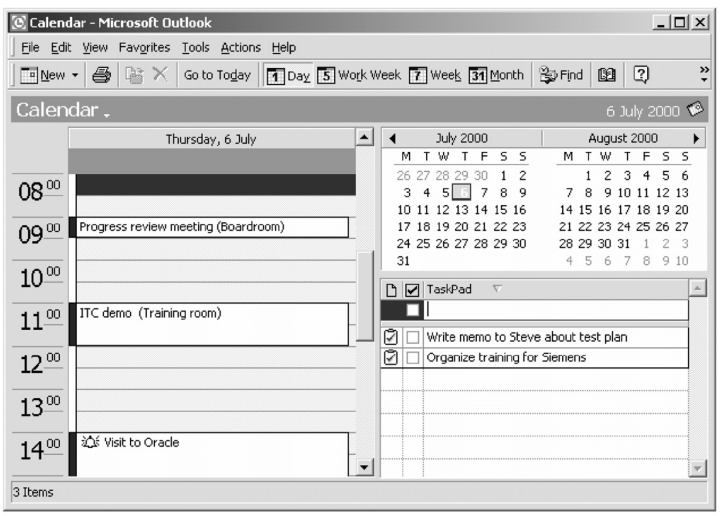


Рис. 7.39. Окно Calendar *в программе Microsoft Outlook. (Копия экрана воспроизводится с разрешения компании Microsoft Corporation.)*

В главном окне должны быть перечислены мероприятия, запланированные на день для пользователя системы. Средства управления окна Calendar (Календарь) могут выводить на экран как прошлые, так и будущие мероприятия. События, назначенные на определенное время (запланированные события), должны выводиться на левой панели экрана. Однако в окне должны демонстрироваться и обрабатываться незапланированные, просроченные (в прошлом) и завершенные мероприятия.

Разработайте главное окно для системы управления взаимоотношениями с заказчиками, соответствующее описанным выше требованиям.

УВ32. Обратитесь к задаче 3, посвященной системе управления взаимоотношениями с заказчиками (см. раздел 1.6.3 главы 1), и последующим примерам из главы 4. Рассмотрите упражнение УВ32.

Предположим, что главное окно системы управления взаимоотношениями с заказчиками не допускает некоторых манипуляций событиями. Например, ввод нового события или обновление существующего мероприятия должны осуществляться только через вторичное диалоговое окно.

После двойного щелчка мышью в первичном окне в диалоговом окне должны отобразиться все подробности события. Диалоговое окно выводит не только информацию о событии, но и данные о связанной с этим работе, а также об организации и контактном лице, с которым намечена встреча.

Детальная информация о событии, которая может выводиться на экран и подвергаться изменениям, может содержать тип события (мероприятия), его подробное описание (примечание), дату, время и имя пользователя (сотрудника), отвечающего за выполнение мероприятия, а также время планирования, начала и завершения мероприятия.

Создайте диалоговое окно для манипуляции событиями, соответствующее указанным требованиям.

УВ33. Обратитесь к задаче 3, посвященной системе управления взаимоотношениями с заказчиками (см. раздел 1.6.3 главы 1), и последующим примерам из главы 4. Рассмотрите упражнения УВ31 и УВ32.

Рассмотрите папку с закладками Update Organization (Обновить организацию) на рис. 7.3 (см. раздел 7.1.1). Одна из закладок называется Contacts (Контакты), ее предназначение — открыть доступ к данным о контакте для модификации (класс `EContact`). В противном случае пользователь должен всегда иметь возможность вернуться к главному окну и активизировать вторичное окно Contacts.

Разработайте содержание закладки Contacts.

УВ34. Обратитесь к примерам 7.10 (см. раздел 7.4.3) и 7.11 (см. раздел 7.4.4). Как указано в примере 7.10 со ссылкой на рис. 7.6 (см. раздел 7.2.1), пользователь

может выводить на экран текущий список статусов организаций, дважды щелкнув на опции Organization Status в окне просмотра деревьев. После выполнения этого действия на правой панели окна отобразится список статусов (в окне просмотра строк). Этот список показан на рис. 7.34 (см. раздел 7.4.3).

Если пользователь хочет изменить какой-либо статус, он может выбрать (подсветить) запись о статусе в списке. Существуют три способа открыть диалоговое окно Update Status (Изменить статус): 1) дважды щелкнуть на выбранной записи; 2) выбрать действие Update (Изменить) в строке меню (см. пункт Record (Запись) на рис.7.6); 3) щелкнуть на панели инструментов (см. пиктограмму с перекрывающимися прямоугольниками на рис. 7.6).

Разработайте модель функциональной кооперации UX для изменения статуса организации. Убедитесь, что результаты успешного изменения статуса организации правильно отражаются в списке (см. рис. 7.34). Покажите только третью возможность (кнопка панели инструментов) для навигации по окну Update Status.

УВ35. Обратитесь к примеру УВ34 и рассмотрите его решение (рис. 7.43). Постройте модель структурной кооперации UX, соответствующую модели функциональной кооперации UX, продемонстрированной на рис. 7.43.

Упражнения. Прямой маркетинг по телефону

Дополнительные требования

Рассмотрите следующие дополнительные требования для системы прямого маркетинга по телефону.

- Главным окном интерфейса для прямого маркетинга по телефону является окно Telemarketing Control (Управление маркетингом). В этом окне на экран компьютера оператора выводится список вызовов в текущей очереди. Когда оператор запрашивает вызов из очереди, система устанавливает соединение и оператор получает возможность обработать телефонное подключение. При этом на экране отображается окно Call Summary (Информация о звонке) — на нем выводится время начала, завершения и продолжительность текущего звонка.
- После установления соединения в окне Telemarketing Control отображается информация о текущем звонке: кому был сделан звонок, в рамках какой кампании, какой тип звонка осуществлен. Если для текущего телефонного номера запланировано более одного звонка, оператору предоставляется возможность осуществить цикл этих звонков.

- На любом этапе разговора оператор может просмотреть предысторию спонсора (окно Supporter History (История спонсора)), относящуюся к предыдущим кампаниям. Аналогично можно просмотреть подробности, относящиеся к кампании, в рамках которой произведен текущий звонок (окно Campaign (Кампания)).
- Графический пользовательский интерфейс обеспечивает возможность быстрой фиксации результатов звонка. К возможным результатам относятся: "размещение" (т.е. билеты были заказаны), "обратный вызов", "неудача", "нет ответа", "занято", "автоответчик", "факс", "неверный номер" и "разрыв связи".
- Подробности, относящиеся к кампании, билетам и призам, разыгрываемым в рамках кампании, выводятся в окне Campaign. Подробности, относящиеся к кампании, включают: идентификационный номер, название, дату начала, окончания и розыгрыша призов. Подробности, относящиеся к билетам, включают: количество билетов, участвующих в кампании, количество проданных билетов и имеющихся в наличии. Подробности, относящиеся к призам, включают: описание и цену приза, его место среди других призов (первое, второе или третье).
- Предыстория звонков и участия благотворителя в прошлых кампаниях отображается в окне Supporter History. История звонков включает перечень последних звонков, их типы, результаты, а также идентификаторы кампаний и операторов. Предыстория кампаний содержит информацию о размещении билетов и выигрышах призов благотворителем.
- После выбора действия Placement (Размещение) откроется окно Placement, которое позволяет пользователю выделить билеты благотворителю и зафиксировать платеж.
- После выбора действия No Answer (Нет ответа) или Engaged (Занято) для каждого текущего звонка в системе в качестве результата записываются значения "нет ответа" и "занято" соответственно. Затем звонки перепланируются системой на другое время следующего дня при условии, что каждый звонок не выходит за пределы лимита попыток, установленного для данного типа звонка.
- После выбора действия Machine (Автоответчик) в системе в качестве результата записывается значение "автоответчик". Продолжительность звонка устанавливается только для первого из текущих звонков. Затем звонки перепланируются системой на другое время следующего дня при условии, что каждый звонок не выходит за пределы лимита попыток, установленного для данного типа звонка.

- После выбора действия Fax (Факс) или Wrong (Ошибка) в системе в качестве результата записываются значение "факс" или "неверный номер". Продолжительность звонка устанавливается только для первого из текущих звонков. После этого для каждого благотворителя с текущим номером телефона данные, относящиеся к благотворителю, обновляются, в них заносится признак "неверный номер".
- После выбора действия Disconnected (Разрыв связи) в системе в качестве результата записывается значение "разрыв связи". После этого для каждого благотворителя с текущим номером телефона данные, относящиеся к благотворителю, обновляются, в них заносится признак "неверный телефон".
- После выбора действия Callback (Обратный вызов) в системе в качестве результата записывается значение "обратный вызов". Продолжительность звонка устанавливается только для первого из текущих звонков. Для того чтобы получить время и дату обратного вызова, открывается окно Call Scheduling (Расписание звонков). Затем звонок перепланируется системой (с присвоением ему нового приоритета) на время и дату, полученные с помощью окна Call Scheduling. Новому звонку присваивается тип "обратный вызов".
- Если при выходе из окна Placement все оставшиеся билеты кампании уже распределены, то все последующие звонки благотворителям для данной кампании не имеют смысла. Все подобные звонки должны быть удалены из очереди звонков.

ПМТ1. Обратитесь к задаче 4, посвященной системе прямого маркетинга по телефону (см. раздел 1.6.4 главы 1), и примерам, приведенным в главах 4 и 5. Рассмотрите диаграмму классов из примера 4.7 (см. раздел 4.2.1.2.3 главы 4).

Модифицируйте и расширьте диаграмму классов, показанную на рис. 7.7, для поддержки дополнительных требований, которые сформулированы выше.

ПМТ2. Обратитесь к изложенным выше дополнительным требованиям, а также к решениям, предложенным в ходе решения упражнения ПМТ1.

Разработайте и продемонстрируйте главное окно для системы управления прямым маркетингом по телефону. Это окно должно содержать окно просмотра строк со списком запланированных звонков. Некоторые звонки должны быть явно приписаны конкретному оператору — возможно, по требованию спонсора. Кроме того, окно должно допускать возможность

обновления экрана очереди (опрашивая сервер базы данных), запроса следующего звонка из очереди и переключения на новую кампанию.

ПМТ3. Обратитесь к изложенным выше дополнительным требованиям, а также к решениям, предложенным в ходе решения упражнений ПМТ1 и ПМТ2.

Разработайте и продемонстрируйте главное вторичное окно системы прямого маркетинга по телефону. В этом окне, Current Call (Текущий звонок), отображается основная информация и действия, разрешенные оператору при успешной попытке соединения со спонсором. Командные кнопки в окне следует сгруппировать в три категории: детали звонка, результат звонка и две общие кнопки (Next Call (Следующий звонок) и Cancel (Отмена)).

ПМТ4. Обратитесь к изложенным выше дополнительным требованиям, а также к решениям, предложенным в ходе решения упражнений ПМТ1, ПМТ2 и ПМТ3.

Разработайте и продемонстрируйте окно Supporter History (История спонсора). Оно должно содержать пять групп полей: звонки в рамках кампании, адрес/телефон, история/выигрыши, предпочтительные часы и статус платежа.

ПМТ5. Идентифицируйте основные элементы моделирования UX (стереотипы уровня класса), отражающие содержание главного окна, разработанного при выполнении упражнения ПМТ2.

ПМТ6. Идентифицируйте основные элементы моделирования UX (стереотипы уровня класса), отражающие содержание главного вторичного окна, разработанного при выполнении упражнения ПМТ3.

ПМТ7. Идентифицируйте основные элементы моделирования UX (стереотипы уровня класса), отражающие содержание вторичного окна, разработанного при выполнении упражнения ПМТ4.

ПМТ8. Разработайте модель функциональной кооперации для процесса, определенного в третьем дополнительном требовании. Это требование приведено ниже.

На любом этапе разговора оператор может просмотреть предысторию благотворителя (окно Supporter History (История спонсора)), относящуюся к предыдущим кампаниям. Аналогично можно просмотреть подробности, относящиеся к кампании, в рамках которой произведен текущий звонок (окно Campaign (Кампания)).

ПМТ9. Разработайте модель структурной кооперации для модели функциональной кооперации, созданной в упражнении ПМТ8.

Ответы на контрольные вопросы

Контрольные вопросы 7.1

КВ1. Клиенты графического пользовательского интерфейса разделяются на программируемых клиентов настольных систем и браузерных клиентов на Web-платформах.

КВ2. Интерфейс управляется пользователем.

КВ3. Обратная связь.

Контрольные вопросы 7.2

КВ1. Существование строки меню и панели инструментов в главном окне.

КВ2. Ведомостью свойств в среде Windows называется вкладка в папке вкладок.

КВ3. Пункт меню.

Контрольные вопросы 7.3

КВ1. Пункты меню, кнопки и ссылки.

КВ2. Файлы cookie.

КВ3. Аффорданс ссылки заключается в том, что она обеспечивает переход на другую Web-страницу.

Контрольные вопросы 7.4

КВ1. «`compartment`»

КВ2. С помощью дескриптора `visible`.

КВ3. Диаграмма последовательностей

Ответы к многовариантным тестам

МТ1. г

МТ2. в

МТ3. в

МТ4. в

МТ5. б

МТ6. а

МТ7. а

Ответы на вопросы с нечетными номерами

B1

Бизнес-правила относятся ко всей системе, а не к отдельному окну или отдельной прикладной программе. Попытка удалить объект `Program` может быть выполнена пользователем с помощью окна графического пользовательского интерфейса, но в любом случае она неизбежно завершается вызовом SQL-процедуры, выполняемой в базе данных (а не в клиентском коде). После этого процедура, выполняемая в базе данных, попытается удалить информацию `Program` из базы данных. В этот момент база данных проверяет, соответствует ли процедура удаления бизнес-правилам, как правило, с помощью *триггера*, содержащегося в базе данных. Сообщение, показанное на рис. 7.15, поступает от триггера. Триггеры нельзя вызывать явно. Они запускаются событиями, например при попытке удаления.

С технической точки зрения событие, связанное с удалением, могло бы поступать непосредственно от клиентской программы. Это было бы целесообразно, если бы это событие обслуживалось командой SQL `Delete` (приходящей от клиента прямо в базу данных), а не путем вызова процедуры SQL (хранимой процедуры). Однако это делать не рекомендуется. Помимо остальных недостатков, генерирование SQL-команд клиентом вынудило бы клиентский код обрабатывать любые сообщения об ошибках, возвращаемые сервером (вместо использования хранимой процедуры для интерпретации таких ошибок для клиента).

B3

Клиент и *сервер* — это логические концепции. Они означают не физически существующие компьютеры, а логические процессы/вычисления, выполняемые на компьютерах. Следовательно, клиент-серверное приложение может быть расположено на отдельной машине. Единственное требование, которое при этом должно выполняться, — клиент и сервер должны быть отдельными процессами.

Точки зрения *пользователя* и *разработчика* на клиент и сервер мало отличаются друг от друга. “С точки зрения пользователя, клиент — это *приложение*. Он должен быть полезным, удобным и надежным. Поскольку пользователь возлагает на клиента большие надежды, он должен тщательно выбирать свою клиентскую стратегию, учитывая как технические (например, сеть), так и нетехнические факторы (например, характер приложения)” (Singh et al., 2002).

“С точки зрения разработчика, приложение J2EE может поддерживать многие виды клиентов. Клиенты J2EE могут запускаться на ноутбуках, настольных и карманных компьютерах или мобильных телефонах. Они могут соединяться с внутренней сетью предприятия или с сетью World Wide Web посредством проводной или беспроводной связи или их комбинации. Возможности клиентов варьируются от “тонких”, браузерных или зависящих от сервера до мощных, программируемых и самодостаточных” (Singh et al., 2002).

Программируемый клиент (programmable client) — это приложение, которое находится на пользовательской машине и выполняется на ней, имея доступ к ее ресурсам (файлам и программам). Такой клиент может загружать данные с сервера, выполнять необходимые вычисления и выводить на экран полученные результаты, используя графический пользовательский интерфейс. Программируемый клиент называется также "*толстым*" или "*мощным*" клиентом.

Браузерный клиент (browser client) — это приложение, обновляющее экран графического пользовательского интерфейса, загружая логику с сервера (хотя часть логики может быть запрограммирована в самом клиенте). Браузерный клиент может проверять входные данные пользователя, запрашивать у сервера услуги и управлять диалоговым состоянием приложения. В последнем случае он отслеживает этапы выполнения транзакций (хотя диалоговое состояние может — а часто и должно — управляться сервером). Браузерный клиент также называется *Web-клиентом* или "*тонким*" клиентом.

Различия между программируемым и браузерным клиентом иногда размыты. Клиенты могут иметь разную степень "тонкости" или "толщины". Однако в большинстве случае *браузерный клиент* является очень "тонким" — он выводит данные для пользователя и полагается на серверное приложение. Иначе говоря, приложение браузерного клиента разворачивается на сервере, как правило, на Web-сервере.

С другой стороны, *программируемый клиент* является очень "толстым" и явно инсталлируется (разворачивается) на клиентской машине. Однако программируемый клиент может *разворачиваться по требованию*, например с помощью технологии Java Web Start. После загрузки в кэш-память программируемый клиент можно запустить снова, не загружая вновь.

B5

Понятие *интерфейс* имеет как минимум четыре значения. Во-первых, понятие интерфейса связано с графическим пользовательским интерфейсом (graphical user interface — GUI) — дисплеем, т.е. экраном компьютера, на который выводится информация.

Во-вторых, существует программный интерфейс приложения (Application Programming Interface — API) — набор программ и средств разработки, регламентирующих вызов функций в прикладных программах и открывающих доступ к низкоуровневым модулям (например, операционной системе, драйверам и виртуальным машинам Java — JVM (Java virtual machine)).

В-третьих, существует *открытый интерфейс* (public interface) — *протокол* или набор операций (методов) с открытой видимостью, который могут использовать другие функции, определенные в классах, поддерживающих этот интерфейс.

В-четвертых, понятие интерфейса связано с языками *UML* и *Java*. Оно представляет собой определение семантического типа с атрибутами (возможно, огра-

ниченным лишь константами) и операциями без их объявления (т.е. без реализаций). Интерфейс UML/Java — это способ моделирования/программирования открытого интерфейса. В этом контексте различаются *предлагаемый* и *требуемый интерфейсы*.

B7

Ответ на этот вопрос приведен в книге (Maciaszek and Liong, 2005).

Набор компонентов Swing обеспечивает поставку приложений в *сменном внешнем виде* (pluggable look and feel). Слово *сменный* имеет несколько значений. Оно может означать соответствие платформе графического пользовательского интерфейса, на которой выполняется программа (Windows, Unix и т.д.). Оно может означать также универсальное кроссплатформенное представление, которое на любой платформе выглядит одинаково. В наборе Swing этот внешний вид называется Java. Кроме того, оно может обозначать внешний вид приложения, настраиваемый программистом.

Для обеспечения *сменности* большинство компонентов библиотеки Swing являются платформно-независимыми, или *легковесными* (lightweight). Легковесный компонент программируется без использования какой-либо платформной специфики. К сожалению, не все компоненты библиотеки Swing являются легковесными, есть и *тяжеловесные* (heavyweight). В большинстве случаев, когда используются тяжеловесные компоненты, библиотека Swing создает оболочку, чтобы скрыть платформно-зависимый код и обеспечить легкую смену внешнего вида.

Maciaszek and Liong, 2005

Библиотека Swing содержит множество классов, позволяющих создавать контейнерные объекты. Четыре класса являются тяжеловесными: `JWindow`, `JFrame`, `JDialog` и `JApplet`. В большинстве ситуаций высокоуровневый контейнер программы является экземпляром тяжеловесного контейнера. Для раскраски экрана и обработки событий легковесные контейнерные классы используют тяжеловесные компоненты. К *легковесным* относятся следующие классы: `JInternaFrame`, `JDesktopPane`, `JOptionPane`, `JPanel`, `JTabbedPane`, `JScrollPane`, `JSplitPane`, `JTextPane` и `JTable`.

Maciaszek and Liong, 2005

B9

Сервлет — это код на языке Java, динамически создающий HTML-страницы. Этот Java-код содержит встроенные HTML-элементы. *Серверная страница* (Java Server Page — JSP), наоборот, пишется на языке HTML и содержит встроенный Java-код (дескрипторы и скриплеты) для управления динамическим содержанием страницы и поставки данных (Maciaszek and Liong, 2005).

Если разработчик считает, что код сервлета должен поддерживаться технологией Java Server Page и что страница JSP должна компилироваться на сервлете до его запуска, то разница между этими понятиями становится несущественной. Еще более важно то, что после загрузки сервлета с Web-сервера он может установить

связь с базой данных и поддерживать связь с несколькими клиентами. Это называется *сцеплением сервлетов* (servlet chaining) и позволяет одному сервлету передавать запросы клиента другому сервлету.

Обычно страницы JSP используются для запроса данных у сервлета для вывода на экран. Это объясняется простотой программирования страниц JSP, позволяющего встраивать в код на языке Java дескрипторы HTML и повторно использовать знание языка HTML.

Кроме того, страницы JSP могут вызывать другие страницы JSP и даже включать вывод для других страниц JSP как часть своего собственного вывода. Это позволяет странице JSP комбинировать вывод (HTML) для обогащения содержания дисплея. С другой стороны, сервлет, вызывающий другой сервлет, обычно используется для получения данных, а не для их вывода.

Объяснение упражнений. Управление взаимоотношениями с заказчиками

УВ31

На рис. 7.40 показано главное окно системы управления взаимоотношениями с заказчиками. Для того чтобы сэкономить пространство, приложение `Calendar` разработано как плавающее окно. При желании его можно закрыть. Каждое мероприятие, указанное в левой части окна, сопровождается коротким описанием, а также указанием организации и контактного лица. Для некоторых мероприятий можно привести дополнительную информацию, например название организации, контактный телефон, номер факса или адрес. Несмотря на то что рисунок приведен в черно-белом варианте, мероприятия, указанные на левой панели окна, можно обозначить разными цветами, в зависимости от их приоритета.

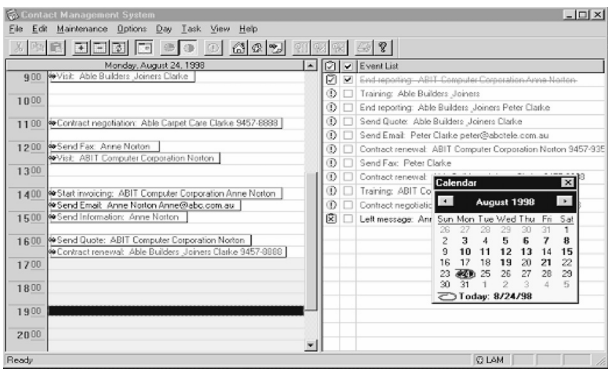


Рис. 7.40. Главное окно системы управления взаимоотношениями с заказчиками. Источник: компания Nielsen Media Research, Сидней, Австралия

Правая панель окна предназначена для решения трех задач. Она содержит информацию о трех категориях мероприятий. *Проведенные* мероприятия удаляются из левой части окна и размещаются в верхней части правой панели. Этот текст выводится зачеркнутым на синем фоне. Основная причина, по которой проведенное мероприятие не удаляется из окна совсем, заключается в том, что оно может быть "незавершенным" (пользователь может преждевременно решить, что мероприятие закончено, а позднее обнаружить, что оно еще не завершено).

Кроме того, на правой панели перечисляются *просроченные мероприятия* (выделены красным цветом). На правой панели выводится также информация о *незапланированных мероприятиях*. Она выводится черным цветом в нижней части панели. Каждая из трех категорий обозначается пиктограммой.

УВ32

Решение примера приведено на рис. 7.41. Обратите внимание на то, что поля Organization (Организация) и Contact (Контакт) нельзя редактировать, поскольку цель мероприятия не может измениться. Аналогично, поля, следующие за приглашением Created (Созданные), также не редактируются.

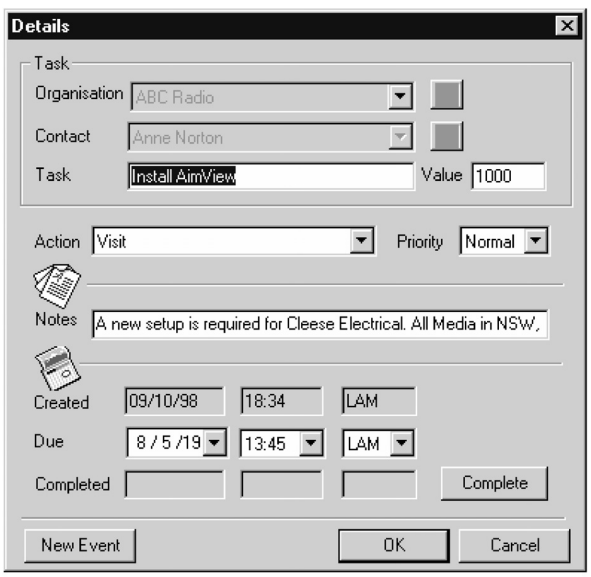


Рис. 7.41. Диалоговое окно системы управления взаимоотношениями с заказчиками. Источник: компания Nielsen Media Research, Сидней, Австралия

Поля, смежные с приглашением Completed (Завершенные), не редактируются, т.е. пользователь не может ввести их тип. Но после щелчка на кнопке Complete (Завершенное) в эти поля автоматически будут вставлены данные, время и ин-

формация о пользователе. После завершения события пользователь по-прежнему имеет возможность объявить его незавершенным, поскольку кнопка Complete (Завершенное) переименовывается в Uncomplete (Незавершенное).

Пользователь может сохранить изменения в базе данных и вернуться к главному окну, щелкнув на кнопке OK. Или же он может отменить изменения, щелкнув на кнопке Cancel (Отмена), и диалоговое окно останется на экране. Пользователь может также щелкнуть на кнопке New Event (Новое мероприятие), которая сохраняет изменения (после подтверждения), очищает все поля в диалоговом окне и позволяет запланировать новое мероприятие (не возвращаясь к главному окну).

УВЗЗ

Как показано на рис. 7.42, вкладка Contacts (Контакты) содержит только имена контактных лиц в организациях. Однако эта вкладка имеет собственный набор командных кнопок (Add (Добавить), Edit (Редактировать) или Delete (Удалить)), которые связаны с выделенным контактом. Щелчки на кнопках Add и Edit открывают вторичное окно Maintain Contacts (Учет контактов) в верхней части окна Maintain Organizations (Учет организаций). Окно Maintain Contacts является модальным по отношению к окну Maintain Organizations.



Рис. 7.42. Папка с вкладками в системе управления взаимоотношениями с заказчиками. Источник: компания Nielsen Media Research, Сидней, Австралия

УВ34

Решение упражнения УВ34 представлено на рис. 7.43 и не требует дополнительных комментариев.

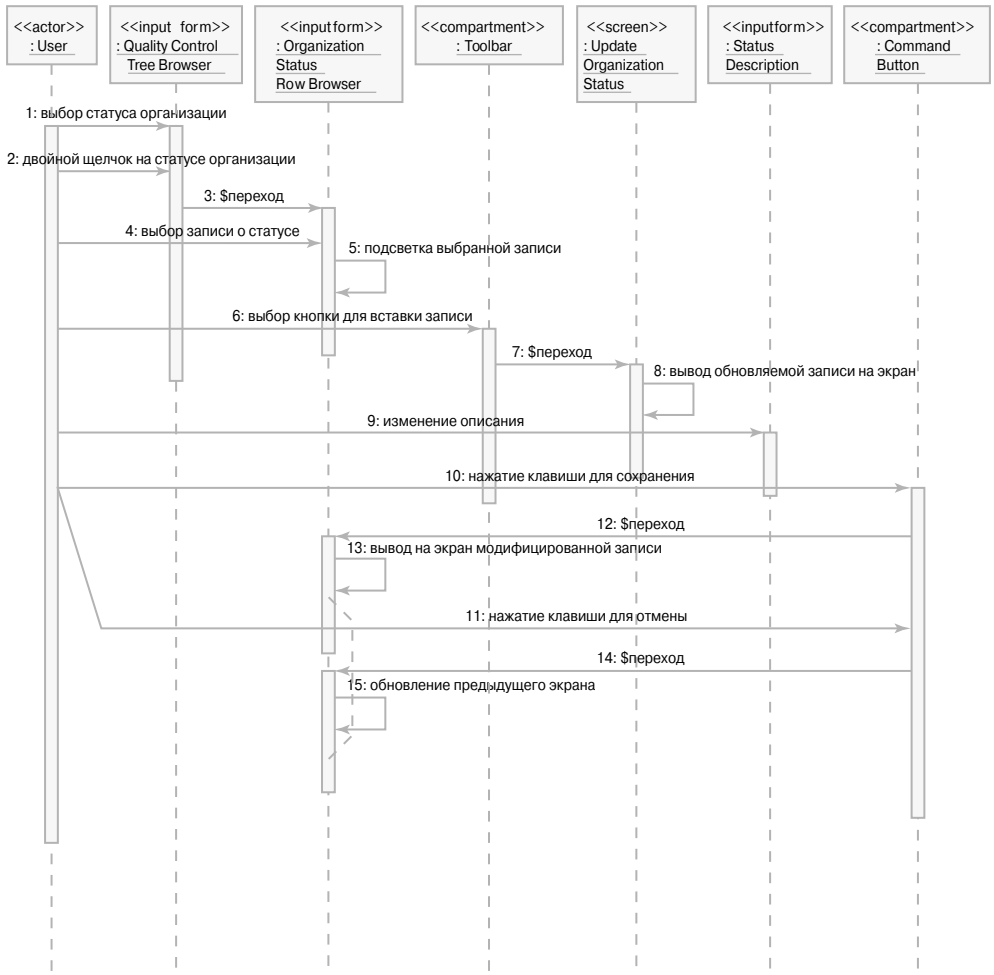


Рис. 7.43. Диаграмма последовательностей для модели функциональной кооперации UX в системе управления взаимоотношениями с заказчиками

УВ35

Решение упражнения УВ35 представлено на рис. 7.43. Это решение также не требует комментариев, но его полезно сравнить с моделью структурной кооперации UX при вставке нового статуса (см. рис. 7.36). Обратите внимание на то, что код статуса теперь является динамическим содержанием окна Update Organization Status (Обновить статус организации), а его описание — динамическим содержа-

нием окна Status Description (Описание статуса). Однако, поскольку окно Status Description содержится в окне Update Organization Status, это описание также включается в динамическое содержание окна Update Organization Status.

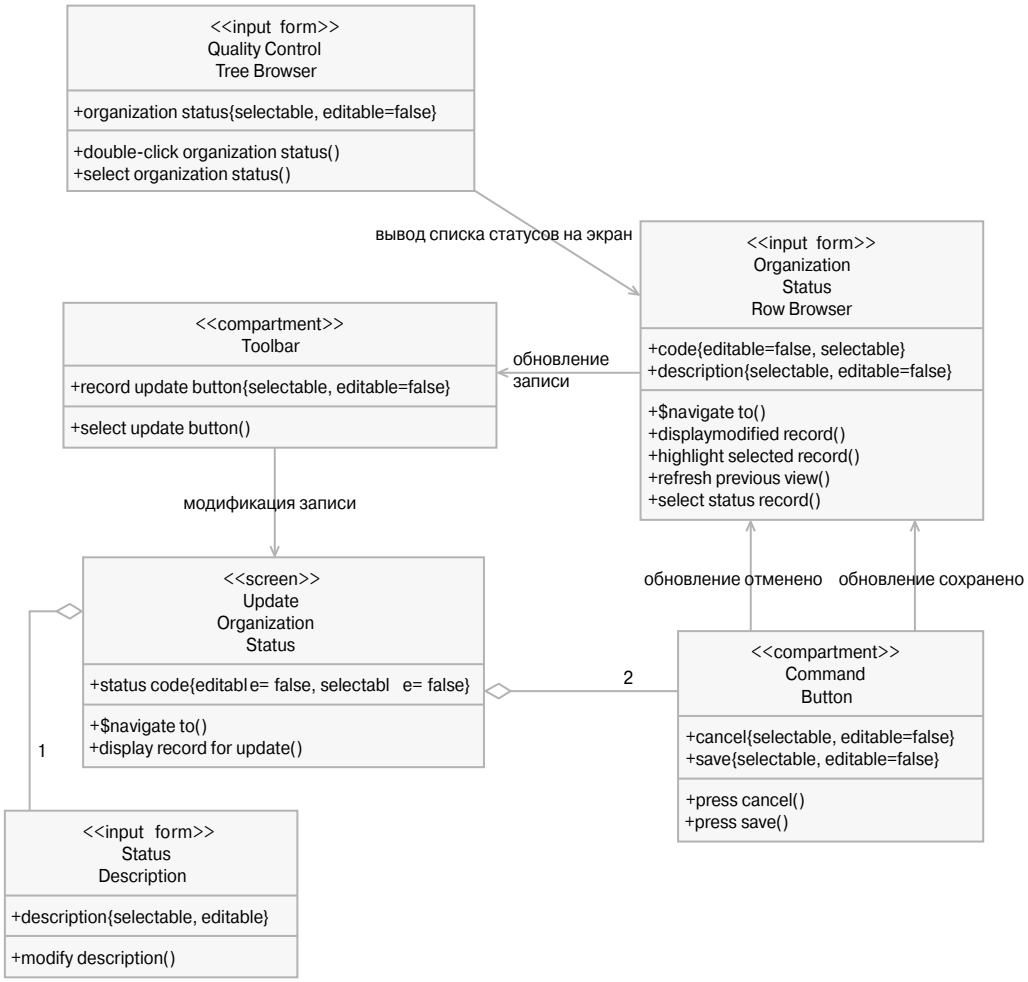


Рис. 7.44. Диаграмма последовательностей для модели функциональной кооперации UX в системе управления взаимоотношениями с заказчиками

ГЛАВА 8

Персистентность и проектирование баз данных

Цели

8.1. Бизнес-объекты и персистентность
8.2. Модель реляционной базы данных
8.3. Объектно-реляционное отображение
8.4. Шаблоны управления персистентными объектами
8.5. Проектирование доступа к базам данных и транзакций
Резюме
Ключевые термины
Многовариантные тесты
Вопросы
Упражнения. Управление взаимоотношениями с заказчиками
Упражнения. Прямой маркетинг по телефону
Ответы на контрольные вопросы
Ответы к многовариантным тестам
Ответы на вопросы с нечетными номерами
Объяснение упражнений. Управление взаимоотношениями с заказчиками

Цели

Информационные системы являются многопользовательскими системами по определению. Для поддержки одновременного доступа к базам данных многих пользователей и прикладных программ используются *системы управления базами данных* (СУБД). Прикладные программы зависят от баз данных, причем не только от данных, но и от функций, предоставляемых базами данных для решения любых конфликтов, обеспечения надежного доступа к данным, гарантии непротиворечивости данных, обработки отказов в ходе транзакции и т.д. Некоторые из этих функций являются неотъемлемой частью программного обеспечения СУБД. Другие функции, от которых зависят прикладные программы, должны кодироваться программистами, разрабатывающими базы данных. Вывод очевиден: правильный проект базы данных, который может объединить и поддерживать *все* прикладные программы, является необходимым условием реализации функциональных возможностей информационной системы.

В языке UML структуры данных, требуемые приложением, определяются диаграммой классов. Структуры данных, которые постоянно находятся в базе данных, моделируются с помощью классов сущностей и отношений между классами сущностей (бизнес-объектов). Классы сущностей необходимо отобразить в структуры данных, распознаваемые базой данных. Эти структуры данных изменяются в зависимости от базовой модели базы данных, которая может быть объектно-ориентированной, объектно-реляционной или реляционной.

Прочитав эту главу, читатели будут

- понимать связи между бизнес-объектами и персистентностью;
- знать модель реляционной базы;
- владеть практическими навыками преобразования объектов в базы данных и наоборот;
- знать основные шаблоны управления персистентными объектами;
- понимать вопросы проектирования и реализации, связанные с доступом к базам данных и обработкой транзакций.

8.1. Бизнес-объекты и персистентность

На протяжении всей книги мы постоянно проводили различие между разработкой клиентского приложения и проектированием сервера баз данных. Мы подчеркивали, что модель классов и подсистемы PCBMER содержат классы приложения, а не структуры хранения баз данных.

Классы сущностей (entity classes) представляют собой постоянные объекты базы данных в приложении, однако они не являются персистентными классами в базе данных. Они *называются* персистентными, потому что еще до завершения работы прикладной программы новые образы объектов сущностей уже будут размещены в базе данных для постоянного хранения. Это позволяет в будущем выполнить ту же самую или другую прикладную программу, загрузив объекты сущностей из базы данных в память программы. Следовательно, необходимо тщательно спроектировать взаимодействие бизнес-объектов и персистентной базы данных.

8.1.1. Инварианты разработки программного обеспечения

Базы данных бывают *реляционными* (например, Sybase, DB2, Oracle8), *объектно-реляционными* (например, UniSQL, Oracle8) или *объектно-ориентированными* (например, ObjectStore, Versant). Маловероятно, чтобы модель хранения для новой современной системы принадлежала к одному из устаревших типов моделей наподобие иерархической (например, IMS), сетевой (например, IDMS), инвертированной или аналогичной модели (например, Total, Adabas). В некоторых случаях, но не в случае действительно современных приложений ИС, персистентность реализуется с помощью текстовых файлов.

В настоящее время на рынке баз данных доминирует *реляционная модель базы данных* (relational database — RDB). Она вытеснила предыдущие иерархические и сетевые модели баз данных. Однако во второй половине 1990-х годов поставщики *систем управления реляционными базами данных* (relational database management systems — RDMBS) объявили о создании *объектной модели базы данных* (object database — ODB), стандартов *Группы управления объектными базами данных* (object database management group — ODMG) и различных *систем управления объектными базами данных* (object database management system — ODBMS).

Впоследствии возникли гибридные *объектно-реляционные системы управления базами данных* (object-relational database management systems — ORDBMS), которые, вероятно, в будущем будут играть доминирующую роль. Традиционные поставщики систем ORDBMS, такие как Oracle и IBM, в настоящее время занимают бо́льшую часть рынка, хотя одновременно постоянно растет влияние компании Microsoft. Тем временем рыночная доля чистых продуктов ORDBMS не растет. Спрос переместился на *программные интерфейсы приложений для хранения объектов* (object storage API), обеспечивающие взаимодействие клиентских приложений с серверными источниками данных, в частности с реляционными базами данных.

Несмотря на то что будущее больше не принадлежит моделям RDB, инерция бизнеса такова, что прошло более десяти лет, прежде чем крупные системы перешли на технологию ORDB или ODB. Кроме того, в будущем будут по-прежнему появляться многочисленные новые приложения, разработанные с помощью технологии RDB, просто потому, что бизнесу не нужны изощренные и сложные для реализации объектные решения.

Действующий стандарт баз данных SQL:1999 называется стандартом объектно-реляционных баз данных (Melton, 2002), хотя он распространяется и на традиционные реляционные базы данных (Melton and Simon, 2001). Стандарт SQL:1999 вносит в реляционную модель объектно-ориентированные свойства, сохраняя ее природу. Объектно-ориентированные характеристики баз данных не оправдали ожиданий разработчиков объектно-ориентированного программного обеспечения, но тем не менее они являются шагом в правильном направлении.

Мы детально рассмотрим реляционную модель, поскольку именно она доминирует на рынке. Довольно забавно, что в первом и во втором издании мы обсуждали объектные и объектно-релчяционные базы данных. Остается лишь надеяться, что эти две модели расширят свое присутствие в информационных системах предприятий, и в будущих изданиях книги мы вернемся к их изучению.

8.1.2. Уровни моделей данных

Специалисты по базам данных выработали свое собственное представление о мире моделирования. Базы данных *хранят данные*. Исторически специалисты по базам данных концентрировались на *моделях данных* (в терминах языка UML — на статических моделях). Современные возможности баз данных по *хранению и выполнению* программ распространили эту перспективу на *модели функционирования* (ориентированные на **триггеры** и **хранимые процедуры**), однако моделирование данных по-прежнему остается краеугольным камнем разработки баз данных.

Модель данных (data model), называемая также *схемой базы данных* (database schema), — это абстракция, представляющая структуры базы данных в терминах, более понятных, чем биты и байты. Распространенная классификация уровней модели данных выделяет три уровня абстракции.

1. Внешняя (концептуальная модель).

2. Логическая модель данных.

3. Физическая модель данных.

Внешняя схема (external schema) представляет обобщенную *концептуальную модель данных*, необходимую для отдельного приложения. Поскольку база данных обычно поддерживает много приложений, конструируется несколько внешних схем. Затем они интегрируются в виде одной концептуальной модели данных.

Наиболее известным методом концептуального моделирования данных являются *диаграммы объект–отношение* (entity-relationship — ER) (Maciaczek, 1990). Несмотря на то что ER-моделирование по-прежнему остается популярным среди разработчиков базы данных, оно постепенно вытесняется моделированием классов UML, представляющим собой концептуальное моделирование приложений и баз данных.

Логическая схема (logical schema) (иногда называемая также *глобальной концептуальной схемой*) представляет собой модель, отражающую логические структуры хранения (таблицы и т.п.) в модели базы данных, используемой для реализации системы (как правило, реляционной). Логическая схема представляет собой глобальную интегрированную модель, поддерживающую текущие и перспективные приложения, которым требуется доступ к информации, хранящейся в базе данных.

Физическая схема (physical schema) отражает специфику конкретной СУБД (например, Oracle или SQL Server). Она определяет способ фактического хранения данных в постоянных устройствах памяти, как правило, на жестких дисках. Физическая схема регламентирует такие аспекты, как использование индексов и кластеризация данных, для повышения эффективности их обработки.

Конструкторские CASE-средства (т.е. инструменты, предназначенные для проектирования и реализации систем) обычно предоставляют разработчикам единый метод моделирования данных для конкретных СУБД. Фактически они представляют собой средства для конструирования комбинированных логических и физических моделей с последующей генерацией соответствующего SQL-кода.

8.1.3. Интеграция приложений и моделирование баз данных

Моделирование *прикладных* программ и моделирование *баз данных* представляют собой разные виды деятельности. Первая выполняется разработчиками прикладных систем, а вторая — администраторами или проектировщиками баз данных. Причина этого разделения заключается в том, что базы данных должны разрабатываться отдельно от приложений (но это не значит, что эта разработка должна выполняться "без оглядки" на приложение). Отдельная база данных должна обслуживать многочисленные и разные прикладные программы. Она должна быть компромиссом, разрешающим любые конфликты и обеспечивающим приложению доступ к данным.

Разработчик прикладной системы обычно имеет модель базы данных, к которой должно обращаться приложение. На ее основе разработчик приложения должен спроектировать соответствующее отображение между моделью программы и моделью базы данных.

На рис. 8.1 показано, как UML-модель приложения связана с моделями персистентных баз данных. Стрелки означают зависимости между элементами моделирования. Принцип нисходящей зависимости (DDP) в архитектуре PCBMER (см. раздел 4.1.3.2 главы 4) расширяет канал связи между приложением и персистентной базой данных.



Рис. 8.1. Интеграция приложений и моделирование баз данных

Подсистема *ресурсов* самостоятельно обеспечивает коммуникацию в базе данных. Все SQL-запросы и вызовы хранимых процедур от приложения генерируются классами ресурсов и передаются серверу базы данных. Все данные и другие результаты, возвращенные сервером базы данных, сначала направляются классам ресурсов, а затем— системе сущностей.

Классы подсистемы *сущностей* представляют *бизнес-объекты*, размещенные в памяти прикладной программы. Правила отображения бизнес-объектов и соответствующие им записи в таблицах базы данных должны быть тщательно определены.

Правила отображения используются *подсистемой-посредником* (mediator subsystem), управляющей кэш-памятью приложения и любыми перемещениями объектов между памятью и базой данных. Это значит, что, когда классу нужен доступ к бизнес-объекту и у него нет ссылки на него, подсистема-посредник является

первым портом вызова. Кроме того, это значит, что подсистема-посредник должна управлять **бизнес-транзакциями** (business transaction), внутри которых осуществляется любая последовательность обращений к базе данных и модификаций.

8.1.4. Отображение объектов в базу данных

Отображение между приложением и базой данных может быть очень запутанным вопросом. В основе трудностей отображения лежат две фундаментальные причины. Во-первых, структуры хранения базы данных могут не иметь практически ничего общего с объектно-ориентированной парадигмой. Во-вторых, база данных почти никогда не проектируется для единственного приложения.

Первая причина равнозначна преобразованию *необъектно-ориентированных сущностей* (как правило, реляционных таблиц) в классы, образующие подсистемы сущностей. Даже если целевая база данных является объектной базой данных, особенности базы данных потребуют аккуратного преобразования.

Вторая причина требует оптимального проектирования базы данных для *всех* приложений, а не только для одного приложения, рассматриваемого в данный момент. Приложениям следует назначить приоритеты с точки зрения их значения для бизнес-процессов, так что структуры базы данных настраиваются на те приложения, которые наиболее важны для организации. Не менее важно, чтобы проектировщик базы данных всегда смотрел вперед, предвидел будущие требования к данным со стороны перспективных приложений и проектировал базу данных таким образом, чтобы адаптировать ее к этим требованиям.

Уровень постоянно хранимых объектов базы данных имеет все шансы оказаться *реляционной базой* данных. Что касается баз данных больших предприятий, то их переход на *объектно-ориентированную технологию разработки баз данных* может совершаться эволюционным путем и должен обязательно включать промежуточный (если не конечный) этап освоения *объектно-реляционной* технологии. Пока мы ограничимся *реляционной моделью базы данных*. С точки зрения *отображения объект–база данных* эта модель характеризуется наиболее жесткими ограничениями и поэтому наиболее трудна для отображения.

Контрольные вопросы 8.1

КВ1. Совпадают ли концепции класса сущностей и персистентного класса?

КВ2. Какие модели баз данных используются как программный интерфейс приложения, управляющего хранением объектов, при взаимодействии клиентских приложений и серверных источников данных?

КВ3. Какой метод концептуального моделирования данных является самым распространенным?

8.2. Модель реляционной базы данных

Базы данных, как и языки программирования, имеют встроенные типы данных, представляющие собой основные конструкции для моделирования и программирования. Такие встроенные типы данных называют **элементарными** (primitive type). Операции, выполняемые базой данных над элементарными типами данных, являются наиболее быстрыми. Кроме того, с их помощью программист может создавать сложные пользовательские типы.

Элементарные типы модели реляционных баз данных действительно элементарны. Простота этой модели, вытекающая из математического понятия *множества*, составляет как сильную, так и слабую стороны этой модели. Математический фундамент, на котором зиждется реляционная модель, определил ее *декларативный* характер (в противоположность *процедурному*). Пользователь просто объявляет, *что* ему требуется получить от базы данных, а не инструктирует систему о том, *как* именно отыскать нужную информацию (система RDBMS знает, как искать данные в своей базе данных).

Однако то, что кажется простым поначалу, становится довольно сложным по мере усложнения решаемой проблемы. Для сложных проблем нет простых решений. У сложных задач не бывает простых решений. Для того чтобы решить их, необходимы более сложные типы данных.

Наверное, лучший способ охарактеризовать модель реляционной базы данных — сформулировать, какие из элементарных типов она *не* поддерживает. Из основных элементарных типов моделирования, применяемых в моделях объектных и/или объектно-реляционных баз данных, реляционная модель не поддерживает следующие.

- Объектные типы и связанные с ними понятия (такие, как наследование или методы).
- Структурированные типы.
- Коллекции.
- Ссылки.

Основным элементарным типом в модели реляционной базы данных является **реляционная таблица** (relational table), состоящая из столбцов. *Столбцы* таблицы могут принимать только *атомарные* значения — структурированные значения или коллекции значений не допускаются.

Модель реляционной базы данных ведет себя весьма жестко по отношению к любым видимым пользователю *навигационным связям* между таблицами — они явно *исключаются*. Отношения между таблицами поддерживаются с помощью сравнения значений в столбцах. Постоянные связи отсутствуют. Полезное свойство объектно-реляционных баз данных поддерживать предопределенные отношения между таблицами называется **ссылочной целостностью** (referential integrity).

На рис. 8.2 показаны элементарные типы модели реляционных баз данных и зависимости между ними. Все понятия выражены с помощью существительных в единственном числе, однако некоторые зависимости применимы более чем к одному экземпляру понятия. Например, ссылочная целостность определяется на одной (или нескольких) таблице. Большая часть понятий, приведенных на рис. 8.2, рассматривается в последующих разделах этой главы. Другие понятия будут рассмотрены более подробно чуть позже.

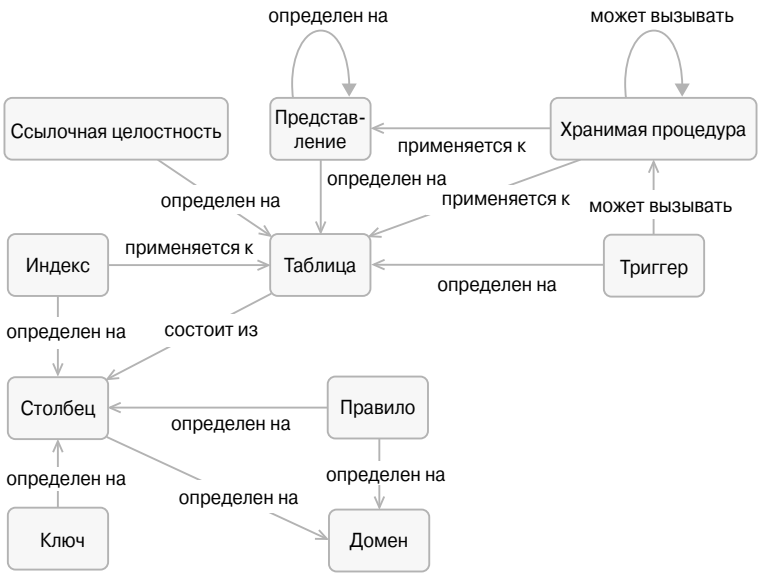


Рис. 8.2. Зависимости между элементарными типами реляционных баз данных

8.2.1. Столбцы, домены и правила

Реляционные базы данных определяют данные в **столбцах** и **строках**. Значение данных, хранимых на пересечении любого столбца и строки, должно быть простым (неделимым) и однократным (не повторяющимся) значением. В этом случае говорят, что столбцы образуют *атомарные домены* (типы данных).

Домен (domain) определяет допустимое множество значений, которое может принимать столбец. Домен может быть безымянным (например, `gender char(1)`) или именованным (например, `gender Gender`). В последнем случае домен `Gender` был определен ранее и используется в определении столбца. Ниже приводится возможный синтаксис определения домена.

```
create domain Gender char(1);
```

Именованный домен можно использовать для определения многих столбцов в различных таблицах. Это способствует достижению непротиворечивости между этими определениями. Изменения, вносимые в определение домена, автоматически

отражаются на определении столбца. Хотя подобная возможность и выглядит на первый взгляд довольно привлекательно, ее использование тут же становится помехой, как только база данных *заполняется* (populated), т.е. загружается данными.

Со столбцами и доменами могут быть связаны некоторые бизнес-правила, накладывающие на них ограничения. *Бизнес-правила* могут определять следующие характеристики данных.

- *Значение, используемое по умолчанию*, — на1пример, если для города не задано никакого значения, предполагается значение "Сидней".
- *Диапазон значений* — например, если допустимое значение для возраста лежит в пределах от 18 до 80.
- *Список значений* — например, если допустимым цветом может быть «`зеленый`», «`желтый`» или «`красный`».
- *Регистр, используемый для представления значения*, — например, если значение должно состоять из символов верхнего или нижнего регистра клавиатуры.
- *Формат значения* — например, если значение должно начинаться с буквы "`K`".

С помощью средств задания правил можно определить только простые правила, относящиеся к единственному столбцу или домену. Более сложные правила, охватывающие таблицы, можно определить с помощью правил ограничения целостности. Основным механизмом определения бизнес-правил являются триггеры.

8.2.2. Реляционные таблицы

Реляционная таблица (relational table) определяется фиксированным множеством своих столбцов. Столбцы могут иметь элементарный тип или тип, определенный пользователем. Таблица может содержать произвольное количество *строк* (записей). Поскольку таблица есть не что иное, как математическое *множество*, повторяющиеся строки в ней отсутствуют.

Значение столбца в определенной строке может принимать значение `null`. Значение `null` означает одну из двух возможностей: "не определенное в данный момент значение" (например, я не знаю даты вашего рождения) или "неприменимое значение" (например, мужчина не может носить женское имя). Значение `null` — это не нуль и не пробел, а специальный набор битов.

Одним из следствий требования к модели реляционной базы данных, которое состоит в недопустимости дублирования строк, является наличие у каждой таблицы **первичного ключа** (primary key). **Ключ** — это *минимальный* набор столбцов и значений в этих столбцах, *единственным способом* идентифицирующих отдельную строку в таблице. Таблица может содержать много ключей. Один из этих ключей произвольным образом выбирается как наиболее важный — первичный. Другие ключи называются *потенциальными* (candidate) или *альтернативными* (alternative).

На практике реляционная таблица не обязана иметь ключ. Это значит, что таблица (без уникального ключа) может содержать идентичные строки, — прекрасное, но бесполезное свойство реляционной базы данных, когда две строки, содержащие одинаковые значения во всех своих столбцах, никак нельзя отличить. В системах объектных и объектно-реляционных баз данных подобное различие обеспечивается за счет наличия идентификатора объекта (два объекта могут быть одинаковыми, но не идентичными, например, как две копии одной книги).

Хотя в языке UML можно ввести стереотипы для моделирования реляционных баз данных, более удобно использовать специально предназначенные для этого диаграммные методы, позволяющие осуществить логическое моделирование реляционных баз данных. На рис. 8.3 показана одна из подобных систем обозначений. В качестве целевой базы данных здесь использовалась база данных DB2.

Employee			
emp_id	CHAR(7)	<pk>	not null
dept_id	SMALLINT	<fk>	not null
family_name	VARCHAR(30)	<ak>	not null
first_name	VARCHAR(20)		not null
date_of_birth	DATE	<ak>	not null
gender	Gender		not null
phone_num1	VARCHAR(12)		null
phone_num2	VARCHAR(12)		null
salary	DEC(8,2)		null

Рис. 8.3. Зависимости между элементарными типами реляционных баз данных

```
--=================================================================
-- Domain: “Gender”
--=================================================================
create distinct type “Gender” as CHAR(1) with comparisons;
--=================================================================
-- Table: “Employee”
--=================================================================
create table “Employee” (
    “emp_id”                CHAR(7)                                 not null,
    “dept_id”               SMALLINT,
    “family_name”           VARCHAR(30)                             not null,
    “first_name”            VARCHAR(20)                             not null,
    “date_of_birth”         DATE                                    not null,
    “gender”                “Gender”                                not null
        constraint “C_gender” check (“gender” in (“F”,“M”,“f”,“m”)),
    “phone_num1”            VARCHAR(12),
    “phone_num2”            VARCHAR(12),
    “salary”                DEC(8,2),
primary key (“emp_id”),
unique (“date_of_birth”, “family_name”)
);
```

Рис. 8.4. Текст программы на языке SQL, сгенерированный для определения таблицы

Таблица Employee состоит из девяти столбцов. Столбец dept_id и последние три столбца допускают значение null. Столбец emp_id представляет собой первичный ключ. Столбец dept_id является **внешним ключом** (foreign key). Столбцы {family_name, date_of_birth} определяют потенциальные (альтернативные) ключи. Столбец gender определен на домене Gender.

Поскольку на реляционную базу данных накладывается ограничение, которое заключается в том, что столбцы могут принимать только атомарные однократные значения, моделирование имени и телефонного номера работника вызывает у нас определенные затруднения. В предыдущем случае мы использовали два столбца: family_name и first_name. Столбцы не сгруппированы и никак не связаны в модели. В последнем случае мы предпочли решение с двумя столбцами — phone_num1 и phone_num2), допускающими наличие максимум двух телефонных номеров у каждого работника.

Как только таблица определена с помощью CASE-средств, можно автоматически сгенерировать программный код для создания таблицы, как показано ниже. Сгенерированный текст программы включает определение домена Gender и определение бизнес-правила, определенного на этом домене.

8.2.3. Ссылочная целостность

Модель реляционной базы данных поддерживает отношения между таблицами посредством ограничения *ссылочной целостности*. Отношения не привязаны к связям отдельных строк между собой. Вместо этого реляционная база данных "выявляет" связи между строками всякий раз, когда пользователь просит систему отыскать соответствующие отношения. Это "выявление" осуществляется с помощью сравнения значений первичных ключей в одной таблице со значениями внешних ключей в той же самой или другой таблице.

Внешний ключ (foreign key) определяется как множество столбцов в одной или более таблицах, значения которых либо равны null, либо должны совпадать со значениями первичного ключа в той же самой или другой таблице. Это соответствие между первичным и внешним ключами называется *ссылочной целостностью*. Первичный и внешний ключи, участвующие в определении ссылочной целостности, должны быть определены на одном и том же домене, но они не обязательно должны иметь одинаковые имена.

На рис. 8.5 показано графическое представление ссылочной целостности. Из-за наличия отношения между таблицами Employee и Department к таблице Employee был добавлен внешний ключ dept_id. Для каждой строки таблицы Employee значение внешнего ключа должно быть либо равно null, либо совпадать с одним из значений dept_id в таблице Department (в противном случае сотрудник может работать в несуществующем подразделении).

Дополнительное описание, сопровождающее линию отношения, *декларативно* определяет поведение, связанное со ссылочной целостностью. Существует четыре

возможных *декларативных ограничения ссылочной целостности*, связанных с операциями `delete` и `update`. Вопрос состоит в том, что делать со строками таблицы `Employee` при удалении или обновлении строк таблицы `Department` (т.е. обновляется столбец `dept_id`). На этот вопрос существует четыре варианта ответа.

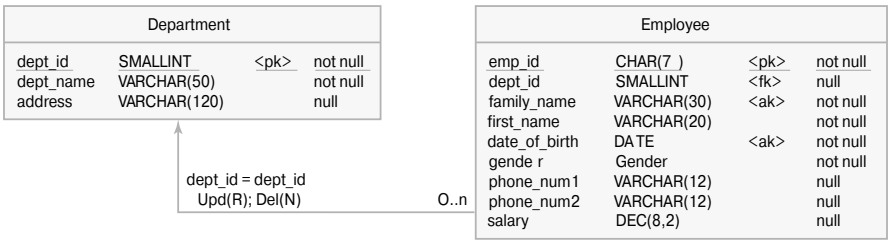


Рис. 8.5. Графическое представление ссылочной целостности

- `Upd(R); Del(R)` — *ограничить* операции обновления или удаления (т.е. не продолжать операции, если только в таблице `Employee` существуют строки, связанные с данным подразделением — `Department`).
- `Upd(C); Del(C)` — *каскадировать* операцию (т.е. удалить все строки таблицы `Employee`, связанные с данным подразделением — `Department`).
- `Upd(N); Del(N)` — *установить значение ключа равным null* (т.е. обновить или удалить строку таблицы `Department` и установить ключ `dept_id` для связанной строки таблицы `Employee` в `null`).
- `Upd(D); Del(D)` — *установить значение ключа равным значению, принятому по умолчанию* (т.е. обновить или удалить строку таблицы `Department` и установить ключ `dept_id` для связанной строки таблицы `Employee` равным значению по умолчанию).

Хотя на рис. 8.5 это не показано, для ссылочной целостности можно определить ограничение, *разрешающее изменение родительской таблицы* (change parent allowed constraint), или *ограничение cpa*. Ограничение *cpa* утверждает, что *дочернюю* (*внешнюю*) таблицу можно связать с другой записью в *родительской* таблице. Например, ограничение *cpa* можно наложить на отношения, указанные на рис. 8.5, так как вполне естественно предположить, что объект `Employee` может быть связан с другим объектом `Department`. Ограничение *cpa* противоположно *замороженному ограничению* (frozen constraint), навязываемому агрегацией *ExclusiveOwns* (см. раздел 5.3.1.1 главы 5).

На рис. 8.6 показан код SQL, автоматически сгенерированный для поддержки ссылочной целостности модели, продемонстрированной на рис. 8.5. Внешний ключ в таблице `Employee` в первом операторе `alter` пропущен, а во втором — восстановлен. Обратите внимание на то, что ссылочная целостность определяется для операций удаления, а не обновления. Причина заключается в том, что `restrict` — это лишь декларативное ограничение, разрешенное для операций обновления, поэтому оно подразумевается неявно.

Моделирование ссылочной целостности усложняется, когда отношения между таблицами являются отношением "множество–множество", как, например, между таблицами `Student` и `CourseOffering` (см. рис. 4.9 в разделе 4.2.3.3 главы 4). Для того чтобы решить проблему, связанную с перенаправлением реляционных баз данных, столбцы которых не могут иметь кратные значения, необходима *таблица пересечений*, например `StdToCrsOff` на рис. 8.7. Единственное предназначение этой таблицы — моделировать отношение "множество–множество" и устанавливать декларативные ограничения ссылочной целостности.

```
alter table "Employee"
      drop foreign key "RefToDepartment";

alter table "Employee"
      add foreign key "RefToDepartment" ("dept_id")
            references "Department" ("dept_id")
            on delete set null;
```

Рис. 8.6. Код SQL, сгенерированный для поддержки ссылочной целостности

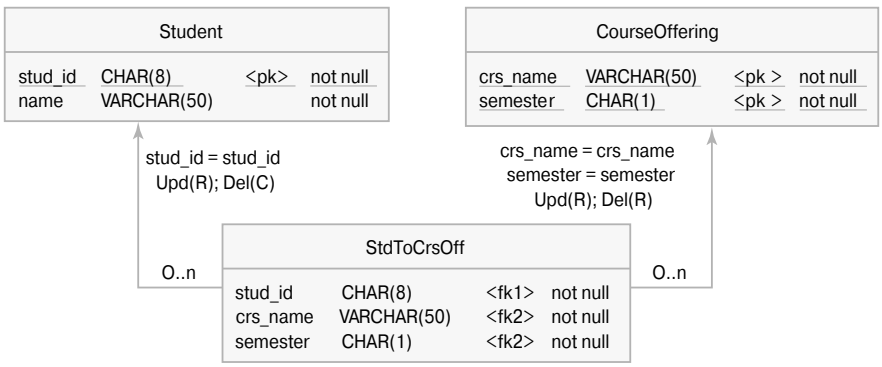


Рис. 8.7. Ссылочная целостность для отношения "множество–множество"

Обратите внимание на то, что, поскольку на рис. 8.7 первичный ключ для таблицы `CourseOffering` состоит из двух столбцов (`crs_name` и semester), соответствующий внешний ключ `StdToCrsOff` также является составным. Несмотря на то что в модели это не показано, первичный ключ для таблицы `StdToCrsOff` состоит из трех столбцов.

8.2.4. Триггеры

Правила и *декларативные ограничения ссылочной целостности* позволяют определять простые бизнес-правила на базе данных. Их недостаточно для определения более сложных правил или каких-либо исключений из правил. Применительно к реляционным базам данных решением данной проблемы (в соответствии со стандартом SQL:1999) является *триггер* (trigger).

Триггер — это небольшая программа, написанная на расширенном языке SQL, которая выполняется автоматически (запускается) в результате модификации таблицы, над которой она определена. Модификацией считается выполнение SQL-оператора `insert`, `update` или `delete`.

Триггер можно использовать для реализации бизнес-правил, выходящих за рамки возможностей оператора `rule` языка SQL (раздел 8.2.1). Например, бизнес-правило, запрещающее вносить изменения в таблицу `Employee` во время выходных, можно запрограммировать в виде триггера. Любая попытка применить к таблице SQL-оператор `insert`, `update` или `delete` в выходные дни приводит к срабатыванию триггера, и база данных отказывается выполнять операцию.

Триггер можно также использовать для приведения в действие более сложных ограничений ссылочной целостности. Например, бизнес-правило может гласить, что при удалении строки из таблицы `Department` (подразделение) строка таблицы `Employee`, представляющая информацию о менеджере этого подразделения, также должна быть удалена. При этом для всех остальных работников (не менеджеров) значения `dept_id` должны быть установлены равным `null`. Подобное бизнес-правило невозможно ввести декларативно. Для того чтобы выполнить его, необходимо задействовать процедурный триггер.

Если для введения ссылочной целостности в базе данных используются триггеры, декларативные ограничения ссылочной целостности обычно не устанавливаются. Смешение процедурных и декларативных ограничений — плохая идея, поскольку она значительно затрудняет их взаимозависимость. Как следствие, в настоящее время преобладает практика программирования ограничений ссылочной целостности только с помощью триггеров. Проблема на самом деле не так страшна, как может показаться на первый взгляд, поскольку мощные CASE-средства способны генерировать бо́льшую часть программного кода автоматически.

Например, ниже приведена программа для триггера, сгенерированная с помощью CASE-средств для системы управления реляционными базами данных Sybase. Триггер реализует декларативное ограничение `Del(r)` — т.е. он не позволяет удалить строку таблицы `Department` до тех пор, пока существует хотя бы одна строка таблицы `Employee`, связанная с ней.

Оператор `if` проверяет, осуществляется ли SQL-операция `delete` (запускающая триггер) для удаления какой-либо строки. Если это не так, триггер прекращает свои действия — никакого вреда нанесено быть не может. Если строка таблицы `Department` может быть удалена, система Sybase запоминает эти (подлежащие удалению) строки в промежуточной таблице `deleted`. Затем триггер выполняет операцию *объединения по равенству* (equality join) по столбцу `dept_id` на таблицах `Employee` и `deleted`, чтобы определить, существует ли хоть один сотрудник, работающий в подразделении, информация о котором подлежит удалению. Если это так, то триггер отменяет удаление, отображает сообщение и осуществляет откат транзакции. В противном случае строку таблицы `Department` разрешается удалить.

```
create trigger keepdpt
    on Department
    for delete
    as
    if @@rowcount = 0
        return /* avoid firing trigger if no rows affected */
    if exists
        (select * from Employee, deleted
        where Employee.dept_id =
            deleted.dept_id)
        begin
            print "Test for RESTRICT DELETE failed. No deletion"
            rollback transaction
            return
        end
        return
go
```

Рис. 8.8. Код SQL, сгенерированный по определению модели данных

Триггер представляет собой особую разновидность *хранимой процедуры* (раздел 8.2.5), которую невозможно вызвать. Триггеры запускаются автоматически во время вставки, обновления или удаления таблицы. Это подразумевает, что каждая таблица должна иметь три триггера. Действительно, в некоторых системах это именно так (например, в системах Sybase и SQL Server). В других системах (например, Oracle или DB2) идентифицируются дополнительные виды событий, поэтому каждая таблица может иметь больше трех триггеров. (Впрочем, возможность иметь больше трех триггеров не повышает выразительность триггерных программ.)

С помощью триггеров можно запрограммировать *любое бизнес-правило*, которое применяется к базе данных и не может быть нарушено клиентскими программами или интерактивными операторами языка SQL Data Manipulation Language (DML). Пользователь клиентской программы может даже не знать о том, что триггеры "наблюдают" за процессом модификации базы данных. Если модификация не нарушает правила, то триггеры программе не видны. Триггер может сам сообщить пользователю о том, что команда DML не может быть выполнена. Он сообщает об этом, выводя информацию на экран приложения и отказываясь выполнять операцию языка DML.

8.2.5. Хранимые процедуры

Впервые *хранимые процедуры* (stored procedures) появились в системе Sybase, но в настоящее время они являются частью любой крупной коммерческой системы управления реляционными базами данных. Хранимые процедуры превращают базу данных в активную программируемую систему.

Хранимые процедуры пишутся на расширенном языке SQL, позволяющем использовать такие программистские конструкции, как переменные, циклы, ветви и операторы присваивания. Хранимая процедура имеет имя, может принимать входные данные и выводить результаты, а также компилироваться и храниться в базе данных. Клиентская программа может вызывать хранимые процедуры наравне с внутренними.

На рис. 8.9 показаны преимущества вызова хранимой процедуры из клиентской программы вместо посылки полного запроса на сервер. Запрос, созданный в клиентской программе, посылается на сервер базы данных через сеть. Запрос может содержать синтаксические и другие ошибки, но клиент не может исправить их, поскольку единственным местом для верификации является система управления базой данных. После верификации система управления реляционными базами данных проверяет, имеет ли отправитель запроса право посылать запросы. Если да, запрос оптимизируется, чтобы определить наилучший маршрут для передачи данных. Только после этого он может быть скомпилирован и выполнен, а результаты — отправлены клиенту.

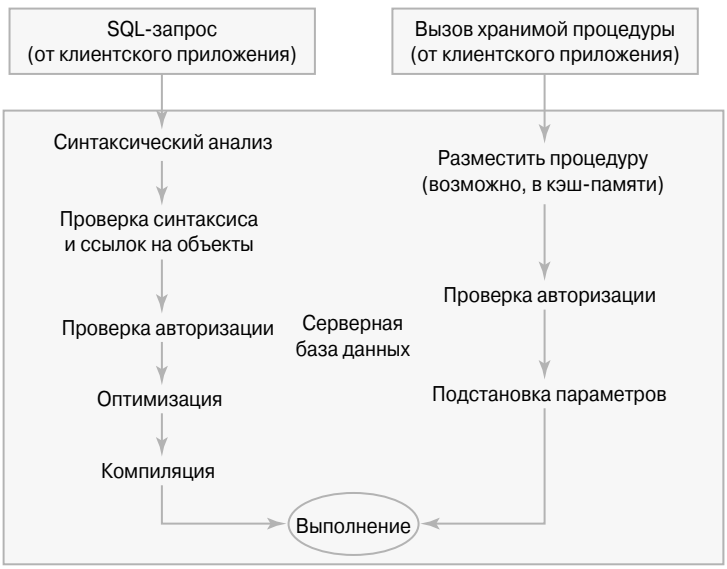


Рис. 8.9. Сравнение вызова клиентской процедуры на языке SQL и хранимой процедуры

С другой стороны, если запрос (или весь набор запросов) сформирован в виде хранимой процедуры, то он оптимизируется и компилируется на сервере базы данных. Клиентская программа теперь не обязана посылать (возможно, большой) запрос через сеть. Вместо этого она посылает короткий вызов с именем процедуры и списком фактических параметров. В случае удачи процедура может разместиться в кэш-памяти системы. Если это невозможно, используется память базы данных.

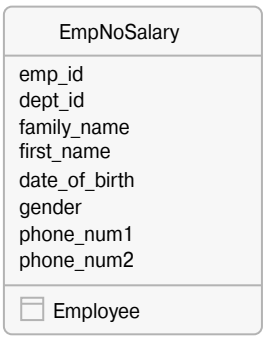
Авторизация пользователя проверяется так же тщательно, как и в случае SQL-запросов. Фактические параметры заменяют формальные, и процедура выполнятся. Результаты возвращаются вызывающей клиентской программе.

Как показано в описанном выше сценарии, хранимые процедуры обеспечивают более эффективный способ доступа к базе данных из клиентской программы. Эти преимущества возникают благодаря экономии сетевого *трафика* и отсутствию необходимости каждый раз осуществлять синтаксический анализ и компиляцию полученного запроса клиента. Что еще более важно, хранимая процедура *занимает одно место* и может вызываться из многих клиентских программ.

8.2.6. Реляционные представления

Реляционное представление (relational view) — это хранимый и поименованный SQL-запрос. Поскольку результатом любого SQL-запроса является переходная таблица, представление может использоваться вместо таблицы в другом SQL-операторе, выводиться на основе одной или более таблиц и/или одного или более других представлений (см. рис. 8.2).

На рис. 8.10 показан графический вид реляционного представления `EmpNoSalary`. Оно отображает всю информацию из таблицы `Employee` за исключением столбца `salary`. Оператор `create view` демонстрирует, что представление фактически является именованным запросом, который выполняется всякий раз при выдаче SQL-запроса или оператора обновления применительно к представлению.




```
-------------------------------------------------------------------------
- - View: "EmpNoSalary"                                          -  -
-------------------------------------------------------------------------
create view "EmpNoSalary" as
        select Employee.emp_id, Employee.dept_id,
                Employee.family_name, Employee.first_name
                Employee.date_of_birth,Employee.gender,
                Employee.phone_num1, Employee.phone_num2
    from Employee;
```

Рис. 8.10. Реляционное представление и сгенерированный код

Теоретически реляционное представление является очень мощным механизмом, который находит множество применений. Оно может использоваться для поддержки безопасности базы данных за счет ограничения круга пользователей, имеющих возможность просмотра таблицы. С его помощью можно представлять данные *с разных точек зрения*. Оно может *изолировать приложение от изменений* определений таблицы, если изменяемое определение не образует часть представления. Кроме того, реляционное представление позволяет упростить выражение *сложных запросов* с помощью метода "разделяй и властвуй" за счет использования нескольких уровней представления.

На практике использование представления в модели реляционных баз данных жестко ограничено из-за невозможности его обновления. *Обновление представления* означает возможность применения операции модификации (SQL-оператор `insert`, `update` или `delete`) к представлению, которое приводит к изменению таблиц базы данных. Язык SQL очень слабо поддерживает обновление представления, отдавая преимущество специальным триггерам.

8.2.7. Нормальные формы

Можно с уверенностью утверждать, что одной из наиболее важных, но в то же время недооцениваемых концепций проектирования РБД является **нормализация** (normalization). Реляционная таблица должна быть представлена в *нормальной форме* (НФ). Существует шесть типов нормальных форм.

- Первая НФ (1NF)
- Вторая НФ (2NF)
- Третья НФ (3NF)
- НФБК (Нормальная форма Бойса-Кодда (Boyce-Codd) — BCNF)
- Четвертая НФ (4NF)
- Пятая НФ (5NF)

Таблицу, представленную в старшей нормальной форме, можно представить во всех более младших НФ. Таблица должна находиться по крайней мере в форме 1NF. Таблица, в которой отсутствуют структурированные или многозначные столбцы, выражена в форме 1NF (и это составляет фундаментальное требование модели реляционных баз данных).

Таблица в младшей нормальной форме может проявлять так называемые **аномалии обновления** (update anomaly). *Аномалия обновления* — это нежелательный побочный эффект, возникающий в результате выполнения одной из операций модификации таблицы (`insert`, `update`, `delete`). Например, если одна и та же информация многократно повторяется в одном и том же столбце таблицы, обновление этой информации должно осуществляться в каждом месте ее хранения, иначе база данных останется в некорректном состоянии. Можно показать, что аномалии обновления постепенно устраняются по мере перехода к старшим нормальным формам.

Итак, каким образом можно нормализовать таблицу к старшей нормальной форме? Таблицу можно привести к старшей нормальной форме, разделив ее по вертикали вдоль столбцов на две или более таблицы меньшего размера. Эти меньшие таблицы, скорее всего, окажутся в старшей нормальной форме и заменят исходную таблицу в проекте модели реляционных баз данных. Исходная таблица, однако, может всегда быть восстановлена с помощью объединения меньших таблиц с использованием SQL-оператора `join`.

Рамки этой книги не позволяют нам рассмотреть теорию нормализации сколько-нибудь подробно. Поэтому читатель может обратиться к книгам Дейта (Date, 2000), Мацяшека (Maciaszek, 1990), Рамакришнана и Герке (Ramakrishnan and Gehrke, 2000) и Зильберкатца с соавторами (Silberschatz et al., 2002). Мы лишь хотели бы подчеркнуть, что правильное проектирование реляционной базы данных проводится на правильно выбранном уровне нормализации.

Что мы подразумеваем под правильным проектом в контексте нормализации? *Правильный проект* означает, что разработчики понимают, как будет использоваться реляционная база данных с точки зрения смеси операций обновления и получения информации. Если база данных является очень динамичной, т.е. подвергается частым операциям обновления, то, естественно, следует создавать небольшие по размерам таблицы, чтобы эффективнее локализовать и выполнять эти обновления. В этом случае таблицы создаются в старшей нормальной форме, и аномалии обновления смягчаются или устраняются.

С другой стороны, если база данных относительно статична, т.е. над ней часто выполняются операции поиска информации, а содержимое обновляется время от времени, имеет смысл разработать *денормализованный проект* базы данных. Это объясняется тем, что поиск в одной таблице большого размера намного эффективнее, чем поиск в нескольких таблицах, которые перед началом поиска необходимо объединить.

Контрольные вопросы 8.2

КВ1. Какая математическая модель лежит в основе концепции реляционной базы данных?

КВ2. Назовите два основных свойства ключа.

КВ3. Может ли внешний ключ иметь нулевые значения?

КВ4. Какие термины используются для описания нежелательного побочного эффекта, который может возникнуть вследствие модификации таблицы?

8.3. Объектно-реляционное отображение

Объектно-реляционное отображение (object-relational mapping) — это отображение модели классов UML в проект реляционной базы данных. Оно должно учитывать ограничения, накладываемые на реляционную модель. Для этого необходимо поступиться некоторой *декларативной семантикой диаграмм классов* в пользу *процедурных решений* в логической схеме проекта. Другими словами, иногда невозможно выразить некоторую встроенную декларативную семантику классов с помощью реляционной схемы. Подобная семантика должна найти процедурное выражение в программах базы данных, т.е. в хранимых процедурах (см. раздел 8.2.5).

Проблемы отображения в модели реляционных баз данных широко изучались в контексте ER-моделирования и расширенного ER-моделирования (например, Elmasri and Navathe, 2000; Maciaczek, 1990). В ходе этих исследований использовались те же принципы и были выявлены все основные вопросы. Отображение должно не просто соответствовать некоторым стандартам (SQL92 или SQL:1999), но также учитывать особенности целевой системы управления реляционными базами данных.

8.3.1. Отображение классов сущностей

Отображение классов сущностей в реляционные таблицы должно соответствовать первой нормальной форме таблиц. Столбцы должны быть атомарными типами. Это ограничение не создает никаких проблем, поскольку на язык UML накладываются такие же ограничения.

Пример 8.1. Управление взаимоотношениями с заказчиками

Обратитесь к спецификациям классов для системы управления взаимоотношениями с заказчиками, указанным в примере 4.6 и на рис. 4.6 (см. раздел 4.2.1.2.3 главы 4). Рассмотрите классы `EContact` и `EEmployee`.

Класс `EContact` обладает атрибутами `familyName` и `firstName`, однако в нем отсутствует имя контактного лица. Аналогично, класс `EEmployee` содержит атрибуты `familyName`, `firstName` и `middleName`, но у нас нет возможности запросить в базе данных имя сотрудника, поскольку этого атрибута в классе нет.

Класс `EContact` обладает также атрибутами `phone`, `fax` и `e-mail`. Такая модель не позволяет контактному лицу иметь более одного номера телефона, факса или адреса электронной почты — на практике подобное предположение нереально.

Отобразите классы `EContact` и `EEmployee` в проект реляционной базы данных, продемонстрировав альтернативные стратегии.

Решение для данного примера приведено на рис. 8.11. Целевой является система управления реляционными базами данных DB2. Решение разработано для системы Oracle. Класс `contact_name` моделируется как атомарный тип данных в таблице `Contact`. Предполагается, что каждое контактное лицо (объект класса `Contact`) имеет только один факс и адрес электронной почты. Однако мы допускаем произвольное количество телефонных номеров. Этой цели служит таблица `ContactPhone`.

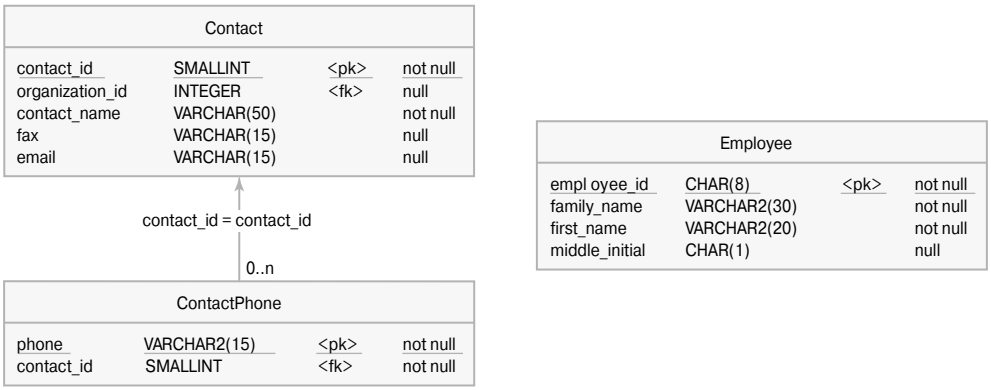


Рис. 8.11. Отображение классов сущностей в проект реляционной базы данных в системе управления взаимоотношениями с заказчиками

В таблице `Employee` сохраняются три отдельных атрибута для имен работника: `family_name`, `first_name` и `middle_initial`. Но в базе данных нет никаких сведений об имени работника `employee_name` как некоторой комбинации этих трех атрибутов.

8.3.2. Отображение отношений ассоциации

Отображение отношений ассоциации в реляционные базы данных связано с использованием ограничений *ссылочной целостности* между таблицами. Любую ассоциацию кратности "один к одному" или "один ко множеству" можно непосредственно выразить с помощью вставки внешнего ключа в одну из таблиц для установления соответствия первичному ключу другой таблицы.

В случае ассоциации "*один к одному*" внешний ключ может быть добавлен к любой из таблиц (решение здесь принимается на основе использования шаблонов ассоциаций), а также может быть желательно составить комбинацию из двух классов сущностей в одной таблице (в зависимости от требуемого уровня нормализации).

Для *рекурсивных* ассоциаций "один к одному" и "один ко многим" внешний и первичный ключи помещаются в одной таблице. Каждая ассоциация "многие ко многим" (независимо от того, рекурсивна она или нет) требует использования перекрестной таблицы (рис. 8.19).

Пример 8.2. Управление взаимоотношениями с заказчиками

Обратитесь к спецификациям ассоциаций для системы управления контактами с клиентами, описанным в примере 4.8 и показанным на рис. 4.5 (см. раздел 4.2.2.2.1 главы 4).

Отобразите диаграмму, показанную на рис. 4.8, на модель реляционной базы данных.

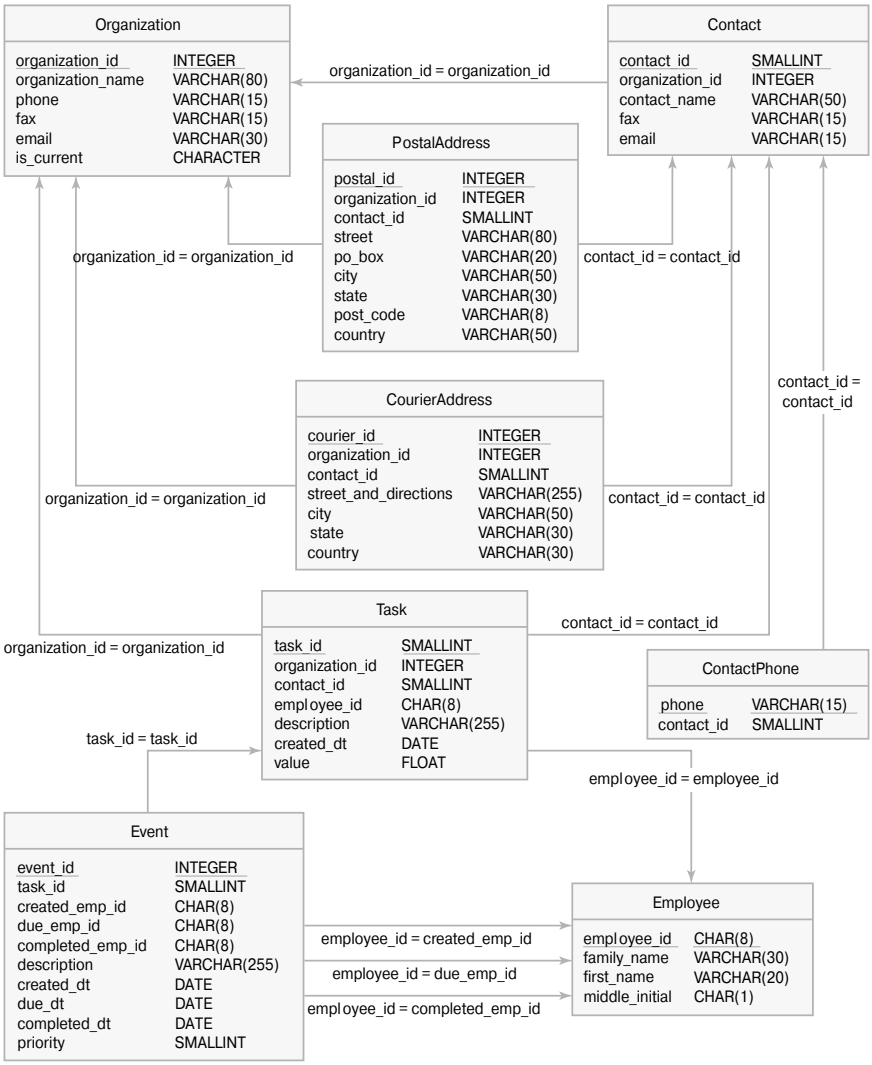


Рис. 8.12. Отображение ассоциаций в проект реляционной базы данных в системе управления взаимоотношениями с заказчиками

Этот пример довольно прост из-за отсутствия ассоциаций "множество ко множеству" в спецификации ассоциаций UML. Диаграмма реляционной базы данных (для системы DB2) показана на рис. 8.12. В соответствии с принципами построения реляционных баз данных мы создали несколько новых столбцов в качестве первичных ключей и решили сохранить модель, приведенную на рис. 8.11, в качестве частичного решения этого примера. Ради экономии места мы не стали показывать столбцы, содержащие значение `null`, и индикаторы ключей.

Ограничения ссылочной целостности между таблицами `PostalAddress` и `CourierAddress`, с одной стороны, и таблицами `Organization` и `Contact`, с другой, моделируются с помощью внешних ключей в таблице адресов. Это отчасти произвольное решение, и ограничения можно было бы моделировать в противоположном направлении (т.е. посредством введения внешних ключей в таблицы `Organization` и `Contact`).

8.3.3. Отображение отношений агрегации

Отношения ассоциации и агрегации в модели реляционной базы данных не различаются, за исключением случаев их процедурной реализации с помощью триггеров или хранимых процедур. На отображение агрегаций распространяются основные принципы отображения ассоциаций (см. раздел 8.3.2). Только в том случае, когда ассоциация может быть преобразована в несколько результирующих реляционных решений, семантика агрегации (как специфической формы ассоциации) может повлиять на решение.

В случае строгой формы агрегации (т.е. композиции) следует попытаться создать комбинацию подмножества и супермножества класса сущности в одной таблице. Это возможно в случае агрегации "один к одному". Для агрегаций типа "один ко множеству" класс подмножества (в сильной и слабой форме агрегации) должен моделироваться в виде отдельной таблицы (с внешним ключом, связывающим ее с ее таблицей-владелицей).

Пример 8.3. Зачисление в университет

Обратитесь к спецификациям агрегаций для системы управления зачислением в университет, изложенным в примере 4.9 и показанным на рис. 4.9 (см. раздел 4.2.3.3 главы 4). Отобразите диаграмму, приведенную на рис. 4.9, на модель реляционной базы данных.

Этот пример содержит отношения агрегации — композицию от класса `Student` к классу `AcademicRecord` и слабую агрегацию от класса `Course` к классу `CourseOffering`. Обе агрегации относятся к типу "один ко множеству" и требуют отдельных таблиц "подмножества".

Для UML-модели, представленной на рис. 4.9, предполагалось (что довольно естественно) наличие косвенной навигационной связи от класса `AcademicRecord` к классу `Course`. В проекте реляционной базы данных может потребоваться установить непосредственную ссылочную целостность между таблицами `AcademicRecord` и `Course`. Кроме того, таблица `AcademicRecord` обладает атрибутом `course_code` как частью ее первичного ключа. Аналогичный атрибут можно включить во внешний ключ в таблице `Course`. Это решение показано на рис. 8.13 (для системы управления реляционными базами данных Informix компании IBM).

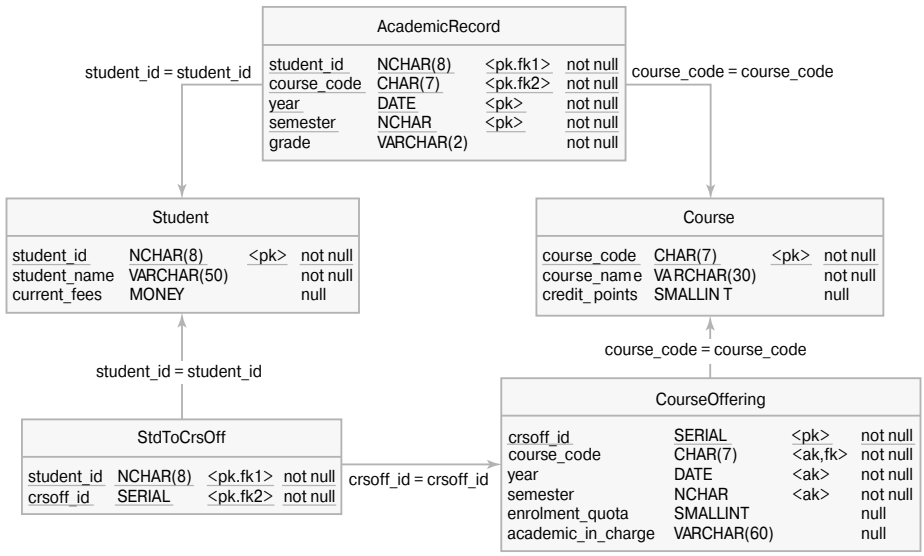


Рис. 8.13. Отображение ассоциаций в проект реляционной базы данных в системе управления зачислением в университет

Ассоциация "множество ко множеству" между классами `Student` и `CourseOffering` ведет нас к другому замечательному наблюдению, хотя и не связанному с отображением агрегации. Результатом ассоциации является перекрестная таблица `StdToCrsOff` с первичным ключом, который образуется в результате конкатенации первичных ключей двух главных таблиц.

Первичный ключ для таблицы `CourseOffering` может выглядеть как `{course_code, year, semester}`. Однако подобный ключ порождает громоздкий первичный ключ для таблицы `StdToCrsOff`. Поэтому мы останавливаем свой выбор на первичном ключе в таблице `CourseOffering`, сгенерированном системой. Он называется `crsoff` и обладает типом `SERIAL` (в системе Informix сгенерированные уникальные идентификаторы относятся к типу `SERIAL`; в других системах управления реляционными базами данных аналогичный тип может называться иначе — например, в Sybase он называется `IDENTITY`, в Microsoft SQL Server — `UNIQUEIDENTIFIER` и в Oracle — `SEQUENCE`).

8.3.4. Отображение отношений обобщения

Отображение отношений обобщения можно осуществить разными способами, но принципы отображения менее сложны, чем можно было бы ожидать. Следует, однако, помнить, что выражение обобщения через структуры данных РБД игнорирует вопросы, которые составляют суть обобщения, — наследование, полиморфизм, повторное использование программного кода и т.д.

Для иллюстрации стратегий отображения обобщения рассмотрим пример, показанный на рис. 8.14. Для преобразования иерархии обобщения в проект модели реляционной базы данных существуют четыре стратегии (хотя для некоторых из этих стратегий возможны дополнительные варианты).

- Отобразить каждый класс в таблицу.
- Отобразить полную иерархию класса в одну таблицу "суперкласса".
- Отобразить каждый конкретный класс в таблицу.
- Отобразить каждый отдельный класс в таблицу.

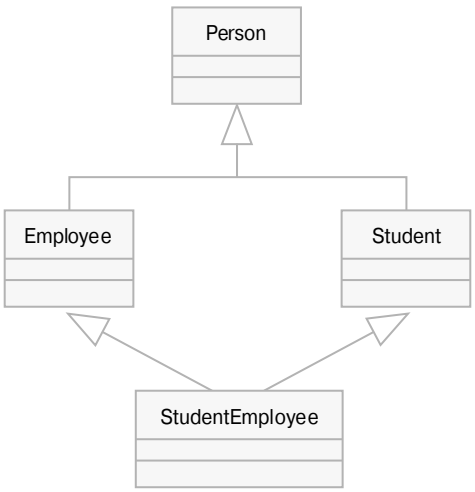


Рис. 8.14. Иерархия обобщения для упрощения отображения в реляционную базу данных

Первая стратегия отображения проиллюстрирована на рис. 8.15. Каждая таблица имеет свой первичный ключ. Представленное решение ничего не говорит о том, "наследует" ли таблица "подкласса" некоторые из ее столбцов от таблицы "суперкласса". Например, хранится ли `person_name` в таблице `Person` и "наследуется" ли таблицами `Employee`, `Student` и `StudentEmployee`? "Наследование" означает в действительности операцию объединения, и штраф за производительность объединения может заставить нас продублировать `person_name` во всех таблицах иерархии.

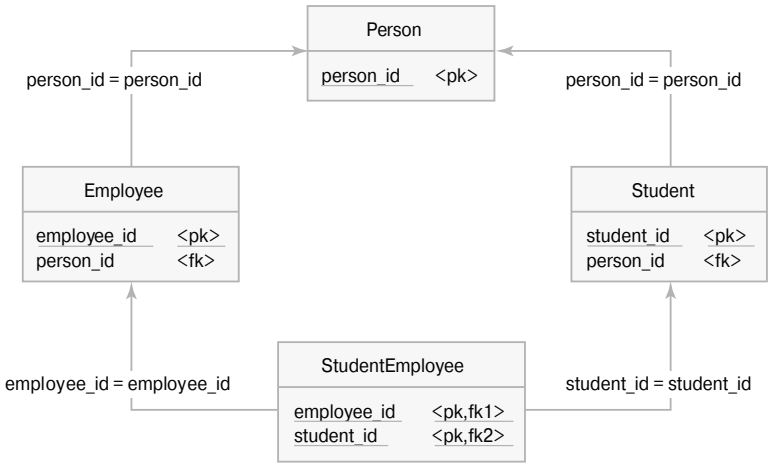


Рис. 8.15. Отображение каждого класса в таблицу

Вторая стратегия отображения проиллюстрирована на рис. 8.16 (для системы управления реляционными базами данных Microsoft SQL Server). Таблица `Person` должна содержать комбинированное множество атрибутов во всех классах иерархии обобщения. Она также содержит два столбца (`is_employee` и `is_student`) для записи статуса: работник, студент или и то и другое.

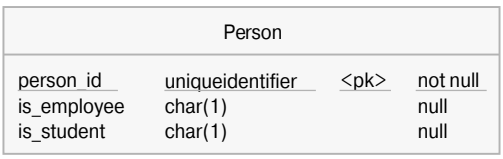


Рис. 8.16. Отображение иерархии классов в таблицу

Для иллюстрации третьей стратегии отображения мы предполагаем, что класс `Person` является абстрактным. Все атрибуты класса `Person` "наследуются" таблицами, соответствующими конкретному классу. Результат аналогичен показанному на рис. 8.17.

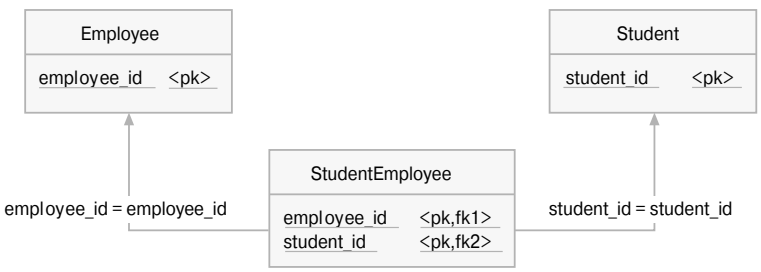


Рис. 8.17. Отображение каждого конкретного класса в таблицу

Последняя из стратегий показана на рис. 8.18 (для системы управления реляционными базами данных Informix), при этом по-прежнему предполагается, что класс `Person` является абстрактным. В противоположность модели, показанной на рис. 8.15, предполагается, что нам всегда известен тот факт, является ли работник также студентом, и наоборот. Отсюда следует запрет на значения `null` для столбцов типа `BOOLEAN`.

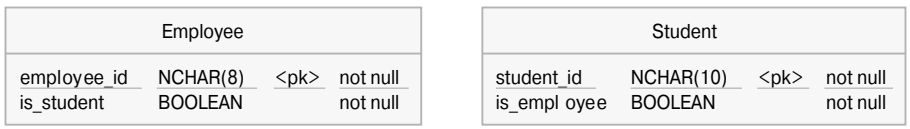


Рис. 8.18. Отображение каждого отдельного конкретного класса в таблицу

Пример 8.4. Магазин видеокассет

Обратитесь к спецификациям обобщения для системы управления работой магазина видеокассет, изложенным в примере 4.10 и показанным на рис. 4.10 (см. раздел 4.2.4.3 главы 4).

Отобразите три класса, показанные на рис. 4.10, в модель реляционной базы данных. Для преобразования каждого класса в таблицу используйте третью стратегию. Отображению подлежат классы `EMovie`, `EVideotape` и `EDVD`.

Подумайте, как обработать производный атрибут `/isInStock` и статический атрибут `percentExcellentCondition`.

Проект реляционной базы данных (для системы Sybase), показанный на рис. 8.19, содержит таблицу для каждого из трех конкретных классов (`Movie`, `Videotape` и `DVD`). Таблицы содержат унаследованные столбцы.

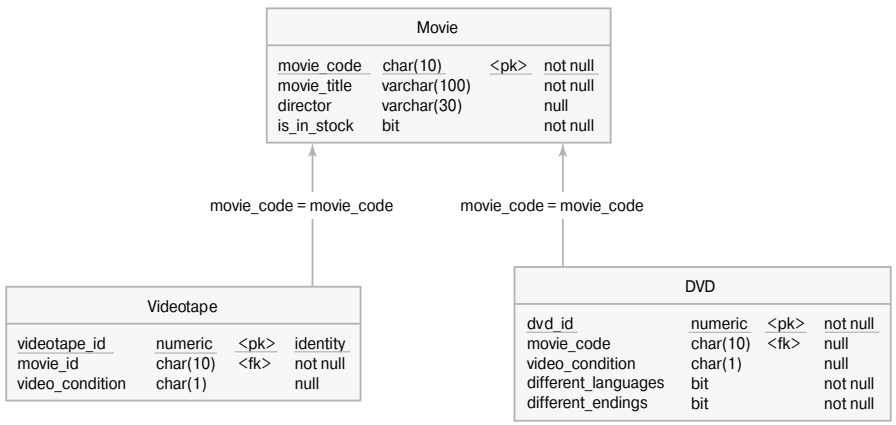


Рис. 8.19. Отображение обобщений в проект реляционной базы данных для системы управления магазином видеокассет

Столбцы `different_languages`, `different_endings` и `is_in_stock` введены с типом `bit`. По определению тип `bit` не допускает значений `null` (столбец типа `bit` принимает два значения: "0" и "1").

Столбец `is_in_stock` устанавливается равным значению "истина" ("1"), если на складе имеется хотя бы одна видеокассета или диск с конкретным фильмом. Это не слишком полезная информация, если клиент интересуется одним из трех видеоносителей. Лучше было бы использовать два столбца типа `bit` или предположить, что эта информация никогда не хранится, т.е. определяется (вычисляется) всякий раз, когда клиент спрашивает ленту или диск.

Статический атрибут `percentExcellentCondition` не хранится ни в одной из таблиц, рассматриваемых в данном проекте. Единственная таблица, в которой он мог бы постоянно храниться, — это таблица `Movie`. Если принимается решение все же хранить его в этой таблице, то к нему применимы те же соображения, что и для атрибута `/is_in_stock`.

Контрольные вопросы 8.3

КВ1. Какой вид отображения требует использования таблиц пересечения?

КВ2. Как выражается полиморфизм в отображении отношений обобщения в реляционную модель?

8.4. Шаблоны управления персистентными объектами

Управление **персистентными объектами** (persistent objects), несомненно, является основной проблемой прикладного программирования. Для решения этой задачи особенно необходимы хорошие *шаблоны проектирования* (design patterns), например *архитектурные шаблоны промышленных приложений* (Patterns of Enterprise Application Architecture — PEAA) (Fowler, 2003). Мацяшек и Лионг (Maciaczek and Liong, 2005) описали несколько из этих шаблонов.

- Коллекция объектов.
- Преобразователь данных.
- Загрузка по требованию.
- Единица работы.

Коллекция объектов (identity map) предлагает решение проблемы присваивания идентификаторов объектов (OID) персистентным объектам, хранящимся в памяти; отображения этих идентификаторов в адреса памяти, выделенной для

этих объектов; отображения других идентифицирующих атрибутов объектов в их идентификаторы и организации единого регистра идентификаторов объектов, которые другие объекты могут использовать для доступа к объектам по их идентификаторам.

Преобразователь данных (data mapper) предлагает решение, в котором программа в любой момент времени знает, находится ли требуемый объект в кэш-памяти, или его следует извлечь из базы данных. Если объект находится в памяти, то преобразователь данных может найти его, если тот является "*чистым*" (clean), т.е. если его состояние (содержание данных) в памяти синхронизировано с соответствующей записью в таблице базы данных. Если же объект является "*грязным*" (dirty), то преобразователь данных инициирует новое извлечение из базы данных. Информация о том, является ли объект "чистым" или "грязным", может храниться в самом преобразователе данных, но лучше хранить ее в коллекции объектов или даже в самом объекте сущности.

Загрузка по требованию (lazy load) предложена Фоулером (Fowler, 2003) как "объект, не содержащий всех данных, необходимых пользователю, но знающий, где их взять". Необходимость этого шаблона возникает из-за того, что данные в базе могут быть весьма взаимозависимыми, а приложение может загрузить в кэш-память лишь ограниченное количество объектов. В то же время очень важно, чтобы программа могла в любой момент времени загрузить больше данных, связанных с объектами, уже находящимися в памяти.

Единица работы (unit of work) — это шаблон, предлагающий решение, в котором программа знает, какие объекты в памяти вовлечены в бизнес-транзакции и, следовательно, должны обрабатываться синхронно, т.е. фиксация всех изменений этих объектов в базе данных должна осуществляться одновременно. Этот шаблон подразумевает, что программа знает тонкости бизнес-транзакций. Она "поддерживает список объектов, вовлеченных в бизнес-транзакцию, а также координирует запись их изменений и решение проблем, связанных с параллельной работой" (Fowler, 2003).

8.4.1. Поиск персистентных объектов

Как указывалось в разделе 8.1.3, архитектура PCBMER предназначена для создания промышленных приложений и может сочетаться с шаблонами PEAA и подобными им. На рис. 8.20 показана диаграмма деятельности, представляющая поток переходов, типичных для ситуаций, когда прикладная программа ищет персистентный объект (Maziaszek and Liong, 2005).

В обычном сценарии пользователь мог бы послать запрос на объект сущности (например, на получение информации о счете-фактуре), вступив в контакт с неким объектом представления (например, окном пользовательского интерфейса). В архитектуре PCBMER такой запрос был бы переадресован *управляющему объекту* (control object), т.е. объекту управляющей подсистемы. Управляющий объект

мог бы попросить *объект преобразователя данных* (data mapper object) получить объект сущности. Объект преобразователя данных обычно размещается в подсистеме-посреднике.

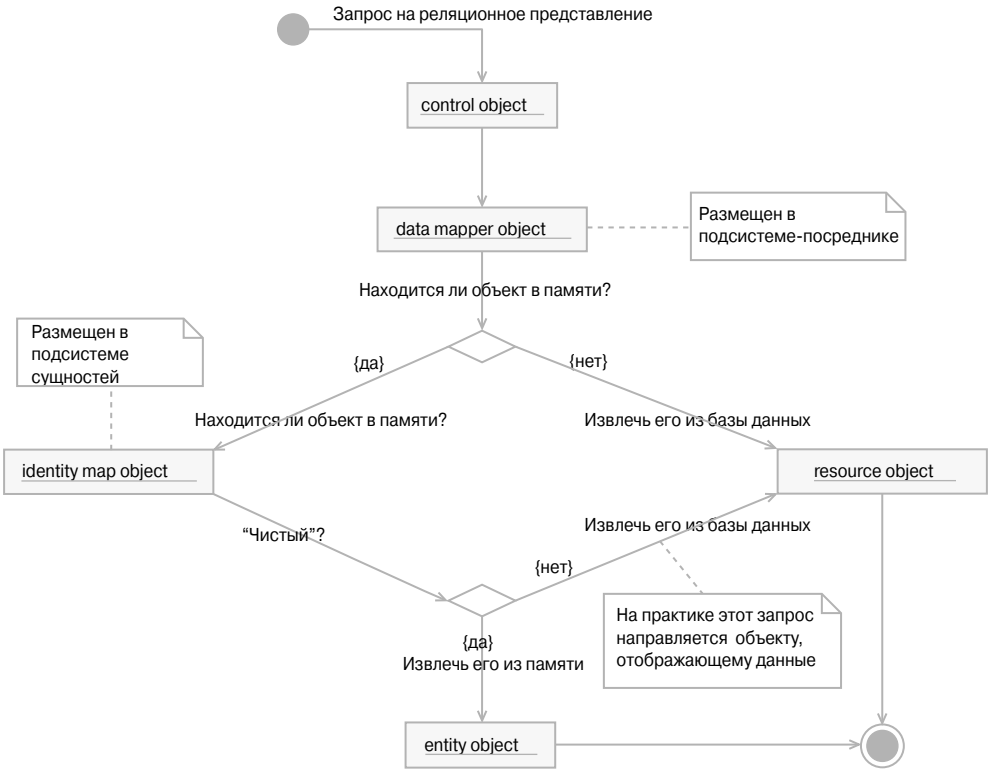


Рис. 8.20. Поиск персистентного объекта

Объект преобразователя данных может иметь большое количество перегруженных методов, реализующих разные стратегии, в зависимости от того, какая информация передается ему управляющим объектом. Как правило, относительно управляющего объекта выдвигаются следующие предположения (Maziaszek and Liong, 2005).

- Управляющий объект знает идентификатор объекта и передает его объекту преобразователя данных.
- Управляющий объект знает значения объекта и передает их объекту преобразователя данных.
- Управляющий объект знает другой объект *X*, содержащий ссылку на искомый объект сущности, и передает объект *X* объекту преобразователя данных.

Обратите внимание на то, что в первом случае управляющий объект может непосредственно запросить объект коллекции объектов и, таким образом, обойтись без преобразователя данных. Это возможно благодаря тому, что управляющая подсистема может непосредственно взаимодействовать с подсистемой сущностей (см. рис. 8.1).

Во втором случае преобразователь данных использует значения атрибутов, полученные от управляющего объекта, чтобы создать соответствующее сообщение объекту коллекции объектов. В свою очередь, этот объект инициирует дальнейший поиск, основываясь на значениях атрибутов.

В третьем случае объект преобразователя данных не может получить непосредственную помощь, поэтому запрос следует делегировать объекту коллекции объектов. Как только будет получен объект, хранящий ссылку на искомый экземпляр, коллекция объектов может выяснить, относится ли ссылка к объекту сущности, хранящемуся в памяти. Если объект находится в памяти, то его можно вернуть управляющему объекту (и далее — объекту представления). Если нет, необходимо извлечь объект сущности из базы данных.

Как показано на рис. 8.20, просто найти объект сущности в памяти недостаточно. Объект должен быть "чистым", т.е. содержать текущие данные, хранящиеся в базе. Информация о том, является ли объект "чистым" или "грязным", обычно хранится в самом объекте (в виде определенного маркера).

Если объект сущности найден, но "грязный", или вообще не найден в кэш-памяти, объект преобразователя данных инициирует поиск в базе данных. Примечание на рис. 8.20 ясно указывает на то, что объект преобразователя данных является посредником во всех попытках поиска в базе данных и, следовательно, подсистема сущностей не взаимодействует непосредственно с подсистемой ресурсов (как в модели PCBMER; см. рис. 8.1).

Поскольку объект преобразователя данных играет настолько важную роль посредника, посылка сообщений от управляющей подсистемы непосредственно подсистеме сущностей следует ограничить ситуациями, в которых управляющий объект уверен, что существующий объект является "чистым" и находится в кэш-памяти. Если же, как в большинстве случаев, управляющий объект не может быть в этом уверен, взаимодействие следует осуществлять при посредничестве объекта преобразователя данных.

8.4.2. Загрузка персистентных объектов

Из диаграммы на рис. 8.20 не ясно, как персистентный объект *загружается* из базы данных в память, если его нет в памяти программы или если он там есть, но помечен как "грязный". Операция загрузки известна также как операция *выписки* (check out), т.е. объект "выписывается" из базы данных в память.

На рис. 8.21 показана диаграмма последовательностей, описывающая загрузку объекта `EContact` из таблицы `Contact` (см. пример 8.1, раздел 8.3.1). Эта мо-

дель предполагает, что объект MDataMapper знает, что объекта EContact нет в памяти, и немедленно обращается за ним к базе данных. Для реализации поиска объект CAdmin посылает атрибут contactName как условие поиска в качестве аргумента операции getContact().

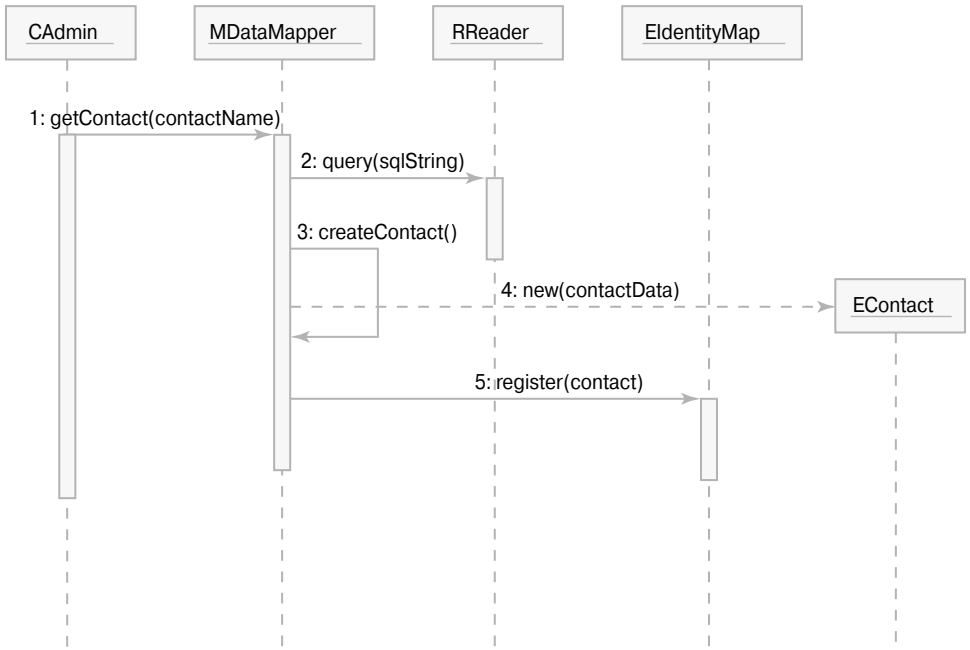


Рис. 8.21. Загрузка персистентного объекта в системе управления взаимоотношениями с заказчиками

В модели, продемонстрированной на рис. 8.21, объект MDataMapper создает строку SQL-запроса и передает его объекту RReader в сообщении query(). После этого объект RReader обрабатывает все обращения к базе данных и получает данные из таблицы Contact. Затем эти данные возвращаются объекту MDataMapper. В целом запрос SQL должен быть полностью создан объектом RReader, а не объектом MDataMapper.

Теперь объект MDataMapper имеет данные для создания объекта EContact. Этот процесс инициируется в методе createContact(). Этот метод отвечает за создание нового объекта EContact с помощью сообщения new().

Для завершения процесса загрузки объект MDataMapper просит объект EIdentityMap зарегистрировать новый объект EContact (и пометить его как "чистый"). Регистрация предусматривает добавление идентификатора объекта EContact в разных отображениях, управляемых объектом EIdentityMap. Наиболее очевидным является коллекция, соответствующая отображению между идентификаторами объекта EContact и самим объектом EContact. Кроме

того, может существовать коллекция, реализующая отображение, связывающее идентификатор EContact с атрибутом contactName (в предположении, что атрибут contacName является уникальным идентификатором).

8.4.3. Выгрузка персистентных объектов

Выгрузка (unloading), или *регистрация* (check-in), — это операция, противоположная загрузке. Существуют три основные ситуации, в которых необходима выгрузка сущности (Maziaszek and Liong, 2005).

- Когда приложение создает новый объект сущности, который должен постоянно храниться в базе данных.
- Когда приложение обновляет объект сущности, и эти изменения должны постоянно храниться в базе данных.
- Когда приложение удаляет объект сущности, и соответствующая запись должна быть удалена из базы данных.

На рис. 8.22 показана последовательность взаимодействий в третьей ситуации, когда объект удаляется приложением. Если объект CAdmin знает, что объект EContact должен быть удален, он вызывает метод deleteContact() из объекта MDataMapper. Объект MDataMapper создает строку SQL для операции delete() и просит объект RWriter обратиться к базе данных и удалить соответствующую запись из таблицы Contact.

Как только объект RWriter вернет объекту MDataMapper информацию о том, что запись в базе данных удалена, объект MDataMapper пошлет объекту EIdentityMap сообщение unregister(). После успешного удаления всей информации об объекте EContact из коллекций, поддерживаемых объектом EIdentityMap, объект MDataMapper отдает команду объекту EContact самоуничтожиться с помощью метода destroy().

Контрольные вопросы 8.4

КВ1. Что означает аббревиатура PEAA?

КВ2. Какой шаблон знает об объектах, существующих в кэш-памяти в данный момент?

КВ3. Какой шаблон предназначен для обработки бизнес-транзакций?

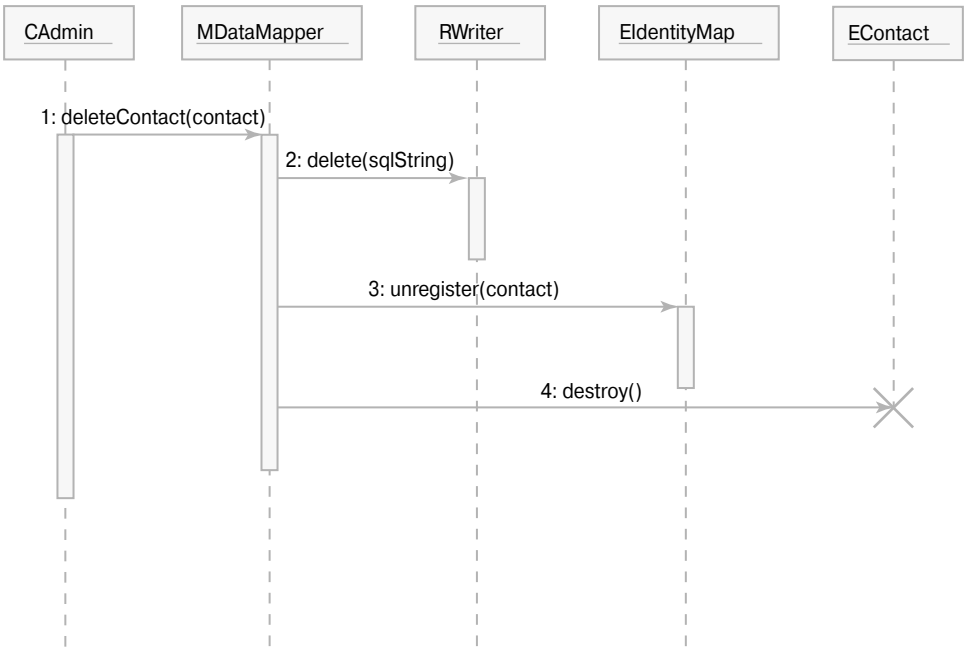


Рис. 8.22. Выгрузка персистентного объекта в системе управления взаимоотношениями с заказчиками

8.5. Проектирование доступа к базам данных и транзакций

Прикладные программы обмениваются данными с базами данных. Для того чтобы иметь доступ к базе данных и модифицировать ее, клиентская программа должна использовать специальный язык — как правило, SQL. Как было показано в разделе 8.4, взаимодействие с базой данных осуществляет подсистема ресурсов (и такие классы, как `RReader` и `RWriter`). Однако в разделе 8.4 не показано, как именно модели взаимодействуют с базами данных. Более того, неясно, как поддерживается *согласованность* бизнес-транзакций.

8.5.1. Уровни программирования на SQL

Для того чтобы понять, как клиентская программа взаимодействует с сервером базы данных, необходимо знать, что язык SQL разделяется на разные диалекты и может использоваться на разных уровнях абстракций. Эти пять уровней языка SQL продемонстрированы на рис. 8.23.

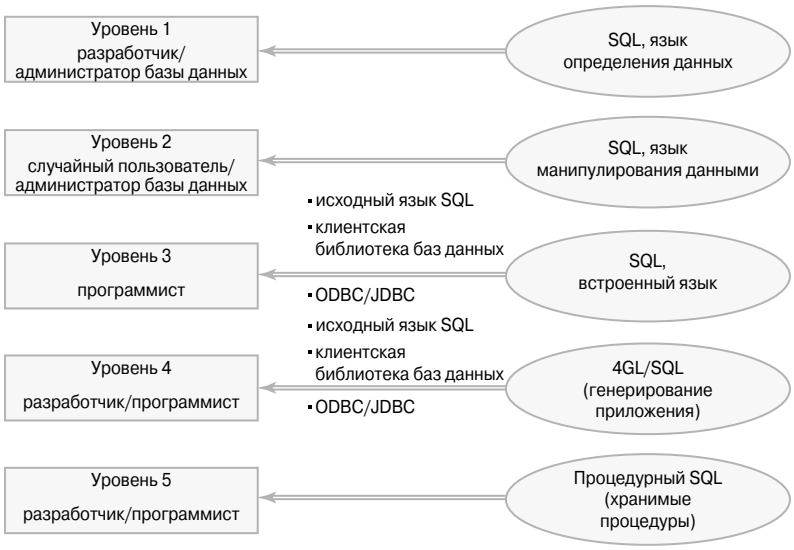


Рис. 8.23. Интерфейсы языка SQL

- Язык *Level 1 SQL* используется как язык определения данных (data definition language — DDL) и предназначен для определения структур баз данных (схем баз данных). Основными пользователями языка Level 1 SQL являются проектировщик и администратор баз данных.
- Язык *Level 2 SQL* используется как язык манипулирования данными (data manipulation language — DML), или *язык запросов* (query language). Термин "язык запросов", однако, неверен, поскольку язык Level 2 SQL предназначен не только для доступа к данным, но и для их модификации (включая операции вставки, обновления и удаления).
- Язык Level 2 SQL применяется широким кругом пользователей, начиная с неискушенных случайных пользователей и заканчивая опытными администраторами баз данных. На этом уровне SQL представляет собой язык *интерактивного* взаимодействия, а это означает, что пользователь может сформировать запрос вне какой-либо среды программирования и немедленно выполнить его над базой данных. Язык Level 2 SQL является отправной точкой в изучении более развитых SQL следующих уровней.
- Прикладные программисты используют SQL на уровнях выше второго. На этих более высоких уровнях SQL позволяет работать в режиме *последовательной обработки записей* (record-at-a-time processing) в дополнение к возможностям *последовательной обработки наборов записей* (set-at-a-time processing) (единственно доступных) на втором уровне. При работе в режиме последовательной обработки наборов записей система управления базами данных берет в качестве входа для запроса одну или более таблицу (набор за-

писей) и возвращает в качестве выхода таблицу. Хотя это довольно мощное средство, использовать его в сложных запросах не только трудно, но и небезопасно.

- Для того чтобы быть уверенным в том, что запрос возвращает верный результат, программист должен иметь возможность просматривать одну за другой записи, возвращаемые как результат запроса, и принимать решение о том, что делать с каждой отдельной записью. Подобная возможность последовательной обработки записей, которая называется *курсором* (cursor), доступна в SQL на уровнях выше второго.
- Язык *Level 3 SQL встроен* в традиционные языки программирования, например C или COBOL. Поскольку компилятор языка программирования не понимает SQL, для трансляции SQL-операторов в вызовы функций библиотеки баз данных, поставляемой изготовителем системы, необходим прекомпилятор (препроцессор). Программист может предпочесть программу, непосредственно использующую функции библиотеки базы данных, при этом необходимость в препроцессоре отпадает.
- Одним из самых распространенных способов обеспечения интерфейса клиентской программы с базами данных является использование стандартов ODBC (open database connectivity — открытый интерфейс доступа к базам данных) или JDBC (Java database connectivity — интерфейс организации доступа Java-приложений к базам данных). Для программирования с использованием этих интерфейсов требуется программный драйвер ODBC или JDBC для определенной СУБД. Интерфейсы ODBC и JDBC обеспечивают стандартную надстройку над языком SQL, которая с помощью драйвера транслируется в "родной" SQL СУБД.
- Интерфейсы ODBC/JDBC обладают тем преимуществом, что обеспечивают независимость программы от "родного" языка SQL СУБД. Если в будущем требуется перенесение программы на другую платформу СУБД, в качестве основного приема осуществления этого процесса служит простая замена драйверов. Что еще более важно, работа с интерфейсами ODBC/JDBC позволяет одному приложению выдавать запрос более чем к одной СУБД.
- Недостатком стандартов ODBC/JDBC является то, что они выступают по отношению к SQL "наименьшим общим знаменателем". Клиентское приложение не может воспользоваться никакими преимуществами специальных средств SQL или расширений, поддерживаемых поставщиком конкретной СУБД.
- Язык *Level 4 SQL* использует ту же стратегию встраивания SQL в клиентские программы, что и SQL третьего уровня. Однако SQL четвертого уровня обеспечивает более мощную среду программирования на основе генератора приложений или графического языка четвертого поколения (fourth generation language — 4GL). Как правило, язык 4GL поставляется оснащенным средствами построения экранных изображений конструирования графического

пользовательского интерфейса. Поскольку прикладные информационные системы требуют развитого графического пользовательского интерфейса, для создания подобных приложений зачастую выбирают комплект 4GL/SQL.

- Язык *Level 5 SQL* дополняет язык SQL третьего и четвертого уровней, обеспечивая возможность переноса некоторых SQL-операторов из клиентской программы на активный (программируемый) сервер баз данных. SQL используется в качестве языка программирования (PL/SQL). Программы сервера можно вызвать из клиентской программы, как будет показано далее.

8.5.2. Проектирование транзакций

Транзакция (transaction) — это логическая единица работы, состоящая из одного или более оператора SQL, выполняемого пользователем. Транзакция также является единицей измерения *непротиворечивости базы данных* (database consistency) — состояние базы данных после завершения транзакции всегда непротиворечиво. Для обеспечения этой непротиворечивости используется программа менеджера транзакций, предназначенная для *восстановления базы данных* (database recovery) и *управления параллельным выполнением операций* (concurrency control).

Согласно стандартам языка SQL транзакция начинается с первого исполняемого SQL-оператора (в некоторых системах требуется явный оператор начала транзакции наподобие транзакции *begin*). Транзакция завершается оператором *commit* или *rollback*. Оператор *commit* записывает изменения в базу данных как постоянные. Оператор *rollback* стирает любые изменения, произведенные транзакцией.

Транзакция является *атомарной* (atomic), т.е. результат выполнения всех SQL-операторов, содержащихся в транзакции, либо полностью *подтверждается* (commit), либо *отвергается* (rollback). Продолжительность (длина) транзакции определяется пользователем. В зависимости от потребностей бизнеса, области приложения, стиля взаимодействия пользователя с компьютером транзакция может быть совсем короткой, состоящей из одного SQL-оператора, или содержать последовательность SQL-операторов.

8.5.2.1. Короткие транзакции

Для большинства традиционных прикладных информационных систем требуются короткие транзакции. Короткая транзакция содержит один или несколько SQL-операторов, которые должны быть выполнены как можно быстрее, так, чтобы не задерживать другие транзакции.

Рассмотрим систему бронирования авиабилетов, в которой несколько туристических агентств принимают заказы от путешественников для полетов по всему миру. Очень важно, чтобы каждая транзакция, связанная с заказом билетов, выполнялась как можно быстрее, чтобы информация о наличии мест быстро обновлялась и база данных была готова к обработке следующей транзакции, ожидающей в очереди.

8.5.2.1.1. Пессимистическое управление параллельным выполнением операций

Архитектура традиционных СУБД, за исключением объектных, ориентирована на короткие транзакции. Эти системы работают в соответствии с *пессимистической концепцией пессимистического управления параллельным выполнением операций* (pessimistic concurrency control). При обработке каждого постоянного объекта, вовлеченного в транзакцию, она устанавливает **блокировку** (lock). Существуют четыре типа блокировок объектов.

- *Эксклюзивная блокировка*, или *блокировка записи* (exclusive (write) lock), вынуждает другие транзакции ожидать, пока транзакция, установившая блокировку, завершится и освободит объект.
- *Блокировка обновления*, или *блокировка намерения* (update (write intent) lock), позволяет другим транзакциям читать объект, одновременно гарантируя транзакции, удерживающей блокировку, возможность выполнить обновление в привилегированном режиме, как только у нее возникнет в этом потребность.
- *Блокировка чтения*, или *разделяемая блокировка* (read (shared) lock), позволяет другим транзакциям читать и, возможно, получать блокировку обновления по объекту.
- *Отсутствие блокировки* (no lock) означает, что другие транзакции могут обновлять объект в любой момент. Подходит только для приложений, допускающих "*грязное чтение*", когда транзакция считывает данные, которые могут быть модифицированы или даже удалены (другой транзакцией) до завершения транзакции.

8.5.2.1.2. Уровни изолированности

Описанным выше четырем типам блокировки соответствуют четыре *уровня изолированности* (levels of isolation) параллельно выполняющихся транзакций. Решение о том, какой уровень изолированности подходит для комбинации транзакций, выполняемых в базе данных, является прерогативой системного проектировщика.

Перечислим уровни изолированности (Khoshafian et al., 1992).

- *Возможность "грязного чтения".* Транзакция `t1` модифицировала объект, но еще не получила подтверждения; транзакция `t2` считывает объект; если транзакция `t1` осуществляет откат, то транзакция `t2` получает объект, который в известном смысле никогда не существовал в базе данных.
- *Возможность неповторяемого чтения.* Транзакция `t1` считала объект; транзакция `t2` обновляет объект; транзакция `t1` считывает тот же объект снова, но в этот момент она получает другое (по сравнению с первоначальным) значение для того же объекта.

- *Возможность фантома.* Транзакция `t1` считала набор объектов; транзакция `t2` вставляет новый объект в набор; транзакция `t1` повторяет операцию чтения и обнаруживает фантомный "объект".
- *Возможность повторяемого чтения.* Транзакции `t1` и `t2` могут продолжать выполнение, но перемежающееся выполнение этих транзакций даст тот результат, что и их выполнение одна за другой (это называется *сериализуемым выполнением* (serializable execution)).

Типичная интерактивная прикладная информационная система, основанная на графическом пользовательском интерфейсе, требует коротких транзакций. Однако уровни изолированности различных транзакций в одном и том же приложении могут отличаться. Для этой цели можно использовать SQL-оператор `set transaction`. Сущность компромисса ясна — увеличение уровня изолированности ведет к общему снижению уровня параллелизма системы. Одно из критически важных проектных решений, однако, не зависит от рассмотренных выше вопросов. Начало транзакции всегда должно откладываться до последней секунды. Нельзя начать транзакцию из окна клиентской программы, а затем заставлять транзакцию ждать, пока она не получит некоторой дополнительной информации от пользователя прежде, чем сможет фактически завершить работу.

Пользователь может очень медленно вводить информацию или отключить компьютер, хотя транзакция продолжает выполняться. *Превышение временного лимита транзакции* (transaction timeout) в конце концов приведет к ее откату, но это успеет негативно отразиться на общей производительности системы.

8.5.2.1.3. Автоматическое восстановление

Закон Мерфи гласит, что если неприятность может произойти, то она обязательно произойдет. Программы могут содержать ошибки, выполняющийся процесс может зависнуть или быть прерванным, электропитание может дать сбой, головки жесткого диска могут поломаться и т.д. К счастью, в большинстве ситуаций СУБД имеет возможность обеспечить *автоматическое восстановление* системы после отказа. Только в случае физической утраты данных на диске требуется вмешательство администратора базы данных, чтобы дать указание СУБД о восстановлении данных с помощью *последней резервной копии базы данных*.

В зависимости от состояния транзакции в момент отказа СУБД автоматически выполняет *откат* (rollback) или *повтор всех завершенных операций* (roll forward) транзакции сразу после устранения причин возникшей проблемы. Процесс восстановления протекает автоматически, но администратор базы данных может контролировать время, необходимое на восстановление, за счет установления частоты контрольных точек. **Контрольная точка** (checkpoint) — это действие, предпринимаемое СУБД для идентификации текущих (выполняемых) транзакций. В этом момент в **регистрационный журнал** (log) заносится контрольная запись. Как следует из его названия, регистрационный журнал представляет собой по-

следовательность записей о выполненных транзакциях. Процесс восстановления должен как минимум сканировать все записи, вплоть до последней контрольной записи, и лишь потом решать, какие действия предпринять дальше.

На рис. 8.24 показаны основные моменты, связанные с автоматическим восстановлением системы после сбоя (Kirkwood, 1992). Транзакция `t1` прошла контрольную точку, но не завершилась до момента сбоя. Поскольку СУБД не известно, были ли физически записаны в базу данных все изменения, произошедшие после контрольной точки, после восстановления системы она *повторяет* транзакцию `t1`.

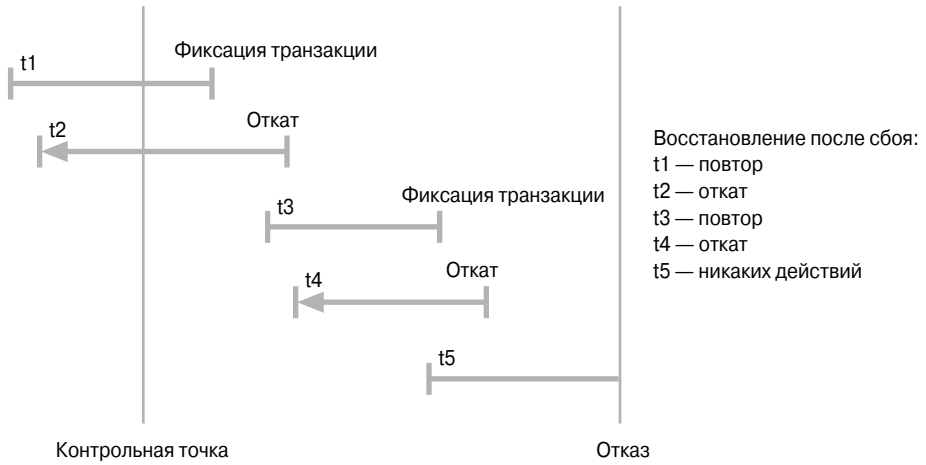


Рис. 8.24. Автоматическое восстановление

К транзакции `t2` применяется откат между контрольной точкой и моментом сбоя. Как и в случае транзакции `t1`, системе не известно, были ли записаны изменения на диске, и она вновь выполняет откат.

Остальные транзакции стартуют после контрольной точки. Для транзакции `t3` *вновь повторяются* все завершенные операции, чтобы убедиться, что произведенные ею изменения достигли базы данных. Аналогично повторяется транзакция `t4`, т.е. для нее выполняется *откат*.

Транзакция `t5` может не требовать никаких корректирующих действий со стороны СУБД, поскольку она выполнялась в момент сбоя. Любые изменения, произведенные транзакцией `t5` до отказа, не были записаны в базу данных. Все промежуточные изменения были записаны только в регистрационный журнал. Пользователь поставлен в известность о том, что транзакция выполнялась в момент сбоя, и может самостоятельно перезапустить транзакцию после того, как СУБД восстановится и вновь начнет функционировать.

8.5.2.1.4. Программируемое восстановление

Несмотря на то что после непредвиденных отказов СУБД выполняет автоматическое восстановление, проектировщики и программисты должны контролировать и прогнозировать любые проблемы, связанные с транзакциями. Система управления базами данных обеспечивает разнообразные возможности отката, которые можно задействовать программным способом, так что проблемы восстановления могут быть решены довольно изящно, при этом пользователь, возможно, даже не заметит, что в некоторый момент произошел сбой.

Для начала заметим, что принципы разработки графического пользовательского интерфейса, такие как управление со стороны пользователя или терпимость к ошибкам (см. раздел 7.1.2 главы 7), требуют, чтобы программа позволяла пользователям совершать ошибки и была способна обеспечить восстановление системы. Откат, контролируемый программистом и применяемый в нужном месте программы, может восстановить предыдущее состояние базы данных (т.е. отменить ошибочные действия) при условии, что транзакция не подтверждена.

Если транзакция *подтверждена*, то у программиста все еще остается возможность написать *компенсирующую транзакцию* (compensating transaction). Впоследствии пользователь может потребовать выполнения компенсирующей транзакции для отмены изменений в базе данных. Компенсирующие транзакции проектируются специально для того, чтобы разрешить программируемое восстановление, и должны быть отражены в моделях прецедентов.

Точка сохранения (savepoint) — это оператор программы, разделяющий длинную транзакцию на более короткие части. *Именованные точки сохранения* (named savepoint) размещаются в стратегически важных пунктах программы. Впоследствии программист имеет возможность осуществить откат выполненной работы к точке сохранения, а не к началу транзакции. Например, программист может поместить точку сохранения прямо перед операцией `update`. Если операция `update` завершится неудачно, программа выполнит откат к точке сохранения и попытается повторить обновление. Вместо этого программа может предпринять любые другие действия, чтобы избежать полного аварийного прекращения транзакции.

В программах большого объема точки сохранения можно помещать перед каждой подпрограммой. Если подпрограмма завершается неудачно, имеется возможность осуществить откат к началу подпрограммы и повторить ее выполнение с исправленными параметрами. При необходимости специально спроектированные и запрограммированные подпрограммы восстановления могут выполнить очистку, так что транзакция может возобновить выполнение.

Триггерный откат (trigger rollback) представляет собой особый вид точки сохранения. Как объяснялось в разделе 8.2.4, триггер можно использовать для программирования бизнес-правил любой сложности. В некоторых случаях откат всей транзакции неприемлем, когда триггер отказывается модифицировать таблицу (из-за того, что транзакция пытается нарушить бизнес-правило). Транзакции мо-

жет потребоваться предпринять корректирующие действия. Вследствие приведенных выше причин СУБД может обеспечить возможности программирования триггера для отката либо всей транзакции, либо только триггера. В последнем случае программа (возможно, хранимая процедура) может проанализировать проблему и принять решение о дальнейших действиях. Даже если необходимо постепенно свернуть всю транзакцию, программа будет иметь возможность более тщательной интерпретации причины ошибки и отображения более осмысленного сообщения для пользователя.

8.5.2.1.5. Проектирование хранимых процедур и триггеров

Принцип *оконной навигации* (window navigation), изложенный в разделе 7.4 главы 7, можно применить к логике прикладной программы, связанной с управлением состояниями транзакций. Возникающие при этом модели *программной навигации* (program navigation) могут идентифицировать хранимые процедуры и триггеры. Для этого необходимо сформулировать цель, определение и детализированный проект каждой хранимой процедуры и триггера. В частности, для определения алгоритмов можно было бы применить псевдокод.

В качестве примера рассмотрим алгоритм для хранимой процедуры `DeleteEvent` в системе управления взаимоотношениям с заказчиками (рис. 8.25). Процедура проверяет, создавал ли информацию о мероприятии сотрудник, пытающийся ее *удалить*. Если это не так, операция удаления отклоняется. Процедура также проверяет, является ли мероприятие единственным оставшимся мероприятием для данного задания. Если это так, то задание также удаляется.

```
BEGIN
INPUT PARAMETERS (@event_id, @user_id)
Select Event (where event_id = @event_id)
IF @user_id = Event.created_emp_id
      THEN
            delete Event (where event_id = @event_id)
            IF no more events for
                  Task.task_id = Event.task_id AND
                  Event.event_id = @event_id
            THEN
                  delete that Task
            ENDIF
      ELSE
            raise error ("Only the creator of the event can
                                    delete that event")
ENDIF
END
```

Рис. 8.25. Псевдокод для хранимой процедуры в системе управления взаимоотношениями с заказчиками

Хранимая процедура `DeleteEvent` содержит операторы `delete` для удаления записей из таблиц `Event` и `Task`. Эти операторы должны запускать триггеры удаления на этих таблицах, если они существуют. Если алгоритм для этих триггеров выходит за пределы обычной проверки ссылочной целостности, проектировщик также должен предоставить для них спецификацию псевдокода (включая решение по стратегии отката — триггерный откат или откат транзакции).

8.5.2.2. Проектирование хранимых процедур и триггеров

Некоторые новые классы прикладных информационных систем поощряют кооперативные взаимодействия пользователей. Эти приложения известны как *системы групповых вычислений* (workgroup computing) или *компьютерные средства коллективной работы* (computer-supported cooperative work — CSCW). В качестве примеров можно привести многие офисные приложения, системы творческой кооперации, системы автоматизированного проектирования, CASE-системы и т.д.

Во многих отношениях приложения для рабочих групп выдвигают требования к базам данных, которые являются ортогональными по отношению к традиционным моделям баз данных. Последние отличаются короткими транзакциями и изолируют пользователей друг от друга. Приложения для рабочих групп требуют длинных транзакций, управления версиями, управления кооперативной параллельностью и т.д.

Модель объектных баз данных описывает схему групповых вычислений, и многие объектные СУБД нацелены именно на эту область приложений. Пользователи, работающие с приложениями для рабочих групп, совместно используют информацию и осведомлены о работах, выполняемых ими над общими данными. Они работают в рамках собственной рабочей среды, используя персональные базы данных, включающие данные, *выписанные* (скопированные) из общей базы данных рабочей группы; работают с *длинными транзакциями*, которые могут занимать несколько сеансов работы с компьютером (пользователь может прерваться, затем возвратиться и продолжить работу с той же длинной транзакцией).

Отличительной чертой длинной транзакции является то, что для нее недопустим автоматический откат вследствие сбоев без отслеживания системой хода транзакции. Для того чтобы понять это требование, представьте мое отчаяние, если бы для этого учебника сейчас был произведен “откат” из-за компьютерного сбоя! Откат для длинных транзакций контролируется пользователями с помощью процедур точек сохранения, которые постоянно сохраняют объекты в личной базе данных пользователя.

Понятие короткой транзакции не исчезло из сферы приложений для рабочих групп. Короткие транзакции необходимы для обеспечения атомарности и изолированности во время операций извлечения данных из групповой базы данных в личную базу и возврата данных (после их обработки в собственной рабочей среде) из личной базы данных в общую базу данных. После выполнения описан-

ных коротких транзакций короткие блокировки освобождаются, а групповая база данных накладывает длинные постоянные блокировки на все извлеченные из нее объекты.

Модель длинных транзакций включает несколько взаимосвязанных целей, среди которых можно, в частности, выделить следующие (Hawryszkiewycz et al., 1994; Maciaczek, 1998).

- Обеспечение обмена информацией (даже если она является временно противоречивой) между сотрудничающими пользователями.
- Обнаружение противоречий в данных и их согласованное разрешение.
- Использование преимуществ поддержки множества версий объектов для обеспечения управляемого совместного использования информации без потери результатов работы в случае отказа системы.

Контрольные вопросы 8.5

КВ1. На каком уровне языка SQL возможна последовательная обработка записей?

КВ2. Назовите две основные обязанности менеджера транзакций в системе управления базами данных.

КВ3. Какой уровень изолированности допускает сериализацию выполнения транзакций?

КВ4. Каким образом администратор базы данных может управлять временем ее восстановления?

КВ5. Каким образом программист может контролировать влияние отката в ходе длинной транзакции?

Резюме

В данной главе описана чрезвычайно важная роль, которую играют базы данных при разработке программного обеспечения. Рассмотрены все основные вопросы, связанные с взаимодействием приложение–база данных.

Существуют три уровня моделей баз данных: внешняя, логическая и физическая. В этой главе мы сконцентрировались на логической модели. Под *отображением объектов в базу данных* мы понимаем отображение классов UML-модели в логическую модель базы данных.

Отображение в логическую модель реляционной базы данных иногда оказывается сложным. Модель реляционных баз данных не поддерживает объектные типы,

наследование, структурированные типы, коллекции или ссылки. Данные хранятся в таблицах, связанных ограничениями *ссылочной целостности*. Для выражения ограничений, связанных с моделированием, которые невозможно выразить посредством табличных структур, используются *хранимые процедуры* и *реляционные представления*. Кроме того, на отображение может влиять *нормализация* таблиц.

Прикладная программа должна общаться с базой данных. Это общение не должно нарушать целостность выбранной *архитектурной модели*. Модель PCBMER очень хорошо сочетается с проектированием баз данных. Существуют разные шаблоны проектирования для управления *персистентными объектами* в прикладном коде. Показано моделирование поиска, загрузки и выгрузки персистентных объектов.

При проектировании взаимодействия приложения и базы данных необходимо учитывать пять уровней *SQL-интерфейса*. *Уровень 5* SQL представляет особый интерес, поскольку он позволяет пользователю непосредственно программировать базу данных. Хранимые процедуры и триггеры сильно влияют на серверный аспект в проектировании программы.

Транзакция — это логическая единица работы с базой данных, начинающаяся, когда база данных пребывает в согласованном состоянии, и гарантирующая, что после ее завершения следующее состояние базы данных также будет согласованным. Транзакции гарантируют *параллельный доступ к базам данных* и *восстановление баз данных*. Как правило, приложения для работы с базами данных требуют *коротких транзакций*, хотя некоторые приложения предусматривают *длинные транзакции*.

Ключевые термины

Аномалия обновления (update anomaly). Нежелательный побочный эффект, возникающий в результате операции модификации таблицы (вставки, удаления или обновления).

Бизнес-транзакция (business transaction). Логическая единица работы с точки зрения делового приложения, которая может состоять из большого количества (системных) транзакций.

Блокировка (lock). Действие системы управления базами данных, которое "блокирует" записи (и другие внутренние структуры данных), размещая их в отдельном SQL-операторе, входящем в состав транзакции, чтобы изолировать выполнение этого оператора по отношению к конкурирующим транзакциям.

Внешний ключ (foreign key). Ключ, включенный в определение ограничения ссылочной целостности (ключ, ссылающийся на первичный ключ в таблице ссылок).

Домен (domain). Допустимый набор значений, который может содержаться в столбце.

Единица работы (unit of work). Шаблон, "поддерживающий список объектов, вовлеченных в бизнес-транзакцию, а также координирующий запись их изменений и решение проблем, связанных с параллельной работой" (Fowler, 2003).

Загрузка по требованию (lazy load). Шаблон, определенный как "объект, содержащий не все данные, необходимые пользователю, но знающий, как их найти" (Fowler, 2003).

Запись (record). См. *Строка*.

Ключ (key). Столбец или набор столбцов, определяющих определенное ограничение целостности, наложенное на таблицы и столбцы в реляционной базе данных.

Коллекция объектов (identity map). Шаблон, "гарантирующий, что каждый объект будет загружен только один раз, сохраняя каждый загруженный объект в определенной коллекции. При ссылке на объект шаблон производит перебор объектов с помощью коллекции" (Fowler, 2003).

Контрольная точка (checkpoint). Действие базы данных, записывающей в *регистрационный журнал* информацию о текущих транзакциях, чтобы сократить время восстановления после сбоя.

Модель данных (data model). Модель структур данных, хранящихся в базе данных. Она может также определять функциональные структуры, например триггеры и хранимые процедуры.

Нормализация (normalization). Процесс проектирования базы данных, позволяющий избежать аномалий обновления.

Объектно-реляционное отображение (object-relational mapping). Устройство или программа, предназначенные для отображения объектов приложения в базу данных, и наоборот.

Первичный ключ (primary key). Ключ, однозначно идентифицирующий строки в таблице. Каждая таблица может иметь только один первичный ключ.

Персистентный объект (persistent object). См. *Персистентный объект* в разделе "Ключевые термины" главы 6.

Представление (view). Хранимый и именованный SQL-запрос, представляющий себя пользователю в виде виртуальной таблицы.

Преобразователь данных (data mapper). Шаблон, определяющий "уровень преобразователей, осуществляющих перемещенные данных между объектами и базой данных, сохраняя их независимость как друг от друга, так и от преобразователя" (Fowler, 2003).

Регистрационный журнал (log). Специальный файл, поддерживаемый системой управления базой данных и содержащий все записи из базы данных, измененные транзакцией (образы записей до и после изменения).

Реляционная таблица (relational table). Основная единица определения данных и хранения данных в реляционной базе данных. Все данные, доступные пользователю, хранятся в таблицах.

Система ER (Entity-Relationship). Система сущность–отношение.

Ссылочная целостность (referential integrity). Правило, определенное для внешнего ключа в таблице, гарантирующее, что значения в этом ключе соответствуют значениям в первичном ключе связанной таблицы (ссылочное значение). Способ реализации ассоциаций в реляционных базах данных.

Стандарт JDBC (Java database connectivity). Стандарт интерфейса организации доступа Java-приложений к базам данных.

Стандарт ODBC (open database connectivity). Стандарт открытого интерфейса доступа к базам данных.

Столбец (column). Именованная колонка таблицы, имеющая определенный тип и демонстрирующая конкретный домен данных. См. также *Строка*.

Строка (row). Коллекция информации о столбцах, соответствующих одной строке в таблице. Реляционный эквивалент объекта в языке программирования. См. также *Столбец*.

Точка сохранения (savepoint). Оператор в программе базы данных, обозначающий точку в программе, в которой возможен откат транзакции без потери результатов, полученных после достижения предыдущей точки сохранения.

Транзакция (transaction). Логическая единица работы, состоящая из одного или нескольких операторов SQL, которые фиксируются или выполняют откат одновременно.

Триггер (trigger). См. *Триггер* в разделе "Ключевые термины" главы 6.

Хранимая процедура (stored procedure). Программа, хранящаяся в базе данных и способная воздействовать на базу данных после ее вызова.

Шаблоны PEAA (patterns of enterprise application architecture). Архитектурные шаблоны промышленных приложений.

Элементарный тип (primitive type). Встроенный тип данных, предусмотренный в языке программирования или в базе данных для поддержки основных операций. Программист может использовать элементарные типы для создания сложных пользовательских типов.

Многовариантные тесты

MT1. К какой базе данных относится стандарт SQL:1999?

а. Реляционной.

б. Объектно-реляционной.

в. Объектно-ориентированной.

г. Ко всем вместе.

MT2. Что из перечисленного ниже не поддерживается моделью реляционных баз данных?

а. Структурные типы.

б. Ссылки.

в. Коллекции.

г. Ничего из выше перечисленного.

Для чего можно использовать представление?

а. Для программирования бизнес-правил.

б. Для поддержки безопасности базы данных.

в. Для определения доменов.

г. Ни для чего из перечисленного выше.

MT3. Какая стратегия не разрешается для отображения отношений обобщения?

а. Отображение всей иерархии классов в одну таблицу суперкласса.

б. Отображение каждого отдельного конкретного класса в таблицу.

в. Отображение каждого абстрактного класса в таблицу.

г. Допускаются все перечисленные выше стратегии.

MT4. Какой шаблон определен как “объект, содержащий не все данные, но знающий, где их найти”?

а. Единица работы.

б. Коллекция объектов.

в. Преобразователь данных.

г. Загрузка по требованию.

MT5. Какой вид блокировки допускает “грязное чтение”?

а. Блокировка намерения.

б. Блокировка чтения.

в. Разделяемая блокировка.

г. Ни одна из перечисленных выше.

Вопросы

B1. Опишите три уровня моделей данных.

B2. Объясните смысл зависимостей между подсистемой ресурсов, схемой базы данных и программами базы данных на рис. 8.1.

B3. Что такое ссылочная целостность? Чем она может быть полезна при отображении из модели классов UML?

B4. Опишите четыре типа декларативных ограничений ссылочной целостности.

В5. Что такое триггер? Как он связан со ссылочной целостностью?

В6. Может ли хранимая процедура быть триггером? Обоснуйте свой ответ.

В7. Что считается правильным уровнем нормализации базы данных? Обоснуйте свой ответ.

В8. Вернитесь к рис. 8.11 (см. раздел 8.3.1). Предположим, что каждый объект класса `ContactPhone` должен быть связан с объектом класса `Contact` и не должен переключаться на другой объект класса `Contact`. Как эти ограничения влияют на базу данных?

В9. Вернитесь к рис. 8.12 (см. раздел 8.4.3). Рассмотрите ситуацию, в которой объект сущности необходимо выгрузить из базы данных в результате операции обновления, а не удаления. Как выглядит диаграмма последовательностей в этом случае?

В10. Кратко опишите пять уровней интерфейсов программирования на языке SQL.

В11. Каковы преимущества вызова хранимой процедуры над SQL-запросом, посланным клиентской программой базе данных? Существуют ли ситуации, в которых SQL-запрос предпочтительнее вызова хранимой процедуры?

В12. Кратко опишите блокировки в пессимистическом сценарии управления параллельной работой.

В13. Кратко опишите уровни изоляции транзакций.

В14. Может ли разработчик или администратор базы данных управлять временем восстановления базы данных? Обоснуйте свой ответ.

В15. Что такое компенсирующая транзакция? Как ее можно использовать при проектировании программ?

В16. Что такое точка сохранения? Как она используется в проектировании программ?

Упражнения. Управление взаимоотношениями с заказчиками

УВ31. Вернитесь к примеру 7.10 (см. раздел 7.4.3 главы 7) и упражнению УВ34 в разделе "Упражнения: управление взаимоотношениями с заказчиками" в конце главы 7. Проанализируйте рис. 7.43, на котором приведено объяснение этого упражнения. Обратите внимание на сообщение 8 на рис. 7.43 (`вывод обновляемой записи на экран`). Усовершенствуйте диаграмму последовательностей (начиная с сообщения 8), чтобы обеспечить управление персистентными объектами в соответствии с принципами, изложенными в разделе 8.4. Объясните эту модель.

УВ32. Вернитесь к примеру 7.10 (см. раздел 7.4.3 главы 7) и упражнению УВ34 в разделе "Упражнения: управление взаимоотношениями с заказчиками" в конце главы 7. Проанализируйте рис. 7.43, на котором приведено решение этого упражнения. Обратите внимание на сообщение 10 на рис. 7.43 (`нажатие клавиши для сохранения записи`). Усовершенствуйте диаграмму последовательностей (начиная с сообщения 10), чтобы обеспечить управление персистентными объектами в соответствии с принципами, изложенными в разделе 8.4. Объясните эту модель.

УВ33. Вернитесь к задаче 3, в которой описывается система управления взаимоотношениями с заказчиком (см. раздел 1.6.3 главы 1), и последующим примерам в главах 4, 5, 7 и 8. В частности, рассмотрите пример 5.3 (см. раздел 5.1.1.3 главы 5). Идентифицируйте триггеры базы данных для таблицы `Event` (Мероприятие).

УВ34. Вернитесь к задаче 3, в которой описывается система управления взаимоотношениями с заказчиком (см. раздел 1.6.3 главы 1), и к последующим примерам в главах 4, 5, 7 и 8. Идентифицируйте триггеры базы данных для таблицы `Task` (Задание).

УВ35. Вернитесь к задаче 3, в которой описывается система управления взаимоотношениями с заказчиком (см. раздел 1.6.3 главы 1), и к последующим примерам в главах 4, 5, 7 и 8. В частности, рассмотрите пример 5.3 (см. раздел 5.1.1.3 главы 5). Идентифицируйте хранимые процедуры, воздействующие на таблицу `Event`. Сформулируйте псевдокод алгоритма для идентифицированных хранимых процедур.

Упражнения. Прямой маркетинг по телефону

ПМТ1. Вернитесь к упражнению ПМТ1 в разделе "Упражнения. Прямой маркетинг по телефону" в конце главы 7. Проанализируйте диаграмму классов, построенную в результате выполнения этого упражнения. Отобразите эту диаграмму классов в модель реляционной базы данных. Объясните это отображение.

ПМТ2. Вернитесь к упражнению ПМТ8 в разделе "Упражнения. Прямой маркетинг по телефону" в конце главы 7. Проанализируйте диаграмму последовательностей, построенную в результате выполнения этого упражнения. Идентифицируйте на этой диаграмме все события (сообщения), связанные с управлением персистентными объектами в соответствии с принципами, изложенными в разделе 8.4. Усовершенствуйте эту диаграмму, включив в нее средства для управления персистентными объектами. Объясните эту модель.

Ответы на контрольные вопросы

Контрольные вопросы 8.1

КВ1. Нет, это не то же самое. Класс сущностей "приговорен" быть персистентным и именно так хранится в базе данных, но он не является персистентным по своей природе. Это различие еще больше размывается в объектно-ориентированных базах данных, хранящих классы сущностей как объекты (реляционные базы данных хранят их в таблицах как записи).

КВ2. Модель объектно-ориентированной базы данных.

КВ3. Диаграммы сущность–отношение (ER-диаграммы).

Контрольные вопросы 8.2

КВ1. Теория множеств (логика предикатов).

КВ2. Ключ должен быть уникальным и минимальным.

КВ3. Да, может.

КВ4. Аномалия обновления.

Контрольные вопросы 8.3

КВ1. В отображении ассоциаций "множество–множество".

КВ2. Никак не выражается — он просто игнорируется.

Контрольные вопросы 8.4

КВ1. Архитектурные шаблоны промышленных приложений.

КВ2. Шаблон *Преобразователь данных*.

КВ3. Шаблон *Единица работы*.

Контрольные вопросы 8.5

КВ1. Начиная с третьего уровня.

КВ2. Восстановление базы данных и управление параллельным выполнением операторов.

КВ3. Повторяемое чтение.

КВ4. Устанавливая частоту контрольных точек.

КВ5. С помощью точек сохранения, обеспечивающих сохранение персистентных объектов в закрытых базах данных пользователей.

Ответы к многовариантным тестам

МТ1. б

МТ2. г

МТ3. б

МТ4. в

МТ5. г

МТ6. г (“грязное чтение” возможно лишь в отсутствие блокировок).

Ответы на вопросы с нечетными номерами

В1

Разработчики моделей данных — люди, специализирующиеся на моделировании структур баз данных, — выделяют три уровня моделей данных: внешние, логические и физические.

Внешние и логические модели являются *концептуальными*. В них не рассматриваются особенности конкретной системы управления базами данных, а лишь обеспечивается соответствие с доминирующей моделью — моделью реляционных баз данных. Эта связь с реляционной моделью часто отражается в урезанной выразительности диаграмм сущность–отношение, используемых для создания внешних и логических моделей (“урезание” возникает из-за семантической простоты реляционной модели).

Внешняя модель данных ориентируется на отдельную прикладную систему. Это приложение, как правило, представляет собой отдельную выполняемую программу, которая объединяется с базой данных. Внешняя модель данных определяет структуры данных, необходимые для приложения.

Как правило, одна база данных поддерживает многие приложения. Требования базы данных к этим приложениям могут перекрываться и конфликтовать. Следовательно, необходимо интегрировать внешние модели в единую *логическую модель данных*. Хорошо продуманная логическая модель данных практически не связана с конкретными приложениями и обеспечивает структуру базы данных, на которой можно построить как существующие, так и потенциальные приложения.

Физическая модель данных возникает в результате преобразования логической модели в проект, соответствующий конкретной системе управления базами данных и конкретной версии этой СУБД. Итоговая физическая модель позволяет пользователям генерировать код для создания схем баз данных (включая триггеры и индексы) и, возможно, загружать тестовые данные в таблицы баз данных.

B3

Ссылочная целостность — это основной метод, гарантирующий, что данные, хранящиеся в реляционной базе, являются точными и корректными. Ссылочная целостность означает, что база данных не должна содержать значения внешних ключей, не соответствующие некоторым значениям первичного ключа.

Ссылочная целостность — это способ реализации ассоциаций между структурами данных в реляционных базах данных. Следовательно, ассоциации (и агрегации) в диаграммах классов UML отображаются в отношения первичный ключ–внешний ключ в моделях реляционных баз данных.

B5

Триггер — это процедурный (программный) способ реализации бизнес-правил, относящихся к данным. Некоторые бизнес-правила, регламентирующие отношения между данными в базе данных, невозможно реализовать с помощью декларативных средств. В этих ситуациях триггеры являются единственным способом программирования правил, касающихся ссылочной целостности данных.

Фактически триггеры обеспечивают *декларативную ссылочную целостность*. С помощью автоматической генерации кода триггера можно учесть любое ограничение, касающееся декларативной ссылочной целостности. Для того чтобы учесть более "экзотические" бизнес-правила, этот код затем можно расширить или перепрограммировать вручную.

B7

Правильный *уровень нормализации* обеспечивает оптимальность схемы базы (с точки зрения производительности и легкости эксплуатации) по отношению к способу ее использования. Поскольку шаблоны использования баз данных состоят из комбинации операций извлечения и обновления, правильный уровень нормализации подразумевает компромисс между извлечением и обновлением, при котором наиболее важные операции извлечения и обновления получают приоритет.

Динамические базы данных (содержание которых часто изменяется) должны иметь высокий уровень нормализации. С другой стороны, относительно *статичные* таблицы (в которых обновление производится лишь с целью создания резервных копий) должны иметь низкий уровень нормализации. В результате некоторые таблицы в базе данных могут иметь высокий уровень нормализации, а другие — низкий.

B8

Вопрос носит несколько гипотетический характер, поскольку степень обновления не указана и точная схема классов сущностей `EIdentityMap` (Коллекция объектов) и `EContact` (Контакт) неизвестна. Например, неясно, следует ли создать только один экземпляр класса `EIdentityMap` в программе или создать

отдельные экземпляры для каждого класса сущностей. В последнем случае статус объекта сущности ("чистого" или "грязного") может храниться в объекте `EIdentityMap`, а не в самом объекте сущности.

Основная разница между сценариями удаления и обновления заключается в том, что выгрузка в ходе обновления, как правило, не приводит к разрушению объекта сущности. Этот объект выгружается в базу данных с помощью применения SQL-операции `update` к таблице `Contact` (Контакт). Например, объект `EContact` является "чистым" и должен иметь соответствующую метку. Такая маркировка осуществляется с помощью сообщения `setClean()` или подобного ему сообщения, посланного объектом `MDataMapper` либо объекту `EIdentityMap`, либо непосредственно объекту `EContact`.

B11

Хранимые процедуры имеют важное преимущество над SQL-запросами, посылаемыми клиентом. Хранимые процедуры позволяют намного сократить трафик, они оптимизированы, скомпилированы и готовы к "запуску".

В крупных системах хранимые процедуры позволяют собирать код из блоков, использовать его повторно, внедрять в более крупные хранимые процедуры и вызывать из разных прикладных программ.

SQL-запрос может по-прежнему использоваться в клиентских программах в ситуациях, когда пользователь имеет возможность создавать *динамические запросы* к базам данных (неявно с помощью пользовательского интерфейса), иначе говоря, когда клиентская программа заранее не знает, в какой момент времени пользователь потребует выполнить обработку (критерий запроса).

B13

Язык SQL поддерживает четыре уровня *изолированности транзакций*. Эти четыре уровня в книге могут иметь разные названия, чтобы читатели могли следить за логикой. Перечислим эти четыре уровня (в скобках приведены синонимы, использованные в главе).

1. Незафиксированное чтение (возможно "грязное чтение").
2. Зафиксированное чтение (возможно неповторяемое чтение).
3. Повторяемое чтение (возможны фантомы).
4. Сериализуемый уровень (повторяемое чтение).

Незафиксированное чтение (read uncommited) означает, что в ходе чтения база данных не генерирует блокировок чтения (разделяемых блокировок). Следовательно, другая транзакция может считывать незафиксированную транзакцию, которая впоследствии может быть отменена (т.е. возможно "грязное чтение").

Зафиксированное чтение (read committed) означает, что база данных устанавливает блокировку чтения на считываемые данные. Считываться могут лишь

зафиксированные данные, однако данные могут быть изменены до завершения транзакции.

Повторяемое чтение (repeatable read) означает, что на все данные, используемые в транзакции, устанавливается блокировка, и другие транзакции не могут их обновлять. Однако другая транзакция может вставить новые элементы в данный набор данных и, если транзакция считает данные из этого набора снова, может возникнуть "фантомное чтение".

Сериализуемый уровень означает, что на все данные, используемые в транзакции, устанавливается блокировка и что другие транзакции не могут их обновлять или вставлять элементы в этот набор.

Обратите внимание на то, что если база данных поддерживает только *страничную блокировку* (т.е. не поддерживает блокировку на уровне строк), то уровни повторяемого чтения и сериализуемый уровень являются синонимами. Это объясняется тем, что другие транзакции не могут вставлять отдельные строки данных до завершения первой транзакции, поскольку вся страница данных блокирована.

B15

Компенсирующая транзакция отменяет уже зафиксированное действие транзакции. Компенсация является важным механизмом в ситуациях, когда пользователь должен отменить изменения, внесенные в базе данных, и ожидает, что программа позволяет выполнить откат (т.е. в пункте меню есть соответствующий пункт). В некоторых случаях возможно и желательно предусмотреть в программе компенсирующий механизм, позволяющий отменять сразу несколько уровней (т.е. выполнять ряд откатов).

В некоторых *современных моделях транзакции*, предусматривающих точки сохранения, допускаются *субтранзакции* (subtransaction), которые могут выполняться без каскада отмен других (возможно, многочисленных) транзакций.

Объяснение упражнений. Управление взаимоотношениями с заказчиками

УВ31

Окно Update Organization Status (Обновить статус организации), показанное на рис. 7.43, похоже на окно, представленное на рис. 7.34 (см. раздел 7.4.3 главы 7), за исключением того, что поле `statusCode` также показано как нередактируемое. Когда поток управления программы начинает навигацию по экрану, возникает необходимость найти последнее значение `statusDescription`. Для этого методу `getOrgStatus()` объекта `CAdmin` посылается сообщение (рис. 8.26). Объект `CAdmin` пересылает этот запрос объекту `MDataMapper`.

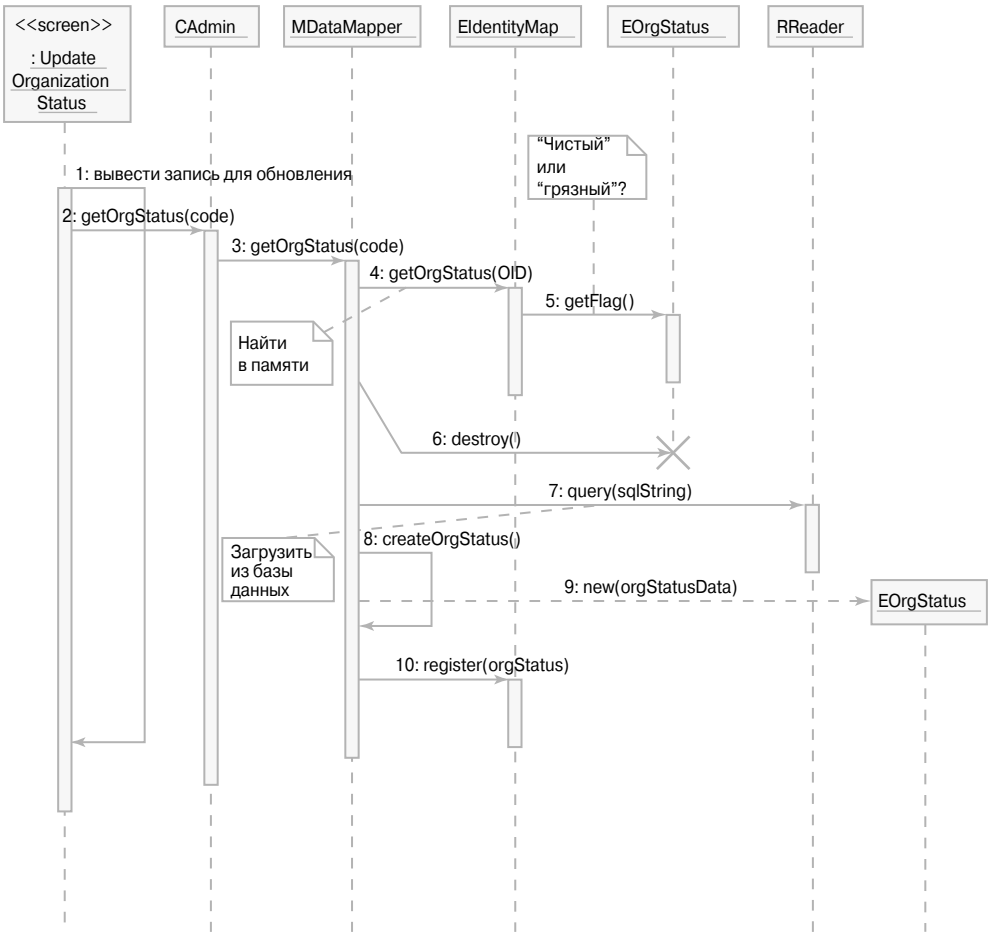


Рис. 8.26. Управление персистентностью после обращения к дисплею в системе управления взаимоотношениями с заказчиками

Как указывалось в разделе 8.4.1 (см. рис. 8.20), объект `MDataMapper` пытается получить объект `EOrgStatus` из кэш-памяти. Если объект `EOrgStatus` существует в памяти и является "чистым", то обработка прекращается (после чего объект `EOrgStatus` возвращается и выводится в окне Update Organization Status (рис. 8.21).

УВ32

Для изменения записи о статусе организации необходимо обновить базу данных и модифицировать объект сущности, находящийся в кэш-памяти. Как показано на рис. 8.27, запрос `save()` пересылается объектом `CAdmin` объекту `MDataMapper`. Зная эту информацию, объект `MDataMapper` может обратиться к объекту `RWriter` для обновления базы данных.

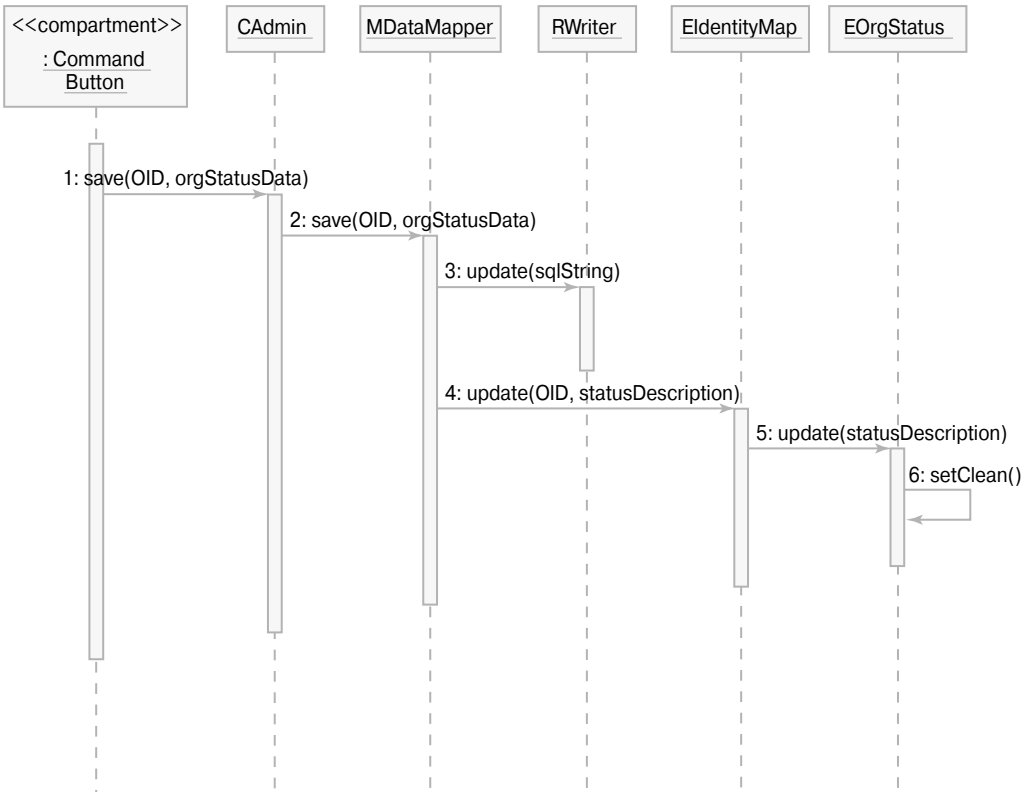


Рис. 8.27. Управление персистентностью после запроса на обновление в системе управления взаимоотношениями с заказчиками

Если обновление базы данных произошло успешно, то объект `MDataMapper` обращается к объекту `EIdentityMap`, а объект `EOrgStatus`, находящийся в памяти, модифицируется новым значением `statusDescription`. Модель, продемонстрированная на рис. 8.27, построена на основе предположения, что объект `EOrgStatus` хранит флаг, означающий, "чистым" является объект или нет. Этот флаг модифицируется с помощью сообщения `setClean()`.

УВЗЗ

В таблице `Event` нужны два *триггера*.

1. Для реакции на сообщение `insert(ti_event)`.
2. Для реакции на сообщение `update(tu_event)`.

Триггер *вставки* гарантирует следующее.

1. Внешний ключ `Event.task_id` соответствует существующему ключу `Task.task_id`.

2. Внешний ключ `Event.created_emp_id` соответствует существующему ключу `Employee.emp_id`.
3. Внешний ключ `Event.due_emp_id` соответствует существующему ключу `Employee.emp_id`.
4. Внешний ключ `Event.completed_emp_id` соответствует существующему ключу `Employee.emp_id` или равен `null`.

Триггер *обновления* гарантирует, что при попытке обновить столбцы внешнего ключа к нему будут применены те же самые бизнес-правила, что и к триггеру *вставки*. Столбцы внешнего ключа перечислены ниже.

- `Event.task_id`
- `Event.created_emp_id`
- `Event.due_emp_id`
- `Event.completed_emp_id`

УВ34

В таблице `Task` нужны три триггера.

1. Для реакции на сообщение `insert(ti_event)`.
2. Для реакции на сообщение `update(tu_event)`.
3. Для реакции на сообщение `delete(td_event)`.

Триггер *вставки* гарантирует следующее.

1. Внешний ключ `Task.contact_id` соответствует существующему ключу `Contact.contact_id`.
2. Внешний ключ `Task.created_emp_id` соответствует существующему ключу `Employee.emp_id`.

Триггер *обновления* гарантирует следующее.

1. Внешний ключ `Task.contact_id` не может быть модифицирован несуществующим ключом `Contact.contact_id`.
2. Внешний ключ `Task.created_emp_id` не может быть модифицирован несуществующим ключом `Employee.emp_id`.
3. Основной ключ `Task.task_id` не может быть модифицирован, если ассоциированные события по-прежнему существуют в таблице `Event`.

Триггер *вставки* гарантирует следующее.

1. Внешний ключ `Task.contact_id` соответствует существующему ключу `Contact.contact_id`.
2. Внешний ключ `Task.created_emp_id` соответствует существующему ключу `Employee.emp_id`.

Триггер *удаления* гарантирует следующее.

1. Все ассоциированные события в таблице `Event` также удаляются (каскад удалений прекращается в таблице `Event`).

УВ35

Хранимые процедуры, предназначенные для усиленной поддержки целостности транзакций в таблице `Event` (т.е. процедуры, модифицирующие таблицу `Event` и взаимодействующие с триггерами), приведены ниже.

- Сохранение события (`SaveTaskEvent_SP`).
- Удаление последнего события (`DeleteEvent_SP`).

К *хранимым процедурам*, модифицирующим таблицу `Event` без риска повреждения ссылок внешнего ключа на другую таблицу, относится следующая процедура (обратите внимание на то, что хранимая процедура, модифицирующая таблицу без риска повреждения ссылок внешнего ключа, либо не изменяет никаких столбцов внешнего ключа, либо прикладная программа гарантирует, что значения внешнего ключа являются корректными, например, используя списки для выбора значений, предлагаемые таблицей данных).

- Сохранить запись о факте, что мероприятие завершено (`CompleteEvent_SP`).

К *хранимым процедурам*, лишь извлекающим информацию из базы данных и "касающихся" таблицы `Event`, относится следующая.

- Найти ежедневные действия сотрудников в подсистеме управления взаимоотношениями с заказчиками (`DailyActivity_SP`).

Процедура `SaveTaskEvent_SP` выполняет следующий *алгоритм*.

```
BEGIN
 INPUT PARAMETERS (все известные (не нулевые) поля, соответствующие
столбцам таблиц Task и Event)
 IN/OUT PARAMETERS (@task_id, @event_id)
 IF @task_id нулевое
   AND другие поля таблицы Task не нулевые
     (за исключением поля @value, которое может быть нулевым)
   THEN
         вставить в таблицу Task
   ELSE
         IF @task_id не нулевое
             AND по крайней мере одно из других полей Task не нулевое
             THEN
                    обновить таблицу Task
         ENDIF
 ENDIF
 IF @task_id не нулевое
```

```
  if @event_id нулевое
   AND другие требуемые поля таблицы Event не нулевые
   THEN
         вставить в таблицу Event
   ELSE
         IF @event_id не нулевое
               THEN
               обновить таблицу Event
         ENDIF
   ENDIF
 ENDIF
 Любой другой вариант, не предусмотренный приложением, но допускаемый
триггерами,
 приведет в откату транзакции и выводу сообщения об ошибке.
END
```

Алгоритм процедуры `DeleteEvent_SP` выглядит так:

```
 BEGIN
 INPUT PARAMETERS (@event_id, @user_id)
 Select Event (где event_id = @event_id)
 IF @user_id = Event.created_emp_id
   THEN
         delete Event (где event_id = @event_id)
         IF мероприятий больше нет
         Tast.task_id = Event.task_id AND
         Event.event_id = @event_id
      THEN
         delete таблицу Task
      ENDIF
   ELSE
     raise error ("Удалить мероприятие может только тот, кто его запланировал")
 ENDIF
 END
```

Алгоритм процедуры `CompeteEvent_SP` выглядит так:

```
 BEGIN
 INPUT PARAMETERS (@event_id, @completed_dt, @completed_emp_id)
 IF @все параметры не нулевые
   THEN
       update Event (где event_id = @event_id)
   ELSE
       raise error ("Удалить мероприятие может только тот, кто его
запланировал")
 ENDIF
 END
```

Алгоритм процедуры `DailyActivity_SP` выглядит так:

```
BEGIN
INPUT PARAMETERS (@date, @user_id)
OUTPUT PARAMETERS (все столбцы из таблиц Rask, Event и выбранные столбцы
из таблиц Contact и PostalAddress)
LOOP (пока события в таблице Event не будут исчерпаны)
  Select Event (где (due_dt <= @date_OR
               completed_dt = @date) AND
               due_emp_dt = @user_id)
  Select Task для данного поля Event.task_id
  Select ActionRef для данного поля Event.action_id
  Select Contact для данного поля Task.contact_id
  Select PostalAddress для данного поля Contact.contact_id
ENDLOOP
END
```

ГЛАВА

9 Управление качеством и изменениями

Цели

9.1. Управление качеством
9.2. Управление изменениями
Резюме
Ключевые термины
Многовариантные тесты
Вопросы
Ответы на контрольные вопросы
Ответы к многовариантным тестам
Ответы на вопросы с нечетными номерами

Цели

В данной главе мы возвращаемся к основным вопросам разработки систем, описанным в главе 1, и, в определенном смысле, закрываем жизненный цикл разработки. Вопросы, рассматриваемые здесь, — управление качеством и изменениями — определяют, способна ли организация достичь двух наивысших уровней технологической зрелости см. раздел 1.1.2.2.2 главы 1).

Управление качеством разделяется на *поддержку качества* (quality assurance) и *контроль качества* (quality control). *Поддержка качества* представляет собой профилактические методы обеспечения качества системы программного обеспечения. *Контроль качества* сводится к (в основном реактивным) способам проверки качества системы программного обеспечения.

Управление изменениями (change management) — это фундаментальный аспект управления проектом в целом. Оно предусматривает, что запросы на изменения должны быть документированы, а влияние каждого изменения на артефакты разработки должно отслеживаться и перепроверяться после их реализации.

Прочитав эту главу, читатели будут

- понимать взаимосвязь между управлением качеством и управлением изменениями;
- понимать разницу между поддержкой качества и контролем качества;
- знать общепринятые методы поддержки качества — контрольные списки, экспертизу и аудит;
- владеть методом поддержки качества на основе разработки посредством тестирования;
- владеть тестированием как основным методом контроля качества;
- осознавать разницу между тестированием системных функций и тестированием системных ограничений;
- осознавать необходимость управления изменениями по формальному запросу;
- понимать, что трассируемость является необходимым условием внесения изменений.

req_analysis_system design.indb 660 05.06.2008 14:43:34

9.1. Управление качеством

Управление качеством (quality control) представляет собой часть *процесса управления программным обеспечением* (software process management) наряду с такими видами деятельности, как управление персоналом, риском и внесением изменений. Некоторые аспекты управления качеством переплетаются с другими проблемами управления.

Исключение составляет *управление проектом* (составление календарного плана, оценка сметы, отслеживание прогресса). Желательно, чтобы управление качеством осуществлялось параллельно и дополняло управление проектом. Управление качеством должно иметь свой собственный бюджет и календарный план. Одной из его задач является гарантия качества управления проектом. Действия и результаты, полученные в рамках управления качеством и изменениями, образуют *базу для оценки эффективности процесса* (performance measurement baselines) (Heldman, 2002).

Целью управления качеством являются *программные продукты*, а также *процессы*, используемые при их разработке. Существует много желательных *характеристик программного обеспечения*, важность которых в разных проектах варьируется. Эти свойства являются дополнительными и должны соответствовать наиболее важным целям системы, т.е. функциональным требованиям, которым должен удовлетворять программный продукт. В книге (Maciaszek and Liong, 2005) перечислены следующие желательные характеристики систем программного обеспечения.

- Корректность.
- Надежность.
- Устойчивость.
- Эффективность.
- Удобность.
- Понятность.
- Легкость эксплуатации (восстановления).
- Масштабируемость (возможность развития).
- Возможность повторного использования.
- Мобильность.
- Способность к взаимодействию.
- Производительность.
- Своевременность.
- Обозримость.

Трем из этих характеристик в книге уделено особое внимание, поскольку они являются наиболее важными для долговременной живучести и конкурентоспособности компаний. Это понятность, легкость сопровождения и масштабируемость, которые в совокупности обеспечивают *адаптивность* системы.

9.1.1. Поддержка качества

Поддержка качества (quality assurance) связана с определением процессов управления и стандартов, гарантирующих качество готовой продукции. В этом смысле стандарты и модели процессов, рассмотренные в разделе 1.1.2.2, относятся к поддержке качества.

Поскольку эти процессы и стандарты не зависят от управления проектом, поддержка качества должна осуществляться отдельным подразделением — *группой поддержки качества программного обеспечения* (software assurance quality team — SQA team), или *группой SQA*. В группу SQA должны входить самые опытные и квалифицированные специалисты организации. Эта группа не должна вовлекаться в процесс разработки проекта, за исключением аспектов, связанных с поддержкой качества. Именно группа SQA (а не разработчики проекта!) несет ответственность за качество окончательного продукта. Она должна отчитываться перед руководством на уровне, соответствующем важности и масштабу разрабатываемого проекта, например на оперативном, тактическом и даже стратегическом уровнях.

9.1.1.1. Контрольные списки, экспертиза и аудит

Тремя наиболее распространенными способами поддержки качества являются **контрольные списки** (checklists), **проверки** (reviews) и **аудит** (audit). *Контрольный список*, как следует из его названия, — это заранее определенный список обязательных пунктов, которые должны быть скрупулезно проверены в ходе разработки. Любая компания. специализирующаяся в области информационных технологий и применяющая устоявшийся процесс разработки программного обеспечения, имеет свой собственный контрольный список действий, которые обязан выполнить разработчик. Стандарты организаций часто также содержат пункты контрольных списков, так что компании могут проверять соответствие стандартам, сверяя свои действия с этими списками.

Поскольку не бывает двух одинаковых проектов, эти процессы могут изменяться от проекта к проекту. Следовательно, контрольные списки не фиксируются раз и навсегда — они должна заново "генерироваться", отдельно для каждого нового проекта. Кроме того, "базовые" контрольные списки должны учитывать новые информационные технологии и изменения парадигм разработки информационных систем.

Проверка — это разновидность пассивного тестирования. Она осуществляется путем проведения формального совещания разработчиков и, возможно, менеджеров, посвященного тестированию работы продукции или процесса. Существуют два распространенных вида проверок: **сквозной контроль** (walkthrough) и **инспекции** (inspections).

Сквозной контроль представляет собой один из видов формальной проверки артефактов методом "мозгового штурма", который может проводиться на любом этапе разработки. Он проводится в ходе тщательно спланированной дружеской

встречи разработчиков, имеющей ясно определенные цели, повестку дня, продолжительность и состав участников. Многие группы разработчиков информационных систем проводят сквозной контроль каждую неделю.

За несколько дней до совещания по сквозному контролю участникам раздают материалы (описания моделей, документы, программы и т.д.), подлежащие проверке на совещании. Материалы собирает и раздает участникам модератор сквозного контроля. Участники изучают материалы и возвращают их модератору со своими комментариями до начала совещания.

Само совещание проходит относительно недолго (не дольше двух-трех часов). Во время совещания модератор представляет сделанные разработчиками замечания и открывает обсуждение по каждому вопросу. Цель совещания — выявить проблемы, а не искать виновных! Разработчик, создавший проблему, в данном случае не важен и может вообще оставаться анонимным (хотя обычно это не так). Вся идея состоит в том, чтобы подтвердить существование проблемы. При этом не следует предпринимать даже попыток решить эту проблему на совещании.

Выявленные проблемы заносятся в *список вопросов, подлежащих сквозному контролю* (Pressman, 2005), который раздают разработчикам после совещания. Этот список используется разработчиками для решения проблем или пересмотра процесса. После исправления ошибок разработчик информирует об этом модератора, который самостоятельно решает, проводить ли новое совещание или нет.

Существует множество свидетельств полезности и эффективности сквозного контроля. Он придает процессу разработки строгость и профессионализм, вносит существенный вклад в достижение высокой производительности труда, способствует своевременному выполнению работы, значительно повышает информированность участников проекта и в конечном итоге позволяет повысить качество разрабатываемого программного обеспечения.

Подобно сквозному контролю, *инспекция* (inspection) осуществляется на дружеском совещании, но в отличие от сквозного контроля это совещание проводится под строгим контролем со стороны руководства проекта. Ее целью также является выявление всех **дефектов**, подтверждение того, что они действительно являются дефектами, их фиксация, назначение сроков устранения и ответственных лиц.

В отличие от сквозного контроля инспекции проводятся менее часто, могут быть посвящены только отдельным, имеющим критическое значение, вопросам и носят более формальный и строгий характер. Инспекция состоит из нескольких этапов. Она начинается со *стадии планирования*, на которой определяются участники инспекции и целевая область инспектирования.

До начала инспекционного заседания необходимо провести короткую информационную встречу. Во время информационного совещания разработчик, продукт которого подлежит инспектированию, описывает суть вопроса. Инспекционные материалы раздают участникам во время информационного совещания или перед ним.

Информационная встреча обычно проводится за неделю до инспекционного совещания. Это дает инспекционной группе время на изучение материалов и подготовку к совещанию. Во время совещания дефекты идентифицируются, фиксируются и нумеруются. Сразу после совещания модератор готовит *журнал дефектов* (defect log) — в идеале он ведется с использованием *средства управления изменениями*, связанного с проектом.

Обычно разработчик должен быстро устранить дефекты и зафиксировать принятое решение с помощью средства управления изменениями. Модератор должен проверить, что дефект устранен, и решить, требуется ли *повторное инспектирование*. Если модератор удовлетворен решением проблемы, то, проконсультировавшись с руководителем проекта, он направляет разрабатываемый модуль группе поддержки качества (если она существует).

Группа SQA должна состоять из лучших специалистов, имеющихся в организации. Группа никак не должна быть связана с проектом за исключением своей роли по обеспечению качества. Группа (а не истинные разработчики!) несет ответственность за конечное качество продукта.

Аудит разработки системы — это процесс поддержки качества, напоминающий обычный бухгалтерский аудит. Это очень формальный процесс, имеющий массу нюансов. Аудит заранее тщательно планируется и предусматривает исследование продукции и/или процессов, проведение собеседований и инспекций.

Аудит всегда входит в сферу общего управления разработкой программного обеспечения. Он связан с управлением риском, оценкой стратегического значения информационных технологий для предприятия и ориентируется на руководство подразделений информационных технологий. В ходе аудита осуществляется оценка соответствия проверяемой продукции инвестициям в контексте общего предназначения предприятия и целей бизнеса.

Анхелкар (Unhelkar, 2003) выявил следующие отличия между аудитом и другими методами поддержки качества.

- Производителем продукции или исполнителем процесса, подлежащих аудиту, обычно является группа, а не отдельный человек.
- Аудит можно проводить в отсутствие производителя продукции.
- В ходе аудита широко используются контрольные списки и собеседования, и менее широко — проверки.
- Аудит может быть внешним, т.е. его может проводить внешняя организация.
- Аудит может проводиться в течение дня, недели и даже дольше, но он всегда имеет четкие границы и цели.

9.1.1.2. Разработка посредством тестирования

Разработка посредством тестирования (test-driven development), ставшая широко распространенной благодаря методам ускоренного проектирования программного обеспечения, — это очень практичный способ поддержки качества.

Она обеспечивает качество программного обеспечения за счет прямого требования, чтобы тестовый код создавался раньше прикладного и чтобы приложение проходило обязательное тестирование.

Методы ускоренного проектирования программного обеспечения (см. раздел 1.5.4 главы 1) способствовали повышению популярности разработки посредством тестирования. В ее основе лежит идея писать **тестовые спецификации** (test case) и **тестовые сценарии** (test scripts), а также тестовые программы до разработки (проектирования и программирования) кода приложения (тестируемой единицы). В противоположность обычной последовательности действий, при разработке посредством тестирования код приложения создается в ответ на тестовый код, а тестовый код можно использовать для проверки кода приложения, как только это станет возможным.

Разработка посредством тестирования имеет много преимуществ, не говоря уже о том, что она позволяет выяснять требования пользователя (и спецификации прецедентов использования) еще до того, как программист напишет первую строку кода приложения. Тестовый код содержит *точки верификации* (verification point), позволяющие проверить, всем ли требованиям пользователя удовлетворяет приложение. В некотором смысле тестовый код создается для того, чтобы испытать и выявить недостатки кода приложения.

Поскольку тестовый код разрабатывается заранее, программист может написать код приложения, ориентируясь на точки верификации, чтобы обеспечить требуемые функциональные свойства. Следовательно, разработка посредством тестирования носит очень активный характер и фактически является средством создания программного обеспечения, а не просто верификации программного обеспечения.

Популярность разработки посредством тестирования привела к появлению соответствующих шаблонов и каркасов, а также библиотек классов и интерфейсов. Одним из наиболее распространенных *тестовых каркасов* для разработки Java-приложений является открытое программное обеспечение JUnit (JUnit, 2004).

Библиотека JUnit — это каркас, разработанный для одного из наиболее известных шаблонов проектирования — *композиционного шаблона* (Gamma et al., 1995). Каркас JUnit содержит Java-интерфейс `Test`, который реализуется двумя классами — `TestCase` и `TestSuite`, связанными с классом `TestCase`, собирающим результаты тестирования. Модель классов для каркаса JUnit показана на рис. 9.1. Эта модель использует композиционный шаблон проектирования.

Поскольку этот каркас соответствует композиционному шаблону, класс `TestSuite` является композиционным классом, реализующим интерфейс `Test`, но может также содержать один или несколько интерфейсов `Test`. Поскольку интерфейс `Test` реализуется классами `TestCase` и `TestSuite`, объект `TestSuite` может содержать один или несколько объектов `TestCase` и/или один или несколько объектов `TestSuite`.

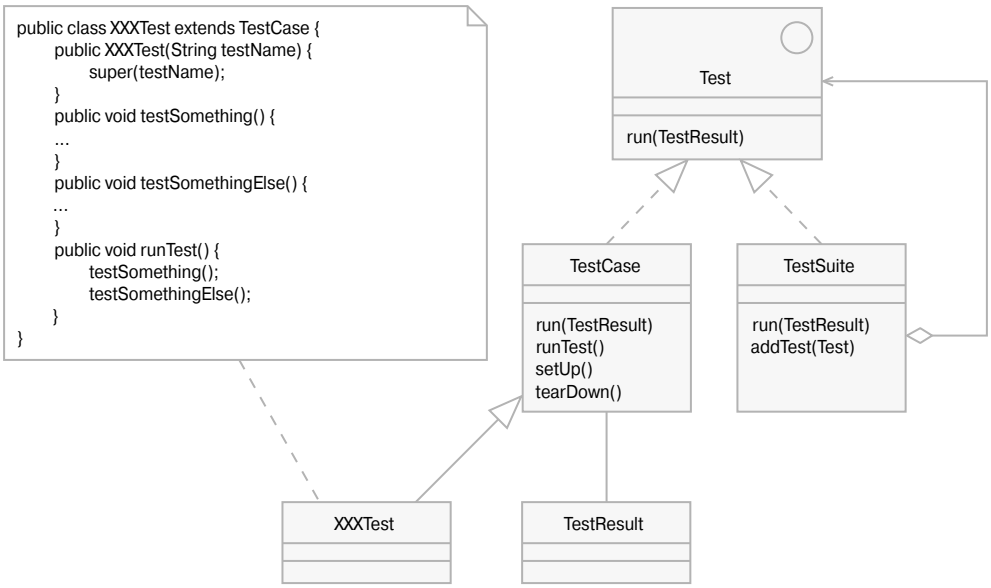


Рис. 9.1. Каркас JUnit как композиционный шаблон. Источник: Maciaszek and Liong, 2005. Воспроизводится с разрешения компании Pearson Education Ltd.

Тестируемый модуль (test unit), например `XXXTest`, реализуется как подкласс класса `TestCase`, способный манипулировать конкретными объектами, входящими в композицию посредством интерфейса `Test`. Фактически класс `XXXTest` реализует тестовые спецификации для проверки модуля, указанного с помощью префикса `XXX` (`XXX` может быть, например, именем тестируемого класса). Как показано во фрагменте кода на рис. 9.1, класс `XXXTest` должен использовать метод `runTest()` для запуска тестируемого модуля в рамках требуемых тестовых спецификаций. Каркас JUnit предлагает визуальный интерфейс для запуска тестов, демонстрации хода их выполнения и отчета о результатах тестирования (Maciaszek and Liong, 2005).

9.1.2. Контроль качества

В то время как целью поддержки качества является ее внедрение в продукты программного обеспечения или процессы разработки, **контроль качества** (quality control) предназначен для тестирования продукции или процессов. Поддержка качества носит активный характер, а контроль качества — реактивный. Поддержка качества имеет стратегическое значение, а контроль качества — тактическое и даже оперативное.

Поскольку контроль качества сводится в основном к *тестированию программного обеспечения,* он охватывает весь жизненный цикл проекта. На этапах, предшествующих появлению какой-либо продукции, подлежащей тестированию,

тестирование принимает вид поддержки качества и применяется к артефактам моделирования программного обеспечения и проектной документации. Одним из ранних проявлений тестирования являются контрольные списки и проверки.

Как только появляются продукты программного обеспечения (код приложения), начинается тестирование модулей. Как указывает Пфлегер (Pfleger, 1998), последовательность тестирования артефактов программного обеспечения противоположна последовательности этапов их производства (рис. 9.2). Соответственно, тестирование интеграции связано со спецификациями проекта, функциональное тестирование проистекает из анализа спецификаций, тестирование ограничений связано с нефункциональными требованиями (ограничениями) и т.д.

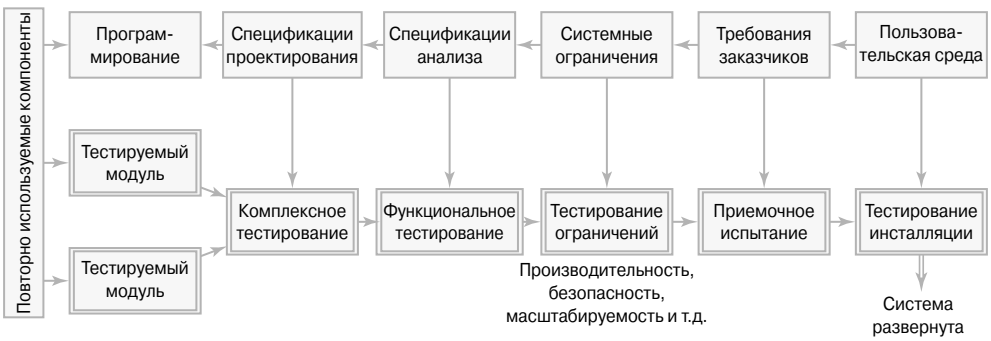


Рис. 9.2. Этапы тестирования и стадии разработки. Источник: Maciaszek and Liong, 2005. Воспроизводится с разрешения компании Pearson Education Ltd.

9.1.2.1. Концепции и методы тестирования

Тестирование — важная часть общего плана управления качеством. В **плане тестирования** (test plan) должны быть описаны расписание испытаний, тестовые задания и ресурсы. План тестирования должен влиять на процессы управления, например устранение дефектов и **совершенствование проекта** (enhancements).

Документация, относящаяся к тестированию и управлению изменениями, является составной частью другой системной документации, включая спецификации прецедентов использования (рис. 9.3). *Свойства* системы, указанные в модели бизнес-прецедента (см. раздел 2.5.2 главы 2), можно использовать для создания первичного плана тестирования. Модель прецедента использования можно использовать для создания **тестовых спецификаций** и определения **тестовых требований** (test requirements). *Дефекты*, выявленные в ходе тестирования, фиксируются в документах о дефектах. Любые нереализованные требования к прецедентам использования перечисляются в документах об усовершенствованиях.



Рис. 9.3. Основные документы, касающиеся контроля качества

При работе с CASE-средством разработчики могут использовать следующие возможности.

- Создавать описательные документы, а затем использовать их для формулировки требований (тестовых требований, требований прецедента использования и т.д.) в CASE-репозитарии.
- Использовать CASE-средство для ввода требований в репозитарий с последующей генерацией документации.

На рис. 9.4 показан фрагмент документа тестового прецедента для ввода тестовых требований в репозитарий. Как и требования бизнес-прецедента, тестовые требования имеют порядковый номер и образуют иерархию. Многие тестовые требования прямо соответствуют требованиям прецедента использования. По этой причине основная часть документа, продемонстрированного на рис. 9.4, называется “Conformance to use case specs” (“Соответствие спецификациям прецедента использования”).

В других разделах документа тестового прецедента идентифицируются требования к тестированию графического пользовательского интерфейса, баз данных и обобщенных компонентов, допускающих повторное использование. Эти виды тестирования должны быть включены в документ, описывающий функциональное тестирование, по двум причинам. Во-первых, для тестирования графического пользовательского интерфейса, базы данных или обобщенных компонентов (например, компонентов динамически подключаемых библиотек) необходимо, чтобы исходные и результирующие данные имели смысл в функциональном контексте.

Во-вторых, дефекты графического пользовательского интерфейса, баз данных и обобщенных компонентов могут проявляться только в контексте некоторых, но не всех функциональных тестов.

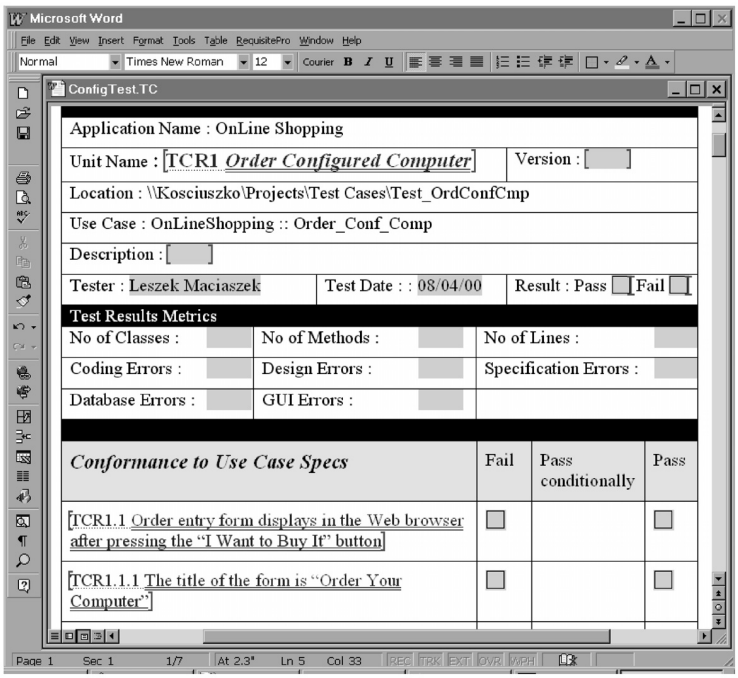


Рис. 9.4. Документ тестового прецедента. (Снимок экрана воспроизводится с разрешения компании Microsoft Corporation.)

Тестовые спецификации используются для записи результатов тестирования в трех столбцах, следующих после каждого тестового требования, указанного на рис. 9.4. Тест может выявить нарушение тестового требования, а также подтвердить его условное (в этом случае потребуются дополнительные объяснения) или безусловное выполнение.

С помощью прецедентов использования и результирующих тестовых требований можно создать *тестовые спецификации*, идентифицированные ранее в *плане тестирования*. Требования в тестовых спецификациях должны допускать отслеживание вплоть до тестовых требований и требований прецедентов использования.

Тестовые спецификации реализуются в виде *тестовых сценариев*. Для того чтобы гарантировать, что программный продукт соответствует тестовым требованиям, в этих сценариях указываются *этапы*, которые должен пройти испытатель, и *точки верификации* (т.е. вопросы, на которые испытатель должен найти ответы). Тестовые сценарии объединяются в более крупные **тестовые наборы**

(test suite), образующие иерархию, в которых более крупные тестовые наборы содержат более мелкие.

Взаимосвязь между концепциями тестирования показана на рис. 9.5 в виде модели классов (Maciaszek and Liong, 2005). В этой модели классов показано также, что некоторые тестовые сценарии можно автоматизировать, а другие можно проверить только вручную.

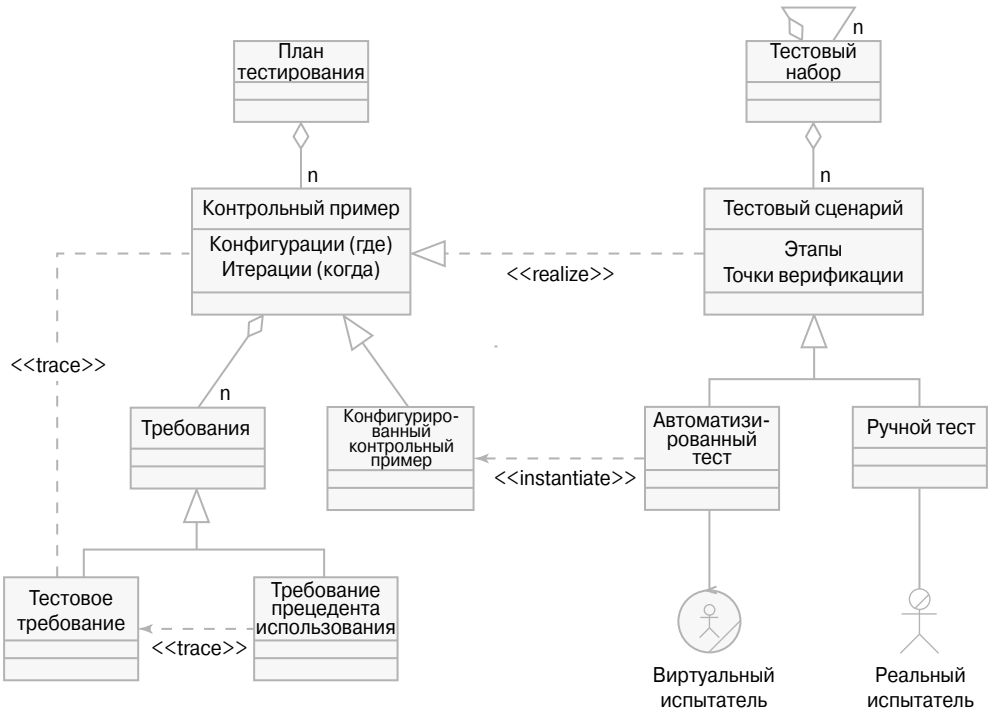


Рис. 9.5. Среда тестирования. Источник: Maciaszek and Liong, 2005. (Воспроизводится с разрешения компании Pearson Education Ltd.)

Ручной тест проводится человеком, просто запускающим модуль (прикладную программу) на тестовом примере и наблюдающем за результатами. *Автоматизированный тест* выполняется виртуальным испытателем, который представляет собой специальную рабочую группу, запускающую модуль на тестовом примере, и автоматически выполняет этапы, проходя точки верификации тестового сценария.

Виртуальный испытатель — это *инструмент записи-воспроизведения*. Сначала этот инструмент создает тестовый сценарий, фиксируя события, связанные с графическим пользовательским интерфейсом, а также другие события, возникающие в ходе тестирования модуля. Затем он воспроизводит записанный сценарий и проверяет модуль снова и снова. Соответственно, автоматизированное тестирование

в большой степени является регрессивным. "**Регрессионное тестирование** представляет собой повторное выполнение соответствующих приемочных тестов в ходе последовательных итераций. Цель регрессионного тестирования — убедиться, что итеративные расширения (приращения) кода не порождают нежелательных побочных эффектов и ошибок в старых частях кода" (Maciaszek and Liong, 2005).

9.1.2.2. Тестирование системных функций

Выбор *методов тестирования* продукции и процессов зависит от многих факторов. Основным фактором является природа тестируемой продукции и процесса. Очевидно, что для проверки программ, моделей и документов нужны разные методы тестирования. Вторым по важности фактором является размах и тщательность теста. Кроме того, необходимы разные методы тестирования *системных функций* и *системных ограничений* (см. раздел 9.1.2.3).

Шах (Schach, 2005) проводит различие между *неформальным* и *методическим* тестированием системных функций. Каждый разработчик выполняет *неформальное тестирование* при моделировании или реализации системных функций. По своему характеру неформальное тестирование несовершенно. Разработчик сервиса менее всего заинтересован в поиске собственных ошибок.

Значение неформального тестирования пренебрежимо мало и должно быть дополнено методическим тестированием. Существуют два основных вида *методического тестирования* (Schach, 2005).

1. Тестирование без выполнения программы — формальные проверки (см. раздел 9.1.1.1).
   - Сквозной контроль.
   - Инспекции.
2. Тестирование, основанное на выполнении программы.
   - Тестирование по спецификации.
   - Тестирование по коду.

Тестирование по спецификации (testing on specs) представляет собой одну из разновидностей тестов, основанных на выполнении программы. Оно применимо к выполняемым программным продуктам, а не документам или моделям. Этот вид тестирования также известен как *тестирование по методу "черного ящика"*, функциональное тестирование, тестирование по входу-выходу и т.д.

Принцип тестирования по спецификации заключается в том, что разработчик рассматривает тестируемый модуль как "черный ящик", получающий некоторую входную информацию и вырабатывающий некий результат. При этом не делается никаких попыток понять программную логику или вычислительные алгоритмы.

При тестировании по спецификации необходимо выработать *тестовые требования* (test requirements), исходя из *требований прецедентов использования*, а затем идентифицировать и зафиксировать их в отдельных документах, пред-

ставляющих план тестирования и тестовые спецификации. Эти документы предоставляют в распоряжение специалиста по тестированию *сценарий тестирования* (test scenario). Эти сценарии можно записывать с помощью специальных *средств записи-воспроизведения* (capture/playback tool), а затем использовать для *регрессионного тестирования*.

Тестирование по отношению к спецификации применяется для выявления дефектов, которые трудно выловить другим способом. В частности, тестирование по спецификации позволяет обнаружить *пропущенные функциональные свойства* — нечто, что (вероятно) было указано в документах как требование прецедента использования (а следовательно, тестовое требование), однако не было запрограммировано. Этот вид тестирования позволяет также выявить пропущенные функциональные свойства, которые не были оформлены документально в описании прецедентов использования, но явно пропущены в реализации системы.

Тестирование по коду (testing to code) представляет собой второй вид тестирования, основанный на выполнении программы. Он также известен как *тестирование по методу "прозрачного ящика"*, *тестирование по методу "стеклянного ящика"*, *логическое тестирование программы* и тестирование ветвей программы.

Тестирование по коду начинается с тщательного анализа алгоритмов программы. Для *выполнения программы* создаются тестовые прецеденты, т.е. гарантии проверки всех возможных выполняемых ветвей программы. Для выполнения программы придумывается специальный набор данных.

Тестирование по коду поддерживается средствами записи-воспроизведения и регрессионного тестирования. Однако характер тестирования по коду требует широкого вмешательства программистов. Многие сценарии воспроизведения тестов должны быть написаны программистами, а не сгенерированы средствами автоматизации. Даже если сценарии генерируются автоматически, они могут потребовать существенной модификации с привлечением программиста.

Подобно всем прочим видам тестирования, основанным на выполнении программы, тестирование по коду не может быть исчерпывающим из-за резкого увеличения количества возможных тестовых прецедентов, даже при умеренном росте сложности программы. Даже если существует возможность тестирования каждой выполняемой ветви программы, нет никаких гарантий, что все дефекты будут обнаружены. Известная мудрость гласит: тестирование способно устранить лишь некоторые ошибки, но не в состоянии доказать правильность программы!

9.1.2.3. Тестирование системных ограничений

Тестирование системных ограничений (testing of system constraints) основано на *выполнении программы*. Его цель — проверить, что системные ограничения были реализованы в соответствии с требованиями заказчиков и тестовой документации. Тестирование системных ограничений включает ряд вопросов; некоторые из них приводятся ниже.

- Тестирование пользовательского интерфейса.
- Тестирование баз данных.
- Тестирование средств авторизации.
- Тестирование производительности.
- Стрессовое тестирование.
- Тестирование отказов.
- Тестирование конфигураций.
- Тестирование инсталляции.

Первые два типа тестирования системных ограничений — *тестирование пользовательского интерфейса* и *баз данных* — очень тесно связаны с тестированием системных функций. Они обычно проводятся параллельно с тестированием системных ограничений. По этой причине они включаются в тестовые документы, создаваемые для тестирования системных услуг.

9.1.2.3.1. Тестирование пользовательского интерфейса

Тестирование графического пользовательского интерфейса пронизывает весь процесс разработки программного обеспечения. Оно начинается на ранней стадии этапа выработки требований, сопровождая такие виды деятельности, как работа с архивными документами, включение эскизов окон в документы описания прецедентов и разработка прототипов GUI-интерфейса. Эти ранние тесты графического пользовательского интерфейса концентрируются на удовлетворении функциональных требований и удобстве использования приложения.

Позднее, после реализации системы, требуется *послереализационное тестирование* графического пользовательского интерфейса. Вначале тестирование проводится разработчиками, затем специалистами по тестированию и — перед выпуском программного обеспечения — заказчиками (*пилотное тестирование*). Ниже приведен примерный перечень вопросов тестового документа, разработанного для послереализационного тестирования графического пользовательского интерфейса (Bourne, 1997).

- Соответствует ли название окна его предназначению?
- Является ли окно модальным или немодальным? Каким оно должно быть?
- Существует ли визуальное различие между обязательными и необязательными полями?
- Можно ли изменить размеры окна, переместить его и восстановить? Необходимо ли это?
- Пропущены ли какие-либо поля?
- Есть ли орфографические ошибки в названиях, метках, именах приглашений для ввода и т.д.?
- Нет ли противоречий в использовании командных кнопок (OK, Cancel, Save, Clear и т.д.) в каких-либо диалоговых окнах?

- Возможно ли отменить текущую операцию (включая операцию удаления)?
- Все ли статические поля защищены от изменения пользователем? Может ли приложение изменить статический текст, корректно ли это делается?
- Применяются ли для статических текстовых полей согласованные шрифты и размеры?
- Соответствуют ли размеры полей редактирования диапазону принимаемых ими значений?
- Все ли поля редактирования инициализированы верными значениями при открытии окна?
- Проверяются ли значения, вводимые в поля редактирования, клиентской программой?
- Правильно ли заполняются из базы данных значения выпадающих списков?
- Используются ли маски редактирования в полях ввода в соответствии с определением?
- Доходчивы ли сообщения об ошибках и легко ли работать с ними?

9.1.2.3.2. Тестирование баз данных

Аналогично тестированию графического пользовательского интерфейса, тестирование баз данных связано со многими другими видами тестирования. Например, многие методы тестирования по спецификации основаны на использовании входной и выходной информации, хранящейся в базе данных. Тем не менее необходимы отдельные методы тестирования баз данных.

Послереализационное тестирование баз данных (post-implementation database testing) включает в себя всестороннее тестирование по коду. Наиболее существенной частью тестирования баз данных является тестирование транзакций. Тестирование некоторых других аспектов баз данных — например, производительности и безопасности/авторизации — иногда выделяют в отдельные тесты.

Аналогично тестам для проверки графического пользовательского интерфейса, некоторые тесты баз данных должны проводиться повторно для всех функций приложения. Вопросы, требующие рассмотрения при тестировании баз данных, необходимо оформить в виде общего документа. Затем этот общий документ следует присовокупить к описанию всех функциональных тестов (тестов системных услуг). Ниже приводится примерный перечень вопросов, на которые следует обратить внимание при тестировании баз данных (Bourne, 1997).

- Проверить, выполняется ли транзакция надлежащим образом при правильных входных данных? Корректна ли обратная связь системы с графическим пользовательским интерфейсом?
- Корректно ли содержимое базы данных после транзакции?
- Проверить, выполняется ли транзакция надлежащим образом при неправильных входных данных? Корректна ли обратная связь системы с графическим пользовательским интерфейсом?

- Корректно ли содержимое базы данных после транзакции?
- Прервать транзакцию до ее завершения. Корректна ли обратная связь системы с графическим пользовательским интерфейсом? Корректно ли содержимое базы данных после транзакции?
- Запустить одну и ту же транзакцию параллельно во многих процессах.
- Сознательно заставить одну из транзакций заблокировать ресурс, необходимый другим транзакциям. Получают ли пользователи понятное объяснение ситуации?
- Корректно ли содержимое базы данных после транзакций?
- Извлечь каждый из клиентских SQL-операторов из клиентской программы и выполнить их над базой данных в интерактивном режиме. Совпадают ли полученные результаты с ожидаемыми, а также соответствуют ли они результатам при выполнении SQL-операторов в составе программы?
- Выполнить интерактивное тестирование каждого более сложного SQL-запроса (выдаваемого из клиентской программы или хранимой процедуры) по методу "прозрачного ящика", включая внешние соединения, объединения, подзапросы, нулевые значения, функции агрегации и т.д.

9.1.2.3.3. Тестирование авторизации

Тестирование авторизации (authorization testing) можно рассматривать как естественное расширение тестирования первых двух типов системных ограничений. Как клиентские (пользовательский интерфейс), так и серверные (базы данных) объекты должны быть защищены от несанкционированного использования. Тестирование авторизации должно обеспечить проверку того, что механизмы безопасности, встроенные в клиентскую и серверную части системы, в действительности защищают систему от несанкционированного проникновения.

Несмотря на то что нарушение безопасности безусловно сказывается на базах данных, защита начинается с клиентской части системы. *Пользовательский интерфейс* программы должен быть в состоянии динамически настраивать собственную конфигурацию в соответствии с *уровнем полномочий* текущего пользователя (который *аутентифицируется* идентификатором и паролем). Пункты меню, командные кнопки и даже целые окна могут стать недоступны пользователям, если у них нет соответствующих прав.

Не все "проходы" в защите системы могут быть обращены в сторону пользователей. Поддержка авторизации является существенным компонентом любой СУБД. *Права доступа к серверу* (*привилегии*) разделяются на две категории.

- Доступ к отдельным *серверным объектам* — таблицам, представлениям, столбцам, хранимым процедурам и т.д.
- Выполнение SQL-операторов — `select`, `update`, `insert`, `delete` и т.д.

Полномочия для пользователей можно назначать непосредственно на *уровне пользователей* (user level) или *групп* (group level). Группы позволяют администратору безопасности предоставлять права доступа группе пользователей с помощью однократного ввода соответствующих параметров. Пользователь может как не принадлежать ни к одной группе, так и принадлежать к нескольким группам.

Для обеспечения большей гибкости при работе с авторизацией большинство СУБД вводят дополнительный уровень авторизации — *ролевой уровень* (role level). Роли позволяют администратору безопасности назначать права доступа всем пользователям, играющим определенную роль в организации. Роли могут быть вложенными, т.е. права, предоставленные разным ролевым именам, могут перекрываться.

Для больших прикладных информационных систем проектирование авторизации требует скрупулезной работы. Зачастую наряду с прикладной базой данных создается база данных авторизации для хранения и манипулирования клиентскими и серверными полномочиями. После входа пользователя в систему прикладная программа обращается к базе данных, чтобы установить уровень полномочий пользователя и настроить свою конфигурацию применительно к этому пользователю.

Любые изменения в правах доступа к базам данных осуществляются через базу данных авторизации — т.е. никто, включая администратора системы защиты, не имеет возможности непосредственно изменить права доступа к базе данных без предварительного обновления базы данных авторизации.

9.1.2.3.4. Тестирование других ограничений

Помимо описанных выше, тестирование системных ограничений включает следующие виды тестирования.

- Тестирование производительности.
- Стрессовое тестирование.
- Тестирование отказов.
- Тестирование конфигураций.
- Тестирование инсталляции.

Тестирование производительности направлено на измерение ограничений производительности, требуемых заказчиком. Ограничения связаны со скоростью транзакций и пропускной способностью. Тестирование проводится при различной рабочей загрузке систем, включая предполагаемую пиковую загрузку. Тестирование производительности является важной составляющей настройки системы.

Стрессовое тестирование проектируется так, чтобы вывести систему из строя при предъявлении к ней завышенных требований — из-за недостатка ресурсов, необычной конкуренции за ресурсы, непредусмотренной частоты, величины или объема требований к ресурсам. Стрессовое тестирование часто сочетается с тестированием производительности и может потребовать соответствующей аппаратной и программной оснастки.

Тестирование отказов направлено на изучение реакции системы на различные аппаратные, сетевые или программные сбои. Этот вид тестирования тесно связан с процедурами восстановления, поддерживаемыми СУБД.

Тестирование конфигураций связано с проверкой функционирования системы при различных аппаратных и программных конфигурациях. Для большинства производственных сред предполагается, что система способна функционировать на различных клиентских рабочих станциях, которые подключаются к базе данных с использованием различных сетевых протоколов. На клиентских рабочих станциях может быть инсталлировано различное программное обеспечение (например, драйверы), которое может конфликтовать с предусмотренными установками.

Инсталляционное тестирование является расширением конфигурационного тестирования. Оно связано с проверкой надлежащего функционирования системы на каждой из платформ, на которых она инсталлируется. Это означает фактическое повторение тестирования системных услуг.

Контрольные вопросы 9.1

КВ1. Какой метод поддержки качества известен также как тестирование, не связанное с выполнением программ?

КВ2. Какой метод поддержки качества предусматривает создание журнала дефектов?

КВ3. Назовите популярный каркас для создания приложений на языке Java.

КВ4. Как реализуются тестовые спецификации?

КВ5. Какой вид тестирования позволяет обнаружить пропущенные функциональные свойства?

9.2. Управление изменениями

Смысл термина *управление изменениями* (change management) так же многозначен, как и смысл самого понятия *изменение*. Изменения вездесущи и влияют на все аспекты работы предприятия. Они имеют как деловой, так и системный аспект.

Управление изменениями можно рассматривать с точки зрения *деловой ценности системы* (Laundon and Laundon, 2006). Это понятие применяется к финансовому анализу и моделям составления смет. Управление изменениями осуществляется в контексте календарных планов, смет и программ капиталовложений, включая проекты по созданию программного обеспечения. Деловой аспект управления изменениями помещает в центр внимания функциональные и организационные

последствия изменений, связанных с новой информационной системой. Он связан с вопросами управления персоналом — мотивацией, заинтересованностью сотрудников в изменениях, а также формах взаимодействия между отдельными сотрудниками и группами.

В более узком смысле управление **изменениями** (как указано далее) нацелено на обеспечение *соответствия системы целям компании* в условиях изменяющихся требований пользователей и выявленных недостатков. В этом смысле управление изменениями — это процесс управления программным обеспечением и процессами, а также управления работой, связанной с усовершенствованием программной системы. Как показано на рис. 9.6 (Maciaszek and Liong, 2005), управление изменениями тесно связано с управлением качеством.

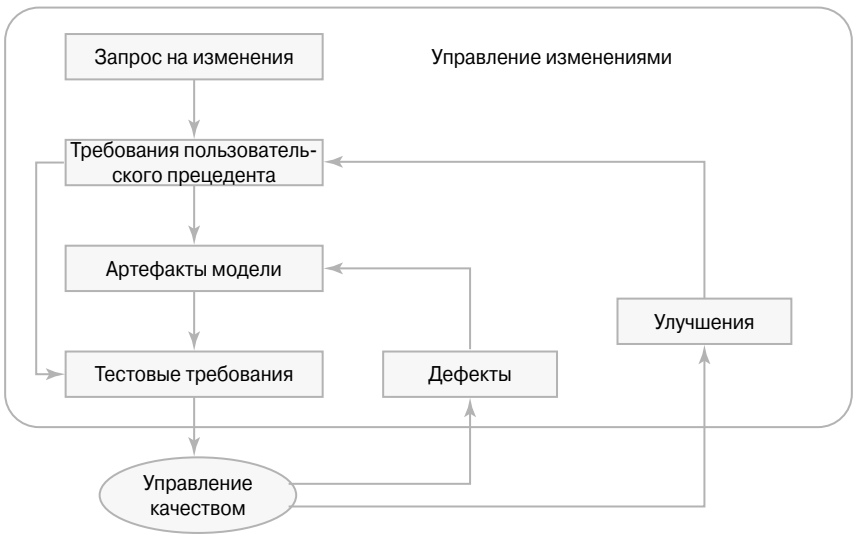


Рис. 9.6. Связь между изменениями и качеством. Источник: Maciaszek and Liong, 2005. (Воспроизводится с разрешения компании Pearson Education Ltd.)

Контроль качества выявляет *дефекты*, которые необходимо устранить. Для этого дефекты следует представить в виде *запроса на изменения* (change request) и передать его разработчикам. Некоторые запросы на изменения могут быть связаны с *усовершенствованиями*, а не дефектами. Как дефекты, так и усовершенствования подразумевают изменение системы, могут иметь приоритеты и владельцев, а также должны отслеживаться по тестовым спецификациям и прецедентам использования.

9.2.1. Инструменты управления запросами на изменения

Управление изменениями для любого программного проекта, в котором принимает участие большое количество разработчиков, представляет собой сложную и ответственную задачу. Рассмотрим сценарий, в рамках которого исправление

двух разных дефектов поручено двум разработчикам. При этом оказалось, что исправление этих на первый взгляд несвязанных дефектов требует внесения изменений в один и тот же программный компонент. Если разработчики не знают об этом возможном конфликте, то они могут одновременно работать над исправлением, и в конце концов более позднее исправление приведет к отмене исправления, сделанного раньше.

Для надлежащего управления изменениями необходимо применение *средств управления запросами на изменения* (рис. 9.7). Эти средства обеспечивают возможность интерактивного управления изменениями и гарантируют работу всех разработчиков с последними версиями документов. Изменения, внесенные в документ одним из участников проекта, тут же становятся достоянием остальных разработчиков. Потенциальные конфликты разрешаются с помощью механизмов блокировки или управления версиями. В первом случае заблокированный документ становится временно недоступен другим разработчикам. В последнем случае возможно создание нескольких версий одного документа, а конфликты между версиями разрешаются позднее путем согласования.

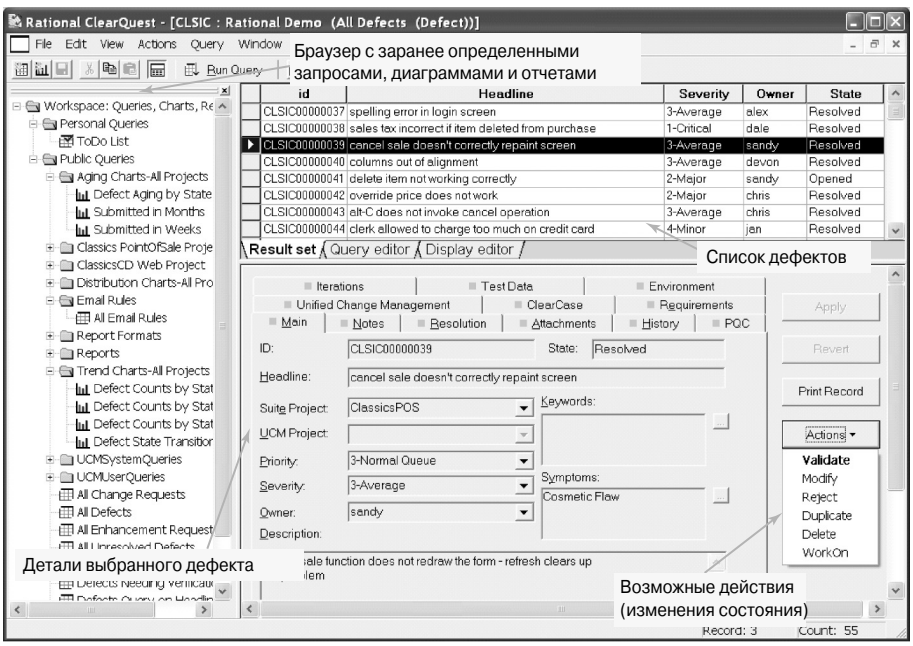


Рис. 9.7. Управление изменениями с помощью программы IBM Rational ClearQuest. Источник: Maciaszek and Liong, 2005. (Воспроизводится с разрешения компании Pearson Education Ltd.)

9.2.1.1. Отправка запроса на изменение

Обычно *запрос на изменение* связан либо с дефектом, либо с усовершенствованием. Запрос на изменение вводится в проектный репозитарий. После ввода в репозитарий разработчики могут отслеживать продвижение запроса на изменение, наблюдать за его *статусом* и действовать в соответствии с ним. *Действия*, выполняемые над запросом на изменение, зависят от текущего статуса запроса.

На рис. 9.8 показана основная вкладка диалогового окна ввода информации о дефектах (Rational, 2000). Дефекты нумеруются и подробно описываются. Информацию о приоритете, серьезности, проекте и ответственном исполнителе можно ввести с помощью подходящих значений выпадающего списка (значения атрибутов, содержащихся в выпадающем списке и в других местах формы, можно настроить в соответствии с потребностями проекта). Остальные поля позволяют вводить описательную информацию, включая возможность присоединения документации, имеющей отношение к дефекту, в частности фрагментов программного кода.

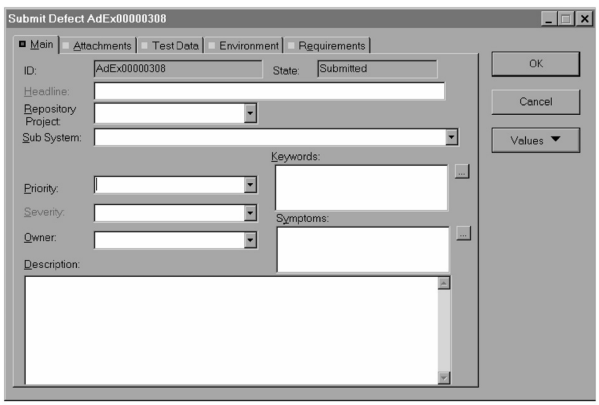


Рис. 9.8. Интерфейс ClearQuest для представления дефектов

Подача запроса на изменение может привести к автоматическому уведомлению участников проектной бригады по *электронной почте*. После этого запрос на изменение переходит в состояние `Submitted` (Подан). Руководство проекта может настроить инструментальные средства таким образом, чтобы в каждом состоянии выполнялись заранее определенные действия. Например, в состоянии `Submitted` могут быть допустимы следующие действия (рис. 9.9).

- `Assign` — назначить задание участнику группы.
- `Modify` — модифицировать некоторые детали запроса.
- `Close` — закрыть, возможно, после исправления.
- `Duplicate` — дублировать по другим идентификаторам.
- `Postpone` — отложить выполнение.

- `Delete` — удалить без исправления.
- `WorkOn` — продолжать работу.

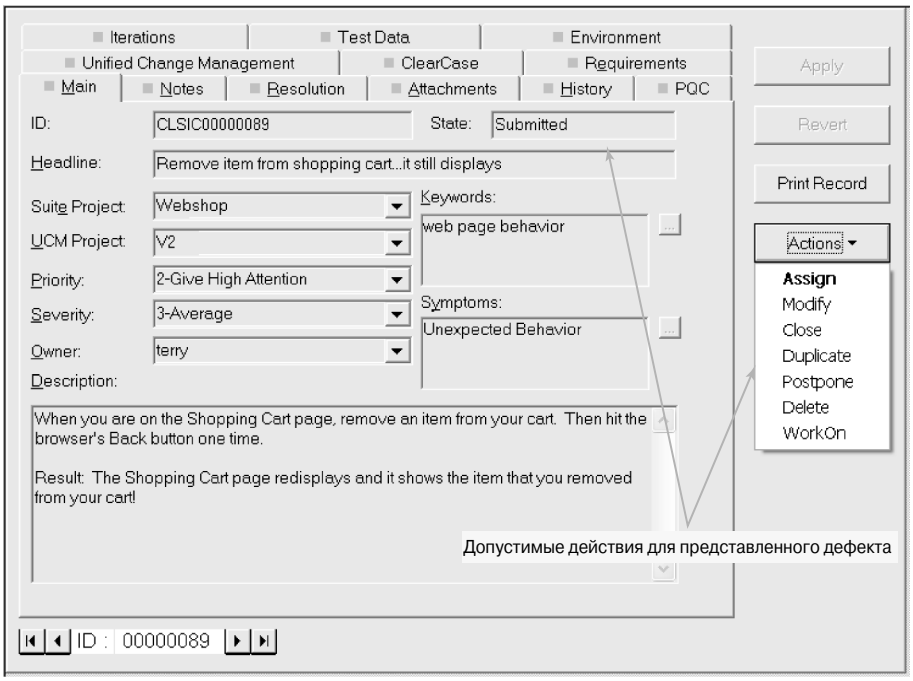


Рис. 9.9. Управление дефектами с помощью программы IBM Rational ClearQuest. Источник: Rational Suite Tutorial (Rational, 2002); Maciaszek and Liong, 2005. (Воспроизводится с разрешения компании Pearson Education Ltd.)

9.2.1.1. Отслеживание запросов на изменение

Каждый запрос на изменение назначается члену группы. Он может выполнить команду `Open` и открыть запрос на изменение. Если запрос находится в состоянии `Open`, никто из других членов группы не может модифицировать статус запроса.

После внесения изменений разработчик может выполнить действие `Resolve`. Подробности решения можно ввести в форму и отправить по электронной почте уведомление руководству проекта и специалистам по тестированию. Последним может потребоваться верифицировать выполненный запрос на изменение с помощью команды `Verify`.

На любом этапе средства управления изменениями помогают отслеживать запросы и создавать выразительные диаграммы и отчеты, демонстрирующие *характеристики проекта*. Диаграммы и отчеты могут содержать данные по оценке количества не распределенных дефектов, помогают выявить загрузку каждого члена группы, показать, сколько осталось неисправленных дефектов, и т.д.

На рис. 9.10 показана диаграмма распределения актуальных дефектов по приоритетам. Шесть дефектов должны быть разрешены немедленно, на 55 дефектов следует обратить более пристальное внимание, 67 дефектов находятся в обычной очереди и 68 имеют низкий приоритет.

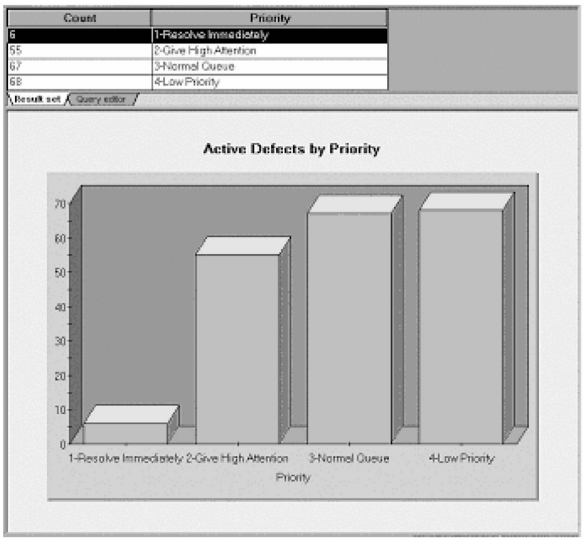


Рис. 9.10. Распределение дефектов по приоритетам

9.2.2. Трассируемость

В основе тестирования и управления изменениями лежит **трассируемость** (traceability). Его цель — зафиксировать, установить связь и сопоставить все важные артефакты разработки, включая требования заказчика. Конечной целью является создание полной, корректной и непротиворечивой документации — от технического задания до руководства пользователя.

Единицей трассирования могут быть текстовые описания или графические модели. Трассируемость означает, что между этими элементами существуют явные связи, прямые или косвенные. Эти связи позволяют анализировать последствия изменения одного из трассируемых элементов.

В предыдущих главах мы различали системные сервисы и системные ограничения. Трассируемость, а также управление качеством и изменениями часто ассоциируют с системными сервисами, выраженными в требованиях прецедентов использования. Однако не следует забывать, что усиление системных ограничений также необходимо тестировать и контролировать.

Трассируемость, а также управление качеством и изменениями — не самоцель, поэтому их не следует переоценивать. Разработчики должны сосредоточиться на разработке, а не на трассировании, тестировании или управлении изменениями.

Кроме того, поддержка этих качеств проекта требует значительных затрат. В то же время пренебрежение этими свойствами приводит к значительным долговременным расходам.

Поскольку трассируемость связана с управлением качеством и изменениями, для определения глубины и размаха трассировки необходимо провести *анализ затрат и результатов* (cost-benefit analysis). По меньшей мере необходимо поддерживать возможность сопоставления требований прецедентов использования и дефектов. В более сложных моделях в это число следует включить тестовые требования. В еще более изощренных моделях трассируемость может распространяться на системные функции, тестовые прецеденты, усовершенствования, точки тестовой верификации и другие артефакты разработки программного обеспечения.

В оставшейся части главы рассматривается модель трассируемости, соответствующая связям между системной документацией, показанной на рис. 9.3. *Системные характеристики* (system futures) перечисляются в описании прецедентов использования, *тестовые спецификации* — в плане тестирования, Связи между системными характеристиками и *требованиями прецедентов использования* фиксируются в описании прецедентов. *Тестовые требования*, описанные в соответствующем документе, можно связать с тестовыми прецедентами и требованиями прецедентов использования. Тестовые требования связываются с *дефектами*, а *усовершенствования* — с требованиями прецедентов использования. Связь между дефектами до *усовершенствованиями* не нужна.

9.2.2.2. Связи системных возможностей с тестовыми прецедентами и тестовыми требованиями

Функциональная возможность системы представляет собой общую часть функциональных свойств, подлежащих реализации, т.е. бизнес-процесс, представляющий собой существенную часть системы. Как правило, функциональная возможность системы соответствует определенному *бизнес-прецеденту использования* в модели бизнес-прецедентов (см. раздел 2.5.2 главы 2). Если модель бизнес-прецедентов формально не разработана, то функциональные возможности системы идентифицируются в *документе, описывающем концепцию системы* (который иногда называют стратегическим планом).

Каждая функциональная возможность системы реализуется с помощью набора *требований прецедентов использования*, выраженных *в одном или нескольких прецедентах использования*. Связывание прецедентов использования с потребностями участников проекта (выраженных в функциональных возможностях системы) помогает проверить обоснованность и правильность модели прецедентов. Эта стратегия "очерчивает рамки" фиксации требований и помогает завершить этап установления требований. Она также помогает осуществлять поступательную разработку и поставку программного продукта заказчикам.

Данная стратегия может стать источником проблем, если требования прецедентов использования в рамках каждого прецедента связаны с функциональными возможностями лишь косвенно. Это может привести к ситуации, в которой между функциональной возможностью и прецедентом использования существует связь, хотя бо́льшая часть требований прецедентов использования не имеет отношения к данной функциональной возможности. Принятие решения о том, обоснованно ли существование связи между функциональной возможностью и прецедентом, может оказаться неразрешимой задачей.

Для того чтобы избежать масштабирования и долговременных проблем, связанных с этой стратегией, *матрица связей* (traceability matrix) должна отслеживать функции не только по отношению к прецедентам, но также непосредственно по отношению к требованиям прецедентов. Это возможно в том случае, если сам прецедент рассматривается как обобщенное требование прецедента использования, под которым находится иерархия специализированных требований прецедентов.

Эта ситуация продемонстрирована на рис. 9.11. Столбцы содержат прецеденты использования и требования прецедентов использования. Иерархическое представление требований прецедентов использования можно свернуть или раскрыть. Стрелки обозначают связи функциональных возможностей с прецедентами использования и требованиями прецедентов использования. Некоторые стрелки перечеркнуты. Это — *подозрительные связи* (suspect traces). Связь становится подозрительной, если она направлена *от* изменений требований или *к* ним. Прежде чем разорвать подозрительные связи, разработчику следует проанализировать их.

9.2.2.2. Связи планов тестирования с тестовыми спецификациями и тестовыми требованиями

План тестирования (test plan) играет по отношению к тестовым спецификациям ту же роль, что и документ описания бизнес-прецедентов для прецедентов использования. План тестирования идентифицирует обобщенную проектную информацию и программные компоненты, которые требуется протестировать. План тестирования также описывает стратегию тестирования проекта, ресурсы, усилия и затраты, необходимые для тестирования.

Каждое тестовое задание, идентифицированное в плане тестирования, должно быть описано в виде отдельного документа. Установление связей между тестовыми требованиями, с одной стороны, и тестовыми спецификациями и планами тестирования, с другой, дает те же преимущества, что и установление соответствия между функциональными возможностями, прецедентами использования и требованиями прецедентов использования. Это позволяет установить область тестирования, его масштаб и т.д.

На рис. 9.12 показана матрица связей плана тестирования с тестовыми прецедентами и требованиями тестовых прецедентов. Иерархическое представление требований тестовых прецедентов можно свернуть или раскрыть.

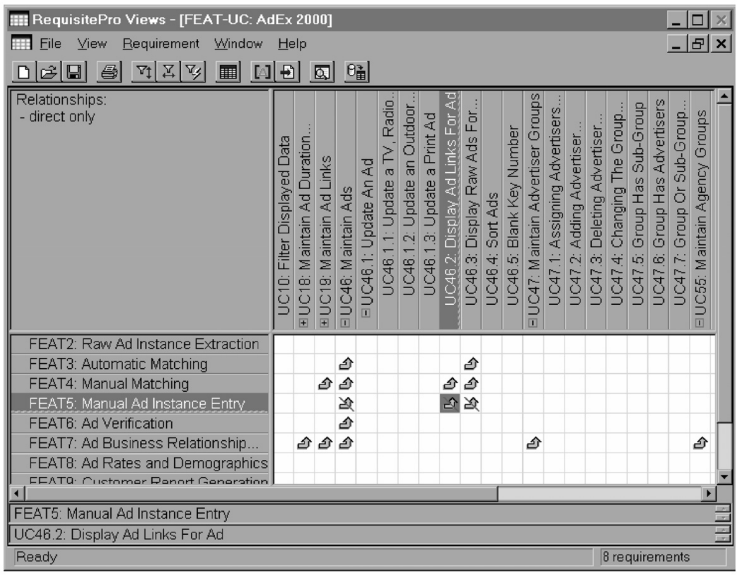


Рис. 9.11. Связывание функциональных свойств с прецедентами использования и требованиями прецедентов использования. Источник: компания Nielsen Media Research, Сидней, Австралия

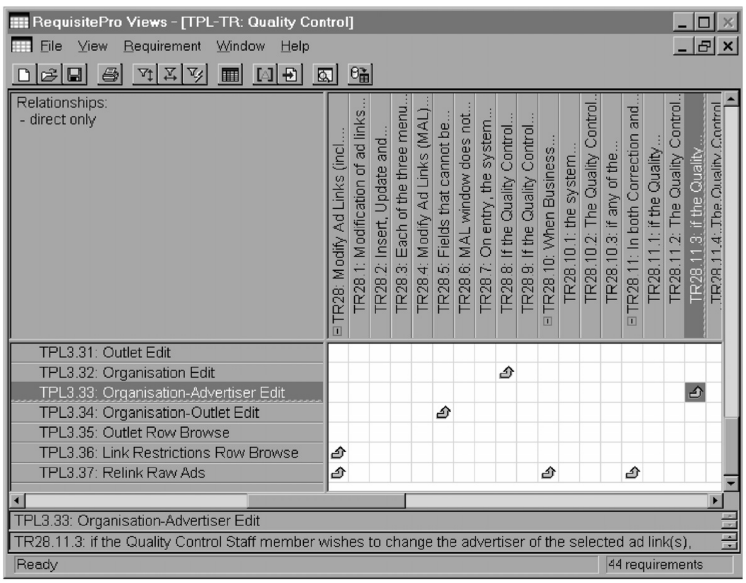


Рис. 9.12. Связывание плана тестирования с тестовыми спецификациями и тестовыми требованиями. Источник: компания Nielsen Media Research, Сидней, Австралия

9.2.2.2. Связи от UML-диаграмм с документами и требованиями

Сопоставление требований и управление ими применимо не только к описательным документам и требованиям, представленным в текстовом виде и хранящимся в CASE-репозитарии. В репозитарии хранятся также UML-модели. Графические объекты UML-диаграмм могут быть связаны гиперссылками с документами и требованиями.

Сопоставление визуальных артефактов UML с любыми другими записями репозитория (в частности, документами и требованиями) можно провести для разных *графических пиктограмм* UML. Пожалуй, наиболее важными из них являются пиктограммы прецедентов на диаграммах прецедентов.

На рис. 9.13 показано диалоговое окно для установления гиперссылки графической пиктограммы прецедента (`Maintain Ads`) на документ. Гиперссылка устанавливается в UML-диаграмме прецедентов использования. В качестве связанного документа может выступать любой документ репозитория, включая документы, показанные на рис. 9.13.

На рис. 9.14 показано диалоговое окно для установления гиперссылки графической пиктограммы (`Maintain Ads`) на требование прецедента использования. В общем случае возможно установить связь между пиктограммой и любым типом требования.

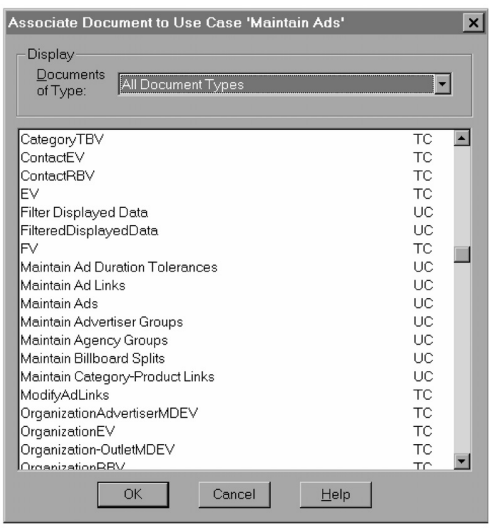


Рис. 9.13. Установление гиперссылки на графическую пиктограмму прецедента использования на документ. Источник: компания Nielsen Media Research, Сидней, Австралия

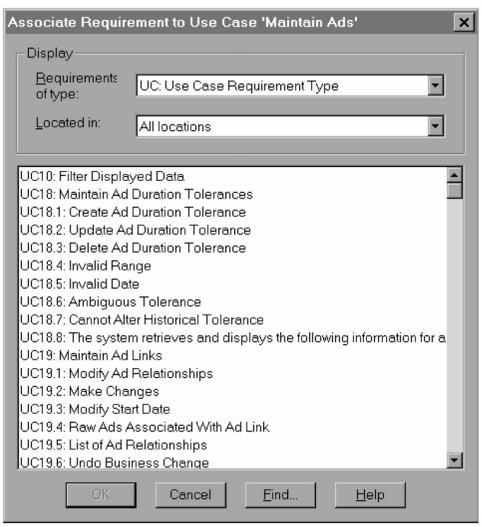


Рис. 9.14. Установление гиперссылки графической пиктограммы прецедента использования на требование. Источник: компания Nielsen Media Research, Сидней, Австралия

9.2.2.4. Связи между требованиями прецедентов использования и тестовыми требованиями

Связи между требованиями прецедентов использования и тестовыми требованиями являются критическим фактором при оценке того, удовлетворяет ли приложение бизнес-требованиям, установленным для него. Связи между этими двумя типами требований позволяют пользователю прослеживать связи между дефектами и тестовыми требованиями, а также требованиями прецедентов использования и функциональными возможностями системы (см. рис. 9.3).

На рис. 9.15 показана матрица связей между требованиями прецедентов использования и тестовыми требованиями. Обратите внимание на то, что требования прецедентов использования, как и тестовые требования, организованы иерархически. Иерархические уровни, на которых устанавливаются связи, могут быть определены заранее.

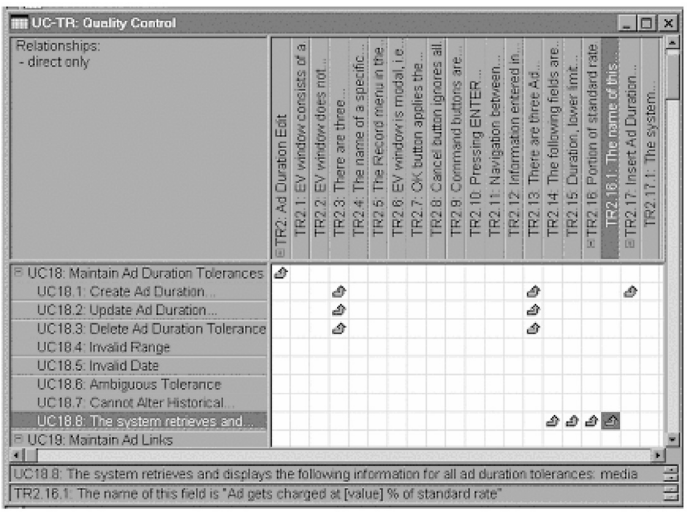


Рис. 9.15. Связи между требованием прецедента использования и тестовым требованием. Источник: компания Nielsen Media Research, Сидней, Австралия

9.2.2.5. Связи между тестовыми требованиями и дефектами

Тестовая спецификация представляется в виде сценариев, содержащих тестовые требования, которые подлежат верификации при тестировании. Сценарии используются *при ручном тестировании*, но многие из этих сценариев можно *автоматизировать* (закодировать) для использования в тестовых средствах "*записи-воспроизведения*". Тестовые требования в документации по тестовым прецедентам можно затем использовать для установления точек верификации в этих автоматизированных тестах.

Точка верификации (verification point) — это требование, содержащееся в сценарии, которое используется (при регрессионном тестировании) для подтверждения состояния тестового объекта в рамках различных версий (скомпонованных версий) тестируемого приложения. Существуют различные типы точек верификации (Rational, 2000). Точка верификации может быть установлена для проверки отсутствия изменений в тексте, точности числового значения, идентичности двух файлов, отсутствия изменений в пунктах меню, соответствия полученных результатов вычислений ожидаемым и т.д.

Автоматизированное тестирование требует работы с двумя файлами данных — эталонным файлом данных и файлом фактических данных. Во время *записи* при проверке точки верификации в *эталонный файл данных* записывается информация об объекте. Эта информация представляет собой основу для сравнения при последующих тестах (воспроизведении). Результаты этого сравнения запоминаются в *файле фактических данных*. Каждая *точка верификации, не прошедшая проверку*, должна быть изучена снова и, при необходимости, внесена в репозитарий средств управления изменениями в качестве *дефекта*.

В конечном счете все дефекты, обнаруженные либо с помощью средств автоматизации, либо вручную, должны быть сопоставлены с тестовыми требованиями. На рис. 9.16 показано средство, которое отображает все дефекты в окне для просмотра строк в верхней части экрана. Текущий выбранный дефект можно сопоставить с одним или более тестовым требованием. В приведенном примере показана связь между двумя тестовыми требованиями к выделенным дефектам.

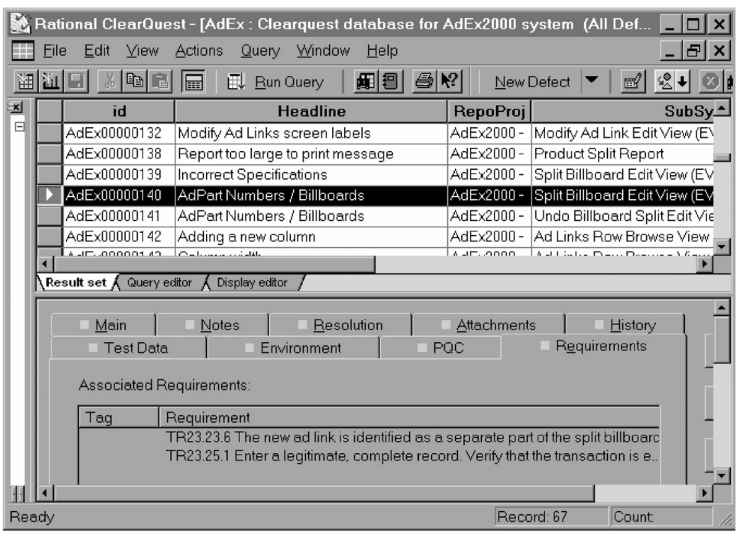


Рис. 9.16. Связи между тестовым требованием и дефектами. Источник: компания Nielsen Media Research, Сидней, Австралия

9.2.2.6. Связи между требованиями прецедентов использования и усовершенствованиями

Дефекты необходимо связывать непосредственно с тестовыми требованиями. Усовершенствования (которые должны быть реализованы в последующих выпусках продукта) должны содержать соответствующие пояснения о требованиях прецедентов использования. Изредка, когда дефект был устранен путем усовершенствования, связь между требованиями прецедентов использования и тестовыми требованиями может позволить пользователю проследить путь дефекта до усовершенствования.

На рис. 9.17 продемонстрирован тот же самый инструмент (см. рис. 9.16), позволяющий управлять усовершенствованиями и дефектами, как, впрочем, и любыми другими запросами на изменения.

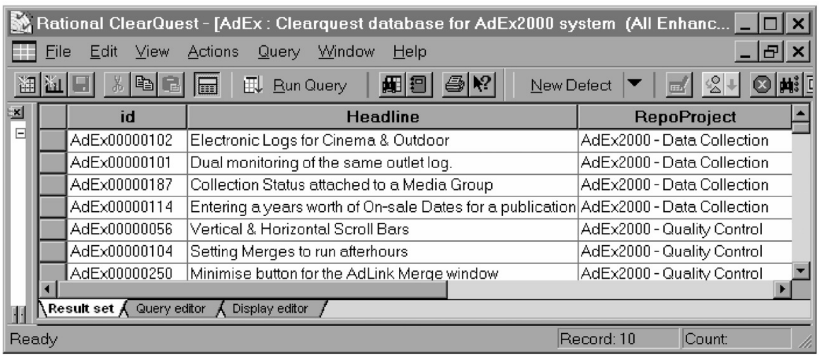


Рис. 9.17. Усовершенствования. Источник: компания Nielsen Media Research, Сидней, Австралия

Контрольные вопросы 9.2

КВ1. Что необходимо сделать, чтобы внести изменения?

КВ2. Какой метод используется для определения масштаба и глубины трассируемости проекта?

КВ3. Какой инструмент позволяет автоматизировать тестовые сценарии?

Резюме

В этой главе рассмотрены вопрос тестирования и управления изменениями. Эти виды деятельности охватывают весь жизненный цикл разработки программного обеспечения. Они требуют специальной *документации*, например планов тестирования, тестовых спецификаций, а также журналов регистрации дефектов и усовершенствований. Тестовые требования идентифицируются в *тестовых спецификациях* и связываются с требованиями прецедентов использования.

Управление качеством подразделяется на два независимых направления. Оно может носить реактивный характер (*post factum*), когда предусматривается использование механизма *контроля качества*. Однако оно может принимать форму превентивной *поддержки качества*, если используется в рамках *разработки посредством тестирования*.

Контроль качества связан с тестированием системных функций и системных ограничений. *Тестирование системных функций* может быть основано на выполнении программы или проводиться без выполнения программы. *Тестирование без выполнения программы* включает в себя сквозной контроль и инспекции. *Тестирование с выполнением программы* может быть *тестированием по спецификациям* или *тестированием по коду*.

Тестирование системных ограничений включает в себя широкий набор относительно самостоятельных тестов, которые относятся к таким вопросам, как пользовательский интерфейс, базы данных, авторизация, производительность, стрессовый режим, поведение при отказах, конфигурация и инсталляция приложения. Некоторые из тестов системных ограничений проводятся параллельно с тестированием системных услуг, другие выполняются независимо.

Запрос на изменение обычно возникает в связи с *дефектом* или *усовершенствованием*. Средства управления изменениями позволяют отправлять запросы на изменения и отслеживать их продвижение после обращения к ним разработчика. Очень важная часть средств управления изменениями связана с установлением *связей* между запросами на изменения и другими системными артефактами, в частности тестовыми требованиями и требованиями прецедентов использования.

Ключевые термины

Аудит (audit). Формальный процесс поддержки качества, возможно. выполняемый компетентными внешними аудиторами.

Дефект (defect). Изменение, требующее исправления.

Изменение (change). Любое ожидаемое или неожиданное событие, проявляющееся в модификации системных требований и/или требующее внимания при реализации кода и моделей проекта.

Инспекция (inspection). Более формальное совещание по вопросам проверки качества под контролем руководства.

Контроль качества (quality control). Процесс верификации качества разработанных артефактов, продуктов и процессов программирования.

Контрольный список (checklist). Метод поддержки качества, основанный на использовании заранее определенного списка пунктов, которые необходимо тщательно проверить в ходе разработки.

План тестирования (test plan). Документ, использующийся при управлении качеством, в котором идентифицируются тестовые спецификации и определяется календарный план, смета и ресурсы, необходимые для тестирования.

Поддержка качества (quality assurance). Процесс превентивного обеспечения качества программных продуктов и процессов.

Проверка (review). Метод поддержки качества с помощью формального протоколируемого совещания разработчиков и, возможно, менеджеров с целью проверки продукции или процессов.

Разработка посредством тестирования (test-driven development). Метод ускоренной разработки программного обеспечения, предусматривающий создание тестовых спецификаций и тестовых программ до создания кода приложения.

Регрессионное тестирование (regression testing). Повторение ранее проведенных приемочных испытаний для проверки старых или новых дефектов, не проявлявшихся в ходе последовательных итераций.

Сквозной контроль (walkthrough). Неформальное совещание, посвященное проверке продукции и процессов и не контролируемое менеджерами проекта.

Тестирование по коду (testing to code). Вид тестирования, основанный на выполнении программы и тщательном анализе ее логики.

Тестирование по спецификациям (testing to specs). Вид тестирования, основанный на выполнении программы, в ходе которого она интерпретируется как "черный ящик", просто принимающий значения и выводящий некие результаты.

Тестовая спецификация (test case). Документ, в котором фиксируются тестовые требования.

Тестовое требование (test requirement). Функциональное требование или нефункциональное ограничение, предназначенное для тестирования.

Тестовый набор (test suite). Набор тестовых сценариев.

Тестовый сценарий (test script). Сценарий, предусматривающий ручное или автоматическое выполнение, в котором определена последовательность этапов тестирования и точки верификации.

Трассируемость (traceability). Процесс, охватывающий весь жизненный цикл проекта, в ходе которого связываются и отслеживаются все важные артефакты системы, включая ее требования.

Усовершенствование (enhancement). Изменение, планируемое в будущем.

Многовариантные тесты

MT1. Какой из перечисленных ниже методов относится к поддержке качества?
- **а.** Сквозной контроль.
- **б.** Контрольный список.
- **в.** Ретроспекция.
- **г.** Инспекция.

MT2. Как называются вопросы, на которые должен ответить испытатель?
- **а.** Точки верификации.
- **б.** Запросы сценария.
- **в.** Тестовые запросы.
- **г.** Все ответы неправильные.

MT3. Как называется повторение приемочных испытаний последовательных версий кода?
- **а.** Тестирование по методу "прозрачного ящика".
- **б.** Регрессионное тестирование.
- **в.** Тестирование по методу "черного ящика".
- **г.** Тестирование покрытия.

MT4. Как называется тестирование по методу "черного ящика"?
- **а.** Тестирование по спецификациям.
- **б.** Функциональное тестирование.
- **в.** Тестирование на основе ввода-вывода.
- **г.** Все ответы неправильные.

MT5. Как называется тестирование по методу "прозрачного ящика"?
- **а.** Функциональное тестирование.
- **б.** Тестирование по спецификациям.
- **в.** Тестирование ветвей программы.
- **г.** Тестирование покрытия.

Вопросы

В1. Управление качеством и изменениями охватывает весь жизненный цикл проекта. Какие еще виды деятельности относятся ко всему жизненному циклу проекта? Объясните цель этих видов деятельности.

В2. Вернитесь к рис. 9.3 (см. раздел 9.1.2.1). Объясните взаимосвязь между усовершенствованиями и дефектами. Почему спецификации усовершен-

ствования связываются со спецификациями прецедентов использования, а спецификации дефектов — с тестовыми спецификациями? Не следует ли установить связь между спецификациями усовершенствований и спецификациями дефектов?

B3. Вернитесь к рис. 9.5 (см. раздел 9.1.2.1). Почему виртуальным тестером должна быть специальная рабочая станция?

B4. Чем сквозной контроль отличается от инспекции?

B5. В чем заключается роль группы поддержки качества (группы SQA) в организации?

B6. Что такое база данных авторизации? Какова ее роль в разработке и тестировании системы?

B7. Какие еще виды тестирования системных ограничений тесно связаны со стрессовым тестированием? Обоснуйте свой ответ.

B8. Какие еще виды тестирования системных ограничений тесно связаны с тестированием инсталляции? Обоснуйте свой ответ.

B9. Какие действия возможны в ответ на отправленный запрос на изменения?

B10. Посетите Web-сайт, посвященный библиотеке JUnit (`www.junit.org`). Опишите последние новшества, внесенные в библиотеку JUnit.

B11. Что такое точка верификации?

B12. Что такое "подозрительная связь"? Приведите пример.

B13. Объясните различие между эталонным файлом данных и фактическими данными.

B14. Укажите место и роль трассируемости в управлении проектом? На какие вопросы отвечает это понятие?

Ответы на контрольные вопросы

Контрольные вопросы 9.1

KB1. Проверка (сквозной контроль и инспекция).

KB2. Инспекция.

KB3. Библиотека JUnit.

KB4. Посредством тестовых сценариев.

KB5. Тестирование по спецификациям.

Контрольные вопросы 9.2

КВ1. Подать формальный запрос на изменения.

КВ2. Анализ затрат и результатов.

КВ3. Инструменты записи-воспроизведения.

КВ4. Посредством тестовых сценариев.

Ответы к многовариантным тестам

МТ1. в

МТ2. а

МТ3. б

МТ4. г

МТ5. в

Ответы на вопросы с нечетными номерами

В1

К важным видам деятельности, охватывающим весь жизненный цикл проекта, кроме тестирования и управления изменениями, относятся следующие (Maciaszek and Liong, 2005).

- Планирование проекта (см. раздел 1.4.3.1 главы 1).
- Измерения (см. раздел 1.4.3.2 главы 1).
- Управление конфигурацией.
- Управление персоналом.
- Управление риском.

Планирование проекта (и слежение) — это деятельность, охватывающая весь жизненный цикл проекта, нацеленная на оценку (и верификацию) потребления временны*х*, денежных, трудовых и других ресурсов в ходе выполнения проекта. В более широком смысле планирование проекта включает в себя также поддержку качества, управление персоналом, анализ рисков и управление конфигурацией.

Для того чтобы спланировать выполнение проекта в будущем, необходимо измерять его показатели в прошлом. Сбор *показателей* — это деятельность, связанная с измерениями характеристик программной продукции и процессов. Измерения — это сложная и запутанная область, поскольку не любое свойство

программного обеспечения допускает количественную оценку. Несмотря на это, лучше иметь приблизительную количественную оценку, чем не иметь никакой. Существует множество свидетельств того, что собирание показателей является необходимым условием успеха при разработке системы.

Управление конфигурацией тесно связано с управлением изменениями. Управление конфигурацией добавляет "коллективную" компоненту в управление изменениями. Цель управления конфигурацией — хранение версий программных артефактов, созданных группой разработчиков, обеспечение доступа к этим версиям для всех участников группы, а также объединение версий в конфигурируемые модели и продукты.

Информационная система имеет социальный характер. Невозможно создать успешную программу, не уделив должного внимания человеческому фактору. *Управление персоналом* включает в себя подбор кадров, создание группы и ее мотивацию, установление эффективных отношений между людьми, решение конфликтов и обеспечение внешней обратной связи.

Риски — это "потенциально опасные обстоятельства, вредящие процессу разработки и качеству продукции" (Ghezzi et al., 2003). "Управление риском — это деятельность, связанная с принятием решений и оценкой рисков, связанных с этими решениями. Она позволяет оценить возможные результаты проекта и вероятности их достижения" (Maciaszek and Liong, 2005). Очевидно, что управление риском охватывает весь жизненный цикл проекта — от начала до завершения. Оно может привести к досрочному прекращению проекта, если риск становится слишком высоким и неотвратимым.

В3

Виртуальный испытатель — это вычислительная среда записи-воспроизведения, предназначенная для регрессионного тестирования. Она воспроизводит предварительно записанные тестовые сценарии и следит за поведением кода приложения, т.е. генерирует ли код те же самые события и результаты, что и записанные в тестовом сценарии. Следовательно, важно, чтобы вся среда тестирования была неизменной и устойчивой.

Только рабочая станция, специально предназначенная для тестирования и не используемая для других целей, может гарантировать неизменность или устойчивость среды тестирования. Важно, чтобы рабочая станция использовала ту же самую версию операционной системы при выполнении всех тестов на данной машине и требовала минимального дополнительного программного обеспечения.

В частности, рабочая станция не должна быть соединена с Интернетом, серверами электронной почты и т.д. Это объясняется тем, что события Интернета могут перехватываться тестируемой программой и интерпретироваться как неожиданные события, сгенерированные прикладной программой.

B5

Группа SQA (поддержки качества) отвечает за планомерную и систематическую оценку качества программных продуктов и процессов. Эта группа проверяет продукцию и процессы на соответствие существующим стандартам на протяжении всего жизненного цикла проекта. Для того чтобы гарантировать, что цель группы SQA будет достигнута, ответственность за качество поставляемой продукции возлагается именно на нее, а не на разработчиков.

Качество программного обеспечения оценивается с помощью мониторинга процессов, формальных проверок, аудита и тестирования. Оценка проводится в *точках проверки качества* (approval points), установленных для разработки программного обеспечения и процессов контроля. Результаты работы группы SQA излагаются в *аудиторском отчете*, чтобы руководство проекта, ознакомившись с итогами тестирования и рекомендациями, привела процессы разработки в соответствие со стандартами и нормативами.

B7

Стрессовое тестирование приводит систему в необычный режим, для которого характерны минимальные ресурсы, предельные нагрузки, нестандартные частоты или показатели. Стрессовое тестирование тесно связано с *тестированием производительности*, поскольку падение производительности часто возникает в условиях, когда система находится в условиях напряженной работы.

На практике стрессовое тестирование и тестирование производительности можно выполнять параллельно, используя схожие тестовые сценарии, аппаратное обеспечение и программный инструментарий.

B9

Действия, допустимые для открытого запроса на усовершенствование, зависят от принятой практики. Список разрешенных действий обычно настраивается с помощью CASE-средства, выбранного для разработки системы. Возможный список может выглядеть так, как показано ниже.

- `Close`. Закрыть в результате решения проблемы или по указанию менеджера.
- `Modify`. Установить некоторые детали усовершенствования.
- `Delete`. Удалить запрос без последствий; возможна запись об ошибке.
- `WorkOn`. Продолжить работу.
- `Postpone`. Отложить на будущее.

B11

Точка верификации — это пункт в сценарии регрессионного тестирования, показывающий, выполняется ли тестовое требование или нет. Она используется для верификации состояния тестируемого объекта в разных версиях программы. Точки верификации могут устанавливаться для достижения следующих целей.

- Тестирование числовых и других значений, принадлежности числа указанному диапазону, заполнения поля и т.д.
- Сравнение содержания двух файлов или других наборов данных.
- Проверка существования файла, таблицы базы данных и т.д.
- Фиксация и сравнение состояний пунктов меню графического пользовательского интерфейса, средств управления, окон и т.д.
- Проверка существования окон графического пользовательского интерфейса или специального программного модуля в памяти компьютера.
- Фиксация и сравнение Web-сайтов.

В ходе *фиксации* (записи) программы в точках верификации происходит перехват информации о тестируемом объекте и ее сохранение в качестве образца ожидаемого поведения. В ходе *воспроизведения* в точках верификации происходит сравнение текущей информации о тестируемом объекте с образцом. *Регрессионный тест* проходит успешно, если сценарий воспроизведения подтверждает, что программа выполняется правильно, а точки верификации возвращают правильные данные.

B13

Большинство точек верификации создают *эталонный файл данных*, если в ходе воспроизведения зафиксированные данные отличаются от эталонных, точка верификации считается неисправной и создается *файл фактических данных*.

Если сценарий воспроизводится много раз и много раз завершается неудачно, каждый раз создается отдельный файл данных (как правило, с тем же именем, но с последовательно возрастающими номерами). Имя фактического файла обычно совпадает с именем файла, соответствующего точке верификации.

ГЛАВА 10

Систематизация и закрепление учебного материала

Цели

10.1. Моделирование прецедентов использования
10.2. Моделирование деятельности
10.3. Моделирование классов
10.4. Моделирование взаимодействия
10.5. Моделирование конечных автоматов
10.6. Модели реализации
10.7. Разработка кооперации объектов
10.8. Проектирование навигации по окнам
10.9. Проектирование баз данных
Резюме
Упражнения. Интернет-магазин

req_analysis_system design.indb 699 05.06.2008 14:43:38

Цели

Данная глава посвящена разбору учебного примера. Она иллюстрирует все важные модели и процессы жизненного цикла разработки программного обеспечения. Изложение опирается на конкретное приложение — Интернет-магазин — и является полным и исчерпывающим. Последовательность тем, вопросов и решений, рассмотренных в главе, следует принципам, принятым в книге. Действительно, можно (и даже нужно) использовать эту главу для закрепления знаний при переходе от одной главы к другой.

Кроме учебных целей, эта глава имеет дополнительную ценность. Она демонстрирует все важные артефакты разработки программного обеспечения (модели, диаграммы, документы и т.д.) и показывает, как они работают в сочетании друг с другом. Поскольку учебный пример посвящен Web-приложению, он описывает самые современные достижения технологии проектирования.

Прочитав эту главу, читатели будут

- знать, что результаты моделирования связаны друг с другом и частично перекрываются;
- лучше понимать изменения, происходящие на разных уровнях абстракции при переходе от анализа к проектированию и реализации;
- знать о типичных задачах анализа и проектирования, возникающих в области коммерческих Web-приложений;
- владеть систематизированными знаниями о диаграммах UML и способах их взаимодействия.

10.1. Моделирование прецедентов использования

Постановка задачи: Интернет-магазин (обработка заказов)

Производитель компьютеров предлагает клиентам покупать свою продукцию через Интернет. Пользователи могут выбирать серверы, настольные системы и ноутбуки на Web-странице производителя либо со стандартной конфигурацией, либо описав желательную конфигурацию в интерактивном режиме. Компоненты конфигурации (например, микросхемы памяти) представляются в виде выпадающих списков. Для каждой новой конфигурации система может вычислить ее стоимость.

Для того чтобы разместить заказ, клиенты должны ввести информацию о покупке и форме оплаты. Поставщик принимает оплату с помощью кредитных карточек или чеков. После размещения заказа система посылает клиентам по электронной почте подробное описание заказа. Ожидая прибытия компьютеров, заказчики могут проверять статус своих заказов в оперативном режиме.

Обрабатывая заказ, система последовательно проверяет кредитоспособность клиента и надежность способов оплаты, запрашивает заказанную конфигурацию на складе, распечатывает счет-фактуру и отдает складу распоряжение поставить компьютеры заказчику.

10.1.1. Действующие лица

Этап 1. Интернет-магазин

Обратитесь к приведенной выше постановке задачи и, рассмотрев следующие расширенные требования, определите действующих лиц в приложении.

- Для знакомства со стандартной конфигурацией выбираемого сервера, настольного компьютера или ноутбука клиент использует Web-страницу Интернет-магазина. При этом также приводится цена конфигурации.
- Клиент выбирает детали конфигурации, с которыми он хочет ознакомиться, возможно, с намерением купить готовую или составить более подходящую конфигурацию. Цена для каждой конфигурации может быть подсчитана по требованию пользователя.
- Клиент может выбрать вариант заказа компьютера через Интернет либо попросить, чтобы продавец связался с ним для объяснения деталей заказа, договорился о цене и т.п. прежде, чем заказ будет фактически размещен.
- Для размещения заказа клиент должен заполнить электронную форму, указав адрес для доставки товара и отправки счета-фактуры, а также детали оплаты (кредитная карточка или чек).
- После ввода заказа клиента в систему продавец отправляет на склад по электронной почте требование, содержащее детали заказанной конфигурации.
- Детали сделки, включая номер заказа и номер счета клиента, отправляются по электронной почте клиенту, так что заказчик может проверить состояние заказа через Интернет.
- Склад получает счет-фактуру от продавца и отгружает компьютер клиенту.

На рис. 10.1 показаны три действующих лица, явно указанных в спецификации: Клиент, Продавец и Склад.



Рис. 10.1. Действующие лица в системе Интернет-магазина

10.1.2. Прецеденты использования

> **Этап 2. Интернет-магазин**
>
> Обратитесь к этапу 1 (см. раздел 10.1.1) и выберите прецеденты для системы управления Интернет-магазином.

Для выполнения этой задачи можно создать таблицу распределения функциональных требований по субъектам и прецедентам. Обратите внимание на то, что некоторые потенциальные бизнес-функции могут выходить за рамки приложения — они не подлежат преобразованию в прецеденты.

В табл. 10.1 приведены функциональные требования к системе Интернет-магазина, распределенные по действующим лицам и прецедентам использования. Сборка конфигурации компьютера и отправка его клиенту относятся к функциям, которые в данном случае выходят за рамки приложения.

Таблица 10.1. Распределение требований по действующим лицам и прецедентам использования системы управления Интернет-магазином

Номер требования	Требование	Действующее лицо	Прецедент
1	Для знакомства со стандартной конфигурацией выбираемого сервера, настольного или портативного компьютера клиент использует Web-страницу Интернет-магазина. При этом также приводится цена конфигурации	Клиент	Вывод стандартной конфигурации

Окончание табл. 10.1

Номер требования	Требование	Действующее лицо	Прецедент
2	Клиент выбирает детали конфигурации, с которыми он хочет ознакомиться, возможно, с намерением купить готовую или составить более подходящую конфигурацию. Цена для каждой конфигурации может быть подсчитана по требованию пользователя	Клиент	Формирование конфигурации
3	Клиент может выбрать вариант заказа компьютера по Интернету либо попросить, чтобы продавец связался с ним для объяснения деталей заказа, договорился о цене и т.п. прежде, чем заказ будет фактически размещен	Клиент Продавец	Заказ конфигурации Установление контакта с продавцом
4	Для размещения заказа клиент должен заполнить электронную форму с адресами для доставки товара и отправки счета-фактуры, а также деталями, касающимися оплаты (кредитная карточка или чек)	Клиент	Заказ конфигурации Проверка и обработка платежа
5	После ввода заказа клиента в систему продавец отправляет на склад электронное требование, содержащее детали, касающиеся заказанной конфигурации	Продавец Склад	Пересылка заказа на склад
6	Детали сделки, включая номер заказа и номер счета клиента, отправляются по электронной почте клиенту, так что заказчик может проверить состояние заказа через Интернет	Продавец Клиент	Заказ конфигурации Вывод статуса заказа
7	Склад получает счет-фактуру от продавца и отправляет компьютер клиенту	Продавец Склад	Печать счета-фактуры

Прецеденты использования системы Интернет-магазина показаны на рис. 10.2.



Рис. 10.2. Прецеденты использования системы Интернет-магазина

10.1.3. Диаграммы прецедентов использования

Этап 3. Интернет-магазин

Обратитесь к предыдущим этапам и постройте диаграмму прецедентов для системы управления Интернет-магазином.

Решение этой задачи можно получить, воспользовавшись информацией, полученной на предыдущих этапах. Единственное, что при этом необходимо дополнительно учесть, — это отношения между прецедентами. Диаграмма прецедентов представлена на рис. 10.1.3. Смысл отношения «`extend`» (расширяет) состоит в том, что прецедент `Заказ конфигурации` может быть расширен действующим лицом `Клиент` с помощью прецедента `Установление контакта с продавцом`.

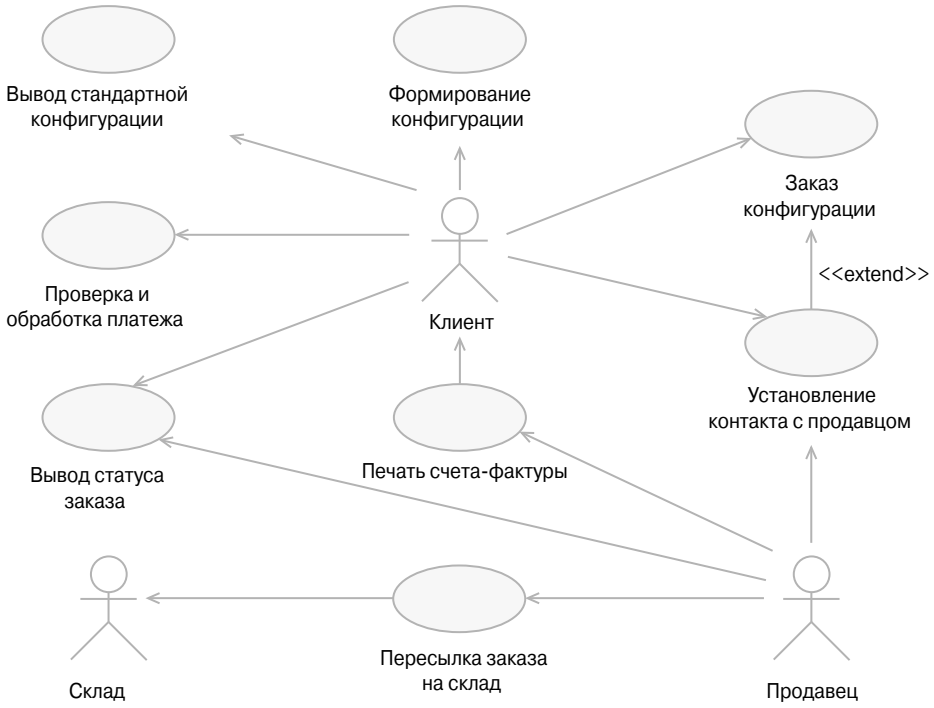


Рис. 10.3. Диаграмма прецедентов использования системы Интернет-магазина

10.1.4. Документирование диаграмм прецедентов использования

> **Этап 4. Интернет-магазин**
>
> Обратитесь к предыдущим этапам и подготовьте спецификацию прецедента `Заказ конфигурации`. Для того чтобы установить детали, которые явно не сформулированы в требованиях, используйте общие знания или информацию о типичной задаче обработки заказов.

Таблица 10.2. Распределение требований по действующим лицам и прецедентам использования системы управления Интернет-магазином

Прецедент использования	Заказ конфигурации
Краткое описание	Прецедент дает возможность `Клиенту` ввести заказ на покупку. Заказ включает адреса доставки товара и оплаты счета, а также детали условий оплаты
Действующее лицо	`Клиент`

Окончание табл. 10.2

Прецедент использования	Заказ конфигурации
Предусловия	Клиент с помощью Интернет-браузера выбирает страницу производителя компьютеров для ввода заказа. На Web-странице отображается подробная информация о сконфигурированном компьютере вместе с его ценой
Основной поток	Прецедент начинается с того, что Клиент заказывает конфигурацию компьютера с помощью выбора функции Continue (Продолжить). В это время на экране компьютера отображается детализированная информация о заказе. Система просит Клиента ввести детали покупки, в том числе имя продавца (если оно известно); детали, касающиеся доставки (имя и адрес клиента); детальную информацию об оплате (если она отличается от информации о доставке); способ оплаты (кредитная карточка или чек) и произвольные комментарии. Клиент выбирает функцию Purchase (Покупка) и отправляет заказ производителю. Система присваивает уникальный номер заказа и клиентский учетный номер заказу на покупку и заносит информацию о заказе в базу данных. Система отправляет Клиенту по электронной почте номер заказа и клиентский номер вместе со всеми деталями, относящимися к заказу, в качестве подтверждения принятия заказа
Альтернативный поток	Клиент инициирует функцию Purchase до того, как введет всю обязательную информацию. Система отображает на экране сообщение об ошибке и просит ввести пропущенную информацию. Клиент выбирает функцию Reset (Сброс), для того чтобы вернуться к исходной форме заказа на покупку. Система дает возможность клиенту вновь ввести информацию
Постусловия	Если прецедент был успешным, заказ на покупку записывается в базу данных. В противном случае состояние системы остается неизменным

10.2. Моделирование деятельности

10.2.1. Действия

Этап 5. Интернет-магазин

В табл. 10.3 изложены события, относящиеся к основному и альтернативному потокам, заимствованным из документа описания прецедента, а также обозначены состояния видов деятельности. Обратите внимание на то, что каждому виду деятельности присвоено имя, исходя из системной точки зрения, а не с точки зрения субъекта.

Таблица 10.3. Идентификация действий в основном и альтернативном потоках прецедентов использования системы управления Интернет-магазином

Номер	Прецедент использования	Действие
1	Прецедент начинается с того, что `Клиент` заказывает конфигурацию компьютера с помощью выбора функции `Continue` (Продолжить). В это время на экране компьютера отображается детализированная информация о заказе	`Вывести текущую конфигурацию` `Получить бланк заказа`
2	Система просит `Клиента` ввести детали покупки, в том числе имя продавца (если оно известно); детали, касающиеся доставки (имя и адрес клиента); детальную информацию об оплате (если она отличается от информации о доставке); способ оплаты (кредитная карточка или чек) и произвольные комментарии	`Вывести на экран бланк заказа`
3	`Клиент` выбирает функцию `Purchase` (Покупка) и отправляет заказ производителю	`Детализировать информацию о покупке`
4	Система присваивает уникальный номер заказа и клиентский учетный номер заказу на покупку и заносит информацию о заказе в базу данных	`Запомнить заказ`
5	Система отправляет `Клиенту` по электронной почте номер заказа и клиентский номер вместе со всеми деталями, относящимися к заказу, в качестве подтверждения принятия заказа	`Отправить детальную информацию о заказе`

Окончание табл. 10.3

Номер	Прецедент использования	Действие
6	`Клиент` выбирает функцию `Reset` (Сброс), для того чтобы вернуться к исходной форме заказа на покупку. Система дает возможность клиенту вновь ввести информацию	`Послать описание заказа по электронной почте`
7	Если прецедент был успешным, заказ на покупку записывается в базу данных. В противном случае состояние системы остается неизменным	`Вывести на экран бланк заказа`

Виды деятельности, приведенные в табл. 10.3, показаны на рис. 10.4.



Рис. 10.4. Действия в прецеденте использования `Заказ конфигурации` *системы Интернет-магазина*

10.2.2. Диаграмма деятельности

> **Этап 6. Интернет-магазин**
>
> Обратитесь к этапам 4 (см. раздел 10.1.4) и 5 (см. раздел 10.2.1). Постройте диаграмму видов деятельности для прецедента `Заказ конфигурации`.

Диаграмма деятельности для этапа 6 показана на рис. 10.5.

Первым действием является операция `Вывести текущую конфигурацию`. Это действие выполняется, пока не истечет заранее определенный интервал времени или не активизируется действие `Получить бланк заказа`. Если бланк заполнен не полностью, система вновь переходит в состояние `Вывести на экран бланк заказа`. В противном случае система переходит в состояние `Запомнить заказ`, а затем — в состояние `Послать описание заказа по электронной почте`.

Обратите внимание на то, что на диаграмме показаны только те условные переходы, которые всегда появляются на выходах из состояния вида деятельности. Условные переходы, которые являются внутренними для состояния вида деятель-

ности, не показаны явно на диаграмме. Об их существовании можно догадаться по наличию нескольких исходящих переходов, которые, возможно, сопровождаются именем условия в квадратных скобках (например, таким, как [просрочен] на выходе из состояния `Вывести на экран бланк заказа`).

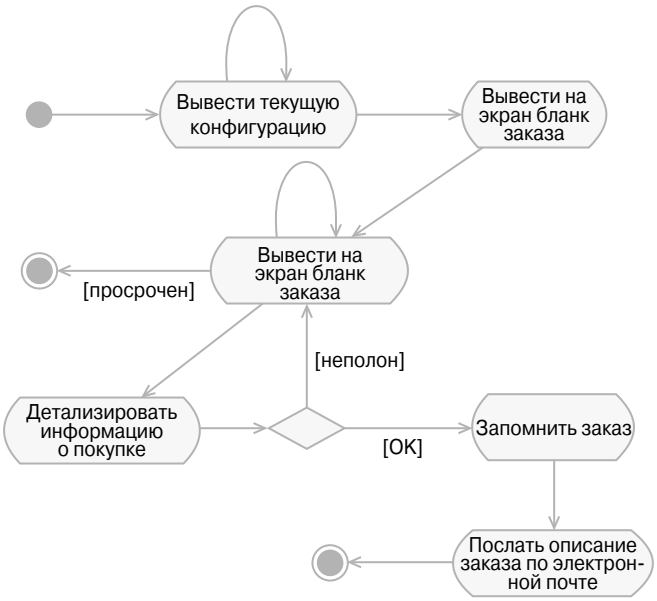


Рис. 10.5. Диаграмма деятельности в прецеденте использования `Заказ конфигурации` *системы Интернет-магазина*

10.3. Моделирование классов

10.3.1. Классы

Этап 7. Интернет-магазин

Обратитесь к требованиям, сформулированным в постановке задачи в начале главы и на этапе 1 (см. раздел 10.1.1). Определите потенциальные классы для системы управления работой Интернет-магазина.

Распределение функциональных требований по классам сущностей показано в табл. 10.4.

Таблица 10.4. Распределение требований среди классов сущностей системы управления Интернет-магазином

Номер	Требование	Класс сущностей
1	Для знакомства со стандартной конфигурацией выбираемого сервера, настольного или портативного компьютера клиент использует Web-страницу Интернет-магазина. При этом также приводится цена конфигурации	Customer (Клиент), Computer (Компьютер), (StandardConfiguration (Стандартная конфигурация), Product (Товар))
2	Клиент выбирает детали конфигурации, с которыми хочет ознакомиться, возможно, с намерением купить готовую или составить более подходящую конфигурацию. Цена для каждой конфигурации может быть подсчитана по требованию пользователя	Customer (Клиент), ConfiguredComputer (Конфигурация компьютера), (ConfiguredProduct (Укомплектованный товар)), ConfigurationItem (Элемент конфигурации)
3	Клиент может выбрать вариант заказа компьютера по Интернету либо попросить, чтобы продавец связался с ним для объяснения деталей заказа, договорился о цене и т.п. прежде, чем заказ будет фактически размещен	Customer (Клиент), ConfiguredComputer (Конфигурация компьютера), Order (Заказ), Salesperson (Продавец)
4	Для размещения заказа клиент должен заполнить электронную форму с адресами для доставки товара и отправки счета-фактуры, а также деталями, касающимися оплаты (кредитная карточка или чек)	Customer (Клиент), Order (Заказ), Shipment (Заказ), Shipment (Поставка), Invoice (Счет-фактура); Payment (Платеж)
5	После ввода заказа клиента в систему продавец отправляет на склад электронное требование, содержащее детали, касающиеся заказанной конфигурации	Customer (Клиент), Order (Заказ), Salesperson (Продавец), ConfiguredComputer (Конфигурация компьютера)Ю, ConfigurationItem (Элемент конфигурации)
6	Детали сделки, включая номер заказа, номер счета клиента, отправляются по электронной почте клиенту, так что заказчик может проверить состояние заказа через Интернет	Order (Заказ), Customer (Клиент), (OrderStatus (Состояние заказа))
7	Склад получает счет-фактуру Invoice от продавца и отправляет компьютер клиенту Shipment. Выделение классов представляет	Invoice (Счет-фактура), (Shipment (Поставка)), Computer (Компьютер),

Окончание табл. 10.4

Номер	Требование	Класс сущностей
	собой итеративную задачу, и первоначальный перечень предполагаемых классов, как правило, претерпевает изменения. При определении того, являются ли понятия, присутствующие в требованиях, искомыми классами, могут помочь ответы на некоторые вопросы.	`Customer (Клиент)`

- Чем отличаются классы `ConfiguredComputer` и `Order`? Например, следует ли хранить объект класса `ConfiguredComputer`, если заказ не был сделан?
- Совпадает ли смысл класса `Shipment`, приведенный в требованиях 4 и 7? Вероятно, нет. Необходим ли класс `Shipment`, если известно, что поставка является обязанностью склада и, таким образом, выходит за рамки приложения?
- Не является ли класс `ConfigurationItem` просто атрибутом класса `ConfiguredComputer`?
- Является ли класс `OrderStatus` самостоятельным или же атрибутом класса `Order`?
- Является ли класс `Salesperson` самостоятельным или же атрибутом классов `Order` и `Invoice`?

Ответить на эти и подобные вопросы нелегко. Для этого необходимы глубокие знания в прикладной области. С целью обучения мы выбрали список классов, представленных на рис. 10.6.

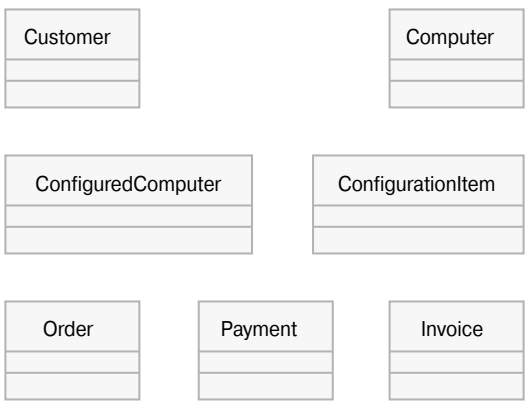


Рис. 10.6. Классы в системе Интернет-магазина

10.3.2. Атрибуты

Этап 8. Интернет-магазин

Обратитесь к этапам 5–7. Предложите атрибуты классов, показанных на рис. 10.6. Рассмотрите только атрибуты, имеющие элементарный тип (см. раздел А.3.1 приложения А).

На рис. 10.7 показаны классы с элементарными (наиболее интересными) атрибутами. Атрибуты класса `ConfigurationItem` требуют некоторых пояснений. Значениями атрибута `itemType` являются типы элементов конфигурации, такие как процессор, память, монитор, жесткий диск и т.д. Атрибут `itemDescr` содержит более подробное описание типа элемента. Например, процессор, входящий в конфигурацию, может представлять собой процессор марки Intel с частотой 4000 МГц, имеющий кэш объемом 2048 Кбайт.



Рис. 10.7. Элементарные атрибуты в классах системы Интернет-магазина

10.3.3. Отношения ассоциации

Этап 9. Интернет-магазин

Обратитесь к предыдущим заданиям. Рассмотрите классы, показанные на рис. 10.7. Подумайте, какие пути доступа между этими классами необходимы для прецедентов использования? Добавьте ассоциации в модель классов.

На рис. 10.8 показаны наиболее очевидные ассоциации между классами. При определении *кратности* ассоциаций был сделан ряд предположений (см. раздел А.5.2 приложения А). Заказ поступает от одного клиента, но клиент может разместить несколько заказов. Заказ не принимается до тех пор, пока не определены реквизиты платежа, поэтому возникает ассоциация "один к одному". Заказ не должен ассоциироваться со счетом-фактурой, однако счет-фактура всегда связан с единственным заказом. Заказ делается на один или несколько сконфигурированных компьютеров, а компьютер с данной конфигурацией может заказываться многократно или не заказываться вовсе.

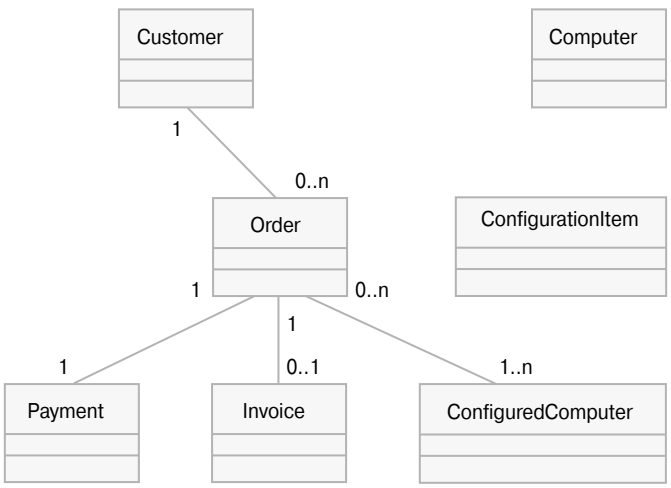


Рис. 10.8. Ассоциации между классами системы Интернет-магазина

10.3.4. Отношения агрегации

Этап 10. Интернет-магазин

Обратитесь к предыдущим заданиям. Рассмотрите модели, представленные на рис. 10.7 и 10.8. Добавьте к модели классов отношения агрегации.

На рис. 10.9 показаны отношения агрегации, включенные в модель. Компьютеры могут отличаться одним или более элементами конфигурации. Подобно этому, сконфигурированный компьютер состоит из одного или нескольких элементов.

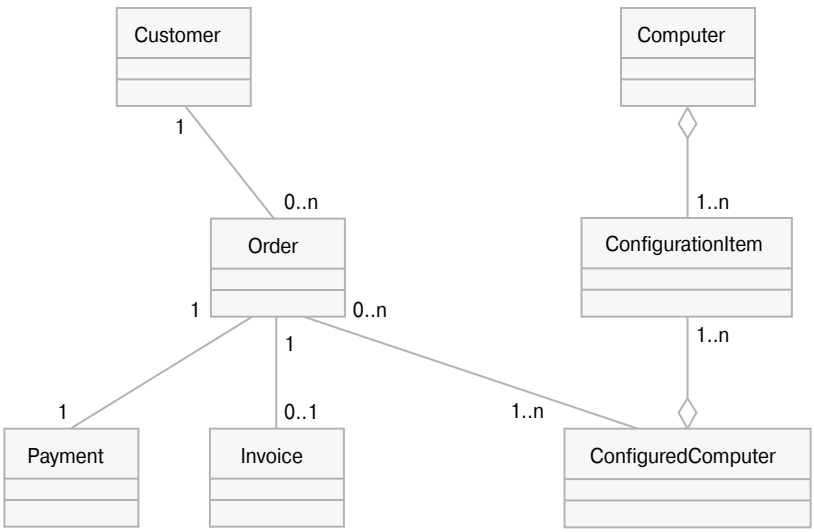


Рис. 10.9. Отношения агрегации между классами системы Интернет-магазина

10.3.5. Отношения обобщения

Этап 11. Интернет-магазин

Обратитесь к предыдущим этапам. Рассмотрите модели на рис. 2.28 и 2.30 (см. главу 2). Подумайте, каким образом можно вычленить какие-либо общие атрибуты в существующих классах и перенести их в класс более высокого уровня. Введите в модель классов обобщение.

На рис. 10.10 показана модифицированная модель, в которой класс `Computer` стал *абстрактным* классом с двумя *конкретными* подклассами: `StandardComputer` и `ConfiguredComputer`. Теперь классы `Order` и `ConfigurationItem` связаны с классом `Computer`, который может быть либо классом `StandardComputer`, либо классом `ConfiguredComputer`.

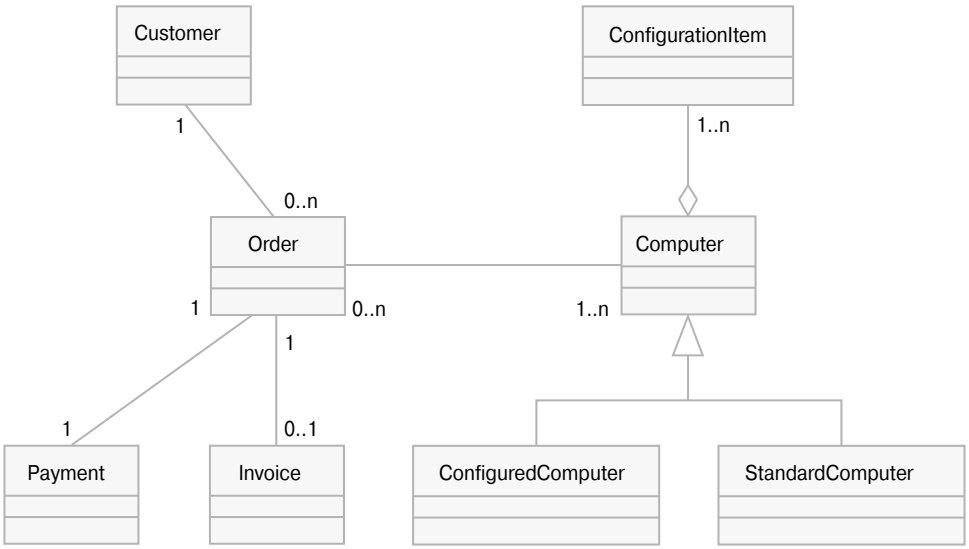


Рис. 10.10. Отношения обобщения между классами системы Интернет-магазина

10.3.6. Диаграмма классов

Этап 12. Интернет-магазин

Обратитесь к предыдущим этапам. Скомбинируйте модели, представленные на рис. 10.7 и 10.10, так, чтобы показать полную диаграмму классов. Измените содержимое атрибутов классов, как того требует введение обобщающей иерархии.

На рис. 10.11 показана диаграмма классов для приложения Интернет-магазина. Это еще не законченное решение, поскольку, например, для практического решения может потребоваться введение дополнительных атрибутов.

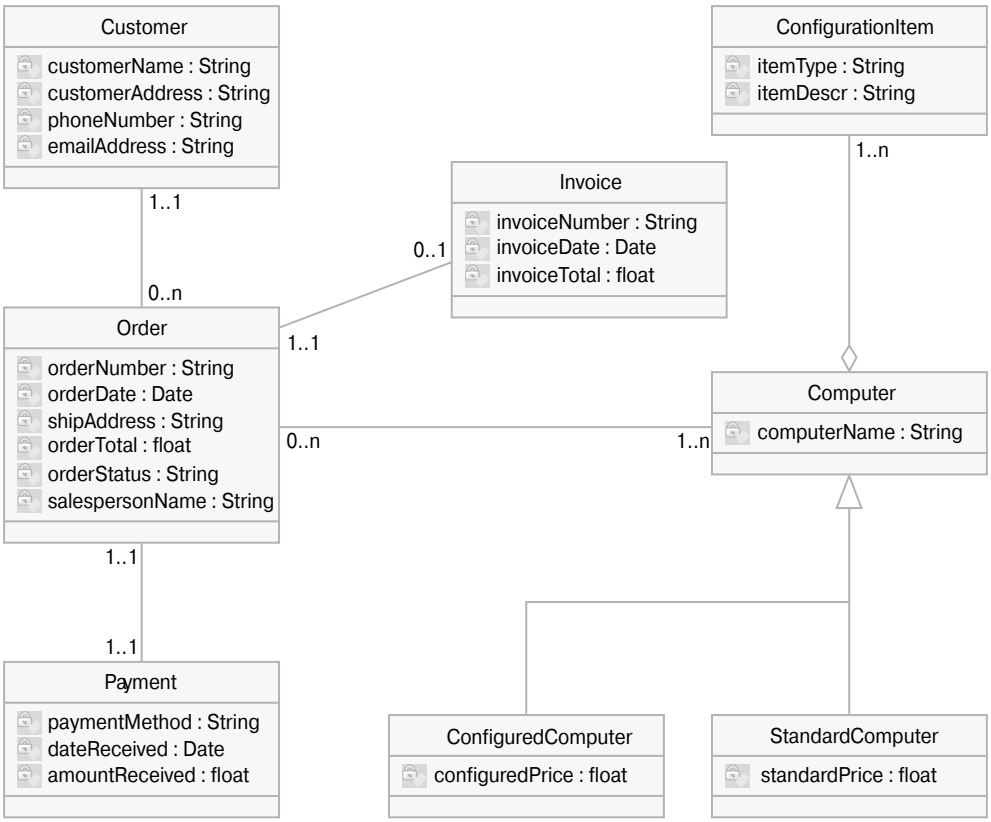


Рис. 10.11. Диаграмма классов в системе Интернет-магазина

10.4. Моделирование взаимодействия

10.4.1. Диаграмма последовательностей

Этап 13. Интернет-магазин

Обратитесь к диаграмме деятельности, представленной на рис. 10.5 (см. раздел 10.2.2). Рассмотрите первое действие на диаграмме: `Вывести на экран текущую конфигурацию`. Постройте диаграмму последовательностей для этого шага. В модели PCBMER перед именем класса указывается префикс, обозначающий уровень, которому принадлежит данный класс. Ограничьтесь классами представления и сущностей, предполагая, что уровень представления может непосредственно связываться с уровнем сущностей.

Для лучшего понимания задачи обратитесь к рис. 10.12 и 10.13. На рис. 10.12 (Sony, 2004) показана возможная Web-страница, с помощью которой пользователь может выбрать требуемую конфигурацию. На рис. 10.13 (Sony, 2004) показана Web-страница с описанием конфигурации, возникающая после того, как пользователь щелкнет на кнопке Submit.

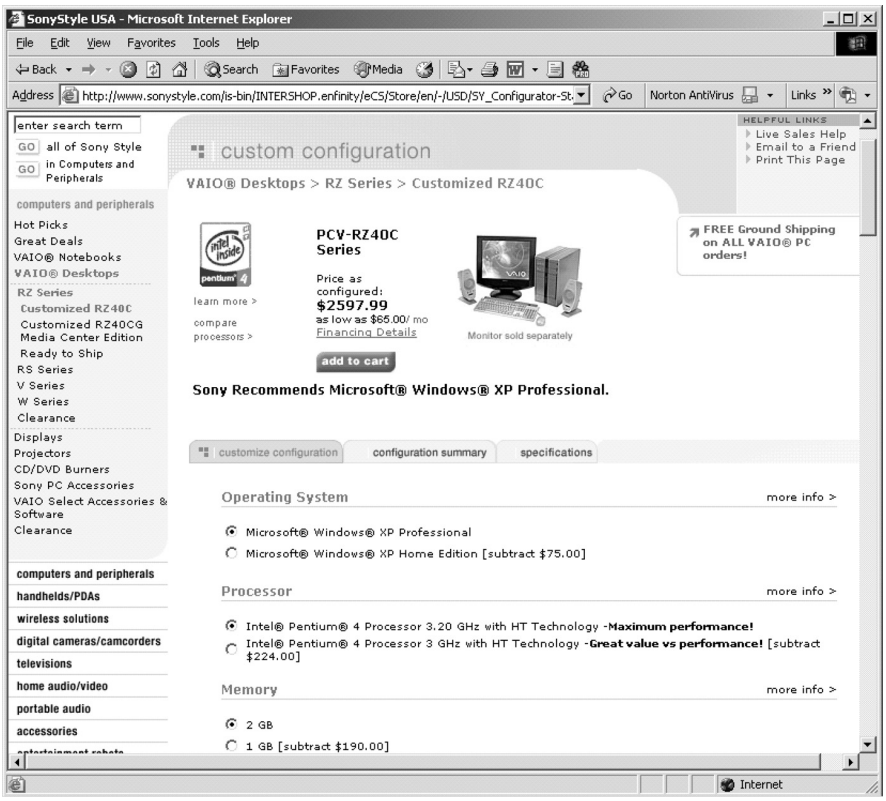


Рис. 10.12. Пример Web-страницы для выбора конфигурации в системе Интернет-магазина. Источник: Sony Electronic Inc. (2004)

Диаграмма последовательностей для "отображения текущей конфигурации" показана на рис. 10.14. Когда внешнее действующее лицо (`Клиент`) принимает решение вывести на экран конфигурацию компьютера, он щелкает на кнопке Submit, расположенной на Web-странице PCustomConfiguration. Это событие обслуживается методом `submit()`.

Поскольку Web-страница *владеет* объектом `EComputer`, она может запросить все детали непосредственно. Однако в модели, продемонстрированной на рис. 10.14, класс `EComputer` содержит всю информацию (`computerName`,

itemDescr и price) и возвращает ее классу PCustomConfiguration в виде коллекции. Фактически класс EConfigurationItem представляет собой коллекцию объектов, а метод getItemDescr() применяется к этой коллекции. Модель не объясняет, как именно работает метод getPrice().

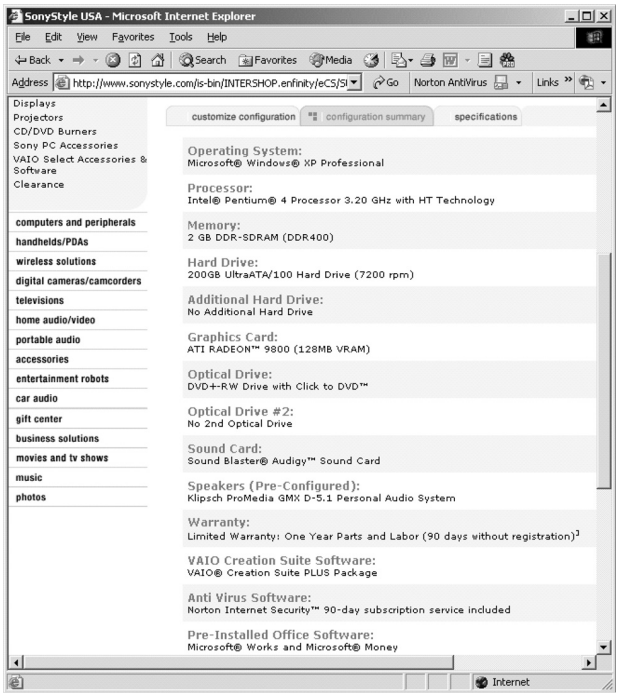


Рис. 10.13. Пример Web-страницы с описанием конфигурации в системе Интернет-магазина. Источник: Sony Electronic Inc. (2004)

Получив всю требуемую информацию, класс PCustomConfiguration создает новую Web-страницу PConfigurationSummary. Его конструктор получает всю информацию через аргументы метода new(). Конструктор класса PConfigurationSummary содержит метод display(), предназначенный для вывода текущей конфигурации на экран.

10.4.2. Диаграмма коммуникации

Этап 14. Интернет-магазин

Преобразуйте диаграмму последовательностей, представленную на рис. 10.14, в диаграмму коммуникации.

Диаграмма коммуникации, альтернативная диаграмме последовательностей, представленной на рис. 10.14, приведена на рис. 10.15. Класс `EConfiguration-Item` в этой модели представлен в виде коллекции объектов.

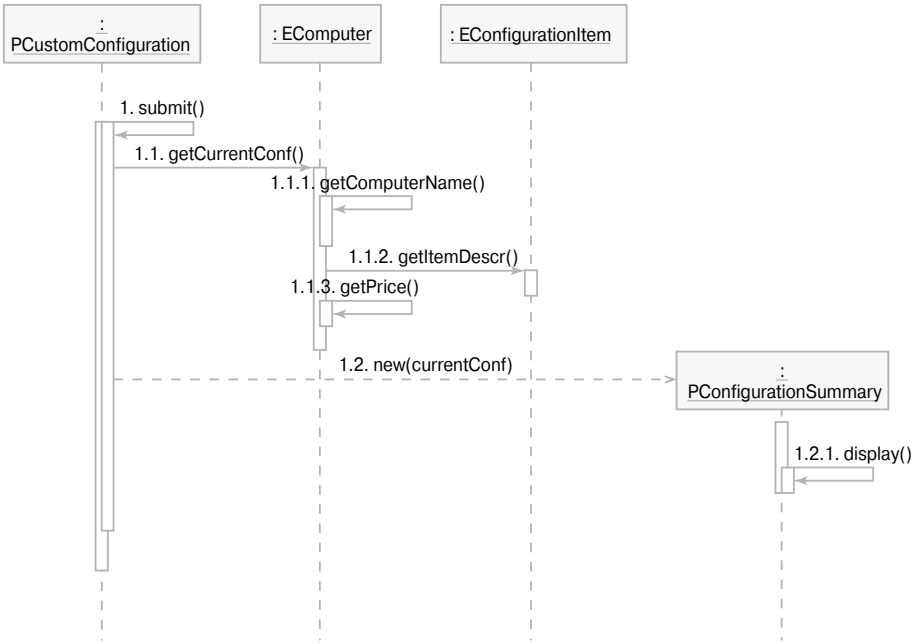


Рис. 10.14. Диаграмма последовательностей для действия `Вывести на экран текущую конфигурацию` *в системе Интернет-магазина*

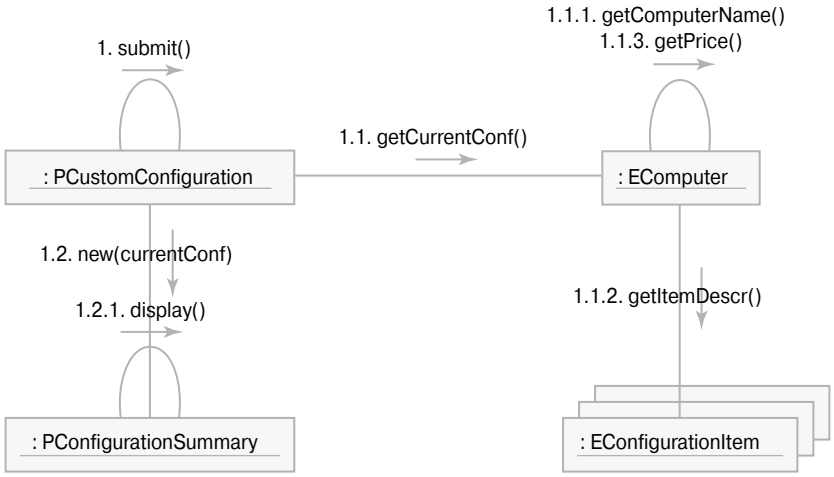


Рис. 10.15. Диаграмма коммуникации для действия `Вывести на экран текущую конфигурацию` *в системе Интернет-магазина*

Этап 15. Интернет-магазин

Обратитесь к диаграмме классов, показанной на рис. 10.11, и диаграмме коммуникации, приведенной на рис. 10.15. Не переделывайте всю диаграмму классов — покажите лишь классы, содержащие дополнительные операции. Покажите отношения зависимости между классами, ранее не связанными друг с другом.

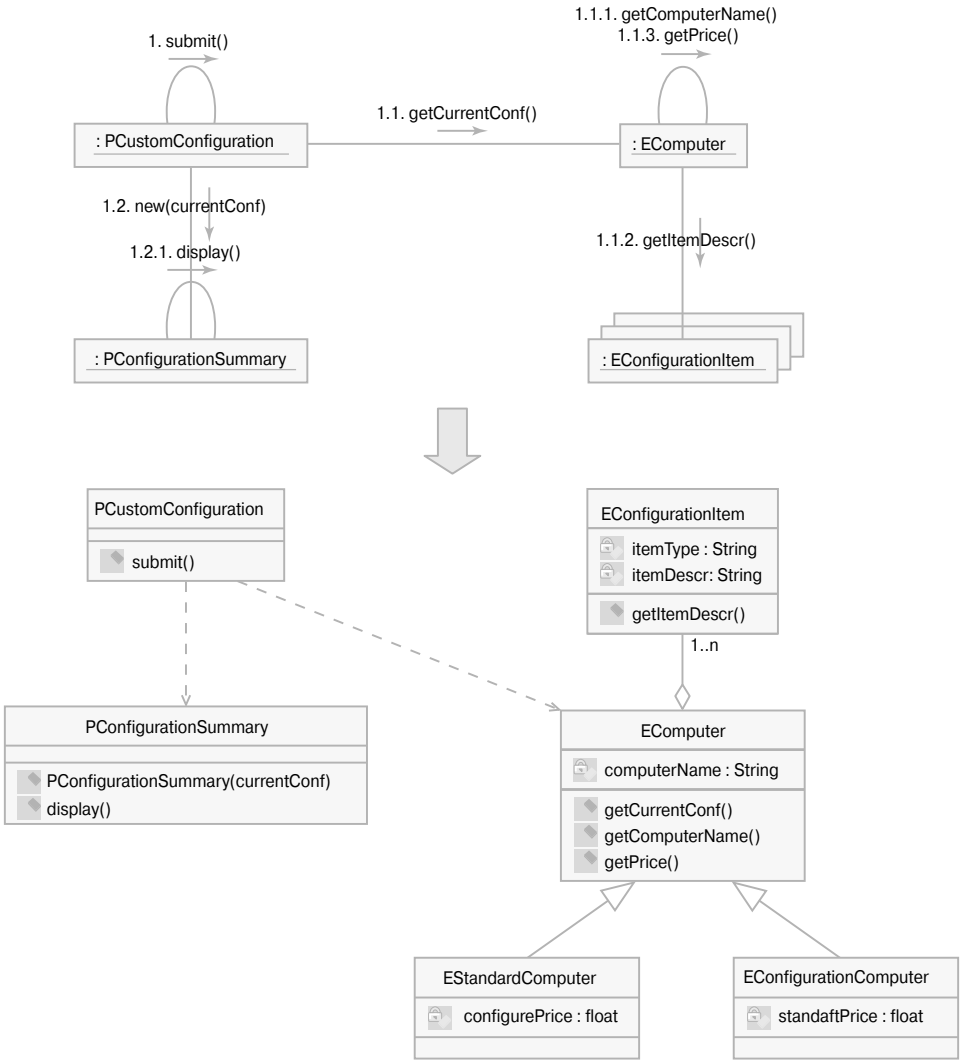


Рис. 10.16. Использование взаимодействия для добавления операций в классы системы Интернет-магазина

Решение этой задачи представлено на рис. 10.16. Операции добавлены в соответствии с указанными требованиями. Обратите внимание на то, что сообщение `new()` приводит к вызову конструктора класса `PConfigurationSummary`. Класс `PCustomConfiguration` зависит от классов *`EComputer`* и `PConfigurationSummary`.

Класс `EComputer` — это *абстрактный* класс. Операции *`getCurrentConf()`* и *`getPrice()`* — абстрактные операции, унаследованные и реализованные подклассами `EConfiguredComputer` и `EStandardComputer`.

10.5. Моделирование конечных автоматов

10.5.1. Состояния и переходы

Этап 16. Интернет-магазин

Рассмотрите класс `Invoice` (Счет-фактура), относящийся к системе Интернет-магазина. Из модели прецедентов известно, что клиент определяет способ оплаты (кредитная карточка или чек) за компьютер, когда бланк заказа заполнен и отослан производителю. Это приводит к генерации заказа и последующей подготовке счета-фактуры. Однако диаграмма прецедентов не проясняет, когда же производится фактическая оплата в соответствии со счетом-фактурой. Можно, к примеру, предположить, что оплата возможна до или после того, как счет-фактура выдана, а также предположить возможность частичной оплаты.

Из модели класса нам известно, что счет-фактура для заказа подготавливается продавцом и в конечном итоге передается на склад. Склад отправляет счет-фактуру клиенту одновременно с доставкой компьютера. Важно отметить, что состояние оплаты счета-фактуры отслеживается в системе, так что счет-фактура снабжен надлежащими комментариями.

Постройте диаграмму состояний, которая фиксирует возможные состояния счета-фактуры в той мере, в которой это касается платежей.

На рис. 10.17 показана модель конечного автомата для класса `Invoice`. Начальным состоянием объекта `Invoice` является состояние `Unpaid` (Заказ не оплачен). Из состояния `Unpaid` возможны два перехода. При поступлении частичной оплаты объект `Invoice` переходит в состояние `Partly Paid` (Заказ оплачен частично). Допускается только одна частичная оплата. Поступление окончательного платежа, когда объект `Invoice` находится в состоянии `Unpaid` или `Partly Paid`, инициирует переход в состояние `Fully Paid` (Заказ полностью оплачен). Это конечное состояние.

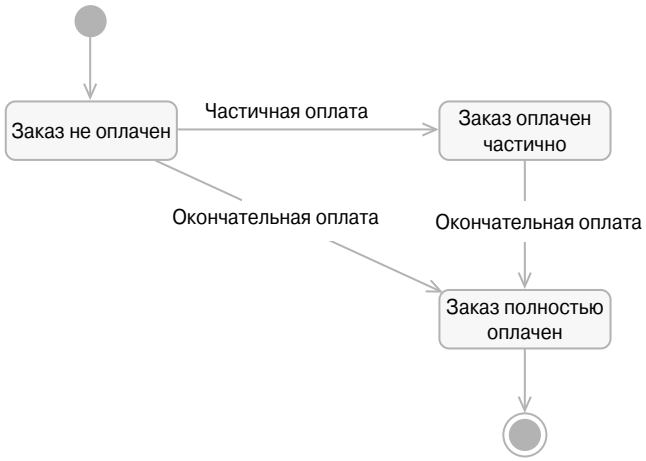


Рис. 10.17. Состояния и события, связанные с классом `Invoice` *в системе Интернет-магазина*

10.5.2. Состояния и переходы

Этап 17. Интернет-магазин

Обратитесь к предыдущим этапам. Рассмотрите класс `Order` (Заказ), относящийся к системе Интернет-магазина. Обдумайте, в каких состояниях может находиться класс `Order`, начиная с отправки заказа в систему и заканчивая его заполнением.

Рассмотрите ситуации, когда заказанный компьютер находится на складе в готовом виде или же его необходимо сконфигурировать в отдельности, чтобы удовлетворить требования клиента. Клиент может также задать определенный день, в который он бы хотел получить компьютер, даже если он уже есть на складе.

Клиент может отменить заказ в любое время, до того как компьютер будет доставлен ему. Моделировать ситуацию взимания штрафа, связанную со слишком поздним отказом, не требуется. Постройте диаграмму состояний для класса `Order`.

Диаграмма конечного автомата для класса `Order` продемонстрирована на рис. 10.18. Начальное состояние называется `New Order` (Новый заказ). Оно является частью состояния `Pending` (Незавершенные заказы) наряду с состояниями `Future Order` (Будущий заказ) и `Back Order` (Невыполненный заказ). Из этих трех состояний есть два возможных перехода.

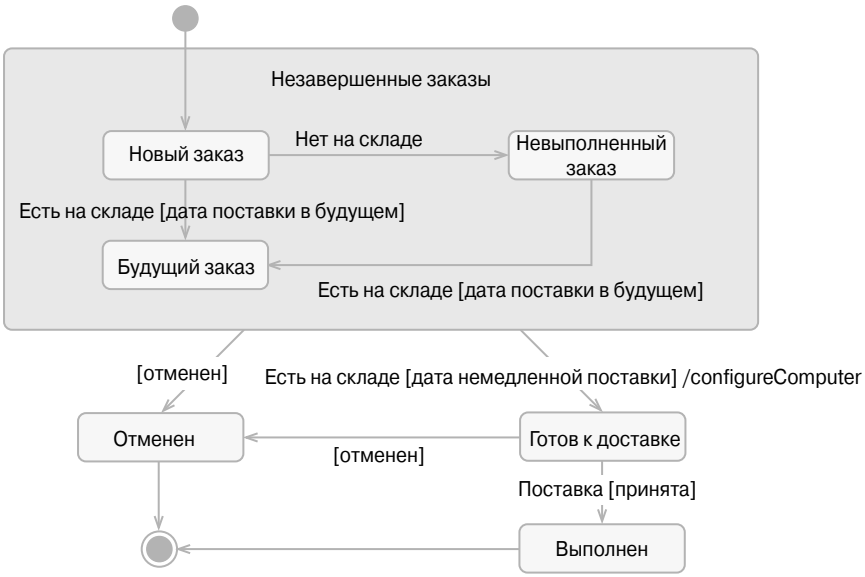


Рис. 10.18. Диаграмма конечного автомата для класса `Order` *в системе Интернет-магазина*

Переход в состояние `Canceled` (Отменен) защищено условием [`отменен`]. Должна существовать возможность — без изменения правил моделирования состояний — заменить защитное условие событием `cancel` (отмена). Переход в состояние `Ready to Ship` (Готов к доставке) помечен полным описанием, содержащим событие, защиту и действие.

10.6. Модели реализации

10.6.1. Подсистемы

Этап 18. Интернет-магазин

Уровни в архитектурной модели PCBMER моделируются как подсистемы. Это соответствует условию, что сервисы подсистемы должны инкапсулироваться интерфейсами.

Дополните архитектурную диаграмму модели PCBMER, представленную на рис. 4.3 (см. раздел 4.1.3.1 главы 4), указав предлагаемые и требуемые интерфейсы. Используйте обозначение “леденца”(см., например, рис. 3.19 в разделе 3.6.2 главы 3).

Рис. 10.19 не требует дополнительных пояснений. Например, подсистема `Bean` (Компоненты) обеспечивает интерфейсы, требуемые уровнями `Presentation` (Представление) и `Control` (Управление).

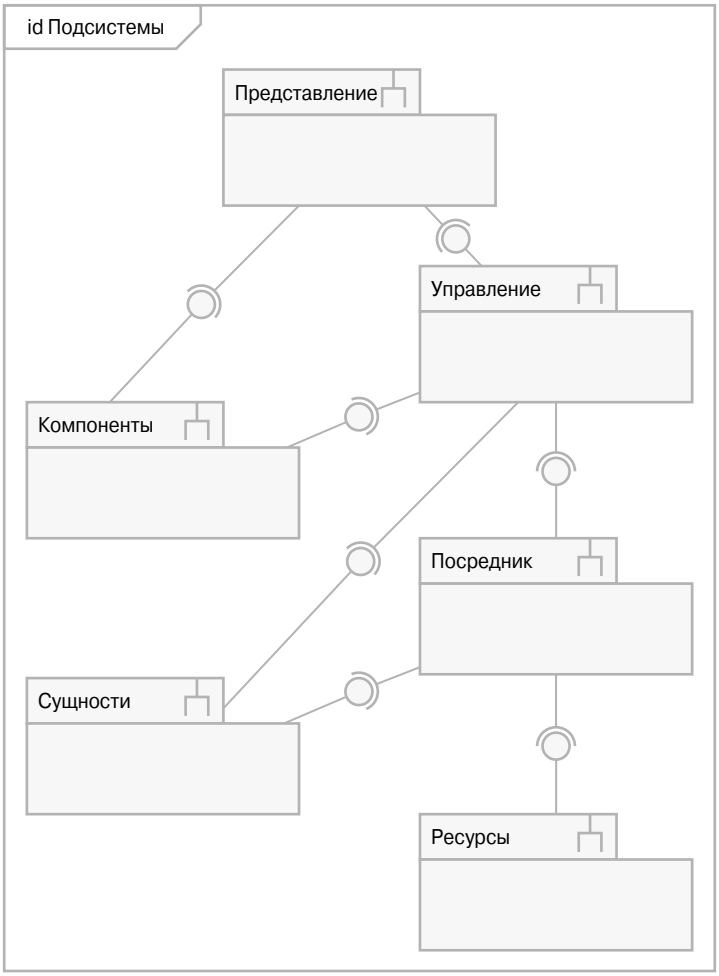


Рис. 10.19. Подсистемы PCBMER со своими интерфейсами в системе Интернет-магазина

10.6.2. Пакеты

Этап 19. Интернет-магазин

Обратитесь к предыдущим этапам разбора учебного примера, посвященного Интернет-магазину. Вполне естественно, что большинство классов, определенных там, представляют собой персистентные бизнес-объекты. Более полная модель для системы может потребовать идентификации других классов прикладной программы. Это можно сделать постепенно в ходе проектирования системы. Даже если классы прикладной программы еще не определены, можно сделать предположения относительно пакетов, которые должны группировать классы в связанные модули в соответствии с подходом PCBMER.

Обдумайте возможный состав и структуру пакетов для Интернет-магазина и основные зависимости между ними. Дополните модель, представленную на рис. 10.19, включив в нее пакеты.

Лучший способ решить задачу — "сымитировать систему" и представить, что необходимо сделать, чтобы принять заказ клиента на конфигурацию компьютера. Первый очевидный вывод состоит в том, что система справляется с двумя отдельными функциями: конфигурированием компьютера и вводом заказа. Эти две функции требуют отдельных окон графического пользовательского интерфейса, поэтому можно создать два пакета: `просмотр конфигурации` и `просмотр заказа`, представленных на рис. 10.20.

"Деловая часть" спектра классов показана на диаграмме классов (см. рис. 10.11). Классы подсистемы *Сущности* можно естественным образом сгруппировать в виде трех пакетов сущностей: `клиенты`, `компьютеры` и `заказы` (последний может также включать классы `Invoice` и `Payment`). Более того, необходим пакет, присваивающий уникальные идентификаторы объектов `OID` объектам сущностей и поддерживающий их соответствие (см. раздел 8.4 главы 8). Назовем этот пакет `коллекция объектов`.

Далее необходимо идентифицировать пакеты, соединяющие классы представления и сущностей, — пакеты *–Control*. Нам необходим пакет для конфигурирования компьютеров и вычисления стоимости конфигурации. Назовем его `configuration provider`. Кроме того, нужен пакет, отвечающий за ввод и запись заказов, — пакет `order monitor` (монитор заказов).

О подсистеме *Посредник* известно немного, но, как следует из фактов, касающихся шаблонов управления персистентными объектами (см. раздел 8.4 главы 8), необходимы по крайней мере три пакета этой подсистемы. Эти пакеты можно назвать, следуя именам шаблонов, — `данные`, `преобразователь`, `загрузка по требованию` и `единица работы`.

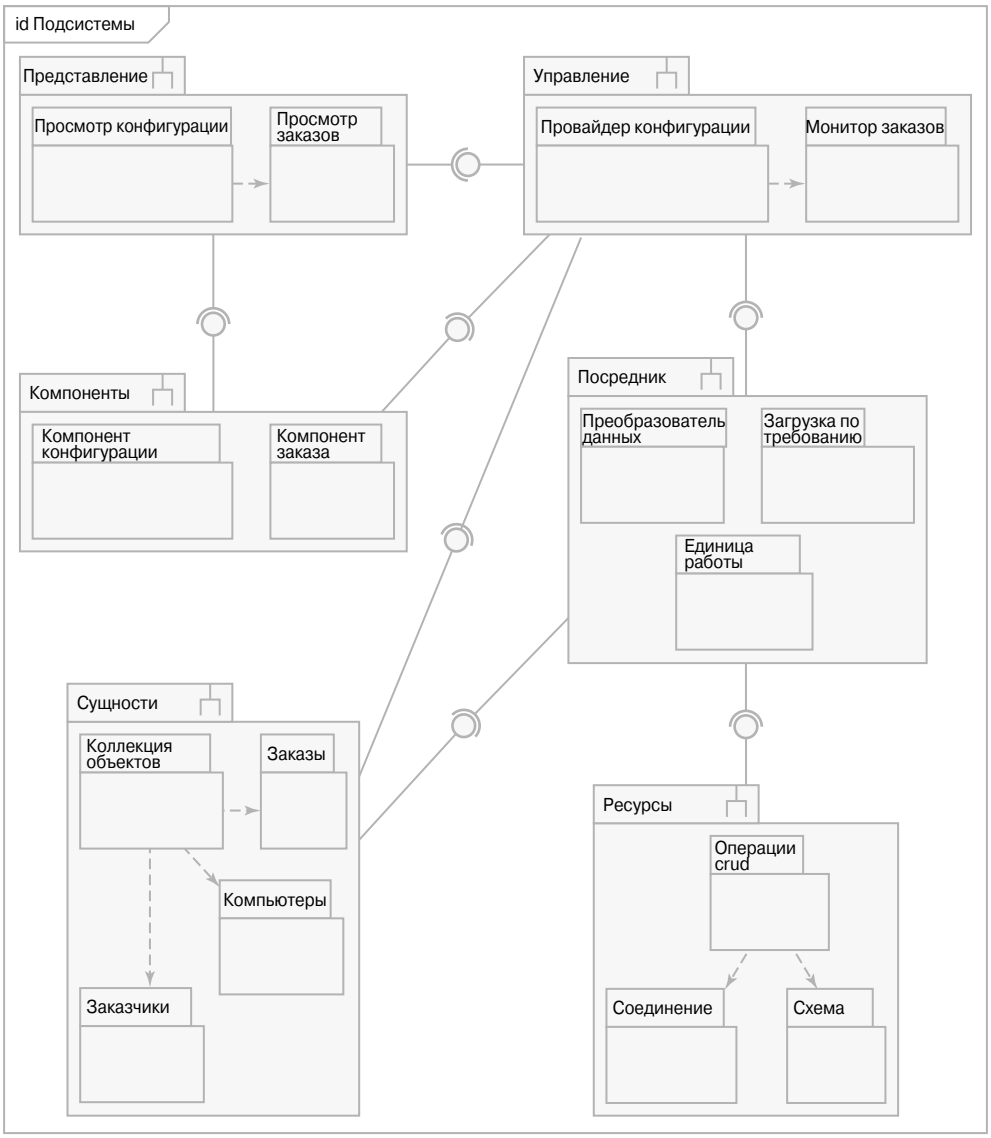


Рис. 10.20. Пакеты системы Интернет-магазина

Наконец, необходим один или несколько пакетов подсистемы `Ресурсы`. Основной пакет баз данных можно назвать пакетом `crud` — Create–Read–Update–Delete (создать–читать–обновить–удалить (см. раздел 4.3.4.1 главы 4)). Пакет `crud` обращается к таблицам базы данных всякий раз, когда приложению требуется доступ к ее содержимому или его модификация.

Пакет `crud` зависит от двух других пакетов баз данных — `соединение` и `схема`. Классы пакета `соединение` отвечают за установление соединения, авторизацию и транзакции. Пакет `схема` содержит текущую информацию об объектах схемы базы данных — таблицах, столбцах, хранимых процедурах и т.д. Приложение может порождать объекты пакета `схема` при запуске, так что оно в состоянии проверить, что объекты базы данных существуют в базе, прежде чем осуществить попытку фактического доступа к базе данных (например, перед вызовом хранимой процедуры приложение может проверить, используя находящиеся в памяти объекты схемы, что хранимая процедура по-прежнему существует).

10.6.3. Компоненты

> **Этап 20. Интернет-магазин**
>
> Разработайте диаграмму компонентов для бизнес-объектов в описанном примере. Напомним, что компонент — это связная функциональная часть системы с понятным интерфейсом, допускающая замену. Поскольку платформа реализации системы управления Интернет-магазином до сих пор не указывалась, идентификация более мелких компонентов (например, библиотеки и хранимых процедур) на этом этапе еще невозможна.

Для решения задачи естественно рассмотреть типичную последовательность действий клиента, желающего купить компьютер и переходящего с одной Web-страницы на другую. В основе этих переходов лежит анализ прецедентов использования, изложенный в разделе 10.1.

Первая Web-страница, которую может посетить интерактивный пользователь, — это Web-страница поставщика, на которой перечислены категории изделий (серверы, настольные системы, портативные компьютеры), выделены последние предложения и скидки и приводятся ссылки на Web-страницы, на которых представлены перечни изделий и дано их краткое описание. Краткое описание включает также цену стандартной конфигурации изделия. Эта часть системы посвящена рекламе товаров интерактивным покупателям. Это единый функциональный блок, который может образовать компонент `ProductList` (Перечень изделий).

На следующем шаге клиент может запросить технические спецификации выбранного изделия. Сюда входит визуальное отображение изделия под разными углами зрения. Это функционально законченная Web-страница, которая является неплохим претендентом на компонент `ProductDisplay` (Отображение изделия).

Если предположить, что описанные выше Web-страницы привлекли внимание клиента к продукту, то он может сформулировать запрос в отношении различных конфигураций изделия, удовлетворяющих его специфические нужды и требова-

ния по расходам. Подобный запрос можно удовлетворить с помощью динамических Web-страниц, позволяющих в интерактивном режиме построить конфигурацию и отобразить готовое изделие вместе с рассчитанной ценой конфигурации. Это следующий неплохой претендент на компонент, который мы назовем `Configuration` (Конфигурация).

Клиенту, решившему купить изделие, предоставляется бланк заказа на покупку. В нем должны быть, в частности, указаны такие детализированные данные, как имя клиента, адрес, по которому следует доставить изделие и послать счет-фактуру. Одновременно выбирается способ оплаты, и соответствующие подробности пересылаются с использованием некоторого безопасного протокола передачи данных. Это четвертый компонент — `Purchase` (Покупка).

Последний компонент должен отслеживать выполнение заказа. С точки зрения клиента, он должен обеспечивать возможность просмотра состояния заказа с помощью Web-страницы (после ввода номера клиента и номера заказа). Этот компонент можно назвать `OrderTracking` (Отслеживание заказа).

Рассмотренные выше пять компонентов показаны на рис. 10.21. Основные зависимости между компонентами показаны с помощью требуемых и предоставляемых интерфейсов. Компоненты — это реальные модули с самоуправляемым развертыванием. Они требуют тщательного проектирования и реализации даже для небольших систем.

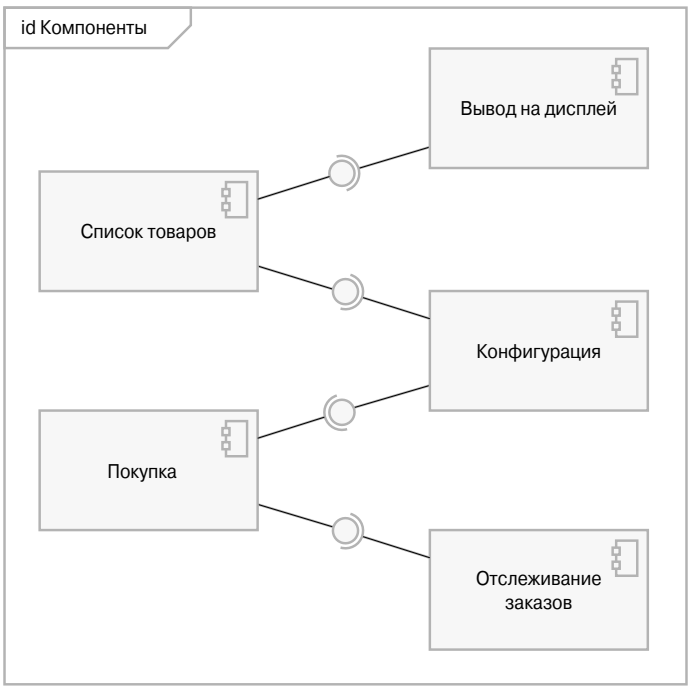


Рис. 10.21. Диаграмма компонентов системы Интернет-магазина

10.6.4. Примечания

Этап 21. Интернет-магазин

Рассмотрите модели реализации, описанные на этапах 18–20, и постройте диаграмму развертывания системы управления Интернет-магазином. В частности, решите, требуется ли сервер приложения.

Поскольку Интернет-системы не требуют прямого соединения, *развертывание* Web-приложений становится значительно более сложной задачей, чем развертывание приложений баз данных в архитектуре "клиент/сервер". Для того чтобы приступить к развертыванию, требуется установить Web-сервер в качестве пункта маршрутизации между всеми браузерами клиентов и базой данных.

Если проблема управления сеансами не может быть удовлетворительно решена с помощью технологии *cookie*, необходимо воспользоваться технологией распределенных объектов. Развертывание распределенных объектов может потребовать размещения отдельных архитектурных элементов — сервера приложений — между Web-сервером и сервером баз данных.

Проект развертывания должен обращаться к вопросам безопасности. Безопасная передача данных и протоколы шифрования — еще один аспект требований со стороны проекта развертывания. Кроме того, необходимо тщательное планирование с учетом сетевой загрузки, Интернет-соединений, резервных копий и т.д.

Архитектура развертывания, способная поддерживать более сложные Web-приложения, включает четыре яруса вычислительных узлов.

1. Клиентский Web-браузер.
2. Web-сервер.
3. Сервер приложений.
4. Сервер баз данных.

Браузер клиентского узла можно использовать для отображения статических или динамических Web-страниц. Страницы, включающие сценарии и аплеты, можно загружать и выполнять в рамках браузера. Клиентский браузер можно оснастить дополнительными функциональными возможностями, такими как элементы управления ActiveX или JavaBeans. Выполнение программы приложения на клиентской машине, но вне браузера, может удовлетворить другие требования к графическому пользовательскому интерфейсу.

Web-сервер обрабатывает запросы на страницы, поступающие от браузера, и динамически генерирует страницы и программный код для выполнения и отображения на клиенте. Web-сервер также обеспечивает настройку и параметризацию сеансов работы пользователя.

Сервер приложений необходим в том случае, когда в реализации используются распределенные объекты. Он управляет бизнес-логикой. Бизнес-компоненты публикуют свои интерфейсы для других узлов через интерфейсы компонентов, такие как CORBA, DCOM или EJB.

Бизнес-компоненты инкапсулируют постоянные объекты, хранимые в базе данных, чаще всего в реляционной базе. Они взаимодействуют с сервером баз данных через протоколы связи с базами данных, например, такими, как JDBC или ODBC. Узел базы данных обеспечивает масштабируемое хранилище данных и многопользовательский доступ к нему.

Как показано на рис. 10.22, Интернет-магазин можно развернуть без отдельного сервера приложений. Программы, содержащиеся в Web-страницах, могут выполняться Web-сервером. Потенциальным преимуществом сервера приложений является то, что хранимые на нем компоненты приложения могут повторно использоваться другими Web-приложениями для вызова той же бизнес-логики. Однако Интернет-магазин — это автономная система, и вряд ли можно указать другие Web-приложения, которые могли бы извлечь пользу из ее бизнес-логики.

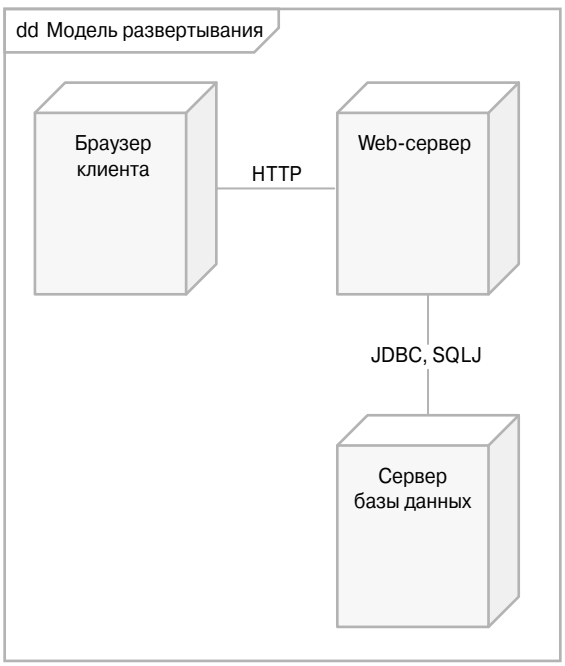


Рис. 10.22. Диаграмма развертывания системы Интернет-магазина

10.7. Разработка кооперации объектов

Взаимодействия определяют реализацию прецедентов использования и более сложных операций (простые операции не следует моделировать как кооперацию). Проектирование кооперации неизменно приводит к *уточнению* (модификации и расширению) существующих диаграмм классов и выработке новых диаграмм кооперации (либо к детализации существующих диаграмм последовательностей). Другие типы диаграмм, в частности диаграммы состояний, также могут потребовать разработки или уточнения.

Важным побочным результатом (или даже предпосылкой) проектирования кооперации является уточнение прецедентов. Прецеденты, зафиксированные в виде документа при проведении анализа требований, как правило, не достаточно детализированы для проектирования кооперации. Спецификации прецедентов необходимо уточнить в проектной документации. Прецеденты, представленные более подробными спецификациями, должны включать требования системного уровня, оставаясь в то же время на точке зрения субъектов.

Если на этапе анализа (см. раздел 2.4 главы 2) управление требованиями уходит на второй план, то теперь открывается последняя возможность достичь большей формализации и строгости в их определении. Требования следует подвергнуть тщательной классификации и пронумеровать их. Для того чтобы обеспечить надлежащее управление изменениями и трассируемость требований, их необходимо поместить в репозиторий CASE-системы. Попытки отследить изменения требований вручную обречены на неудачу. Однако с помощью CASE-средств их легко пронумеровать и реструктуризировать.

На рис. 10.23 показан фрагмент спецификации прецедентов использования. Требования пронумерованы с использованием десятичной системы Девея (Dewey), при этом каждое требование обозначено префиксом UC (use case — прецедент). Префикс полезен в том случае, когда документ описания прецедентов содержит более одного типа требований. Заметим, что требования заключены в квадратные скобки, подчеркнуты и отображаются зеленым цветом.

После ввода в CASE-репозитарий требования можно просматривать и модифицировать с помощью других инструментов, поддерживаемых средствами CASE. На рис. 10.24 показан экран, на котором отображена иерархия требований. Разработчик может использовать этот экран для модификации любых требований или добавления любых атрибутов

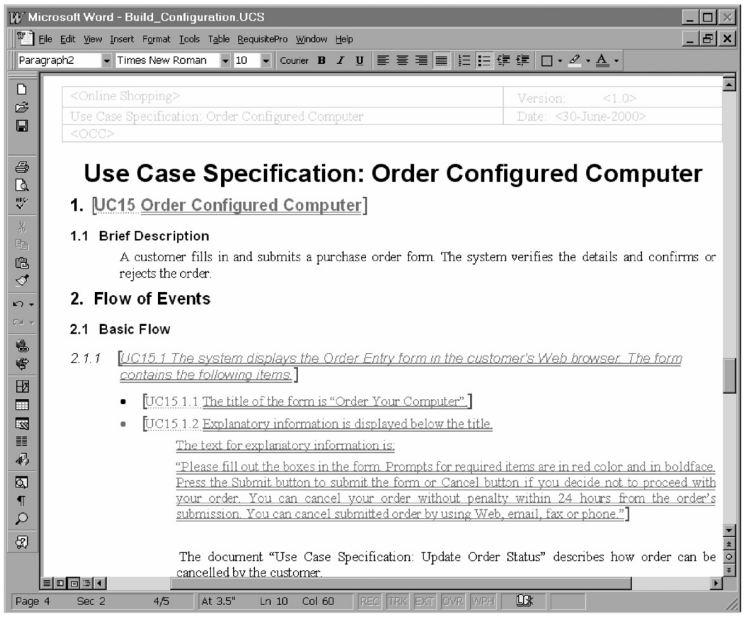


Рис. 10.23. Фрагмент спецификации прецедента использования, поддерживаемой CASE-средством. (Скриншот воспроизводится с разрешения компании Microsoft Corporation.)

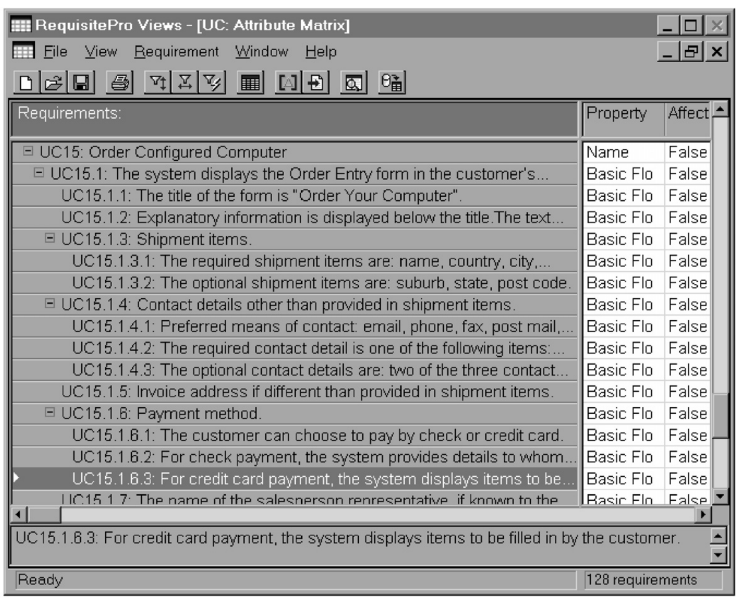


Рис. 10.24. Управление требованиями с помощью CASE-средства

10.7.1. Проектные спецификации прецедентов использования

Этап 22. Интернет-магазин

Обратитесь к спецификации прецедентов использования системы, управляющей Интернет-магазином, приведенной в разделе 10.2.2 (см. табл. 10.2). Этот документ описывает прецедент `Заказ конфигурации`. Однако он недостаточно подробен для разработки модели кооперации объектов.

Следовательно, цель данного этапа заключается в уточнении спецификации прецедентов так, чтобы он представлял собой спецификацию прецедентов, пригодную для проектирования. Уточненный документ должен быть организован так, как показано на рис. 10.23. Фактически рис. 10.23 и 10.24 представляют собой часть решения на данном этапе.

Соответствующий текст документа описания прецедента приведен ниже. Заметим, что этот документ можно представить в другом формате. Например, можно не указывать номера прецедентов.

Спецификация прецедента: заказ конфигурации

1. [UC15 Заказ конфигурации]

1.1. Краткое описание

Клиент заполняет и отправляет форму заказа на покупку. Система проверяет детальную информацию и подтверждает или отклоняет заказ.

2. Поток событий

2.1. Основной поток

[UC15.1 Система отображает форму ввода заказа в окне браузера клиента. Бланк состоит из следующих пунктов.]

[UC15.11.1 форму имеет заголовок “Заказывайте компьютер”.]

[UC15.11.2 Под заголовком отображается поясняющая информация.]

Текст поясняющей информации имеет следующее содержание.

“Заполните, пожалуйста, поля формы. Приглашения для ввода требуемых ответов показаны красным полужирным шрифтом. Для отправки формы щелкните на кнопке Submit; если же вы решили не продолжать заполнение, щелкните на кнопке Cancel. Вы можете отказаться от своего заказа без всякого штрафа в течение 24-х часов с момента отправки заказа. Вы можете отказаться от отправленного заказа, прибегнув к помощи Web, электронной почты, факса или телефона”.]

Описание того, каким образом клиент может отказаться от заказа, содержится в документе "Спецификация прецедента: обновление статуса заказа".

[UC15.11.3 Реквизиты поставки].

[UC15.11.3.1 К требуемым реквизитам доставки относятся: имя, страна, город, улица, курьерские направления.]

[UC15.11.3.2 К дополнительным реквизитам доставки относятся: пригород, штат, почтовый код.]

[UC15.11.4 Контактная информация, отличная от той, которая предоставляется как реквизиты доставки.]

[UC15.11.4.1 Предпочтительные средства для контактов: электронная почта, телефон, факс, почта, курьерская почта.]

[UC15.11.4.2 В качестве обязательной контактной информации следует указать один из следующих реквизитов: электронная почта, телефон, факс.]

UC15.11.4.3 В качестве дополнительной контактной информации можно указать два из трех контактных реквизитов, перечисленных как обязательные, и почтовый адрес (если он отличается от указанного в реквизитах доставки).]

[UC15.11.5 Адрес доставки счета-фактуры, если он отличается от реквизитов доставки.]

[UC15.11.6 Способ оплаты.]

[UC15.11.6.1 Клиент может выбрать в качестве метода оплаты чек или кредитную карточку.]

[UC15.11.6.2 При оплате с помощью чека система предоставляет подробную информацию, кому должен быть оплачен чек и по какому адресу его следует доставить. Она также информирует клиента, что на клиринг чека после получения может потребоваться до трех дней.]

[UC15.11.6.3 При оплате с помощью кредитной карточки система отображает реквизиты, которые должен заполнить клиент. К этим реквизитам относятся: перечень допустимых кредитных карточек, номер кредитной карточки, срок действия кредитной карточки.]

[UC15.11.7 Имя торгового представителя, если оно известно клиенту по предыдущим сделкам.]

[UC15.11.8 Две командные кнопки: Submit и Cancel.]

[UC15.12 Система предлагает клиенту ввести подробные данные, касающиеся заказа, поместив указатель мыши на первое редактируемое поле (реквизит имени).]

[UC15.13 Система позволяет вводить информацию в произвольном порядке.]

[UC15.14 Если клиент не отправляет и не аннулирует форму в течение 15 минут, выполняется альтернативный поток "Клиент не активен").]

[UC15.15 Если клиент щелкает на кнопке Отправить и вся необходимая информация предоставлена, форма заказа отправляется на Web-сервер. Web-сервер связывается с сервером баз данных для того, чтобы поместить заказ в базу данных.]

[UC15.16 Сервер баз данных присваивает заказу на покупку уникальный номер заказа и учетный номер клиента.]

[UC15.17 Если сервер баз данных не в состоянии создать и запомнить заказ, выполняется альтернативный поток "Исключительная ситуация для базы данных".]

[UC15.18 Если клиент отправляет заказ с неполной информацией, выполняется альтернативный поток "Неполная информация".]

[UC15.19 Если клиент в качестве предпочтительного средства коммуникации предоставляет адрес электронной почты, система отправляет клиенту заказ и номера клиента вместе со всеми деталями, касающимися заказа, и подтверждением принятия заказа. Прецедент завершается.]

[UC15.20 В противном случае детали, касающиеся заказа, отправляются клиенту по почте и прецедент также завершается.]

[UC15.21 Если клиент щелкает на кнопке Cancel**, выполняется альтернативный поток "Отмена".]**

2.2. Альтернативные потоки

Клиент неактивен

[UC15.14.1 Если клиент не проявляет активности в течение более чем 15 минут, система закрывает соединение с браузером. Прецедент завершается.]

Исключительная ситуация для базы данных

[UC15.17.1 Если база данных возбуждает исключительную ситуацию, система интерпретирует ее и информирует клиента о характере ошибки. Если клиент разорвал соединение, система отправляет сообщение об ошибке по электронной почте клиенту и продавцу. Прецедент завершается.]

Если с клиентом нет возможности связаться посредством Интернета или электронной почты, продавец должен связаться с клиентом с помощью других средств.

Неполная информация

[UC15.18.1 Если клиент не заполнил все необходимые реквизиты, система предлагает ему предоставить пропущенную информацию. На экране отображается список пропущенных реквизитов. Прецедент продолжается.]

Отмена

[UC15.11.1 Если клиент щелкнул на кнопке Cancel, заполненные поля формы очищаются. Прецедент продолжается.]

3. Предусловия

3.1. Клиент указывает браузеру на Web-страницу системы. Страница отображает детальную информацию о конфигурации компьютера вместе с ее ценой. Клиент щелкает на кнопке Purchase.

3.2. Клиент щелкает на кнопке Purchase не позднее 15 минут после момента запроса на создание и отображение на странице браузера последней конфигурации компьютера.

4. Постусловия

4.1. Если отправка заказа клиента прошла успешно, заказ на покупку записывается в базу данных. В противном случае состояние системы не изменяется.

10.7.2. Создание прототипов пользовательского интерфейса

Этап 23. Интернет-магазин

На практике проектные спецификации прецедента использования на том уровне детализации, который представлен в разделе 10.7.1, еще не могут служить основой для создания проекта кооперации объектов, допускающего программирование. Для того чтобы снабдить программистов дополнительной информацией, существуют два основных метода. Оба они связаны с *раскадровками* (storyboarding). Первый метод напоминает способ, применяемый в кинематографе, — раскадровки UX (см. раздел 7.4.1 главы 7 и раздел 10.8).

Цель этого раздела — разработать прототип Web-страницы, выводящей на экран текущий статус заказа.

На рис. 10.25 показан простой проект страницы для вывода статуса заказа. Более сложный проект должен содержать страницу *регистрации клиентов* (client logon). Страница для регистрации клиентов должна предлагать пользователю ввести свой учетный номер и номер заказа (посланный клиенту при регистрации заказа). После этого система должна провести авторизацию клиента, прежде чем выводить статус его заказа на экран.



Рис. 10.25. Прототип Web-страницы Вывод статуса заказа *в системе управления Интернет-магазином*

Проект, представленный на рис. 10.25, демонстрирует (в строке адреса URL), что статус заказа реализован как *активная серверная страница* (active server page — ASP). Эта схема содержит пять информационных полей. Серый фон этих полей свидетельствует о том, что пользователь не может модифицировать значения, выводящиеся в этих полях (т.е. эти поля предназначены только для чтения). Гиперссылка внизу экрана позволяет пользователю уйти со страницы, на которой выводится статус заказа.

10.7.3. Диаграмма последовательностей

Этап 24. Интернет-магазин

Обратитесь к этапу 23. Постройте диаграмму последовательностей для прецедента `Вывод статуса заказа`.

Поищите в проекте шаблоны управления персистентными объектами (см. разделы 8.4 и "Решения упражнений. Управление взаимоотношениями с заказчиками" в конце главы 8). Приведите проект в соответствие с диаграммой классов, представленной на рис. 10.11, и диаграммой пакетов, продемонстрированной на рис. 10.20. Для простоты можете проигнорировать объекты компонентов.

Предположим, что преобразователь данных знает, что в кэш-памяти нет "чистой" информации о статусе заказа и за ней необходимо обратиться к базе данных. Как только информация о статусе заказа будет получена из базы данных, кэш-память следует обновить. Предположим также, что обновление кэш-памяти означает обновление объекта класса `EOrder` и создание нового объекта класса `EOrderStatus`.

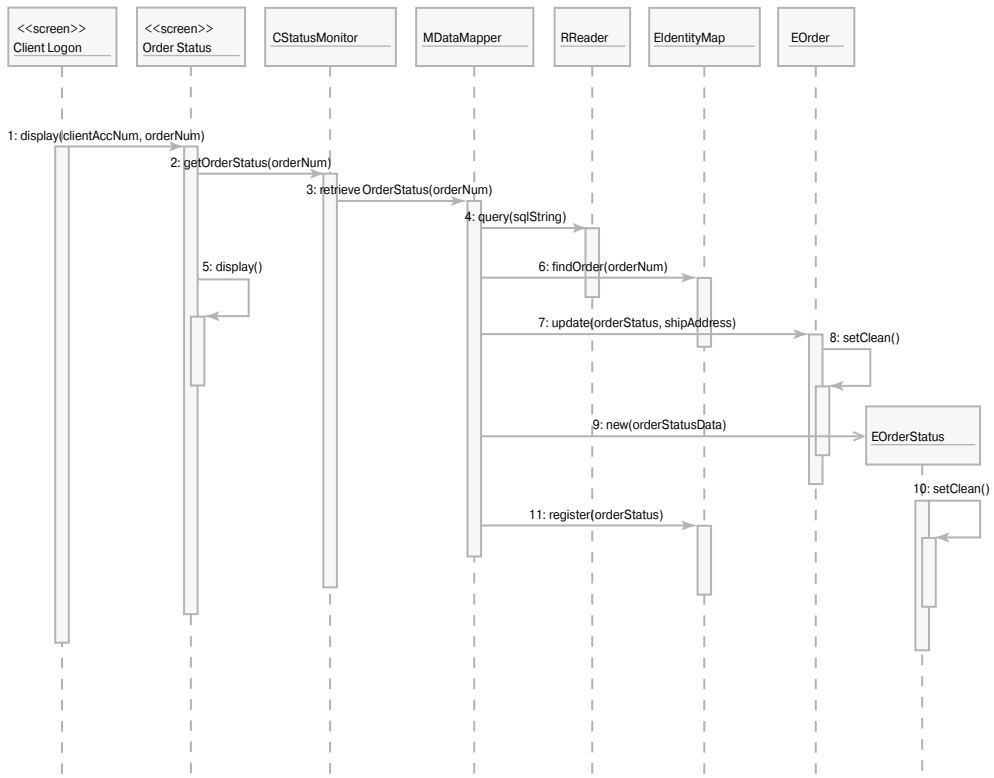


Рис. 10.26. Диаграмма последовательностей для прецедента Вывод статуса заказа *в системе управления Интернет-магазином*

Решение задачи, поставленной на этапе 24, представлено на рис. 10.26. Обратите внимание на то, что объект класса `MDataMapper` непосредственно обращается к базе данных и возвращает данные о статусе заказа объекту `Order Status` через класс `CStatusMonitor` ("экраны" можно было бы моделировать как объекты представления, — например, в виде класса `POrderStatus`, — но мы предпочли использовать систему обозначения раскадровки UX. Для обновления данных экран `Order Screen` использует сообщение `display()`. Одновременно класс `MDataMapper` обновляет объекты сущностей в кэш-памяти. Сначала

он обновляет объект класса `EOrder`, затем создает новый объект класса `EOrderStatus`. Эта модель не объясняет, что следует сделать, чтобы связать объект класса `EOrderStatus` с объектом класса `EOrder`.

10.7.4. Проектная диаграмма классов

> **Этап 25. Интернет-магазин**
>
> Обратитесь к этапу 24. Постройте диаграмму классов для прецедента `Вывод статуса заказа`. Включать данные-члены во все классы, кроме классов сущностей, не обязательно. Достаточно ограничиться методами. Покажите типы значений, возвращаемых методами. Следует также определить отношения между классами, включая отношения зависимости.

Проектная модель классов, показанная на рис. 10.27, является непосредственным следствием модели, приведенной на рис. 10.26. Основным дополнением являются типы значений, возвращаемых методами, и типы данных атрибутов. На диаграмме использованы типы, предусмотренные в языке Java. Атрибут `orderStatus` в классе `Order` (см. рис. 10.11) заменен ассоциацией с классом `EOrderStatus`.

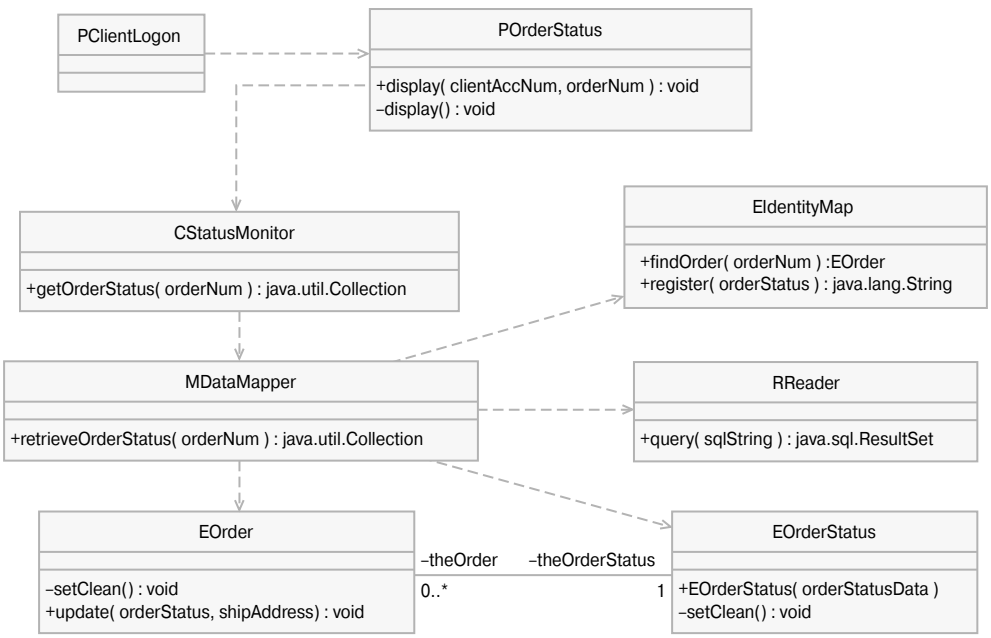


Рис. 10.27. Проектная диаграмма классов для прецедента Вывод статуса заказа *в системе управления Интернет-магазином*

10.8. Проектирование навигации по окнам

10.8.1. Использование элементов UX

Этап 26. Интернет-магазин

Обратитесь в этапу 22. Изучите проектные спецификации прецедента использования и выявите элементы UX. Разработайте стереотипные классы UX для прецедента использования `Заказ конфигурации`.

Диаграмма классов, демонстрирующая элементы UX и отношения между ними, показана на рис. 10.28. Окно Computer Order (Заказ компьютера) содержит одну форму Order Entry (Ввод заказа), две кнопки, Submit (Сделать заказ) и Cancel (Отмена), и, возможно, экран Order Confirmation (Подтверждение заказа). Форма Incomplete Order Entry (Неполные данные) представляет собой разновидность формы Order Entry. Компонент Database Exception моделируется как раздел данных, чтобы показать, что он может использоваться многими экранами.

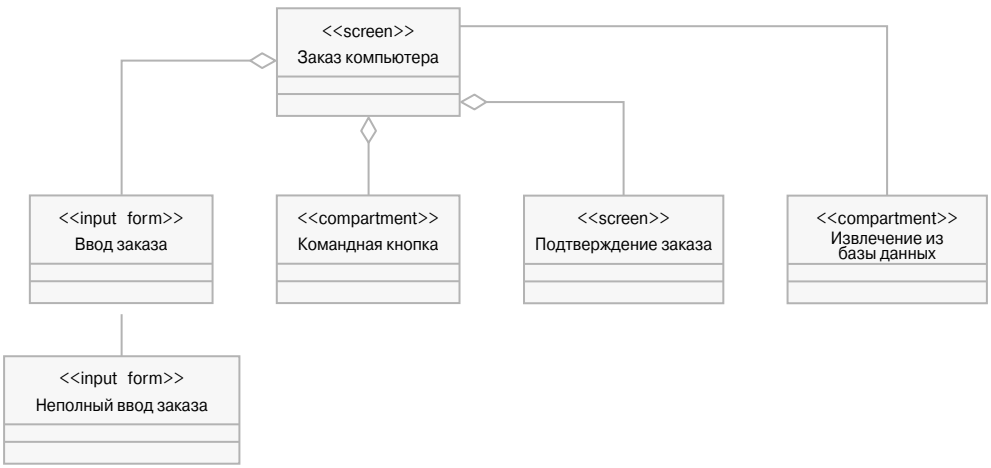


Рис. 10.28. Элементы UX для прецедента Вывод статуса заказа *в системе управления Интернет-магазином*

10.8.2. Функциональная кооперация UX

Этап 27. Интернет-магазин

Обратитесь в этапам 22 и 26. Постройте диаграмму последовательностей для функциональной кооперации UX в прецеденте использования `Заказ конфигурации`. Не следует чрезмерно сосредоточивать внимание на отдельных полях в бланке заказа. Рассматривайте их как группы элементов в спецификациях прецедента использования, т.е. как информацию о поставке, контактную информацию, адрес для пересылки счета-фактуры, способ оплаты и фамилию продавца.

Диаграмма последовательностей, являющаяся решением задачи, поставленной на этапе 27, приведена на рис. 10.29.

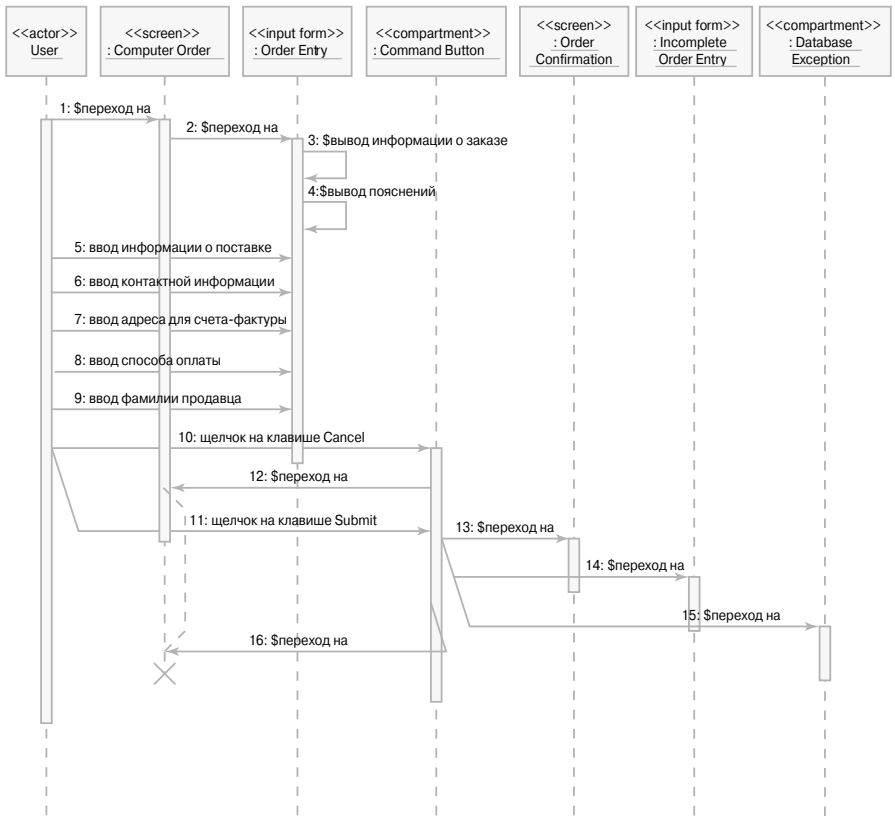


Рис. 10.29. Диаграмма последовательностей для функциональной кооперации UX в прецеденте Вывод статуса заказа *в системе управления Интернет-магазином*

10.8.3. Структурная кооперация UX

Этап 28. Интернет-магазин

Обратитесь в этапам 22, 26 и 27. Постройте диаграмму классов для структурной кооперации UX в прецеденте использования `Заказ конфигурации`. Можно не применять дескрипторы UX к динамическому содержанию UX-элементов (полей в UX-классах).

Решение задачи, поставленной на этапе 28, приведено на рис. 10.30.

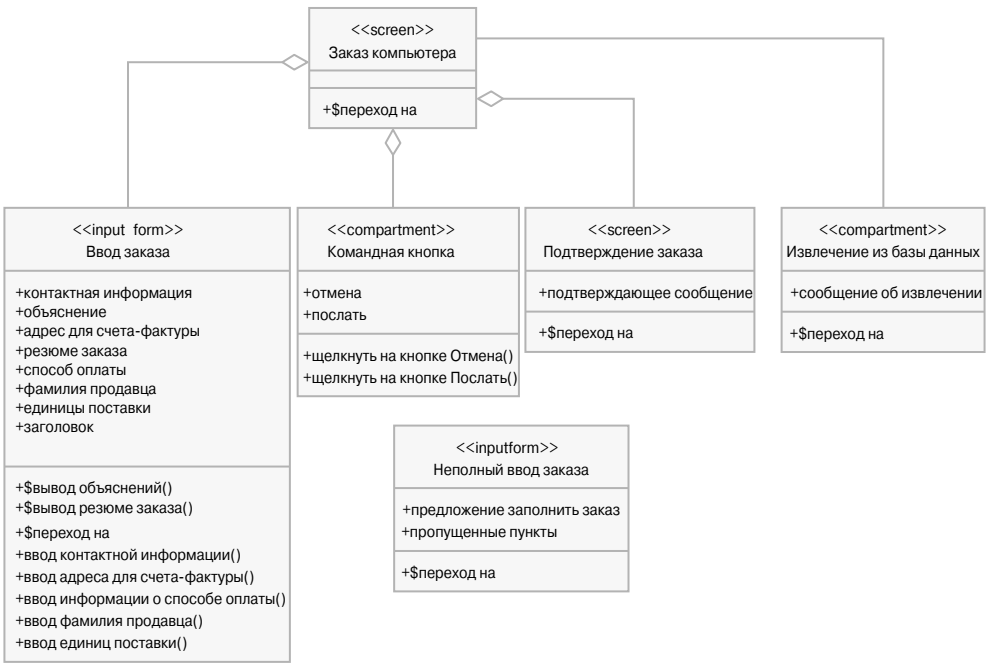


Рис. 10.30. Диаграмма классов для структурной кооперации UX в прецеденте Вывод статуса заказа *в системе управления Интернет-магазином*

10.9. Проектирование баз данных

10.9.1. Объектно-реляционное отображение

Этап 29. Интернет-магазин

Обратитесь к этапу 12. Отобразите диаграмму классов, изображенную на рис. 10.11, в схематическую модель базы данных. Оцените необходимость таблицы, содержащей информацию о статусе заказа, в соответствии с требованиями, выявленными на этапе 25. Создайте таблицы, отношения между ними, типы столбцов и нулевые индикаторы. Кроме того, укажите кратность отношений.

Отображение порождает 12 таблиц, показанных на рис. 10.31. Большинство отображений носят рутинный характер и следуют рекомендациям, определенным в разделе 8.3 (см. главу 8). Обобщение на рис. 10.11 отображено с помощью третьей стратегии, описанной в разделе 8.3.4, т.е. каждый конкретный класс отображен в таблицу.

На рис. 10.31 ассоциация между классами `Order` и `Computer` имеет тип "множество–множество", так что класс `Order` должен быть ассоциирован хотя бы с одним компьютером. Отображение, показанное на рис. 10.31, использует две "таблицы отношений", чтобы преобразовать ассоциацию "множество–множество" в два отношения "один–множество" (отдельно для связывания с классами `StandardComputer` и `ConfiguredComputer` соответственно). В этом отображении ограничение, требующее, чтобы класс `Order` был ассоциирован хотя бы с одним компьютером, в модели базы данных не поддерживается. Примечание на диаграмме означает, что это ограничение должно быть закреплено процедурно (т.е. с помощью триггера).

Удивительно, но очень похожая модель получается для агрегации класса `Computer` с классом `ConfigurationItem` (см. рис. 10.11). Модель классов не показывает, что объект класса `ConfigurationItem` может быть компонентом нескольких объектов класса `Computer` (поскольку класс `ConfigurationItem` определяет тип детали, а не ее конкретный экземпляр). Соответственно, агрегация на самом деле является отношением "множество–множество", как показано на рис. 10.31.

Другое неочевидное решение связано с отношением между таблицами `Order` и `Invoice`. Это отношение можно моделировать с помощью замены внешнего ключа в одной из этих двух таблиц. Однако следует отметить, что внешний ключ (`invoice_number`) допускает нулевые поля, поскольку отношение между таблицами `Order` и `Invoice` имеет кратность `0..1` (см. рис. 10.11).

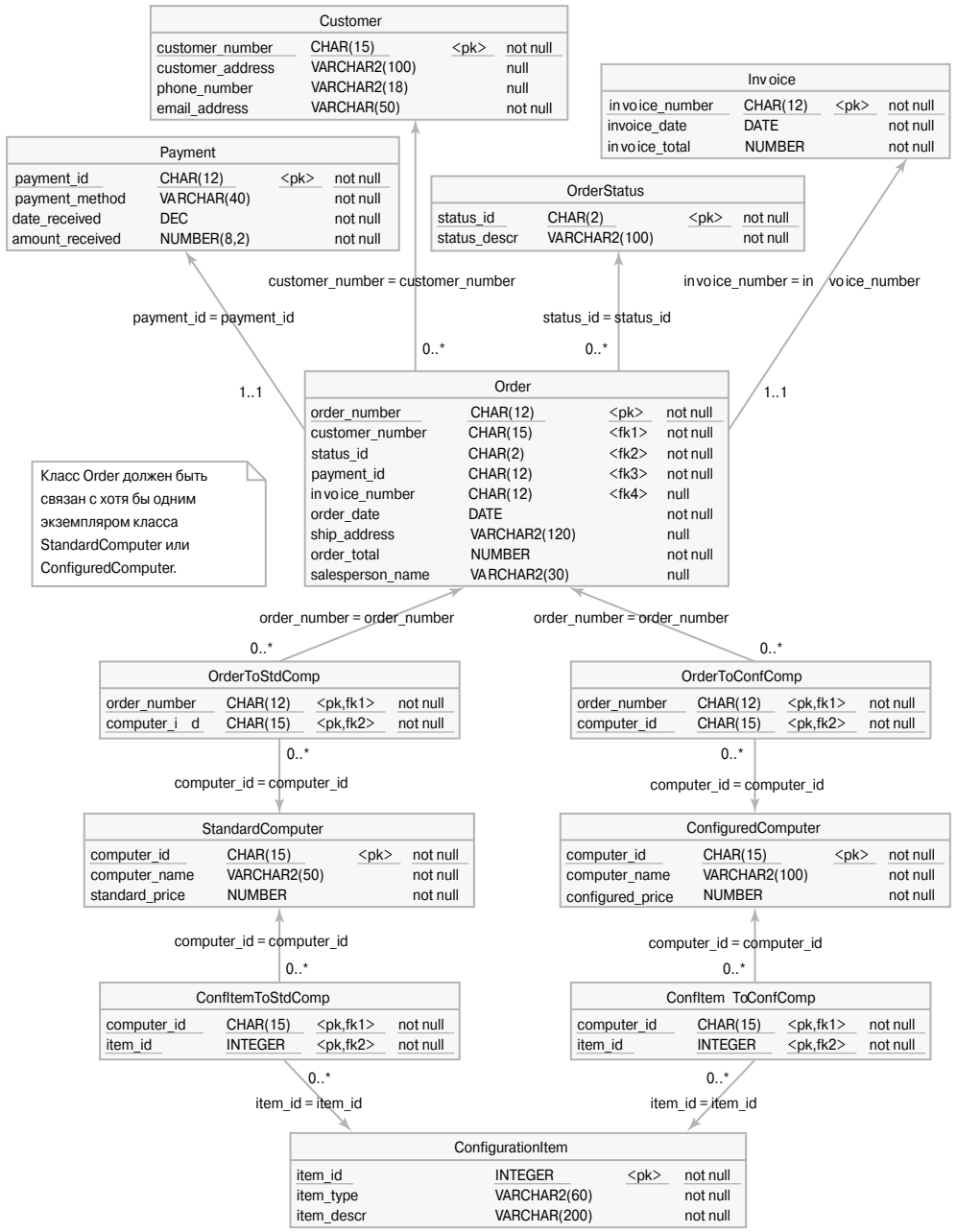


Рис. 10.31. Схематическая модель базы данных для системы управления Интернет-магазином

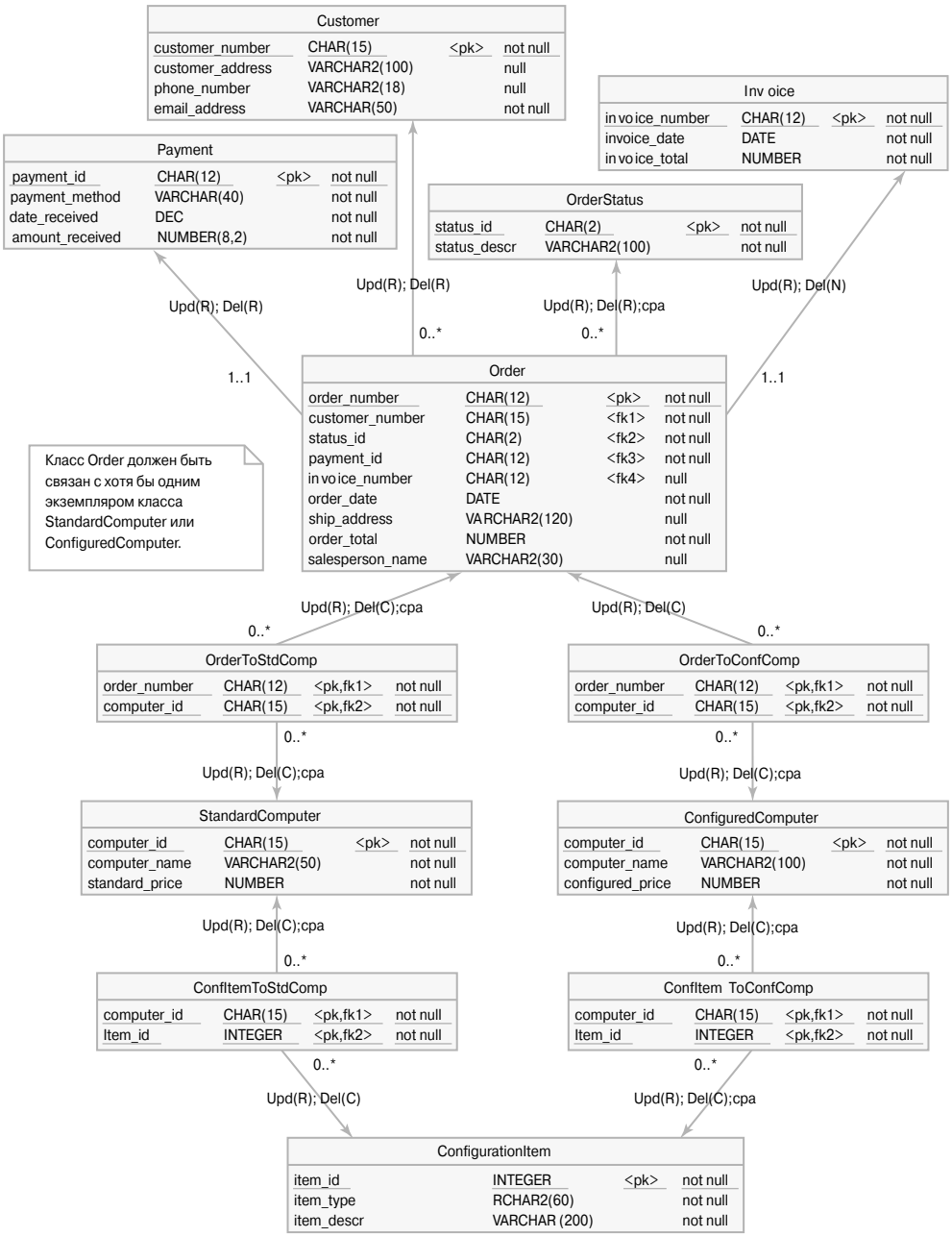


Рис. 10.32. Схематическая модель базы данных, учитывающая требование ссылочной целостности системы управления Интернет-магазином

10.9.2. Проектирование ссылочной целостности

Этап 30. Интернет-магазин

Обратитесь к этапу 29 и схематической модели базы данных, показанной на рис. 10.31. Рассмотрите разные декларативные ограничения ссылочной целостности, наложенные на операции удаления (см. раздел 8.2.3 главы 8). Покажите, какие отношения на рис. 10.31 должны иметь ограничения `Del(C)` или `Del(N)`, а не `Del(R)`. Покажите также на диаграмме, какие отношения допускают "изменение родительских записей" (т.е. разрешают ограничение `cpa` (change parent allowed)).

На рис. 10.32 приведена схематическая модель базы данных, показывающая разрешенные действия, связанные с удалением, и допустимость ограничения *cpa*. Ограничение *cpa* разрешено для отношения между классами `Order` и `Order-Status` (заказ может изменять атрибут `status_id`). Ограничение *cpa* разрешено также для большинства отношений, связывающих "таблицы отношений" в нижней части диаграммы. Существуют два исключения: выбранная конфигурация компьютера не должна изменять заказ и стандартный компьютер не должен изменять элемент конфигурации.

Удаление счета-фактуры обнуляет внешний ключ в таблице `Order`. Записи в "таблицах отношений" можно удалять свободно, если была удалена родительская запись. Этот факт отмечен записью ограничения `Del(C)`, наложенного на эти таблицы. Все остальные ограничения удаления имеют тип `Del(R)`.

Резюме

Последняя глава книги посвящена важному (если не самому важному) аспекту обучения — систематизации и закреплению изученного материала. Этот принцип относится к большинству университетских курсов — последнюю неделю обучения часто посвящают именно систематизации и закреплению пройденного материала. Для таких курсов эта глава окажется очень полезной.

В главе последовательно применяется принцип систематизации и закрепления материала. Для иллюстрации всех важных этапов анализа и проектирования используется одна предметная область — описание работы Интернет-магазина. В совокупности представленный материал описывает тридцать взаимозависимых этапов разработки программного обеспечения. Эти этапы можно сгруппировать в девять последовательных тем.

1. Моделирование прецедентов использования.
2. Моделирование деятельности.
3. Моделирование классов.
4. Моделирование взаимодействий.
5. Моделирование конечных автоматов.
6. Модели реализации.
7. Проектирование кооперации.
8. Проектирование навигации по окнам.
9. Проектирование базы данных.

Эта глава ни в коей мере не может заменить собой остальную часть книги. Она не содержит теоретических сведений (за редким исключением). Кроме того, учебный пример, описанный в одной главе, не может продемонстрировать все аспекты моделирования и проектирования. Он позволяет лишь проиллюстрировать наиболее важные вопросы анализа и проектирования.

Упражнения. Интернет-магазин

A1. Обратитесь к этапу 2 (см. раздел 10.1.2).

В пункте 6 табл. 10.1 утверждается, что клиент получит письмо по электронной почте и проверит статус заказа, однако нет ни одного прецедента использования, который продемонстрировал бы это действие. Следует ли это делать? Обоснуйте свой ответ.

A2. На этапе 3 (см. раздел 10.1.3) идентифицирован прецедент использования `Вывод статуса заказа`, позволяющий клиенту проверять статус своего заказа.

Напишите спецификацию прецедента использования `Вывод статуса заказа`, используя формат документа, показанный на этапе 4 (см. раздел 10.1.4).

A3. Вернитесь к этапу 9 (см. раздел 10.3.5).

На рис. 10.8 класс `Customer` не имеет прямой ассоциации с классами `Payment`, `Invoice` и `ConfigureComputer`. Нужны ли эти ассоциации? Если да, модифицируйте диаграмму. Обоснуйте свой ответ.

A4. Вернитесь к этапу 11 (см. раздел 10.3.5).

На рис. 10.10 показан относительно простой пример обобщения. Какие сложности могут возникнуть, если существует разница между информацией в заказе стандартного компьютера и в заказе специальной конфигурации?

Например, в зависимости от изменений, которые необходимо учесть в системах `ConfiguredComputer`, клиент должен внести дополнительную плату или может получить скидку за оптовую покупку систем `Standard-Computer`.

Модифицируйте диаграмму, чтобы отразить эти изменения. Кратко объясните свой ответ.

A5. Вернитесь к этапу 6 (см. раздел 10.2.2).

Постройте аналитическую диаграмму последовательностей для действия `Вывести на экран бланк` (см. рис. 10.5). Рассмотрите в качестве примера рис. 10.14 (см. этап 13, раздел 10.4.1).

A6. Проанализируйте решение упражнения A4.

Добавьте в классы операции, обозначающие объекты на диаграммах последовательностей. Покажите отношения, включая зависимости между классами.

A7. Вернитесь к этапу 16 (см. раздел 10.5.1).

Диаграмма конечного автомата, показанная на рис. 10.17, соответствует ограничению, допускающему только одну частичную оплату. Предположим, что это не так и что клиент имеет право сделать несколько частичных платежей. Модифицируйте диаграмму соответствующим образом.

Предложите два решения. Первое решение должно предусматривать ситуацию, в которой частичная оплата заранее обозначена как частичная. Второе решение должно учитывать ситуации, в которых система сама определяет, какой является оплата: частичной или полной.

A8. Вернитесь к этапу 12 (см. раздел 10.3.6).

Рассмотрите часть модели с классами `Customer`, `Order` и `Invoice`. Можно ли ввести производные ассоциации в эту модель? Если да, добавьте их в диаграмму.

A9. Вернитесь к этапу 12 (см. раздел 10.3.6).

Рассмотрите часть модели с классами `Order` и *`Computer`*. Измените ассоциацию между этими классами, сделав ее ограниченной, чтобы явно учесть ограничение "один заказ на один компьютер".

A10. Вернитесь к этапу 5 (см. раздел 10.2.1).

Постройте диаграмму последовательностей для действия `Послать описание заказа по электронной почте`, руководствуясь подходом, описанным на этапе 24 (см. раздел 10.7.3).

A11. Вернитесь к этапу 5 (см. раздел 10.2.1).

Постройте проектную диаграмму классов для действия `Послать описание заказа по электронной почте`, руководствуясь подходом, описанным на этапе 25 (см. раздел 10.7.4).

A12. Вернитесь к этапу 3 (см. раздел 10.1.3).

Разработайте проектную диаграмму последовательностей для действия `Проверить и принять платеж клиента`, руководствуясь подходом, описанным на этапе 22 (см. раздел 10.7.1).

A13. Проанализируйте решение упражнения A12.

Исследуйте проектные спецификации прецедентов использования для идентификации UX-элементов. Разработайте соответствующие стереотипизированные UX-классы для прецедента использования `Проверить и принять платеж клиента`, руководствуясь подходом, описанным на этапе 26 (см. раздел 10.8.1).

A14. Проанализируйте решения упражнений A12 и A13.

Постройте диаграмму последовательностей для функциональной кооперации UX для прецедента использования `Проверить и принять платеж клиента`, руководствуясь подходом, описанным на этапе 27 (см. раздел 10.8.2).

A15. Проанализируйте решения упражнений A12–A14.

Постройте диаграмму классов для структурной кооперации UX для прецедента использования `Проверить и принять платеж клиента`, руководствуясь подходом, описанным на этапе 28 (см. раздел 10.8.3).

A16. Вернитесь к этапу 29 (см. раздел 10.9.1).

Разработайте схематическую модель базы данных, альтернативную модели, показанной на рис. 10.31. Попытайтесь создать модель, как можно сильнее отличающуюся от своего аналога, сохранив ее декларативную семантику и эффективность.

A17. Вернитесь к этапу 30 (см. раздел 10.9.2).

Рассмотрите отношение между таблицами `Invoice` и `Order`. Напишите триггеры базы данных (можно в виде псевдокода) для этих двух таблиц, гарантирующие ссылочную целостность между ними, как показано на рис. 10.32.

ПРИЛОЖЕНИЕ

A Основы объектной технологии

Цели

A.1. Аналогия с объектами реального мира
A.2. Объект-экземпляр
A.3. Класс
A.4. Переменные, методы и конструкторы
A.5. Ассоциации
A.6. Агрегация и композиция
A.7. Обобщение и наследование
A.8. Абстрактный класс
A.9. Интерфейс
Резюме
Вопросы
Ответы на вопросы с нечетными номерами

Практически все современные системы программного обеспечения являются объектно-ориентированными и разрабатывались с помощью объектно-ориентированного моделирования. Повсеместное использование объектов в информационных системах требует знаний объектной технологии практически от всех участников проекта — не только разработчиков, но и заказчиков (пользователей и владельцев систем). Для того чтобы правильно понимать друг друга, все участники проекта должны иметь общие представления об объектной технологии и знать язык объектного моделирования. Заказчикам достаточно знать хотя бы основные концепции и конструкции, применяемые в моделировании. Разработчики должны знать объектную технологию намного глубже, на уровне, допускающем создание моделей и их реализацию в виде программного обеспечения.

Основная трудность изучения *объектной технологии* связана с отсутствием очевидной отправной точки и ясного направления исследования. Их невозможно описать с помощью исследования сверху вниз или снизу вверх. При изучении объектной технологии мы вынуждены постоянно отправляться "с середины". Независимо от того, насколько далеко мы продвинулись в процессе изучения, постоянно кажется, что мы находимся где-то посередине (поскольку все время возникают новые вопросы). Первый большой успех в процессе изучения будет достигнут, когда читатель глубоко осознает, что в объектно-ориентированной системе все является объектом.

Концепцию объектной технологии лучше всего объяснять с помощью визуальных презентаций конструкция языка UML. По этой причине в приложении этот язык используется для описания всех основных концепций объектной технологии.

А.1. Аналогия с объектами реального мира

Для того чтобы объяснить объектную ориентацию информационных систем, лучше всего провести аналогию с объектами реального мира. Окружающий нас мир состоит из *объектов*, пребывающих в неких наблюдаемых *состояниях* (определенных текущими значениями атрибутов этих объектов) и демонстрирующих некое *поведение* (определенное операциями (функциями), выполняемыми над этими объектами). Каждый объект однозначно *идентифицируется* среди других объектов.

Например, чашка на столе находится в наполненном *состоянии*, поскольку она приспособлена для хранения жидкостей и в ней все еще есть кофе. Когда в ней не останется кофе, ее состояние станет пустым. Если же чашка упадет на пол и разобьется, то она перейдет в разбитое состояние.

Кофейная чашка, разумеется, пассивна — она не обладает собственным *поведением*. Однако этого нельзя сказать о собаке или эвкалипте. Собака лает, дерево растет и т.д. Итак, некоторые объекты реального мира обладают поведением.

Кроме того, все объекты реального мира обладают *индивидуальностью* — фиксированным свойством, позволяющим отличать один объект от другого. Если на столе стоят две чашки из одного набора, то можно сказать, что они *одинаковые*, но не *идентичные*. Чашки одинаковы, потому что пребывают в одном и том же состоянии, т.е. значения их атрибутов совпадают (например, они имеют одинаковый размер и форму, окрашены в черный цвет и пусты). Однако на объектно-ориентированном языке они не идентичны, поскольку их две, и у пользователя есть выбор.

Реальные объекты, обладающие тремя названными свойствами — состоянием, поведением, идентичностью, — образуют *системы с естественным поведением*. Естественные системы безусловно являются самыми *сложными системами* из всех известных. Ни одна компьютерная система не может сравниться по сложности с животным или заводом.

Несмотря на сложность, естественные системы способны очень хорошо функционировать: они демонстрируют интересное поведение, могут приспосабливаться к внешним и внутренним изменениям, могут эволюционировать со временем и т.д. Вывод очевиден. Наверное, следует конструировать искусственные системы с помощью моделирования структуры и поведения естественных систем (Maciaszek et al., 1996b).

Искусственные системы являются моделями реальности. Кофейная чашка на экране компьютера — всего лишь модель реальной "сущности", так же как собака или эвкалипт на дисплее. Следовательно, модель кофейной чашки может обладать собственным поведением. Она может, например, упасть на пол, если ее уронить. Падение можно моделировать как поведенческую *операцию* чашки. Еще одним "действием" чашки может быть разбивание при ударе об пол. Большинство, если не все, объекты в компьютерной системе "оживают" — они обладают поведением.

А.2. Объект-экземпляр

Объект — это экземпляр некоей сущности. Он может быть одним из множества экземпляров одной и той же сущности. Например, конкретная чашка — это экземпляр множества всевозможных чашек.

Общее описание сущности называется *классом*. Следовательно, объект является экземпляром класса. Однако, как будет показано далее, класс также может иметь конкретное воплощение — он может быть объектом. Поэтому нам необходимо различать *объект-экземпляр* (instance object) и *объект-класс* (class object).

Для краткости объект-экземпляр часто называют просто *объектом* или *экземпляром*. Называть его "экземпляром объекта" было бы неверно. По тем же причинам неправильно говорить "объект класса". Да, класс представляет собой шаблон для объектов с одинаковыми атрибутами и операциями, но сам класс также может быть реализован как объект (было бы странно называть такую операцию созданием "объекта класса объектов").

А.2.1. Объектная нотация

В языке UML объект изображается в виде прямоугольника с двумя отделениями. Верхнее отделение содержит имя объекта и имя класса, которому принадлежит объект.

Синтаксис этой конструкции выглядит так:

objectname: classname.

Нижнее отделение содержит список имен атрибутов и значений. С помощью этого синтаксиса можно также показать типы атрибутов:

attributename: [type] = value.

На рис. А.1 продемонстрированы четыре разных способа графического изображения объекта. На нем изображен объект `Course` с именем `c1`. Этот объект имеет два атрибута. Типы атрибутов не указаны — они заданы в определении класса. Отделения для значений атрибутов можно не указывать, если они не имеют значения для данного аспекта моделирования. Аналогично, демонстрируя анонимные объекты данного класса, можно не указывать их имена. В заключение отметим, что имя класса объекта также можно не указывать. Наличие или отсутствие двоеточия означает, относится ли данная метка к классу или объекту.

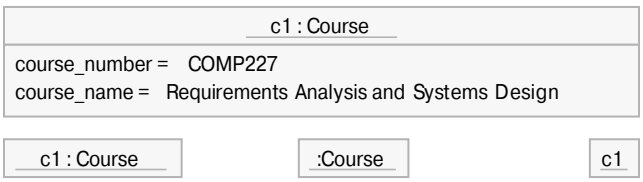


Рис. А.1. Объект-экземпляр

Следует иметь в виду, что объектная нотация не предусматривает в обозначении объекта особого отделения для перечня *операций*, которые может выполнять объект-экземпляр. Это объясняется тем, что операции, выполняемые всеми объектами-экземплярами, идентичны, и поэтому хранить их в каждом объекте-экземпляре было бы нецелесообразно. Операции можно либо хранить в *объекте-классе*, либо связать их с объектами-экземплярами другими средствами (реализованными в объектно-ориентированной системе программного обеспечения).

Кстати, существуют менее известные языки программирования, такие как Self, позволяющие связывать операции с объектами (но не классами) в ходе выполнения программы. Эти языки называют *прототипными* (prototipical) или *основанными на делегировании* (delegation-based language). Для этих ситуаций язык UML допускает использование третьего отделения, содержащего операции в качестве специфического расширения.

А.2.2. Как кооперируются объекты

Количество объектов определенного класса может быть очень большим. Это особенно справедливо для *бизнес-объектов*, представляющих концепции бизнеса и известных как *классы сущностей*, такие как `Invoice` (Счет-фактура) или `Employee` (Сотрудник). Показать большое количество объектов на диаграмме нереально, а иногда просто невозможно.

Объекты изображаются только для того, чтобы привести пример системы в определенный момент времени или проиллюстрировать, каким образом они *кооперируются* с течением времени при выполнении определенных задач. Например, чтобы заказать товар, необходимо установить *кооперацию* между объектами `Stock` (Склад) и `Purchase` (Покупка). Точнее говоря, объекты на диаграмме

кооперации — это *роли*, которые играют объекты, а не объекты сами по себе. Эти роли описывают многие возможные объекты. Графически роли изображаются с помощью обозначения анонимного объекта.

Системные задачи выполняются объектами, которые активизируют *операции* (поведение) друг друга. Мы говорим, что они обмениваются *сообщениями*. Сообщения запускают операции на объектах, которые могут приводить к изменению состояний объектов и вызывать другие операции.

На рис. А.2 показан поток сообщений между четырьмя объектами. Скобки после имени сообщения указывают на то, что сообщение может принимать параметры (аналогично вызову функции в традиционном программировании). Объект `Order` (Заказ) предписывает объекту `Shipment` (Поставка) доставить заказ (для этого объект `Order` передает себя объекту `Shipment` в качестве фактического параметра сообщения `shipOrder()`). Объект `Shipment` отдает объекту `Stock` команду поставить соответствующее количество товаров, посылая сообщение `getProducts()`. Объект `Stock` анализирует объем запасов, выполняя операцию `analyzeLevels()`. Если запас оказывается ниже определенного уровня, объект `Stock` предписывает объекту `Purchase` сделать повторный заказ дополнительного объема товаров с помощью сообщения `reorder()`.



Рис. А.2. Кооперация объектов

Несмотря на то что приведенное объяснение кооперации объектов выглядит как последовательность пронумерованных сообщений, в общем случае на порядок вызовов объектов не накладывается строгих ограничений. Например, сообщение `analyzeStockLevels()` можно активизировать независимо от сообщений `shipOrder()` (доставить заказ) и `getProduct()` (доставить товар). По этой причине в моделях кооперации объектов сообщения обычно не нумеруются.

А.2.3. Индивидуальность и коммуникация между объектами

Каким образом объект распознает *индивидуальность* другого объекта, которому требуется отправить сообщение? Каким образом объект `Order` знает свой объект `Shipment` так, чтобы сообщение `shipOrder()` попало к своему адресату?

Ответ заключается в том, что каждому объекту при создании присваивается *идентификатор объекта* (object identifier — OID). Идентификатор объекта представляет собой *дескриптор* (handle) объекта — уникальный номер, который остается с объектом на протяжении всего времени его существования. Если объекту `X` необходимо отправить сообщение объекту `Y`, то объект `X` каким-либо образом должен узнать OID объекта `Y`. На практике для установления *связей* с помощью идентификаторов существуют два подхода, каждому из которых соответствует определенный тип связи.

- Постоянная связь по OID.
- Временная связь по OID.

Различие между этими видами связи определяется продолжительностью существования объекта. Время жизни некоторых объектов не превышает времени выполнения программы — они создаются программой и уничтожаются во время выполнения программы или по ее завершении. Это так называемые *временные объекты* (transient object). Другие объекты продолжают существовать после завершения программы — они запоминаются в долговременной дисковой памяти и доступны при следующем выполнении программы. Такие объекты называются *персистентными* (persistent object).

А.2.3.1. Персистентная связь

Персистентная связь (persistent link) — это ссылка (или множество ссылок), с помощью которой персистентный объект устанавливает связь с другим персистентным объектом (или со множеством других объектов). Следовательно, для установления персистентной связи объекта `Course` (Курс) с его объектом `Teacher` (Преподаватель) объект `Course` должен содержать атрибут связи, значение которого равно OID объекта `Teacher`. Описанная связь постоянна, поскольку OID физически хранится в объекте `Course`, как показано на рис. А.3.

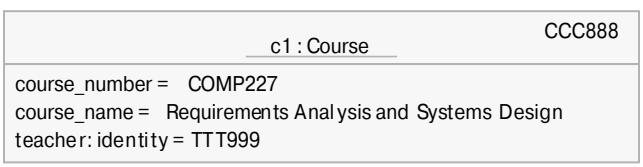


Рис. А.3. Представление персистентной связи

Идентификатор объекта c1 помечен здесь как CCC888. Объект содержит атрибут связи teacher. Данный атрибут имеет тип identity. Его значение равно идентификатору объекта Teacher, т.е. TTT999, и представляет собой логический адрес этого объекта. Его можно реализовать как комбинацию идентификационного номера компьютера и времени, прошедшего с момента активизации объекта (выраженного в миллисекундах). Если объект Teacher является персистентным, то среда языка программирования может преобразовать логический адрес в адрес физической памяти на диске.

Как только объекты Course и Teacher перемещаются в память программы, значение атрибута teacher будет *настроено по адресу* (swizzled) с помощью указателя памяти, устанавливая кооперацию между объектами на уровне памяти. (Термин *swizzling* — это не термин языка UML, он используется в описаниях объектных баз данных, где перемещение объектов между долговременной и оперативной памятью довольно частое явление.)

На рис. A.3 показано, как персистентные связи представляются в объектах. В языке UML связи между объектами можно изображать так, как показано на рис. A.4. Связи представляются как *экземпляры ассоциации* между объектами Course и Teacher.

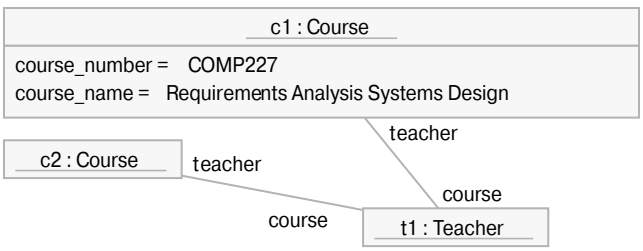


Рис. A.4. Представление персистентной связи в объектной модели UML

Обычно кооперативные связи допускают *навигацию* в обоих направлениях. Каждый объект Course связан со своим объектом Teacher, а объект Teacher может привести к объектам Course. Изредка допускается навигация только в одном направлении.

A.2.3.2. Временная связь

Что если между объектами Course и Teacher не определены персистентные связи, но нам по-прежнему требуется отправить сообщение от объекта t1 к объекту c1 для вызова операции getCourseName()? Прикладная программа должна обладать другими средствами для установления идентичности объекта c1 и создать *временную связь* от объекта t1 к объекту c1.

К счастью, в распоряжении программиста имеется много средств, способных обеспечить инициализацию переменной crsRef с помощью OID объекта c1, по-

стоянно находящегося в памяти. Для начала заметим, что, возможно, связь между объектами `c1` и `t1` была установлена в программе раньше и переменная `crsRef` хранит правильный OID. Например, программа выполнила операцию поиска с использованием атрибута занятости преподавателя `t1` и расписания курсов и определила, что преподаватель `t1` должен вести курс `c1`.

Альтернативная возможность состоит в том, что программа имеет доступ к постоянно хранимой таблице, которая отображает номера курсов на имена преподавателей. Затем программа может выполнить поиск на объектах `Course`, чтобы отыскать все курсы, читаемые преподавателем `t1`, и попросить пользователя определить курс (т.е. объект `Course`), которому должно быть отправлено сообщение `getCourseName()`.

Возможно также, что задача программы как раз и состоит в создании объектов для курсов и преподавателей перед тем, как занести их в базу данных. Между соответствующими объектами нет постоянных связей, но пользователь вводит информацию таким образом, что каждый курс четко определяет ответственного за него преподавателя.

Затем программа может запомнить временную связь в программной переменной (такой, как `crsRef`), а эту переменную можно позднее использовать (во время выполнения этой же программы) для отправки сообщений между объектами `Teacher` и `Course`.

Короче говоря, существуют как программные, так и управляемые пользователем методы установления временных связей между объектами, которые не связаны постоянно с помощью ассоциации между соответствующими классами. *Временные связи* — это программные переменные, которые содержат значения OID объектов, находящихся в текущий момент в оперативной памяти. Отображение (*настройка по адресам*) между временными и персистентными идентификаторами объектов должно быть прерогативой базовой программной среды, например, такой, как система управления объектной базой данных.

A.2.3.3. Передача сообщений

Как только объект связывается с другим объектом, он может послать сообщение по линии связи и запросить у другого объекта *услугу* (service). Иначе говоря, объект может вызвать *операцию* другого объекта, послав ему *сообщение*. В типичном сценарии для ссылки на объект отправитель может использовать переменную, содержащую значение для связи с этим объектом (значение OID). Например, *сообщение*, посланное объектом `Teacher`, чтобы выяснить имя объекта `Course`, может выглядеть следующим образом:

```
crsRef.getCourseName(out crs_name)
```

В этом примере конкретный объект класса `Course`, выполняющий операцию `getCourseName()`, определяется с помощью текущего значения переменной `crsRef`. Выходной (`out`) аргумент `crs_name` — это переменная, подлежащая

инициализации значением, которое возвращается операцией `getCourseName()`, реализованной в классе `Course`.

Этот пример подразумевает, что язык программирования различает аргументы ввода (`in`), вывода (`out`) и ввода-вывода (`in/out`). Популярные объектно-ориентированные языки программирования, такие как Java, не делают таких различий. В языке Java аргументы сообщения, имеющие элементарные типы данных (например, `crs_name`), передаются операциям по значению. *Передача по значению* (pass by value) подразумевает, что существуют реальные аргументы ввода, — операция не может изменять передаваемые ему значения. Это изменение невозможно потому, что операция работает с копией аргумента.

Если аргументы сообщения имеют неэлементарные типы (т.е. аргументы являются ссылками на объекты, определенные пользователем), то передача по значению подразумевает, что операция получает ссылку на аргумент, а не его значение. Поскольку существует только одна ссылка (а не две ее копии), операция может использовать ее для доступа и возможной модификации значений атрибутов в передаваемом объекте. Это фактически исключает надобность в явных аргументах ввода-вывода.

В заключение отметим, что необходимость в наличии явных аргументов вывода в языке Java заменяется типом значения, возвращаемого операцией, вызванной с помощью сообщения. Тип возвращаемого значения может быть как элементарным, так и неэлементарным. Операция может возвращать не более одного значения или вообще ничего не возвращать (в этом случае вместо типа возвращаемого значения указывается тип `void`). Если необходимо вернуть несколько значений, то программист может определить агрегированный объект неэлементарного типа, содержащий все возвращаемые значения.

А.3. Класс

Класс — это дескриптор множества объектов, обладающих одинаковым набором атрибутов и операций. Он служит *шаблоном* для создания объектов. Каждый объект, созданный по шаблону, содержит *значения* атрибута, соответствующие *типу* атрибута, определенному в классе, и может вызвать операции, определенные в классе.

Графически класс представляется в виде прямоугольника с тремя отделениями, разделенными горизонтальными линиями, как показано на рис. А.5. Верхнее отделение содержит имя класса, среднее — объявления всех атрибутов класса, нижнее — определения операций.

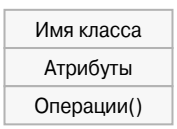


Рис. А.5. Отделение классов

А.3.1. Атрибуты

Атрибут представляет собой пару *тип–значение*. Класс определяет *типы атрибутов*. Объекты содержат *значения атрибутов*. На рис. А.6 показаны два класса с определениями имен и типов принадлежащих им атрибутов. Атрибуты имен в объекте `Course` указаны с помощью обозначений, используемых в теории базы данных. Однако в среде программистов приняты обозначения, показанные в описании объекта `Order`.

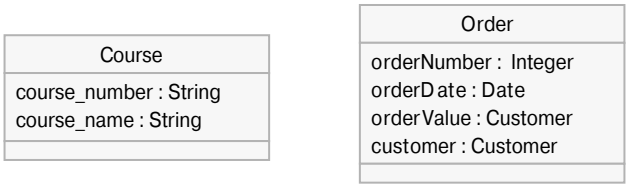


Рис. А.6. Атрибуты

Тип атрибута может быть встроенным *элементарным* или *пользовательским* типом. Элементарный тип непосредственно распознается и поддерживается базовой объектно-ориентированной средой программирования. Все типы атрибутов, показанных на рис. А.6, за исключением типа `customer`, имеют элементарные типы.

А.3.1.1. Тип атрибута, обозначающий класс

Тип атрибута также может обозначать класс. Применительно к конкретному объекту некоторого класса подобный атрибут содержит значение идентификатора объекта (OID), указывающее на объект другого класса. В UML-моделях анализа атрибуты, типы которых обозначают классы, не приводятся в среднем отделении представления класса (в отличие от примитивных типов). Вместо этого они представляются с помощью *ассоциаций* между классами. На рис. А.7 показана *ассоциация* между двумя классами.

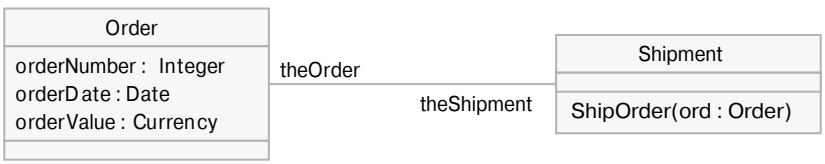


Рис. А.7. Ассоциация между классами с функциональными именами (модель анализа)

Два имени на ассоциативной линии (`theShipment` и `theOrder`) представляют так называемые функциональные имена (конкретная поставка и конкретный заказ). *Функциональное имя* (role name) определяет значение для одного из полюсов ассоциации и используется для *навигации* к объекту другого класса, участвующего в ассоциации.

В реализованной системе функциональное имя (на противоположном полюсе ассоциации) становится атрибутом класса, тип которого является классом, на который указывает функциональное имя. На рис. А.8 показано, как выглядят два класса, продемонстрированные на рис. А.7, после того, как они были реализованы.

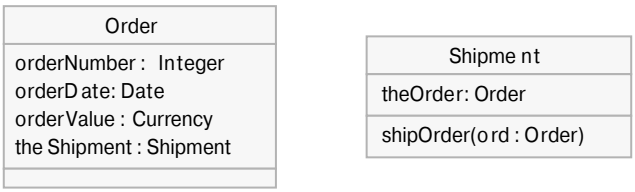


Рис. А.8. Атрибуты, обозначающие классы (модель реализации)

А.3.1.2. Видимость атрибутов

Как отмечалось в разделе А.2.2, объекты взаимодействуют, отправляя друг другу сообщения. Сообщение вызывает операцию класса. Операция обслуживает запрос вызывающего объекта с помощью доступа к значениям атрибутов в своем собственном объекте. Для того чтобы подобный сценарий стал возможен, операции должны быть видимы внешним объектам (сообщения должны видеть операции). Говорят, что подобные операции обладают *открытой видимостью* (public visibility).

В чисто объектно-ориентированной системе (примером которой может служить среда программирования языка Smalltalk) большинство операций относится к *открытым* (public), а большинство атрибутов — к *закрытым* (private). Значения атрибутов скрыты от других классов. Объекты одного класса могут только затребовать услугу (операцию), опубликованную в открытом интерфейсе другого класса. Им не разрешается непосредственно манипулировать атрибутами другого объекта.

Операции *инкапсулируют* (encapsulate) атрибуты. Заметим, однако, что инкапсуляция применима к классам. Один объект не может скрыть (инкапсулировать) ничего от другого объекта того же класса.

Видимость обычно обозначается с помощью знаков “плюс” и “минус”:

+ — для открытой видимости;
– — для закрытой видимости.

В некоторых CASE-средствах эти символы заменены графическими пиктограммами. На рис. А.9 продемонстрированы два графических представления для обозначения видимости атрибутов.



Рис. А.9. Закрытые атрибуты и открытые операции

А.3.2. Операции

Объект содержит *данные* (*атрибуты*) и *алгоритмы* (*операции*) для работы с этими данными. Операция объявляется в классе. Процедура, реализующая операцию, называется *методом* (method). Операция (или метод, если быть точным) вызывается с помощью отправленного ей сообщения. Имя сообщения и имя операции совпадают. Операция может содержать список параметров, которым в вызове сообщения могут быть присвоены определенные значения, и может возвращать значение вызывающему объекту.

Имя операции вместе со списком типов формальных аргументов называется *сигнатурой* (signature) операции. Сигнатура внутри класса должна быть уникальной. Это значит, что класс может содержать множество операций с одним и тем же именем, при условии, что списки типов параметров этих операций отличаются.

А.3.2.1. Операции в рамках кооперации объектов

Объектно-ориентированная программа выполняется, реагируя на случайные события, порождаемые пользователем. События возникают, когда пользователь нажимает клавишу, щелкает мышью, выбирает пункты меню, щелкает на командных кнопках или выполняет действия с любыми другими устройствами ввода данных. *Событие*, сгенерированное пользователем, преобразуется в *сообщение*, отправляемое объекту. Для выполнения задания многие объекты должны вступать в кооперацию. Кооперация объектов обеспечивается с помощью вызова операций других объектов (см. раздел А.2.2).

На рис. А.10 показаны операции классов, необходимые для поддержки кооперации объектов, приведенных на рис. А.2. Каждое сообщение, показанное на рис. А.2, запрашивает операцию класса, обозначенного как адресат сообщения.

В данном простом примере классы `Order` и `Product` (на рис. А.2 класс `Product` не показан) не содержат никаких операций. Объект класса `Order` инициирует себя сам, посылая объекту `Shipment` (Поставка) команду осуществить его поставку. В результате поставки может потребоваться пополнение запаса новыми товарами.



Рис. А.10. Операции в рамках кооперации объектов

А.3.2.2. Видимость операций

Видимость операции определяет, доступна ли она объектам других классов. Если она доступна, то ее видимость называется *открытой*. В противном случае она называется *закрытой*. Пиктограммы, приведенные перед именами операций на рис. А.10, обозначают открытую видимость.

Большинство операций объектно-ориентированной системы должны иметь открытую видимость. Для того чтобы объект мог предоставлять услуги внешнему миру, соответствующая операция должна быть доступной. Однако большинство объектов имеют много внутренних операций. Эти операции должны быть закрытыми. Они доступны только объектам класса, в котором они определены.

Следует различать видимость операции и *область ее действия* (operation scope). Операцию можно вызвать из объекта-экземпляра (см. раздел А.2) или из объекта-класса (см. раздел А.3.3). В первом случае говорят, что операция обладает *областью действия экземпляра* (instance scope), а во втором случае — *областью действия класса* (class scope). Например, операция, предназначенная для определения возраста работника, обладает областью действия экземпляра, а операция вычисления среднего возраста всех работников обладает областью действия класса.

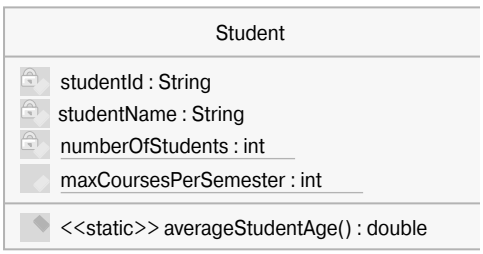
А.3.3. Объекты-классы

В разделе А.2 проведено различие между *объектами-экземплярами* и *объектами-классами*. *Объект-класс* — это объект, область действия атрибутов и/или операций которого совпадает с областью действия класса. Область действия класса означает, что атрибут или операция являются глобальными и вызываются из класса, но необязательно из объекта-экземпляра. Однако следует заметить, что на практике большинство языков программирования не реализуют концепцию объекта-класса и не допускают создания экземпляров таких объектов. Вместо этого они предусматривают синтаксическую возможность ссылаться на имя класса, чтобы получить доступ к атрибутам или операциям, областью действия которых является класс.

Наиболее распространенными глобальными атрибутами являются атрибуты, которые хранят значения, принятые по умолчанию, или агрегированные значения (такие, как суммы, итоги, средние значения). Наиболее распространенной операцией такого рода является операция создания и уничтожения объектов-экземпляров и операции, вычисляющие агрегированные значения.

На рис. А.11 показан класс `Student` с глобальными атрибутом (подчеркнутым) и операцией (идентифицированной стереотипом «`static`»). Обратите внимание на то, что атрибут `numberOfStudents` (количество студентов) является закрытым, а атрибут `maxCourcesPerSemester` (максимальное количество курсов за семестр) — открытым. Каждый студент может прослушать одинаковое максимальное количество курсов за семестр. Операция, вычисляющая средний возраст студентов, обладает глобальной областью действия, поскольку ей необходим доступ к индивидуальному возрасту каждого студента (в объекте-экземпляре `Student`), чтобы извлечь эти величины, просуммировать их и разделить на общее количество студентов, сохранив результат в переменной `numberOfStudents`.

На рис. А.11 показан код на языке Java, соответствующий данной графической модели. Для различия между свойствами экземпляра и класса в языке Java используется ключевое слово `static`. Фактически в одном определении класса язык Java допускает определение двух видов объектов (объекты-экземпляры и объекты-классы) (Lee and Tepfenhart, 2002).




```
public class Student
{
   private String studentId;   //accessible via Student's operations
   private String studentName; //accessible via Student's operations
   private static int numberOfStudents;
   //accessible only to Student's static methods, such as averageStudentAge()
   public static int maxCoursesPerSemester;
   // accessible via Student:: maxCoursesPerSemester

   public static double averageStudentAge()
   { implementation code here }
   //callable by referring to the class name – Student::averageStudentAge()
   //callable also with an object of the class – std.averageStudentAge()
}
```

Рис. А.11. Класс на языке Java с глобальными атрибутами и операциями

Два атрибута экземпляра (`studentId` и `studentName`) хранятся в виде копий (размещенных в памяти) в каждом экземпляре класса. Поскольку эти атрибуты являются закрытыми, они доступны лишь для операций класса `Student`.

Глобальные (статические) атрибуты хранятся в виде одной копии (т.е. занимают одну область памяти). Если статический атрибут является закрытым (как атрибут `numberOfStudent`), то этот участок памяти доступен для всех экземпляров класса `Student` и статических операций этого класса. Если статический атрибут является открытым (как атрибут `maxCourcePerSemester`), то этот участок памяти доступен для всех экземпляров всех классов по имени класса, т.е. `Student::numberOfStudents`.

Открытые статические операции можно вызывать из любого экземпляра любого класса. Их можно вызвать, сославшись на имя или объект класса (например, `std.averageStudenrAge()`).

А.4. Переменные, методы и конструкторы

В ходе предыдущей дискуссии мы, по мере возможности, старались использовать обобщенную терминологию языка UML. Но для того чтобы объяснить принципы реализации объектов, необходимо использовать терминологию проектирования UML-моделей и языков программирования, таких как Java. Часто термины анализа и проектирования имеют одинаковый смысл, но иногда они отличаются друг от друга, поэтому между ними нет взаимно однозначного соответствия.

В данном разделе описываются концепции переменной и метода. Переменная является аналогом атрибута (переменная реализует атрибут). Метод — это аналог операции (метод реализует операцию).

Переменная (variable) — это имя области памяти, которая может содержать значения конкретного типа. Переменную можно объявить в классе или в операции класса (теле метода). В первом случае переменная называется *данным-членом* класса (data member). Во втором случае переменная называется не данным-членом класса, а *локальной переменной* (local variable). Локальная переменная существует только в теле метода (т.е. пока выполняется этот метод).

Данные-члены могут существовать в области действия экземпляра (*переменные экземпляра*) или класса (*переменные класса*) (см. раздел А.3.3). Существуют две категории переменных экземпляра — переменные, реализующие *атрибуты*, и переменные, реализующие *ассоциации*. В первом случае переменные имеют элементарные типы (они хранят значения атрибутов). Во втором случае переменные имеют неэлементарные типы (они хранят ссылки на объекты и, следовательно, реализуют ассоциации).

Важно понимать, что переменные, хранящие ссылки на объекты, не являются объектами, хотя они позволяют выполнять действия с объектами (Lethbridge and Laganiére, 2001). Во время выполнения отдельной программы одна и та же пере-

менная может ссылаться на разные объекты, а на одни и те же объекты могут ссылаться разные переменные. Кроме того, переменная может содержать нулевое значение, т.е. не ссылаться ни на один объект.

Данные-члены (переменные экземпляра и класса) могут инициализироваться как *неконстантные* и *константные* значения/объекты (non-constant or constant). Константная переменная не может изменять свое значение после того, как оно было ему присвоено. Локальные переменные не могут быть константными. В языке Java константы определяются с помощью ключевого слова `final`.

Переменные экземпляра, реализующие атрибуты, могут инициализироваться одновременно со своим определением в классе. Переменные экземпляра, реализующие ассоциации, обычно инициализируются в конструкторе, создающем объекты соответствующего класса.

Метод (method) — это реализация операции (сервиса), принадлежащей классу (Lee and Tepfenhart, 2002). Метод имеет имя и *сигнатуру* (signature) — список формальных аргументов (параметров). Два метода с одним и тем же именем, но разными сигнатурами, считаются разными. Такие методы называются *перегруженными*.

Метод может *возвращать* (вызывающему объекту) одно значение некоего элементарного или неэлементарного типа. Формально говоря, все методы должны иметь тип возвращаемого значения, но этот тип может быть пустым. Имя метода, его сигнатура и тип возвращаемого значения образуют *прототип* метода.

Конструктор (constructor) — это специальный метод (пуристы могут даже сказать, что конструкторы вообще не являются методами). Каждый класс должен иметь хотя бы один конструктор, но конструкторов может быть и несколько (их можно перегружать). Имя конструктора совпадает с именем класса, в котором он объявлен. Конструкторы не имеют типа возвращаемого значения. В языке Java конструктор называется ключевым словом `new`, например:

```
Student std22 = new Student();
```

Конструктор `Student()` в этом примере является так называемым конструктором по умолчанию. Если программист пропустил его в определении класса, то такой конструктор генерируется автоматически. Конструктор по умолчанию создает объект, все переменные класса которого заданы по умолчанию. В языке Java значения, заданные по умолчанию, равны `0` для чисел, `'0'` — для символов, `false` — для булевых переменных и `null` — для объектов.

A.5. Ассоциации

Ассоциация (association) — это разновидность отношений между классами. Другими видами отношений являются *обобщение* (generalization), *агрегация* (aggregation) и *зависимость* (dependency).

Отношение ассоциации устанавливает связь между объектами данных классов. С ее помощью они могут взаимодействовать друг с другом. Обычно сообщения между объектами отправляются по ассоциативной связи. Это дает важное преимущество, а именно: статические структуры (ассоциации), возникающие на этапе компиляции, фиксируют передачу всех возможных динамических сообщений, допустимых в ходе выполнения программы.

На рис. A.12 показана ассоциация *OrdShip* между классами `Order` и `Shipment`. Это отношение позволяет объекту `Order` связаться с несколькими объектами `Shipment` (этот факт обозначен с помощью кратности ассоциации *n*). Объект `Shipment` также может связываться с несколькими объектами `Order`.

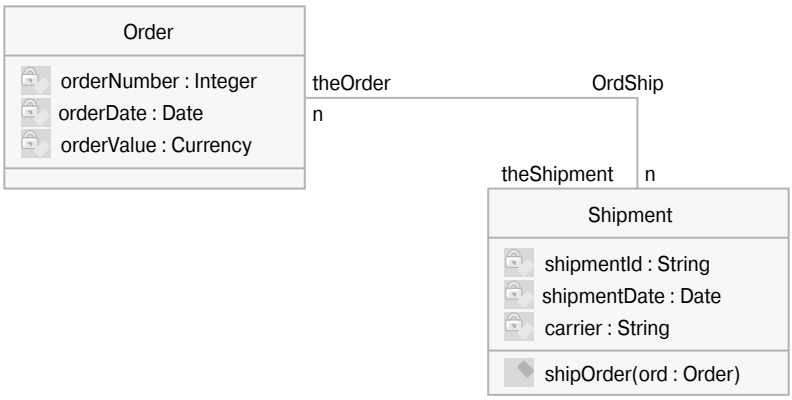


Рис. A.12. Ассоциация

В простейшем случае взаимно однозначной ассоциации между объектом `Order` и объектом `Shipment` сценарий обработки заказа может быть следующим. Объект `Order` должен быть поставлен. Для этого он создает новый объект класса `Shipment`, вызывая один из его конструкторов. В результате создания экземпляра объект `Order` получает ссылку на новый объект `Shipment`.

Теперь объект `Order` может послать объекту `Shipment` сообщение `shipOrder()`, передавая себя объекту `Shipment` как фактический аргумент этого сообщения. Таким образом, объект `Shipment` получает ссылку на объект `Order`. Осталось лишь установить ассоциацию, присвоив ссылку на объект `Shipment` переменной `theShipment` и, наоборот, ссылку на объект `Order` переменной `theOrder`.

Как правило, ассоциации с классами сущностей (бизнес-объектами), как показано на рис. A.12, являются двусторонними. Однако существуют категории классов, между которыми вполне достаточно установить односторонние ассоциации. К ним относятся классы, представляющие окна графического пользовательского интерфейса, программную логику или события, генерируемые пользователем.

A.5.1. Степень ассоциации

Степень ассоциации (association degree) определяет количество классов, соединенных с помощью ассоциации. Наиболее распространенной является ассоциация второй степени — *бинарная ассоциация* (binary association). Ассоциация, показанная на рис. А.11, является бинарной. Ее можно также определить для единственного класса. Такая ассоциация называется *унарной,* или *сингулярной ассоциацией* (unary, or singular association) (Maciaszek, 1990). Унарная ассоциация устанавливает связь между объектами одного класса.

Типичный пример унарной ассоциации показан на рис. А.13. Он описывает иерархическую организационную структуру. Объект `Employee` (Сотрудник) либо подчиняется (`managedBy`) другому объекту `Employee`, либо не подчиняется никому (например, если он работает генеральным директором). Объект `Employee` может руководить (`managerOf`) другими сотрудниками, если он не находится на дне иерархии и ему никто не подчиняется. Возможны также ассоциации третьей степени (*тернарные ассоциации*), хотя их применение не рекомендуется (Maciaszek, 1990).

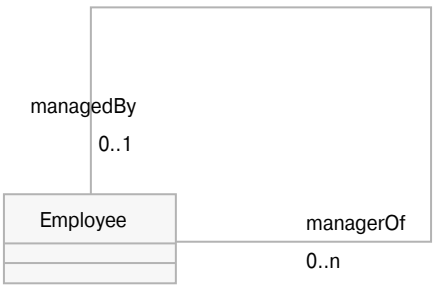


Рис. А.13. Унарная ассоциация

A.5.2. Кратность ассоциации

Кратность ассоциации (association multiplicity) определяет, сколько объектов могут занимать позицию, указанную *функциональным именем*. Кратность говорит о том, сколько объектов целевого класса (указываемых функциональным именем) может быть ассоциировано с одним объектом исходного класса.

Кратность обозначается в виде диапазона целых чисел `i1..i2`. Целое число `i1` определяет минимальное количество связываемых объектов, а `i2` — их максимальное количество (если точное максимальное количество связываемых объектов неизвестно или не фиксировано, то вместо него может быть подставлена буква `n`). Если данная информация не имеет значения для выбранного уровня абстракции (как на рис. А.12), то минимальное количество можно не указывать.

Наиболее часто встречаются следующие значения кратности:

```
0..1
0..n
1..1
1..n
n
```

На рис. A.14 показаны две ассоциации на классах `Teacher` (Преподаватель) и `CourseOffering` (Предлагаемый курс). Одна из ассоциаций фиксирует распределение преподавателей по текущим учебным курсам. Другая определяет, кто из преподавателей отвечает за учебный курс. Преподаватель может вести несколько учебных курсов или не вести ни одного (например, если преподаватель в отъезде). Учебный курс ведут один или несколько преподавателей. Один из этих преподавателей отвечает за курс. В общем случае преподаватель может вести несколько курсов или не вести их совсем. Руководит учебным курсом один и только один преподаватель.

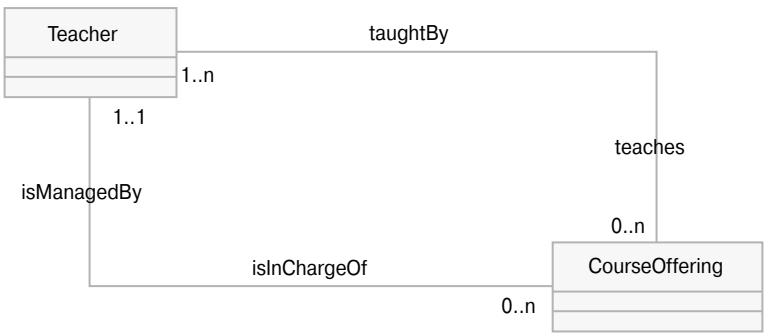


Рис. A.14. Кратность ассоциации

В языке UML термин кратность нельзя понимать буквально. Минимальную кратность, равную нулю или единице, можно интерпретировать как отдельное семантическое понятие *принадлежности* (membership) или *участия* (participation) (Maciaszek, 1990). Нулевая минимальная кратность означает необязательную принадлежность объекта ассоциации. Единичная кратность означает обязательную принадлежность. Например, предлагаемый курс обучения (объект `CourseOffering`) должен проводиться под руководством преподавателя (объект `Teacher`).

Свойство принадлежности обладает интересными семантическими особенностями. Например, конкретная обязательная принадлежность может дополнительно означать, что принадлежность является *фиксированной*, т.е. если объект связан с целевым объектом в ассоциации и не может быть заново связан с другим целевым объектом в той же ассоциации.

А.5.3. Ассоциативная связь и объем ассоциации

Ассоциативная *связь* представляет собой экземпляр ассоциации. Она является *кортежем* (tuple) ссылок на объекты. Кортеж может быть *множеством* или *списком* (упорядоченным множеством) ссылок. В принципе кортеж может содержать только одну ссылку. Как указывалось выше, связь также представляет *функциональное имя*. *Мера* ассоциации — это количество связей в наборе.

На рис. А.15 представлена конкретная реализация ассоциации `OrdShip`, показанной на рис. А.12. На рис. А.15 изображены пять связей. Следовательно, мера ассоциации равна пяти. Понимание сущности ассоциативной связи и меры ассоциации важно для общего представления об ассоциации, однако ассоциативные связи и объемы ассоциаций не предназначены для моделирования и не проявляются явным образом.

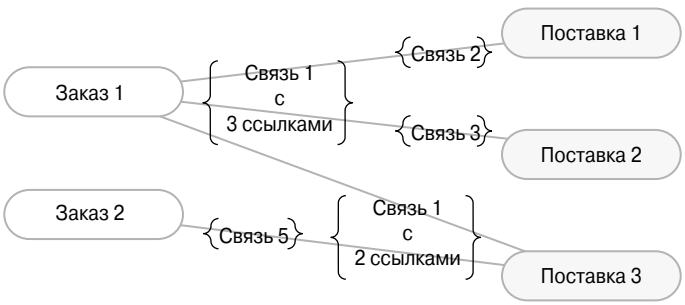


Рис. А.15. Связи и меры

А.5.4. Ассоциативный класс

Иногда ассоциация имеет свои собственные атрибуты (и/или операции). Для моделирования таких ассоциаций необходимо использовать класс (поскольку атрибуты могут быть определены только в классе). Каждый объект *ассоциативного класса* (association class) содержит значения атрибутов и связи с объектами ассоциированных классов. Поскольку ассоциативный класс является разновидностью класса, он может быть ассоциирован с другими классами, существующими в модели, обычным способом.

На рис. А.16 показан ассоциативный класс `Assessment` (Оценка). Объект класса `Assessment` хранит список баллов, общий балл и оценку, полученную студентом (`Student`) по предлагаемому курсу (`CourseOffering`).

Атрибут `mark` (балл) имеет тип `List(Number)`. Это — так называемый *параметризованный тип* (parameterized type). Объект `Number` представляет собой параметр класса `List`, т.е. упорядоченного множества значений. Атрибут `mark` содержит список всех баллов, полученных студентом по данному курсу. Иначе говоря, если студент `Fred` проходит курс обучения по дисциплине “COMP227”, то обязательно

будет создан список (упорядоченное множество) баллов, полученных им в ходе обучения по этому курсу. Этот список баллов запоминается в объекте `A0ssessment`, который представляет собой ассоциацию между объектами `Fred` и `COMP227`.

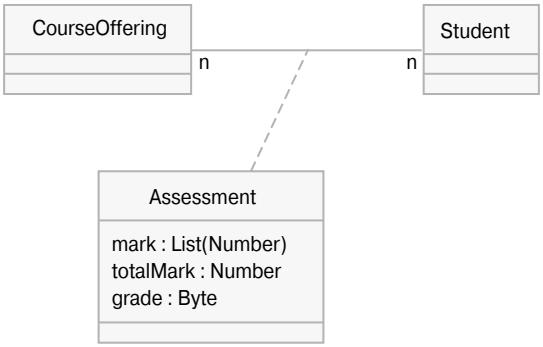


Рис. A.16. Ассоциативный класс

A.6. Агрегация и композиция

Агрегация (aggregation) — это отношение вида *часть-целое* между классом, который представляет набор компонентов (*класс супермножества*), и классами, представляющими компоненты (*классы подмножеств*). *Класс супермножества* (superset class) содержит один или несколько классов *подмножеств* (subset class). Свойство включения может быть сильным (*агрегация по значению*) или слабым (*агрегация по ссылке*). В языке UML *агрегация по значению* (aggregation by value) называется *композицией* (composition), а *агрегация по ссылке* (aggregation by reference) — просто *агрегацией*.

С точки зрения системного моделирования агрегация представляет собой особый случай ассоциации, обладающей дополнительной семантикой. В частности, агрегация обладает свойствами *транзитивности* и *асимметрии*. Транзитивность означает, что если класс `A` содержит класс `B`, а класс `B` содержит класс `C`, то класс `A` содержит класс `C`. Асимметрия означает, что если класс `A` содержит класс `B`, то класс `B` не может содержать класс `A`.

Композиция обладает дополнительным свойством *зависимости по существованию* (existence dependency). Объект класса подмножества не может существовать без связи с объектом класса супермножества. Отсюда следует, что если объект супермножества удален (уничтожен), то объекты его подмножеств также удаляются.

Композиция изображается в виде закрашенного ромба на конце ассоциативной линии, проведенной к классу супермножества. Агрегация, *не* являющаяся композицией, обозначается *пустым ромбом*. Заметим, тем не менее, что пустой ромб можно также использовать, если проектировщик в ходе моделирования не хочет принимать окончательного решения, является ли агрегация композицией или нет.

На рис. А.17 слева показана композиция, а справа — обычная агрегация. Любой объект `Book` (Книга) является композицией объектов `Chapter` (Глава), а любой объект `Chapter` — композицией объектов `Section` (Раздел). Объект `Chapter` не имеет собственной независимой жизни; он существует только в объекте `Book`. Этого нельзя сказать об объектах `BeerBottle` (Пивная бутылка). Объект *может* существовать вне своего контейнера — объекта `Crate` (Ящик).

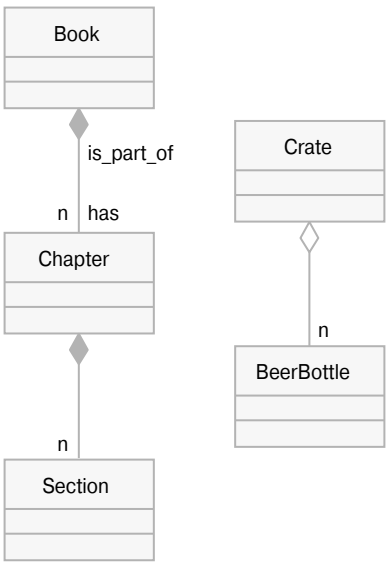


Рис. А.17. Композиция и агрегация

Агрегация и композиция (а также связанное с ними понятие *делегирования*, введенное в разделе А.6.3) представляют собой полезные концепции объектной технологии. К сожалению, коммерческие языки программирования относительно слабо поддерживают эти концепции. Во многих языках реализация агрегации и композиции не отличается от реализации ассоциации, т.е. основана на *скрытых ссылках* (Lee and Tepfenhart, 2002). Язык Java предлагает альтернативную реализацию этих понятий с помощью *внутренних классов* (раздел А.6.2).

А.6.1. Скрытая ссылка

Скрытая ссылка (buried references) реализует агрегацию с помощью переменной, имеющей закрытую видимость и ссылающейся на объект подмножества. Это ничем не отличается от реализации ассоциации с помощью закрытых ссылок. Такая реализация не поддерживает никакой семантики агрегации. Следовательно, например, для того чтобы гарантировать, что удаление объекта супермножества приведет к удалению объектов подмножества, программист должен реализовать соответствующие операции удаления в коде приложения.

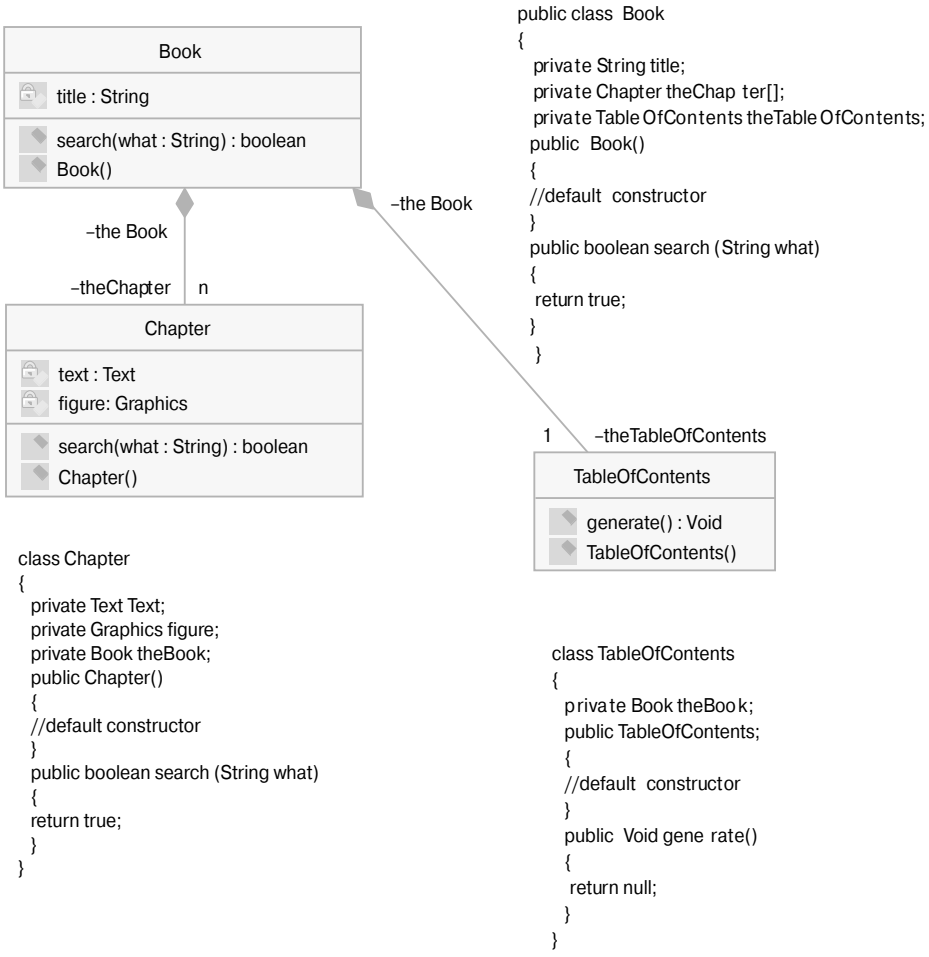


Рис. А.18. Скрытая ссылка

Пример скрытой ссылки приведен на рис. А.18. Объект класса `Book` (Книга) моделируется как композиция многих объектов класса `Chapter` (Глава) и одного объекта класса `TableOfContents` (Содержание). Следовательно, класс `Book` имеет две скрытые ссылки — закрытые переменные `theChapter` и `theTableOfContents`. Сделав эти переменные закрытыми, программист скрывает главы и содержание от других классов. Однако это приносит лишь небольшой выигрыш, поскольку видимость класса не может быть закрытой по отношению к другим классам. В данном примере классы `Chapter` и `TableOfContents` имеют *пакетную* видимость (об этом свидетельствует отсутствие ключевого слова `public` перед именами этих классов). Таким образом, эти классы являются видимыми для других классов, находящихся в том же пакете (в языке Java каждый класс должен находиться только в одном пакете).

Наличие обратных ссылок из классов `Chapter` и `TableOfContents` на класс `Book` является результатом несовершенной реализации агрегации/композиции. Объекты подмножества должны знать, какому супермножеству они принадлежат.

На рис. А.18 показано, как объект супермножества может продемонстрировать внешнему миру, что именно он является владельцем объектов подмножества. В реализации это показано с помощью операции `search()`. В классе `Book` предусмотрена возможность поиска строки в книге. Для этого объекту класса `Book` необходимо послать сообщение `search()`. После этого класс `Book` может переслать этот запрос объектам класса `Chapter`, вызвав метод `search()` из этих объектов. Фактически, ожидая результата, отправитель полагается исключительно на объект класса `Book`.

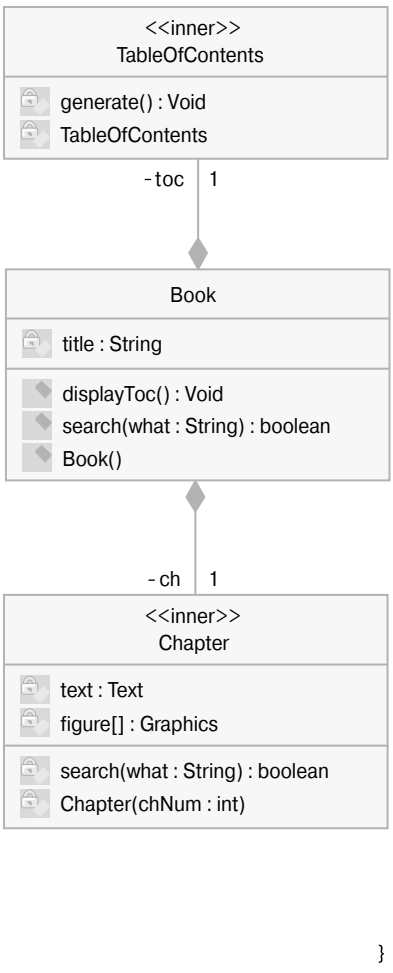
А.6.2. Внутренний класс

В языке Java можно определить класс как внутренний член другого класса. Класс-член может существовать в области действия класса, т.е. определяться с помощью ключевого слова `static`. Такой класс называется *вложенным* (nested class). Кроме того, класс-член может существовать в области действия экземпляра. Такой класс называется *внутренним* (inner class) (Eckel, 2003). Оказывается, что внутренний класс можно использовать в качестве наилучшего механизма реализации агрегации/композиции в языке Java.

Внутренний класс отражает отношение между экземпляром супермножества и его внутренними объектами подмножества. Экземпляр супермножества имеет естественный контроль над объектами подмножества, поскольку он владеет ими. И наоборот, экземпляры подмножества имеют прямой доступ ко всем членам объекта супермножества, включая закрытые.

На рис. А.19 показано, как можно реализовать модель, приведенную на рис. А.18, с помощью внутренних классов. В обычных ситуациях внешний класс получает ссылку на внутренний класс либо создавая внутренний объект в собственном конструкторе, либо предусматривая другой метод для создания внутреннего объекта. В примере описан первый подход. Конструктор `Book()` создает экземпляры объектов `TableOfContents` и `Chapter`, вызывая их конструкторы (хотя он мог бы создать эти объекты с помощью своих закрытых методов, обеспечив тот же самый уровень инкапсуляции). Класс `Book` переадресует все запросы к своему содержанию соответствующим объектам классов `TableOfContents` и `Chapter`.

Внутренние классы имеют дополнительные преимущества в реализации агрегации/композиции — они могут быть *закрытыми* (в то время как обычные классы имеют лишь открытую или пакетную видимость). К закрытым внутренним классам можно получить доступ лишь через внешний класс. Тем самым реализация внутреннего класса полностью скрывается от всех классов, кроме внешнего, а количество зависимостей уменьшается. На рис. А.19 о существовании классов `TableOfContents` и `Chapter` знает только класс `Book`.




```
public class Book
{
  private String title;
  private Chapter[] ch;
  private TableOfContents toc;
  public Book(…)
{ …
   toc = new tableOfContents();
   ch = new Chapter[numberChapters];
   for (int i=0; i<numberChapters; i++)
     ch[i] = new Chapter();
}
  public Void displayToc()
  {
   toc.generate();
   return;
}
public boolean search(String what)
{
   for (int i=0;i<ch.length;i++)
     if( ch[i].search(what))
       return true;
   return false;
}
  private class Chapter
  {
   private Text text;
   private Graphics[] figure;
   private boolean search (String what)
   {…
   }

    }
private class TableOfContents
   {
   private Void generate()
   {…
   }
  }
}
```

Рис. А.19. Внутренний класс

Другие важные преимущества внутренних классов связаны с понятием *интерфейса* и *наследования*. Оба этих понятия обсуждаются в разделах А.7 и А.9. Для читателей, знакомых с этими концепциями, достаточно сказать, что внутренний класс может как реализовать интерфейс, так и расширять класс (наследовать его атрибуты) (Eckel, 2003). Первый способ может еще больше уменьшить зависимости классов, существующих в программе, от внутреннего класса, даже если внутренний класс является открытым. Классам в программе доступны лишь ссылки

на один или несколько интерфейсов, реализованных во внутреннем классе невидимым для внешнего мира способом. Второй метод фактически допускает *множественное наследование реализации*, несмотря на то, что в языке Java этот механизм отсутствует. Множественное наследование является следствием того факта, что внешний класс и его внутренние классы могут независимо друг от друга быть наследниками других классов.

А.6.3. Делегирование

С ассоциацией/композицией связан мощный метод *делегирования* (delegation). Делегация — это хорошая замена наследования с помощью повторного использования кода (Gamma et al., 1995). Несмотря на то что делегирование можно использовать между любыми классами, его лучшие свойства проявляются в классах, связанных агрегацией/композицией.

Идея делегирования заключена в самом его названии. Если объект получает запрос на выполнение одного из его сервисов и не может выполнить его, он делегирует работу одному из его компонентных объектов. Делегирование работы другому объекту не освобождает получателя сообщения от ответственности, таким образом, делегируется *работа*, но не *ответственность*.

На рис. А.19 приведен пример делегирования, в которое вовлечен внутренний класс. Делегирование может относиться к методу `search()`. Метод `search()` содержится в открытом разделе класса `Book`, тем самым класс `Book` предлагает свои услуги внешнему миру. Когда объект класса `Book` получает запрос на выполнение метода `search()`, он делегирует работу объекту класса `Chapter`. Работу выполняет объект класса `Chapter`, а вся слава достается объекту класса `Book`. Запрос на сервис даже не оставляет выбора для адресата сообщения, поскольку класс `Chapter` имеет закрытую видимость.

С технической точки зрения сценарий, приведенный на рис. А.19, называется *пересылкой сообщения* (forwarding message), а не *делегированием*. Делегирование — это более сложная форма пересылки сообщений, в которой объект, делегирующий сервис, передается вместе со ссылкой на него (Gamma et al., 1995). Но в случае внутренних классов ссылку на делегирующий объект передавать не обязательно, поскольку внутренний объект имеет прямой доступ ко всем членам внешнего класса (с помощью скрытых ссылок, реализованных в самом языке программирования).

На рис. А.19 не продемонстрирован аспект, связанный с *повторным использованием* кода и облегченным сопровождением *программ*, опирающихся на этот вид делегирования. Для обеспечения повторного использования кода и облегчения эксплуатации программы делегирование должно сочетаться с интерфейсами и/или абстрактными классами (обе эти концепции обсуждаются в разделах А.8 и А.9).

Идея заключается в том, что изменения в объектах, выполняющих некую работу (т.е. объектах, которым делегируется какая-то работа), не должны влиять ни на остальную программу, ни на саму работу. Для примера предположим, что объек-

ты классов Chapter и Section имеют одинаковый тип и являются наследниками одного и того же суперкласса (абстрактного класса) или реализуют один и тот же интерфейс. В этом случае при выполнении операции search() экземпляры класса Chapter можно заменить экземплярами класса Section. Объект клиента, пославшего сообщение search(), не будет знать об этой замене.

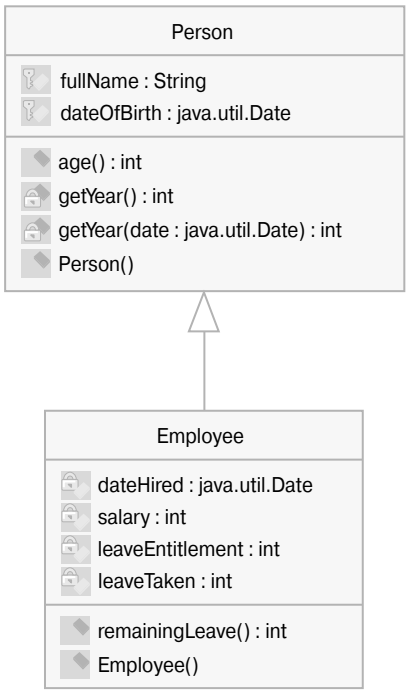
А.7. Обобщение и наследование

Обобщение (generalization) представляет собой разновидность отношения между более общим классом (*суперклассом* или *базовым классом*) и более специализированным видом класса (*подклассом* или *производным классом*). Подкласс является разновидностью суперкласса. Где допускается использование суперкласса, там может использоваться и подкласс. Обобщение делает излишним переопределение уже заданных свойств. Атрибуты и операции, уже определенные для суперкласса, могут *повторно использоваться* в подклассе. Говорят, что подкласс *наследует* (inherit) атрибуты и методы его суперкласса. Обобщение способствует пошаговой спецификации, использованию общих свойств разными классами и лучшей локализации изменений.

Обобщение изображается в виде пустого треугольника на конце линии отношения, проведенной к родительскому классу. На рис. А.20 класс Person (Личность) является суперклассом, а класс Employee (Сотрудник) — подклассом. Класс Employee наследует все атрибуты и операции класса Person. Наследуемые свойства явно не показаны в прямоугольнике, обозначающем подкласс, — отношение обобщения отодвигает наследование на задний план.

Обратите внимание на то, что наследование применимо к классам, а не объектам, т.е. к типам, а не значениям. Класс Employee наследует определения атрибутов fullName (полное имя) и dateOfBirth (дата рождения). Это происходит благодаря тому, что экземпляр класса Employee одновременно является экземпляром класса Person. Следовательно, конструктор Employee() может устанавливать значения атрибутов fullName и DateOfBirth, а также остальных четырех данных-членов.

Однако существует один трюк. Два атрибута в классе Person имеют закрытую видимость. Это значит, что класс Employee не имеет доступа к значениям fullName и dateOfBirth в объекте класса Person. Это логично. Некто по имени Джо Гай может быть либо сотрудником и человеком, либо просто человеком. Если Джо является сотрудником (и человеком), он имеет свои собственные значения атрибутов fullName и dateOfBirth (а также dateHired (дата приема на работу) и т.д.). Если же Джо — просто человек (и не сотрудник), то он также имеет собственные значения атрибутов fullName и dateOfBirth (но не dateHired и т.д.).




```
import java.util.Date;
import java.util.Calendar;
import java.util.GregorianCalendar;
public class Person
{
  protected String fullName;
  protected Date dateOfBirth;
  public Person ()
  {. . .}
  public int age(){
    return getYear() – getYear(dateOfBirth);
  }
  private int getYear(){
    return getYear(new Date(System.currentTimeMil
  }
  private int getyear(Date date){
    Calendar cal = GregorianCalendar.getInstance()
    cal.setTime(date);
    return cal.get(Calendar.YEAR);
  }
}

public class Employee extends Person
{
  private Date dateHired;
  private int salary;
  private int leaveEntitlement;
  private int leaveTaken;
  public Employee()
  {. . .}
  public int remainingLeave(){
    return leaveEntitlement – leaveTaken;
  }
}
```

Рис. А.20. Обобщение

Несмотря на то что экземпляр класса `Employee` не имеет доступа к значениям атрибутов в экземпляре класса `Person`, он может вызвать метод `age()` класса `Person`, не ссылаясь на имя этого класса. Унаследованный метод `age()` имеет доступ к значению `dateOfBirth`, находящемуся в объекте, которому направлен вызов (в объекте класса `Employee`). Другие классы в программе могут также вызывать метод `age()`, поскольку он находится в открытом разделе, но каждый такой вызов должен либо содержать имя класса (`Person::age()`), либо указывать на объект класса `Person` (`person.age()`).

В методе `age()` для вычисления текущего значения атрибута `age` объекта класса `Person` используются две закрытые операции класса `Person` (два метода `getYear()`). В языке Java предусмотрен довольно неудобный механизм работы со значениями данных. Для выполнения этой работы язык Java содержит несколько

библиотек (например, для операторов `import`, показанных на рис. А.20). Первый метод, `getYear()`, возвращает текущий год. Второй параметризованный метод, `getYear(Data date)`, возвращает год для определенной даты в прошлом.

А.7.1. Полиморфизм

Метод, унаследованный подклассом, часто напрямую используется этим подклассом (т.е. без модификации). Операция `age()` идентично работает в объектах классов `Person` и `Employee`. Однако иногда необходимо, чтобы операции в подклассе были *замещены* (модифицированы) в соответствии с семантическими вариациями подкласса. Например, операция `Employee.remainingLeave()` (остаток отпуска сотрудника) вычисляется путем вычитания значения атрибута `leaveTaken` из значения атрибута `leaveEntitlement` (см. рис. А.20). Тем не менее сотрудник, являющийся менеджером, имеет право на ежегодное получение дополнительного отпуска (`leaveSupplement`). Теперь, если добавить в обобщенную иерархию класс `Manager` (как показано на рис. А.21), операция `Manager.remainingLeave()` должна заместить операцию `Employee.remainingLeave()`. Это показано на рис. А.21 с помощью дублирования имени операции в подклассе.

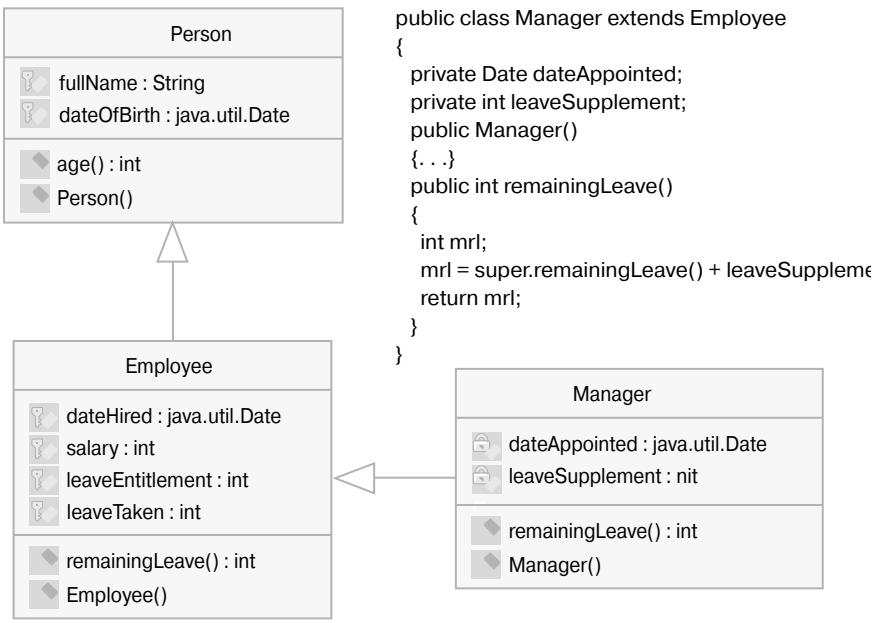


Рис. А.21. Полиморфизм

В этом случае говорят, что операция `remainingLeave()` *замещена* (overridden). Существуют две реализации операции (два *метода*). Теперь можно отправить сообщение `remainingLeave()` объекту `Employee` или `Manager`, и при

этом будут выполняться различные методы. Можно вообще не интересоваться, кто является адресатом: объект `Employee` или `Manager` — для каждого из них будет выполнен соответствующий метод.

Операция `remainingLeave()` *полиморфна* (polymorphic), поскольку существуют два *метода* ее реализации. Оба этих метода имеют одинаковые имена и идентичные *сигнатуры* (signatures) — количество и типы параметров (в данном случае список параметров пуст).

Полиморфизм и наследование тесно взаимосвязаны: полиморфизм без наследования имеет ограниченное применение. *Наследование* (inheritance) позволяет постепенно описывать подкласс путем повторного использования и расширения описания суперкласса. Операцию `Manager.remainingLeave()` можно реализовать с помощью обращения к функциональным возможностям метода `Employee.remainingLeave()` с последующим добавлением к значению, возвращаемому этим методом, величины `leaveSupplement`.

A.7.2. Замещение и перегрузка

Не следует путь замещение и перегрузку. *Замещение* (overriding) — это механизм, используемый для реализации полиморфных операций. Замещенные методы имеют идентичные имена и сигнатуры и размещены в разных классах в рамках одной и той же иерархии наследования. Решение, какой метод вызывать, принимается динамически в ходе выполнения программы. Это решение основано на классе объекта, на который указывает переменная (из которой вызывается метод), а не ее типе (Lee and Tepfenhart, 2002).

Если переменная указывает на объект подкласса в дереве наследования и в этом подклассе существует замещенный метод, то будет вызван замещенный метод. Однако, если этот метод *не объявлен* в подклассе, то среда программирования будет выполнять поиск вверх по дереву иерархии, чтобы найти этот метод в суперклассе данного подкласса. Если метод не был замещен ни в одном из подклассов, то будет вызван метод из базового класса.

Перегрузка (overloading) возникает к ситуации, когда в одном и том же классе объявлено несколько методов с одинаковыми именами. Эти методы имеют одинаковые имена, но разные сигнатуры, а также, возможно, разные типы возвращаемых значений. В отличие от замещения, которое возникает на этапе выполнения программы, перегрузка реализуется на этапе компиляции.

На рис. А.20 проиллюстрирована перегрузка закрытых методов `getYear()` и `getYear(Date date)`. Первый метод возвращает текущий календарный год, второй — календарный год для конкретной даты, переданной как параметр. Программа уже на этапе компиляции решает, какой из методов будет вызван. Фактически оба метода вызываются открытым методом `age()`, объявленным в одном и том же классе (классе `Person`).

А.7.3. Множественное наследование

В некоторых языках, таких как C++, подкласс может быть наследником нескольких суперклассов. Это явление называется *множественным наследованием реализаций* (multiple implementation inheritance). Множественное наследование может порождать конфликты, которые программист должен устранять явно.

На рис. А.22 класс `Tutor` (Наставник) является наследником классов `Teacher` (Преподаватель) и `PostgraduateStudent` (Аспирант). Класс `Teacher` в свою очередь является наследником класса `Person`, как и класс `PostgraduateStudent` (через класс `Student`). В результате класс `Tutor` дважды наследует атрибуты и операции класса `Person`, если только программист не укажет программной среде на необходимость однократного наследования за счет использования левого либо правого "пути" наследования (или если программная среда не применит некоторое правило, принятое по умолчанию и приемлемое для программиста, которое ликвидирует двойное наследование).

Подчеркнем, что язык Java не допускает множественное наследование реализации. Он предусматривает альтернативный механизм, в частности интерфейсы (включая множественное наследование интерфейсов) и внутренние классы, позволяющие реализовать структуры классов и функциональные свойства, соответствующие модели, изображенной на рис. А.22.

А.7.4. Множественная классификация

В большинстве современных объектно-ориентированных программных сред объект может принадлежать только одному классу. Это слишком жесткое ограничение, поскольку в реальности объект может принадлежать одновременно нескольким классам.

Множественная классификация (multiple classification) отличается от множественного наследования. При множественной классификации объект одновременно является экземпляром двух или более классов. При множественном наследовании класс может иметь множество суперклассов, но для каждого объекта должен быть определен единственный класс. О других классах объект может "знать" только благодаря наследованию.

В примере множественного наследования, приведенном на рис. А.22, каждый объект `Person` (например, `Mary` или `Peter`) принадлежит одному классу (наиболее *конкретизированному*, который к нему применим). Если Мэри — аспирантка, а не наставник, то объект `Mary` принадлежит классу `PostgraduateStudent`.

При попытке уточнить значение объекта `Person` для нескольких ортогональных иерархий возникает проблема. Например, объект класса `Person` может быть объектом класса `Employee` (Сотрудник) или `Student` (Студент), `Male` (Мужчина) или `Female` (Женщина), `Child` (Ребенок) или `Adult` (Взрослый) и т.д. Без множественной классификации пришлось бы определить классы для каждой

разрешенной комбинации иерархий, чтобы получить, к примеру, объект `Person`, являющийся объектом класса `Child`, `Female` и `Student` (т.е. класс, который можно было бы назвать `ChildFemaleStudent`) (Fowler, 2004).

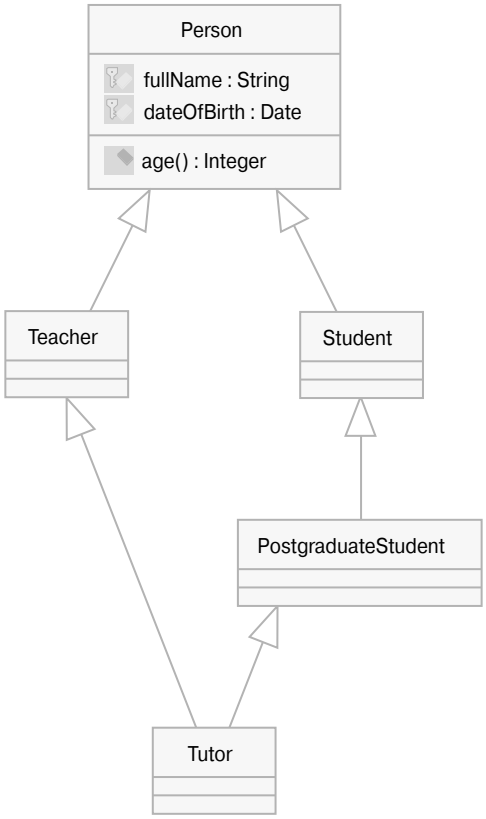


Рис. А.22. Множественное наследование

А.7.5. Динамическая классификация

В большинстве современных объектно-ориентированных программных сред объект не может сменить класс после того, как был реализован (создан). Это еще одно жесткое ограничение, поскольку в реальности объекты могут динамически изменять класс.

Динамическая классификация (dynamic classification) является прямым следствием множественной классификации. Объект может не только принадлежать нескольким классам, но также может в процессе своего жизненного цикла приобретать или утрачивать принадлежность к классу. В случае схемы динамической классификации объект `Person` в один день может быть просто сотрудником, а в другой — менеджером (и сотрудником). Без динамической классификации

такие перемены, как продвижение сотрудников, трудно (или даже невозможно) декларативно моделировать в диаграмме классов. Эта проблема возникает из-за того, что определения идентификаторов объектов (OID) включают в себя информацию о классе, которому принадлежит объект. Динамическая классификация должна была бы допускать изменение идентификаторов объекта, что дискредитирует саму идею уникальных идентификаторов (см. раздел А.2.3).

Язык UML, как и языки программирования, не поддерживает ни динамическую, ни множественную классификацию. Следовательно, все сказанное выше невозможно проиллюстрировать с помощью графических моделей.

А.8. Абстрактный класс

Абстрактный класс (abstract class) — это важная концепция моделирования, логически вытекающая из понятия наследования. Абстрактный класс — это базовый класс, не имеющий непосредственных объектов-экземпляров. Только подклассы абстрактного базового класса могут быть материализованы как экземпляры.

В типичном случае класс становится абстрактным, если хотя бы одна из его операций является абстрактной. *Абстрактная операция* обладает именем и сигнатурой, определенными в абстрактном базовом классе, однако ее реализация (метод) откладывается до того момента, когда будет создан объект *конкретного* производного класса.

Абстрактный класс не может порождать объекты-экземпляры, поскольку он содержит по меньшей мере одну абстрактную операцию. Если бы абстрактный класс создавал объекты, то сообщение, направленное абстрактной операции этого объекта, привело бы к ошибке при выполнении программы (поскольку в этом объекте нет реализации абстрактной операции).

Класс может быть абстрактным только в том случае, если он является суперклассом, который полностью подразделяется на подклассы. Разделение называется полным, если подклассы содержат все возможные объекты, порожденные в наследственной иерархии. В этом случае не существует никаких "блуждающих" объектов (Page-Jones, 2000). Класс `Person` на рис. А.22 не является абстрактным, поскольку может потребоваться реализовать объекты класса `Person`, не являющиеся ни преподавателями (`Teacher`), ни студентами (`Student`). Кроме того, может потребоваться в будущем добавить к подклассы класса `Person` (например, `AdminEmployee`).

На рис. А.23 показан абстрактный класс `VideoMedium` (в языке UML имя абстрактного класса выделяется курсивом). Этот класс содержит абстрактную операцию *`rentalCharge()`*, вычисляющую стоимость проката. Очевидно, что стоимость проката видеокассет и видеодисков вычисляется по-разному. Следовательно, существуют две реализации операции `rentalCharge()` — для классов `Videotape` и `VideoDisk`.

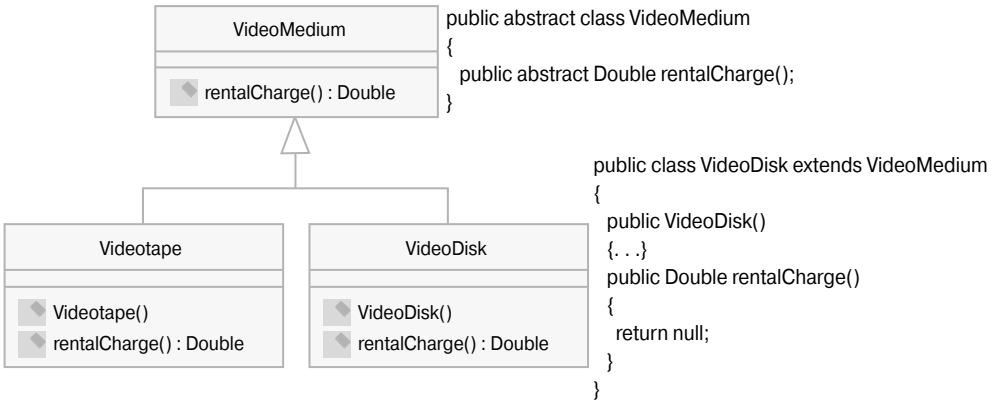


Рис. А.23. Абстрактный класс

Абстрактные классы могут не иметь объектов, но очень полезны при моделировании. Они создают высокоуровневый "словарь" моделирования, без которого язык моделирования будет неполным.

А.9. Интерфейс

Идея абстрактного класса полностью воплощена в интерфейсах Java. *Интерфейс* (interface) — это определение семантического типа с атрибутами (исключительно константными) и операциями, но без фактического объявления операций (т.е. без реализации). Фактические декларации реализуются в одном или нескольких классах, предназначенных для реализации интерфейса.

Программа может использовать переменную интерфейса вместо переменной класса, тем самым отделяя клиентский класс от серверного класса, поставляющего метод реализации. Клиентский объект может определить значение переменной интерфейса и вызвать соответствующий метод из серверного объекта, определенного на этапе выполнения программы.

А.9.1. Интерфейс и абстрактный класс

Абстрактные классы образуют мощный механизм, но они не могут решить проблемы, связанные с множественным наследованием, и не лишены нежелательных побочных эффектов, характерных для наследования реализации (см. раздел 5.2.4 главы 5). Одним из таких побочных эффектов является проблема *хрупкого базового класса* (fragile base class), которая проявляется в том, что при любом изменении реализации базового класса его подклассы подвергаются непредсказуемым влияниям. Поскольку любой абстрактный класс может иметь несколько полностью или частично реализованных методов, он может стать хрупким базовым классом.

На рис. A.24 показано, что абстрактный класс не позволяет решить проблему множественного наследования. Предположим, что магазин видеокассет (см. раздел 1.6.2 главы 1) выдает напрокат не только фильмы, но и оборудование для просмотра видеокассет и дисков. В этом случае проектировщик может попытаться вывести производный класс из класса *VideoMedium*. Однако класс VideoPlayer уже является наследником класса VideoEquipment, поэтому стать наследником класса *VideoMedium* он не может (хотя этот класс и является абстрактным). В языке Java такое наследование невозможно, поскольку в нем предусмотрен лишь механизм одиночного наследования.

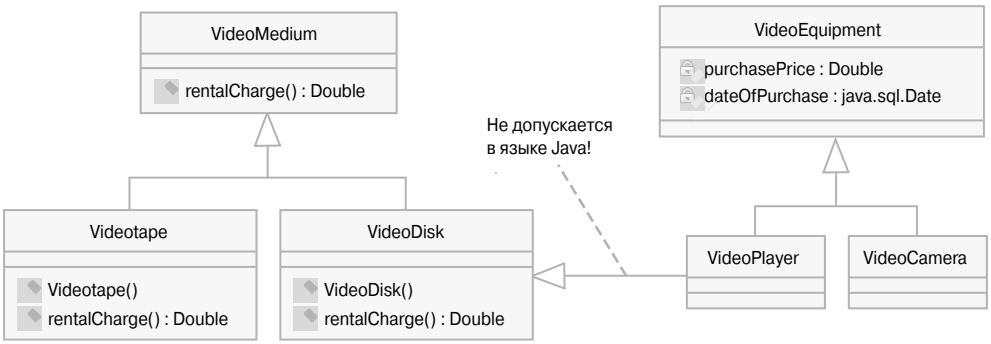


Рис. A.24. Множественное наследование интерфейса в языке Java не допускается

Здесь на помощь приходит понятие *интерфейса*, которое обеспечивает еще несколько преимуществ (Lee and Tepfenhart, 2002; Maciaszek and Liong, 2005). Как и в абстрактном классе, в интерфейсе определяется набор атрибутов и операций, но объекты не создаются. В отличие от абстрактных классов, интерфейсы не реализуют (даже частично) ни один из методов.

Недостаток реализации в интерфейсе напоминает понятие *чисто абстрактного класса*, предусмотренное в языке C#. Однако интерфейс отличается и от этого понятия. В случае чисто абстрактного класса его наследниками должны быть лишь подклассы, реализующие чисто виртуальные методы. В то же время интерфейс может реализовать любой класс системы. Более того, один класс может реализовать любое количество интерфейсов.

A.9.2. Реализация интерфейса

На рис. A.25 показана модель видеомагазина, в которой абстрактный класс, приведенный на рис. A.24, был заменен интерфейсом *VideoMedium*. Графически интерфейс помечается с помощью круга, размещенного в разделе имени (это лишь одна из нескольких возможностей, предусмотренных в языке UML). Все методы интерфейса по умолчанию являются открытыми и абстрактными, поэтому нет необходимости использовать эти ключевые слова в прототипе этих методов.

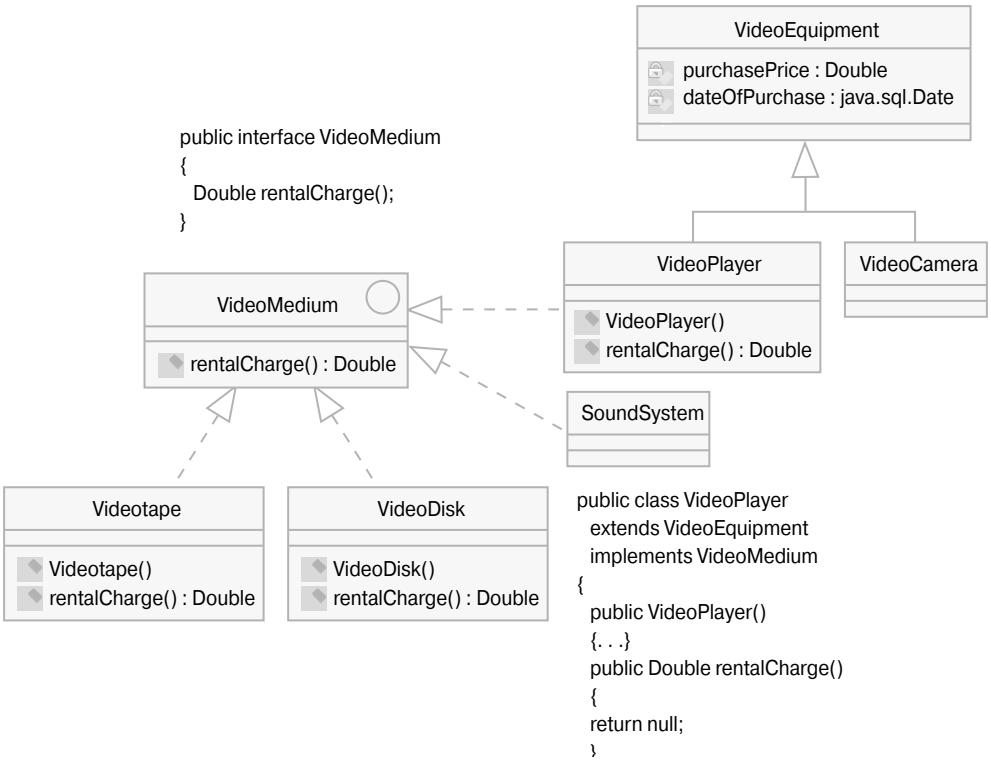


Рис. А.25. Реализация интерфейса на языке Java на примере системы управления магазином видеокассет

Несмотря на то что на рис. А.25 это не показано, интерфейс в языке Java может содержать объявления констант (атрибутов, являющихся открытыми, статическими и финальными). Это ограничивает возможности программиста. Более мощный механизм должен допускать объявление в интерфейсе любого атрибута, тип которого является другим интерфейсом или классом. Это позволило бы создать ассоциации между интерфейсами, а также между интерфейсами и классами. В настоящее время язык Java еще не поддерживает этот механизм, но он уже предусматривается для включения в следующие стандарты языка UML.

Кроме того, на рис. А.25 не показано, что интерфейс может быть наследником другого интерфейса (т.е. он может расширять другой интерфейс). На рис. А.26 продемонстрировано, как один класс (`VideoPlayer`) может расширять другой класс (`VideoEquipment`) и в то же время реализовать один или несколько интерфейсов (`VideoMedium`).

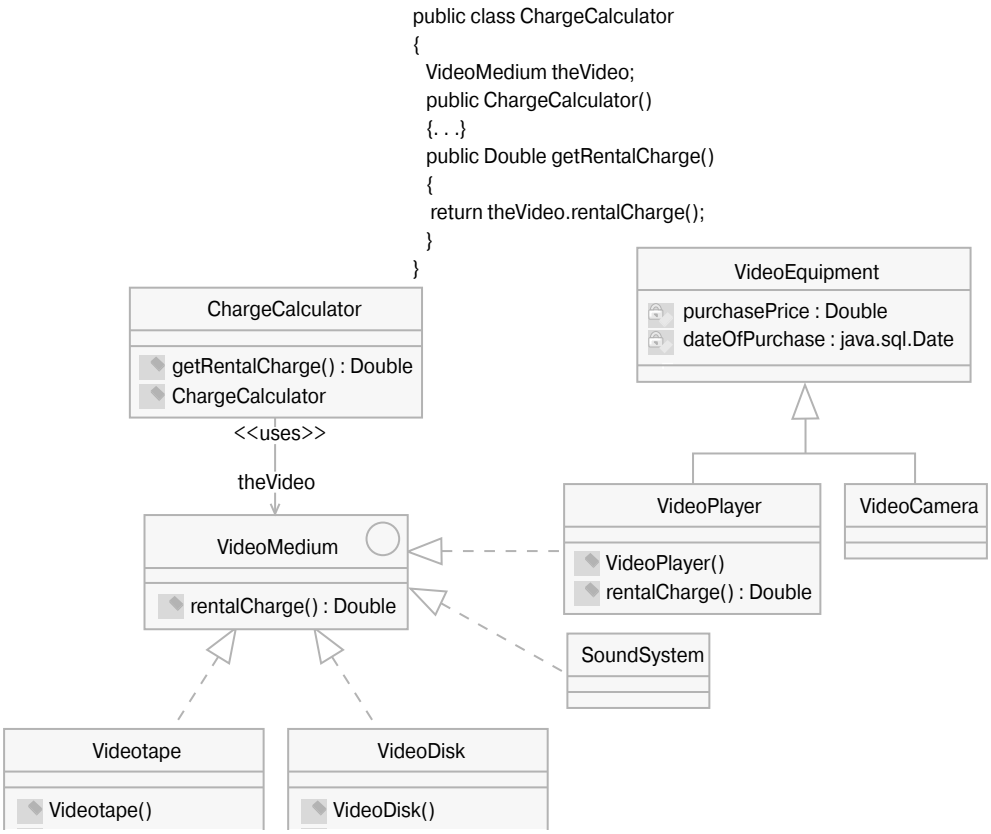


Рис. A.26. Использование интерфейса для исключения зависимости от поставщика в системе управления магазином видеокассет

A.9.3. Использование интерфейса

Мощь интерфейса обусловлена не только за счет удобного решения проблемы множественного наследования реализации. Что еще более важно, интерфейс определяет *ссылочный тип* (reference type), позволяющий отделить клиентские объекты от изменений реализации в объектах сервера.

На имя интерфейса можно ссылаться в любом месте программы, где клиенту требуется сослаться на класс, реализующий данный интерфейс. В результате реализация интерфейса может изменяться, не оказывая никакого влияния на клиентский класс, и даже без уведомления. Это свойство интерфейса существенно повышает *адаптивность* системы.

Отметим, что на рис. A.26 демонстрируется использование односторонних ассоциаций. Ассоциация «`uses`» является односторонней и направлена от класса `ChargeCalculator` к классу `VideoMedium` (этот факт изображается с по-

мощью стрелки). Объект класса `ChargeCalculator` "знает" о существовании объекта класса `VideoMedium` (благодаря переменной `theVideo`), но модель не поддерживает обратную связь от класса `VideoMedium` к классу `ChargeCalculator`.

Резюме

В приложении излагаются основные факты, объясняются фундаментальные термины и концепции объектной терминологии.

Объектная система состоит из взаимодействующих *объектов-экземпляров*. Каждый объект имеет состояние, поведение и идентичность. Концепция *идентичности* является одной из наиболее важных для понимания объектных систем. Кроме того, ее с большим трудом понимают люди, имеющие некоторый опыт разработки обычных компьютерных приложений. Навигация по *связям* — это образ действия объектной технологии, который плохо воспринимают некоторые читатели, владеющие технологией реляционных баз данных.

Класс — это шаблон, предназначенный для создания объектов. Он определяет *атрибуты*, которые объект может содержать, и *операции*, которые он может вызывать. Атрибуты могут иметь элементарные типы или обозначать другие классы. Атрибуты, обозначающие другие классы, называют *ассоциациями*. Ассоциация — это разновидность отношения между классами. Другими видами отношений являются *агрегация* и *композиция*.

Класс может иметь атрибуты и операции, применяемые к самому классу, а не к его объектам. Такие атрибуты и операции требуют понятия *объекта-класса*. Атрибуты и операции, определенные в классах в ходе анализа системы, на этапах проектирования и реализации называются *переменными* и *методами* соответственно.

Полный прототип метода включает в себя имя метода, его *сигнатуру* (список формальных аргументов) и тип возвращаемого им значения. *Конструктор* — это специальный метод, предназначенный для создания объектов класса.

Отношение *ассоциации* создает связь между объектами данных классов. *Степень* ассоциации определяет количество классов, соединенных данной ассоциацией. Кратность *ассоциации* определяет, сколько объектов можно представить с помощью *функционального имени*. Ассоциативная *связь* — это экземпляр ассоциации. Ассоциация, имеющая атрибуты (и/или операции), моделируется как *ассоциативный класс*.

Агрегация — это отношение "часть–целое" между классами, представляющими собой коллекцию компонентов (*классами супермножества*), и классами, представляющими компоненты (*классами подмножества*). В языке UML агрегация по значению называется *композицией*, а агрегация по ссылке — просто *агрегацией*. Агрегация/композиция часто реализуется посредством *скрытых ссылок*. Язык Java

предусматривает альтернативную реализацию агрегации посредством *внутренних классов*. С агрегацией/композицией связан мощный механизм *делегирования*.

Отношение *обобщения* представляет собой разновидность отношения между более общим классом (*суперклассом*, или *базовым классом*) и более специализированным видом этого класса (*подкласса*, или *производного класса*). Обобщение обеспечивает основу для *полиморфизма* и *наследования*. *Замещение* — это механизм реализации полиморфных операций. Коммерческие среды программирования могут поддерживать *множественное наследование*, но они, как правило, не поддерживают ни *множественную*, ни *динамическую классификацию*.

С наследованием связаны понятия абстрактного класса и интерфейса. *Абстрактный класс* — это класс, который может иметь частичную реализацию (т.е. в нем могут быть объявлены не все операции), но не может иметь объектов. *Интерфейс* — это определение семантического типа с атрибутами (исключительно константными) и операциями, не имеющее никакой реализации. Эта реализация должна осуществляться классом, наследующим интерфейс.

Вопросы

B1. Почему следует различать объект-экземпляр и объект-класс?

B2. Что называется идентификатором объекта? Как его можно реализовать?

B3. В чем заключается разница между временным и персистентным объектом?

B4. Что такое временная и персистентная связь? Как они используются в ходе выполнения программы?

B5. Что означает фраза: “Тип атрибута обозначает класс”? Приведите пример.

B6. Почему в хороших объектных моделях большинство атрибутов являются закрытыми, а большинство операций — открытыми?

B7. В чем заключается разница между видимостью и областью действия?

B8. В чем заключается разница между открытым статическим и закрытым статическим членами класса в терминах их доступности? Приведите пример.

B9. В каких ситуациях следует использовать ассоциативный класс? Приведите пример.

B10. Скрытая ссылка и внутренний класс — это два механизма реализации агрегации/композиции. Насколько полно эти механизмы реализуют семантику агрегации/композиции? Обоснуйте свой ответ.

B11. Объясните, почему в типичной среде объектного программирования наследование применяется к классам, а не к объектам?

В12. Как связаны между собой замещение и полиморфизм?

В13. Чем множественная классификация отличается от множественного наследования?

В14. Какие преимущества приносит абстрактный класс при моделировании по сравнению с интерфейсом? Приведите примеры.

Ответы на вопросы с нечетными номерами

В1

Большинство процессов в объектных системах выполняются взаимодействующими объектами-экземплярами. *Объекты-экземпляры* посылают сообщения другим объектам-экземплярам, чтобы активизировать их методы. Следовательно, чтобы попросить о консультации, объект-экземпляр класса `Student` (Студент) может послать сообщение объекту-экземпляру класса `Instructor` (Преподаватель). В системе может существовать много объектов класса `Instructor`, но студент хочет обратиться к конкретному преподавателю.

Тем не менее иногда сообщения необходимо разослать группе объектов-экземпляров. Например, студент может запросить список преподавателей, чтобы решить, к какому из них обратиться. *Объект-класс* `Instructor` — это единственный объект, знающий о существовании своих объектов-экземпляров. Следовательно, чтобы получить список преподавателей, объект-экземпляр класса `Student` должен послать сообщение объекту-классу `Instructor`.

В целом объект-класс содержит сервисы (методы), реализующие эффективный доступ ко всем объектам-экземплярам этого класса. Такие сервисы необходимы, чтобы получить список объектов-экземпляров и выполнить над ними статистические вычисления (например, суммирование, подсчет и усреднение). Не менее важно, что объект-класс несет ответственность за *создание* новых объектов-экземпляров.

В3

Временный объект создается и уничтожается в рамках отдельного выполнения программы. *Персистентный объект* продолжает существовать после завершения выполнения программы. Персистентный объект хранится в месте постоянного хранения, как правило, в базе данных на жестком диске.

Персистентный объект считывается в память программы для обработки и может быть возвращен в область постоянного хранения еще до того, как программа завершится. Разумеется, программа может уничтожить персистентный объект, удалив его из базы данных.

B5

Атрибут класса может принимать значения, имеющие встроенный или пользовательский тип. Набор *встроенных типов*, поддерживаемых объектной средой программирования, известен *заранее*. Встроенные типы могут быть *атомарными* (например, `int` или `boolean`) или *структурированными* (например, `Date` или `Time`).

Пользовательский тип — это новый *тип объекта*, требуемый приложением. Этот тип можно использовать для реализации *класса* с целью создания объектов этого класса.

Атрибуту можно присвоить объект пользовательского типа. Значением такого атрибута является идентификатор OID объекта этого пользовательского типа. Атрибут связан с объектом другого (или этого же) типа. В этом случае говорят, что тип атрибута определяется (пользовательским) классом. Примером является атрибут `theCust` в классе `Invoice` (Счет-фактура). Тип этого атрибута может быть классом `Customer` (Клиент).

B7

Видимость операции означает способность внешних объектов ссылаться на эту операцию. Видимость может быть открытой, закрытой, пакетной (по умолчанию принятой в языке Java) или защищенной (связанной с понятием наследования).

Область действия операции (operation scope) определяет, кто является владельцем операции: объект-экземпляр (*область действия объекта*) или объект-класс (*область действия класса*). Конструкторы, создающие новые объекты, должны иметь область действия класса. Область действия класса означает централизованную глобальную информацию об объектах-экземплярах. Ее следует тщательно контролировать, особенно в распределенных объектных системах.

B9

Ассоциативный класс должен использоваться в ситуациях, когда ассоциация сама по себе имеет свойства (атрибуты и/или операции). Это часто происходит в ассоциациях "множество–множество" и реже — во взаимно-однозначных ассоциациях. Ассоциативный класс для ассоциации "один–множество" встречается очень редко.

Ассоциация "множество–множество" между классами `Employee` (Сотрудник) и `Skill` (Квалификация) может потребовать использования ассоциативного класса, если мы захотим хранить такую информацию, как `dateSkillAcquired` (дата приобретения классификации). Взаимно однозначная ассоциация между классами `Husband` (Муж) и `Wife` (Жена) может реализоваться с помощью ассоциативного класса, предназначенного для хранения атрибутов `marriageDate` (дата бракосочетания) и `marriagePlace` (место бракосочетания).

B11

В типичной среде объектного программирования *наследование* представляет собой поступательное определение класса. Это механизм, с помощью которого более конкретный класс инкорпорирует элементы определения (атрибуты и операции) более общего класса. По этой причине наследование применяется к *типам* (классам), а не объектам.

Объекты создаются после того, как унаследованные элементы будут добавлены в определение класса. Значения унаследованных элементов создаются точно так же, как и значения собственных элементов.

В целом можно себе представить, что наследование можно расширить так, чтобы стало возможным *наследование значений* (*наследование объектов*). Наследование значений может оказаться полезным для установки значений атрибутов, заданных по умолчанию.

Например, объект класса `Sedan` (Седан), производного от класса `Car` (Автомобиль), может наследовать заданные по умолчанию количество колес (четыре), коробку передач (автоматическую) и т.д. При необходимости эти значения можно изменить (заместить). Некоторые инструменты базы знаний поддерживают наследование значений.

B13

Множественная классификация — это режим выполнения программы, в котором объект является экземпляром нескольких классов (и, следовательно, непосредственно принадлежит им). *Множественное наследование* — это семантический вариант обобщения, в котором класс может иметь несколько базовых классов (и, следовательно, наследует свойства всех базовых классов).

Множественная классификация не поддерживается популярными объектными языками программирования. Это создает проблемы в ситуациях, в которых объекты могут играть несколько ролей, например, объект класса `Person` (Человек) может быть одновременно объектом классов `Women` (Женщина) и `Student` (Студент). Не прибегая к множественной классификации, мы могли бы воспользоваться множественным наследованием, чтобы создать класс `FemaleStudent` (Студентка).

Определение классов для каждой допустимой комбинации базовых классов не всегда является желательным, особенно, если объекты со временем могут менять роли. Множественная классификация может сочетаться с *динамической классификацией*, чтобы позволить объектам менять класс на этапе выполнения программы.

Библиография

1. Agile (2006) www.agilealliance.org (last accessed February 2007).
2. Alhir, S.S. (2003) *Learning UML*, O'Reilly & Associates.
3. Allen, p. and Frost, S. (1998) *Component-Based Development for Enterprise Systems: Applying the SELECT Perspective*™, Cambridge University Press.
4. Alur, D., Crupi, J. and Malks, D. (2003) *Core J2EE Patterns: Best practices and design strategies*, 2nd edition, Prentice Hall.
5. Arthur, L.J. (1992) *Rapid Evolutionary Development: Requirements, prototyping and software creation*, John Wiley.
6. Bahrami, A. (1999) *Object Oriented Systems Development*, McGraw-Hill.
7. Beck, K. (1999) *Extreme Programming Explained: Embrace challenge*, Addison-Wesley.
8. Bennett, S., McRobb, S. and Farmer, R. (2002) *Object-Oriented Systems Analysis and Design Using UML*, 2nd edition, McGraw-Hill.
9. Benson, S. and Standing, C. (2002) *Information Systems: A business approach*, John Wiley.
10. *Bloch, J. (2001) Effective Java: Programming language guide, Addison-Wesley.*
11. *Bochenski, B. (1994) Implementing Production-quality Client/Server Systems, John Wiley.*
12. *Boehm, B.W. (1988) "A spiral model of software development and enhancement", Computer, May, p. 61–72.*
13. *Booch, G., Rumbaugh, J. and Jacobson, I. (1999) The Unified Modeling Language: User guide, Addison-Wesley.*
14. *Bourne, K.C. (1997) Testing Client/Server Systems, McGraw-Hill.*
15. *BPMN (2006)* http://en.wikipedia.org/wiki/BPMN *(last accessed February 2007).*

16. *Brainstorming (2003)* www.brainstorming.co.uk/contents.html *(last accessed February 2007).*
17. *Brooks, F.P. (1987) "No silver bullet: essence and accidents of software engineering", IEEE Software, 4, p. 10–19; reprinted in C.F. Kemerer (ed.) Software Project Management: Readings and Cases (1997), Irwin, p. 2–14.*
18. *Buschmann, F., Meunier, R., Rohnert, H., Sommerland, p. and Stal, M. (1996) Patternoriented Software Architecture: A system of patterns, John Wiley.*
19. *CMM (1995) The Capability Maturity Model: Guidelines for improving the software process, Addison-Wesley.*
20. *COBIT (2000) COBIT Framework, 3rd edition, IT Governance Institute.*
21. *COBIT (2005) Aligning COBIT®, ITIL® and ISO 17799 for Business Benefit, IT Governance Institute,* www.itsmf.com/images/news/ITIL-COBiT.pdf *(last accessed February 2007).*
22. *Coad, p. with North, D. and Mayfield, M. (1995) Object Models: Strategies, patterns, and applications, Yourdon Press.*
23. *Conallen, J. (2000) Building Web Applications with UML, Addison-Wesley.*
24. *Connolly, T.M. and Begg, C.E. (2005) Database Systems: A practical approach to design, implementation and management, 4th edition, Addison Wesley.*
25. *Constantine, L.L. and Lockwood, L.A.D. (1999) Software for Use: A practical guide to the models and methods of usage-centered design, Addison-Wesley.*
26. *Date, C.J. (2000) An Introduction to Database Systems, 7th edition, Addison-Wesley.*
27. *Davenport, T.H. (1993) Process Innovation: Reengineering work through information technology, Harvard Business School Press.*
28. *Davenport, T.H. and Short, J. (1990) "The new industrial engineering: Information technology and business process redesign", Sloan Management Review, Cambridge, summer, p. 11, 17.*
29. *Douce, C.R., Layzell, P.J. and Buckley, J. (1999) "Spatial measures of software complexity", Proceedings of the 11th Annual Workshop of Psychology of Programming Interest Group, Leeds, UK,* www.ppig.org/papers/11th-douce.pdf *(last accessed February 2007).*
30. *Eckel, B. (2003) Thinking in Java, 3rd edition, Prentice Hall.*
31. *Elmasri, R. and Navathe, S.B. (2000) Fundamentals of Database Systems, 3rd edition, Addison-Wesley.*
32. *Extreme (2006)* www.xprogramming.com *(last accessed February 2007).*
33. *Feature (2006)* www.featuredrivendevelopment.com *(last accessed February 2007).*
34. *Fenton, N.E. and Pfleeger, S.L. (1997) Software Metrics: A rigorous and practical approach, 2nd edition, PWS Publishing Company.*

35. *Ferm, F. (2003) "The what, how, and why of a subsystem", The Rational Edge, June,* `http://download.boulder.ibm.com/ibmal/pub/software/dw/rationaledge/jun03/TheRationalEdge_June2003.pdf` *(last accessed February 2007).*
36. *Fowler, M. (1997) Analysis Patterns: Reusable object models, Addison-Wesley.*
37. *Fowler, M. (2003) Patterns of Enterprise Application Architecture, Addison-Wesley.*
38. *Fowler, M. (2004) UML Distilled: A brief guide to the standard object modeling language, 3rd edition, Addison-Wesley.*
39. *Fowler, S. (1998) GUI Design Handbook, McGraw-Hill.*
40. *Fowler, S and Stanwick, V. (2004) Web Application Design Handbook: Best Practices for Web-based software, Morgan Kaufmann.*
41. *Galitz, W.O. (1996) The Essential Guide to User Interface Design: An introduction to GUI design principles and techniques, John Wiley.*
42. *Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of reusable object-oriented software, Addison-Wesley.*
43. *Ghezzi, C., Jazayeri, M. and Mandrioli, D. (2003) Fundamentals of Software Engineering, Prentice Hall.*
44. *Glass, R.L. (2005) "IT failure rates – 70 percent or 10–15 percent?", IEEE Soft, May/June, p. 110–112.*
45. *Gold, N.E., Mohan, A.M and Layzell, P.J. (2005) "Spatial complexity metrics: an investigation of utility", IEEE Transactions on Software Engineering, 31 (3) p. 203–212.*
46. *Grady, R. (1992) Practical Software Metrics for Project Management and Process Improvement, Prentice Hall.*
47. *Gray, N.A.B. (1994) Programming with Class, John Wiley.*
48. *Hammer, M. (1990) "Reengineering work: don't automate, obliterate", Harvard Business Review, July/August, p. 104.*
49. *Hammer, M. and Champy, J. (1993a) Reengineering the Corporation: A manifesto for business revolution, Allen & Unwin.*
50. *Hammer, M. and Champy, J. (1993b) "The promise of reengineering", Fortune, 9, p. 94.*
51. *Hammer, M. and Stanton, S. (1999) "How process enterprises really work", Harvard Business Review, November/December, p.108–118.*
52. *Harmon, p. and Watson, M. (1998) Understanding UML: The Developer's Guide: With a Web-based application in Java, Morgan Kaufmann.*
53. *Hawryszkiewycz, I., Karagiannis, D., Maciaszek, L. and Teufel, B. (1994) "RESPONSE: requirements specific object model for workgroup computing", International Journal of Intelligent & Cooperative Information Systems, 3, p. 293–318.*

54. *Heldman, K. (2002) PMP: Project management professional: study guide, Sybex Inc. Henderson-Sellers, B. (1996) Object-oriented Metrics: Measures of complexity, Prentice Hall.*
55. *Heumann, J. (2003) "User experience storyboards: building better UIs with RUP, UML, and use cases", The Rational Edge, November,* `www.-128.ibm.com/developerworks/rational/library/content/RationalEdge/nov03/f_usuability_jh-pdf` *(last accessed February 2007).*
56. *Hirschfeld, R. and Hanenberg, S. (2005) "Open aspects", Computer Languages, Systems & Structures, 32, p. 87–108.*
57. *Hoffer, J.A., George, J.F., and Valacich, J.S. (2002) Modern Systems Analysis and Design, 3rd edition, Prentice Hall.*
58. *Hohpe, G. and Woolf, B. (2003) Enterprise Integration Patterns, Addison-Wesley.*
59. *Horton, I. (1997) Beginning Visual C++ 5, Wrox Press.*
60. *ITIL (2004) The IT Infrastructure Library: An introductory overview of ITIL, Version 1.0a, itSMF,* `www.itsmf.no/bestpractice/itil_overview.pdf` *(last accessed February 2007).*
61. *Jacobson, I. (1992) Object-Oriented Software Engineering: A use case-driven approach, Addison-Wesley.*
62. *Jordan, E.W. and Machesky, J.J. (1990) Systems Development: Requirements, evaluation, design, and implementation, PWS-Kent.*
63. *JUnit (2004)* `www.junit.org` *(last accessed February 2007).*
64. *Khoshafian, S., Chan, A., Wong, A. and Wong, H.K.T. (1992) A Guide to Developing Client/Server SQL Applications, Morgan Kaufmann.*
65. *Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loigntier, J.-M. and Irwin, J. (1997) "Aspect-oriented Programming", Proceeding of the European Conference on Object-Oriented Programming (ECOOP 97), LNCS 1242, Springer, p. 220–242.*
66. *Kifer, M., Bernstein, A. and Lewis, p. (2006) Database Systems: An application-oriented approach: Complete version, 2nd edition, Addison-Wesley.*
67. *Kimball, R. (1996) The Data Warehouse Toolkit: Practical techniques for building dimensional data warehouses, John Wiley.*
68. *Kirkwood, J. (1992) High Performance Relational Database Design, Ellis Horwood.*
69. *Kleppe, A., Warmer, J. and Bast, W. (2003) MDA Explained: The model-driven architecture: practice and promise, Addison-Wesley.*
70. *Koestler, A. (1967) The Ghost in the Machine, Hutchinson.*
71. *Koestler, A. (1978) Janus: A summing up, Hutchinson.*
72. *Kotonya, G. and Sommerville, I. (1998) Requirements Engineering: Processes and techniques, John Wiley.*

73. *Kozaczynski, W. and Thario, J. (2003) "Transforming User Experience Models to Presentation Layer Implementations',* `http://se2c.uni.lu/tiki/se2c-bib.php` *and download paper from [262] of the LASSY Bibliography listing (last accessed February 2007).*
74. *Krasner, G.E. and Pope, S.T. (1988) "A cookbook for using the model view controller user interface paradigm in Smalltalk-80", Journal of Object-oriented Programming, August–September, p. 26–49.*
75. *Kruchten, p. (2003) The Rational Unified Process: An introduction, 3rd edition, Addison-Wesley.*
76. *Kruchten, P., Obbink, H. and Stafford, J. (2006) "The past, present, and future of software architecture", IEEE Software, March/April, p. 22–30.*
77. *Lakos, J. (1996) Large-scale C++ Software Design, Addison-Wesley.*
78. *Larman, C. (2005) "Applying UML and patterns", An Introduction to Object-oriented Analysis and Design and Iterative Development, 3rd edition, Prentice Hall.*
79. *Laudon, K.C. and Laudon, J.P. (2006) Management Information Systems: Managing the digital firm, 9th edition, Prentice Hall.*
80. *Lee, R.C. and Tepfenhart, W.M. (1997) UML and C++: A practical guide to object-riented development, Prentice Hall.*
81. *Lee, R.C. and Tepfenhart, W.M. (2002) Practical Object-oriented Development with UML and Java, Pearson Education.*
82. *Lethbridge, T.C. and Laganiére, R. (2001) Object-Oriented Software Engineering: Practical software engineering using UML and Java, McGraw-Hill.*
83. *Lieberherr, K.J. and Holland, I.M. (1989) "Assuring good style for object-oriented programs", IEEE Software, 9, p. 38–48.*
84. *Linthicum, D.S. (2004) Next Generation Application Integration: From simple information to Web services, Addison-Wesley.*
85. *Maciaszek, L.A. (1990) Database Design and Implementation, Prentice Hall.*
86. *Maciaszek, L.A. (1998) "Object-oriented development of business information systems – approaches and misconceptions", Proceedings of the 2nd International Conference on Business Information Systems BIS '98, Poznan, Poland, p. 95–111.*
87. *Maciaszek, L.A. (2006) "From hubs via holons to an adaptive meta-architecture – the 'AD-HOC' approach", in Software Engineering Techniques: Design for Quality, ed. K. Sacha, Springer, p. 1–13.*
88. *Maciaszek, L.A. and Liong, B.L. (2005) Practical Software Engineering: A case study approach, Addison-Wesley.*
89. *Maciaszek, L.A., De Troyer, O.M.F., Getta J.R. and Bosdriesz, J. (1996a) "Generalization versus aggregation in object application development – the 'ad-hoc' approach", Proceedings of the 7th Australasian Conference on Information Systems ACIS '96, 2, Hobart, Australia, p. 431–442.*

90. *Maciaszek, L.A., Getta, J.R. and Bosdriesz, J. (1996b) "Restraining complexity in object system development – the 'ad-hoc' approach", Proceedings of the 5th International Conference on Information Systems Development ISD '96, Gdansk, Poland, p. 425–435.*
91. *MDA (2006)* `www.omg.org/mda` *(last accessed February 2007).*
92. *Melton, J. (2002) Advanced SQL:1999: Understanding object-relational and other advanced features', Morgan Kaufmann.*
93. *Melton, J. and Simon, A. (2001) SQL:1999: Understanding relational language components, Morgan Kaufmann.*
94. *Meyers, S. (1998) Effective C++: 50 specific ways to improve your programs and design, 2nd edition, Addison-Wesley.*
95. *Michelson, B.M. (2005) Business Process Execution Language (BPEL) Primer: Understanding an important component of SOA and integration strategies, Patricia Seybold Group,* `http://elementallinks.typepad.com/bmichelson/2005/09/view_bpel_proce.html` *(last accessed February 2007).*
96. *Murphy, G. and Schwanninger, C. (2006) "Aspect-oriented programming", IEEE Software, January–February, p. 20–23.*
97. *Olsen, D.R. (1998) Developing User Interfaces, Morgan Kaufmann.*
98. *OMG (2004)* `www.omg.org/uml` *(last accessed February 2007).*
99. *Oz, E. (2004) Management Information Systems, 4th edition, Thomson.*
100. *Page-Jones, M. (2000) Fundamentals of Object-Oriented Design in UML, Addison-Wesley.*
101. *Pfleeger, S.L. (1998) Software Engineering: Theory and practice, Prentice Hall.*
102. *Polikoff, I., Coyne, R. and Hodgson, R. (2006) Capability Cases: A solution envisioning approach, Addison-Wesley.*
103. *Poppendieck, M. and Poppendieck, T. (2003) Lean Software Development: An agile toolkit for software development managers, Addison-Wesley.*
104. *Porter, M. (1985) Competitive Advantage: Creating and sustaining superior performance, Free Press.*
105. *Porter, M.E. and Millar, V.E. (1985) "How information gives you competitive advantage", Harvard Business Review, July/August, p. 149–161.*
106. *Pressman, R.S. (2005) Software Engineering: A practitioner's approach, 6th edition, McGraw-Hill.*
107. *Quatrani, T. (2000) Visual Modeling with Rational Rose 2000 and UML, Addison-Wesley.*
108. *Ramakrishnan, R. and Gehrke, J. (2000) Database Management Systems, McGraw-Hill.*
109. *Rational (2000) Rational Solutions for Windows, online documentation, April 2000 edition, Rational Software.*

110. *Rational (2002) Rational Suite Tutorial, Version 2002.05.00, Rational Software.*
111. *Responsive (2003)* `www.responsivesoftware.com/timelog.htm` *(last accessed February 2007).*
112. *Riel, A.J. (1996) Object-oriented Design Heuristics, Addison-Wesley.*
113. *Robertson, J. and Robertson, S. (2003) Volere Requirements Specifications Template, 9th edition, Atlantic Systems Guild,* `www.atlsysguild.com` *(accessed February 2007).*
114. *Robson, W. (1994) Strategic Management and Information Systems: An integrated approach, Pitman.*
115. *Roy-Faderman, A., Koletzke, p. and Dorsey, p. (2004) Oracle JDeveloper 10g Handbook, McGraw-Hill/Osborne.*
116. *Rozanski, N. and Woods, E. (2005) Software Systems Architecture: Working with stakeholders using viewpoints and perspectives, Addison-Wesley.*
117. *Ruble, D.A. (1997) Practical Analysis and Design for Client/Server and GUI Systems, Yourdon Press.*
118. *Rumbaugh, J. (1994) "Getting started: using use cases to capture requirements", Journal of Object-oriented Programming, September, p. 8–10, 12, 23.*
119. *Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991) Object-oriented Modeling and Design, Prentice Hall.*
120. *Rumbaugh, J., Jacobson, I. and Booch, G. (2005) The Unified Modeling Language Reference Manual, 2nd edition, Addison-Wesley.*
121. *RUP (2003)* `www-306.ibm.com/software/awdtools/rup/` *(last accessed February 2007).*
122. *Rus, I. and Lindvall, M. (2002) "Knowledge management in software engineering", IEEE Software, May/June, p. 26–38.*
123. *Sam-Bodden, B. and Judd, C.M. (2004) Enterprise Java Development on a Budget: Leveraging Java open source technologies, Apress (Springer).*
124. *Schach, S. (2005) Classical and Object-Oriented Software Engineering, 6th edition, McGraw-Hill.*
125. *Schmauch, C.H. (1994) ISO 9000 for Software Developers, ASQC Quality Press.*
126. *Selic, B. (2003) "The subsystem: a curious creature", The Rational Edge, July,* `www-128.ibm.com/developerworks/rational/library/content/RationaleEdge/jul03/k_subsystem_bs.pdf` *(last accessed February 2007).*
127. *Silberschatz, A., Korth, H.F. and Sudershan, S. (2002) Database System Concepts, 4th edition, McGraw-Hill.*
128. *Singh, I., Stearns, B., Johnson, M. and Enterprise Team (2002) Designing Enterprise Applications with the J2EE Platform, 2nd edition, Addison-Wesley.*
129. *Sklar, J. (2006) Principles of Web Design, Thomson Course Technology.*

130. *Smith, J. (2003) "A Comparison of the IBM Rational Unified Process and eXtreme Programming".* www3.software.ibm.com/ibmdl/pub/software/rational/web/ whitepapers/2003/TP167.pdf *(last accessed February 2007).*
131. *Smith, J.M. and Smith, D.C.P. (1977) "Database abstractions: aggregation and generalization", ACM Transactions on Database Systems, 2, p. 105–133.*
132. *Sommerville, I. and Sawyer, p. (1997) Requirements Engineering: A good practice guide, John Wiley.*
133. *Sony (2004)* www.sonystyle.com *(last accessed February 2007).*
134. *Sowa, J.F. and Zachman, J.A. (1992) "Extending and formalizing the framework for information systems architecture", IBM Systems Journal, 3, p. 590–616.*
135. *Stein, L.A., Lieberman, H. and Ungar, D. (1989) "A shared view of sharing: the Treaty of Orlando", in W. Kim and F.H. Lochovsky (eds), Object-oriented Concepts, Databases and Applications, Addison-Wesley, p. 31–48.*
136. *Stelting, S. and Maassen, O. (2001) Applied Java Patterns, Prentice Hall.*
137. *Stevens, p. and Pooley, R. (2000) Using UML Software Engineering with Objects and Components, Addison-Wesley.*
138. *Szyperski, C. (1998) Component Software: Beyond object-oriented programming, Addison-Wesley.*
139. *Treisman, H. (1994) "How to design a good interface design", Software Magazine, Australia, August, p. 32–36.*
140. *UML (2003) OMG Unified Modeling Language Specification, Version 1.5, OMG.*
141. *UML (2005) "Unified Modeling Language: Superstructure", Version 2.0, formal/05-07-04,* www.uml.org/#UML2.0 *(last accessed February 2007).*
142. *Unhelkar, B. (2003) Process Quality Assurance for UML-based Projects, Addison-Wesley.*
143. *Wegner, p. (1997) "Why interaction is more powerful than algorithms", Communications of the ACM, 40(5) p. 80–91.*
144. *White, S.A. (2004) "Introduction to BPMN",* www.bpmn.org/Documents/Introduction%20to%20BPMN.pdf *(last accessed February 2007).*
145. *White, S.A. (2005) "Using BPMN to model a BPEL process",* www.bpmn.org/Documents/Mapping%20BPMN%20to%20BPEL%20Example.pdf *(last accessed February 2007).*
146. *Whitten, J.L. and Bentley, L.D. (1998) Systems Analysis and Design Methods, 4th edition, McGraw-Hill.*
147. *Windows (2000) The Windows Interface Guidelines for Software Design, MSDN Library, CD-ROM collection, Microsoft.*
148. *Wirfs-Brock, R. and Wilkerson, B. (1989) "Object-oriented design: a responsibilitydriven approach", in OOPSLA '89 Proceedings, SIGPLAN Notices, 10, ACM, p. 71–75.*

149. *Wirfs-Brock, R., Wilkerson, B., and Wiener, L. (1990) Designing Object-Oriented Software, Prentice Hall.*
150. *Wood, J. and Silver, D. (1995) Joint Application Development, 2nd edition, John Wiley.*
151. *Yourdon, E. (1994) Object-Oriented Systems Design: An integrated approach, Yourdon Press.*
152. *Zachman, J.A. (1987) "A framework for information systems architecture", IBM Systems Journal, 3, p. 276–292.*
153. *Zachman, J.A. (1999) "A framework for information systems architecture", IBM Systems Journal, 2/3, p. 454–470.*

Предметный указатель

C

CASE-репозиторий *52*
CASE-средство *52*

W

Web
  приложение *546*, *579*
  сайт *547*
  страница *547*, *579*
  хранилище данных *70*, *111*

A

Авторекурсия *394*
Агрегация *219*, *240*, *295*, *297*, *388*, *766*, *771*
  ExclusiveOwns *295, 389*
  Has *296, 390*
  Member *296, 391*
  Owns *296, 390*
  по значению *771*
  по ссылке *771*
Адаптивность *42*, *111, 145*, *261*
Активация *223*, *240*
Анализ
  риска *91, 158*
  требований *77*
Анкетирование *148*
Аномалия обновления *615*, *642*
Аплет *548*, *579*
Аргумент
  входной *225*
  выходной *225*
  фактический *225*
  формальный *225*
Артефакт *181*, *237*, *240*
Архитектура *111*, *502*
  Core J2EE *263*
  Core PCBMER *265*
  MVC *262*, *330*
  PCBMER *265*, *330*
  PCMEF *265*
  “толстого” клиента *434*
  “тонкого” клиента *434*
  информационных систем, ISA *64*, *110*
  клиент/сервер *431*
  мандатная *139*, *182*
  модель-представление-контроллер *262*
  одноранговая *432*
  ориентированная на сервисы SOA *111*
  программного обеспечения *261*, *330*
  промежуточная *435*
  системная *140*
  трехъярусная *80*, *433*
  управляемая моделями, MDA *93*, *111*
  физическая *431*
  ярусная *433*
Асинхронная коммуникация *268*

req_analysis_system design.indb 803 05.06.2008 14:43:47

Аспект *97*
  статический *213*
Ассоциация *71*, *134*, *168*, *172*, *218*, *240*, *766*
  квалифицированная *371*
  производная *292*, *370*
  рекурсивная *293*
  сингулярная *768*
  тернарная *768*
  унарная *768*
Атрибут *216*, *240*, *760*
  идентифицирующий *216*
  описательный *216*
Аудит *662*, *690*
Аффорданс *558*
  кнопки *560*
  меню *558*
  ссылки *558*

Б

База данных *68*
  активная *435*
  объектно-ориентированная *599*
  объектно-реляционная *599*
  распределенная *431*
  реляционная *599*
Базисный набор данных *89*
Библиотека
  Swing *579*
  инфраструктуры информационных технологий, ITIL *111*
Бизнес-анализ *77*
Бизнес-класс *273*
Бизнес-моделирование *163*
Бизнес-объект *181*, *214*
Бизнес-прецедент *162*, *176*
  использования *142*, *166*, *181*
Бизнес-процесс *167*, *181*
  внутренний *132*
  основной *41*
  сотрудничества, B2B *132*
Бизнес-сервис *433*
Бизнес-транзакция *642*
Блок-схема *211*
Блокировка *642*
  записи *635*
  намерения *635*
  обновления *635*
  разделяемая *635*
  чтения *635*
  эксклюзивная *635*

В

Взаимодействие *223*, *240*
Видимость *362*, *409*, *761*
  закрытая *362*, *761*
  защищенная *362*
  открытая *362*, *761*
  пакетная *362*, *773*
Возможность
  деловая *181*
  повторного использования *144*, *181*
Вопрос
  закрытого типа *146*
  многовариантный *148*
  наводящий *147*
  некорректный *147*
  об альтернативных идеях *147*
  о диаграмме запросов *147*
  о других источниках информации *147*
  о конкретных деталях *147*
  о минимально допустимом решении *147*
  о перспективе *147*
  ориентировочный *149*
  открытого типа *146*
  предвзятый *147*
  проектный *175*
  с ранжированием *149*
  триггерный *152*
Воссоединение потока *211*
Восстановление *68*
Выгрузка *630*
Вызов *319*, *517*
  обратный *319*
Выявление требований *143*

Г

Глоссарий *170*
Границы
  проекта *111*
  системы *157*, *177*
Граф
  операций *315*
Группа *135*
  гарантии качества, SQA *88*
  управления объектами, OMG *53*, *111*
Групповые вычисления *75*

Д

Данные *67*, *111*
  персистентные *68*
Действие *132*, *208*, *231*, *240*
  внешнее *572*
  входное *327*
  выходное *327*
  пользовательское *572*
Действующее лицо *142*, *167*, *181*, *202*
  второстепенное *167*
  первичное *167*
Декомпозиция
  функциональная *73*
Делегирование *269*, *393*, *776*
Деловой регламент *68*
Дескриптор *361*
Дефект *663*, *690*
Деятельность *208*, *240*
Диаграмма *130*, *181*
  бизнес-классов *177*
  бизнес-прецедентов использования *177*
  взаимодействий *223*, *318*
  деятельности *210*, *314*
  иерархии процессов *130*, *181*
  классов *79*, *213*, *220*, *233*
  коммуникации *226*, *318*
  композитной структуры *485*, *487*
  компонентов *233*, *236*
  конечного автомата *231*
  контекстная *157*, *164*, *177*, *182*
  навигационная *577*
  объект-отношение *601*
  отношений логических объектов, ERD *73*, *110*
  последовательностей *223*, *318*
  потоков данных, DFD *73*, *110*, *157*
  прецедентов использования *201*, *204*
  развертывания *233*, *237*, *431*, *468*
  структурная *213*, *233*
Домен *605*, *642*
  атомарный *605*
  именованный *605*

Ж

Жизненный цикл ПО *71*, *111*

З

Зависимость *330*, *462*, *502*, *766*
  по видимости *462*
  по доступу *462*
  по использованию *462*
  циклическая *269*
Загрузка *628*
Задача *98*, *129*, *182*
  комплексная *97*, *98*
Заказчик *176*
Закон
  Деметера *472*, *502*
Заменимость *330*
Замещение *379*, *517*, *779*, *780*
Запись *643*
Знание *67*, *70*, *111*
  подразумеваемое *67*
Значение
  меченое *361*, *409*

И

Иерархия
  агрегации *393*
  наследования *365*
  ослабленная *437*
Издатель *269*
Изменение *690*
Изменчивость ПО *40*
Именная группа *272*

Именование
ассоциаций *292*
классов *281*
роли ассоциации *292*
Индивидуальность *752*
Инкапсуляция *362*, *377*, *761*
Инспекция *88*, *662*, *691*
Инсталляция *82*
Интеграция
информационно-ориентированная *55*
непрерывная *96*
ориентированная на интерфейсы *55*
ориентированная на порталы *55*
ориентированная на процессы *56*
Интеллектуальный анализ данных *70*, *111*
Интервью *146*
неструктурированное *146*
структурированное *146*
Интерфейс *41*, *234*, *240*, *376*, *784*
JDBC *264*, *330*
открытый *322*
пользовательский *435*
предлагаемый *235*, *267*, *331*
требуемый *235*, *267*, *332*
Информационная лавка *69*, *111*
Информация *67*, *111*
производная *370*, *409*
Использование
взаимодействия *406*, *409*
повторное *482*, *503*
каркасов *483*
инструментальных средств *482*
кода *379*
Исследование
бизнес-возможностей *137*
деловой возможности *182*
Итерация *44*, *111*

К

Каркас *483*
Классификатор *235*, *240*
Классификация *71*
динамическая *478*, *782*
множественная *781*
требований *160*
Класс *182*, *240*, *753*, *759*
“людей” *273*
абстрактный *299*, *330*, *783*
архитектурный *483*
ассоциативный *372*, *770*
базовый *483*
вида *214*
вложенный *774*
внутренний *774*
границ *214*
дружественный *366*
использующий интерфейс *303*
источник *372*
компонентный *388*
компонентов *271*
компьютерный *273*
логический *273*
материализованный *373*
местоположения *273*
модельный *214*
нерелевантный *272*
нечеткий *272*
организаций *273*
поведенческий *273*
понятий *273*
посредник *214*, *271*
представления *214*, *270*
прикладной *273*
реализующий интерфейс *303*
релевантный *272*
ресурсов *214*, *271*
событий *273*
составной *388*
сущностей *214*, *270*, *599*, *754*
управляющий *214*, *270*
физический *273*
целевой *372*
Кластеризация *71*
Клиент *81*, *431*, *502*
“толстый” *525*
“тонкий” *525*
браузерный *525*, *579*, *590*
программируемый *525*, *580*

Клонирование *393*
Ключ *216*, *606*, *643*
  альтернативный *606*
  внешний *607*, *642*
  первичный *606*, *643*
  потенциальный *606*
Коллеги *458*
Коллективное владение *96*
Коллекция *226*
Комментарий *358*, *409*
Композиция *219*, *240*, *295*, *297*, *388*
Компонент *41*, *112*, *234*, *235*, *240*, *266*, *330*, *431*, *464*, *502*
  визуальный *52*
Компоновка *77*
  поэлементная *82*
Коннектор *133*
Конструирование *91*
Конструктор *766*
Конструкция
  архитектурная *54*
  концептуальная *41*
Контакт *104*
Контейнер *579*
Контракт *80*
Контроллер *267*
Контрольный список *662*, *691*
Контроль
  качества *666*, *691*
Концепция
  3E *137*
  CRM *181*
  возможностей системы *182*
  решения *139*, *182*
Кооперация *485*, *502*, *754*
  UX
    функциональная *573*
    структурная *577*
  композитная *489*
  субординированная *489*
Кортеж *770*
Кратность *293*
  ассоциации *172*, *768*

Л

Легкость сопровождения *178*
Линейка *534*
Линия
  жизни *223*, *241*, *397*
  связи *438*
Логика
  интегральная *436*
  представления *435*
  приложения *68*
  процедурная *72*

М

Маркер
  видимости *363*
  итеративный *226*
Матрица
  взаимодействия *157*
  зависимости требований *157*
Медиатор *267*
Меню *558*
Мера
  ассоциации *770*
  пространственной сложности *439*
    функциональная *439*
    объектно-ориентированная *439*
    когнитивная *439*
Местоположение *273*
Метакласс *409*
Метамодель *409*
  архитектурная *262*, *330*
Метод *241*, *766*
  CRC *275*
  автоматизированного проектирования и создания программ, CASE *110*
  быстрой разработки ПО, RAD *151*, *155*, *181*
  доступа *473*, *502*
  именных групп *272*
  класса *227*
  модернизации бизнес-процессов, BPR *77*, *110*
  модифицирующий *502*
  общих шаблонных классов *272*
  прецедентов использования *274*

разработки *89*
совместной разработки ПО, JAD *151*
ценностных цепочек *60*
Механизм
расширения *359*
Моделирование
архитектурное *462*
ассоциаций *290*
бизнес-процессов *129*, *143*
взаимодействий *223*, *239*, *318*
деятельности *223*, *314*
интерфейса *302*
классов *213*, *239*
объектов *304*
прецедентов использования *307*
программного обеспечения *52*
реализации *462*
состояний объектов *326*
Модель *52*, *112*
активная *267*
анализа *370*
базы данных *214*
бизнес-классов *142*, *171*, *182*
бизнес-прецедентов *142*
использования *166*, *182*. *307*
бизнес-процесса *129*, *182*
бизнес-требований *177*
взаимодействий *229*, *274*
вытягивания-проталкивания *267*
вытягивания *267*
вычислительная *393*
границ системы *182*
данных *600*, *643*
деятельности *208*, *239*
жизненного цикла *89*
изменения состояний *54*
классов *171*, *271*
конечного автомата *229*, *239*
пассивная *267*
поведения *54*
прецедента использования *201*, *239*
прогностическая *71*
проектная *370*
проталкивания *267*
разработки *89*
ретроспективная *71*
состояния *54*
спецификации *79*
спиральная *45*, *90*
технологической зрелости, CMM *46, 110*
Модуль
архитектурный *45*
“Мозговой штурм” *148*, *152*, *182*

H

Наблюдатель *268*, *502*
Наблюдение *149*
активное *149*
объяснительное *149*
пассивное *149*
Набор
тестовый *691*
Надежность системы *144*, *182*
Назначение приоритетов *158*
Наследование *299*, *376*, *377*, *777*
множественное *387*
интерфейса *377*
класса *379*
кода *379*
множественное *300*, *781*
реализации *781*
ограничивающее *380*
расширяющее *379*
множественное *387*
реализации *366*, *379*
удобное *381*
Невидимость ПО *40*
Нормализация *615*, *643*

O

Обобщение *220*, *241*, *299*, *376*, *392*, *462*, *766*, *777*
Обработка
событий *268*, *319*
Объект *54*, *112*, *753*
внешний *393*
внутренний *393*
временный *756*
данных *135*

класс *753*
компонентный *393*
контроллера *262*
модели *262*
персистентный *435*, *503*, *625*, *643*, *756*
потоковый *132*, *183*
представления *262*
связующий *133*, *183*
составной *393*
экземпляр *753*
Объем
ассоциации *770*
Ограничение *358*, *409*, *462*
системное *177*
Окно
всплывающее *534*
вторичное *534*, *579*
главное *534*, *579*
диалоговое *540*
просмотра деревьев *538*
сообщения *543*
Оператор
alt *404*
break *404*
loop *404*
opt *404*
parallel *404*
взаимодействия *404*
Операции CRUD *323*
Операция *220*, *241*, *755*, *762*
абстрактная *299*, *329*, *783*
вспомогательная *103*
дружественная *366*
модификатора *473*
наблюдателя *473*
Определение
дескриптора *361*, *409*
концептуальных возможностей *139*
требований *141*, *143*
Организация *273*
ISO *110*
Осуществимость
календарная *85*
практическая *85*
проекта *158*
техническая *85*
экономическая *85*
Отношение
ассоциации *168*
иерархическое *161*
родитель-потомок *161*
связи *167*
трассировки *162*
Отображение
классов сущностей *617*
объектно-реляционное *617*, *643*
отношений агрегации *620*
отношений ассоциации *618*
отношений обобщения *622*

П

Пакет *234*, *241*, *266*, *465*, *502*
COTS *110*
вложенный *462*
осведомленный *269*
связей *270*
Панель *534*, *579*
инструментов *544*
навигационная *559*
Папка
с вкладками *542*
Перегрузка *518*, *780*
Передача
сообщений *319*
Переменная *765*
класса *765*
локальная *765*
экземпляра *765*
Перепроектирование процессов *62*
Пересмотр требований *144*
Пересылка *269*
Переход *229*, *231*, *241*
Планирование
проекта *85*, *112*
ресурсов предприятий *103*
системное *57, 85*, *143*
План
тестирования *667*, *691*

Поведение системы *220*
Повторное использование кода *379*
Поддерживаемость ПО *112*
Поддержка
  качества *662, 691*
  услуг
    информационная *55*
    операционная *55*
Подкласс *299*, *379*
Подписчик *268*
Подпроцесс *129*
Подсистема *234*, *266*, *330*
Подход
  BPR *62*
  CRC *330*
  CRM *103*
  ISA *64*
  SWOT *58*
  VCM *60*
  комплексный *275*
  объектно-ориентированный *53*, *74*
  ориентированный на процессы *73*
  структурный *73*
  функциональный *274*
Показатель *86*, *112*
Полиморфизм *330*, *379*, *779*
Полоса *534*, *579*
Пользовательская история *95*
Понятие *273*
Порт *267*, *331*
Порядковый номер
  в иерархии документов *160*
  в категории требований *160*
Посредник *152*
Поставка
  основная *97*
  промежуточная *97*
Потоки вычислений
  альтернативные *211*
  параллельные *211*
Поток
  данных *164*
  действий *134*
  логический *439*
  объектов *316*
  откликов *167*
  рекурсивный *211*
  сообщений *134*
  управления *208*, *241*, *316*
Представление *643*
  реляционное *614*
Прецедент
  возможностей *137*, *142, 183*
  использования *87*, *167*, *201*, *203*, *241*
    действующего лица *308*, *331*
    системы *308*, *331*
  тестовый *87*
Приемочный тест *95*
Принцип
  PCBMER
    уведомления снизу вверх (UNP) *268*, *331*
    явной ассоциации (EAP) *269*, *331*
    зависимости сверху вниз (DDP) *268*, *331*
    именования классов (CNP) *269*, *331*
    исключения циклов (CEP) *269*, *331*
    осведомленного пакета (APP) *331*
    пакета связей (APP) *270*
    связей между соседями (NCP) *269*, *331*
  восходящего уведомления *456*
  заменимости *299*
  исключения циклов *442*
  нисходящей зависимости *442*
  осведомленного пакета *456*
  полиморфизма *299*
  проектирования интерфейса *529*
    настройка *532*
    удобство *533*
    обратная связь *533*
    толерантность *532*
    индивидуализация *532*
    ориентация на пользователя *530*
    согласованность *531*
    эстетичность *533*
  связей между соседями *451*
Проверка *662*, *691*
Провозможность *152*, *183*
Программирование *82*
  аспектно-ориентированное *97*

парное *96*
управляемое событиями *75*
целенаправленное *95*
Программное обеспечение
прикладное *106*
промежуточное *80*
системное *106*
Проектирование
архитектурное *80*, *140*, *470*, *485*
физическое *431*
баз данных *470*
возможностей программного обеспечения *183*
возможностей системы *139*
детализированное *81*, *470*, *485*
пользовательского интерфейса *470*
программного обеспечения *72*, *79*
программ *470*
систем *79, 140*, *178*
требований *78*
Проект
технический *139*
Производительность системы *145*, *183*
Прототип *183*, *393*
одноразовый *152*
эволюционный *152*
Профиль *409*
Процедура
хранимая *173*, *435*, *612*, *644*
Процесс *43*, *112*, *183*
RUP *92*, *111*
атомарный *130*
бизнес-моделирования *57*
жизненного цикла *89*
информационный *167*
клиентский *431*, *470*
модернизации бизнес-процессов *57*
приложения *434*
серверный *431*, *470*
составной *130*
стратегического выравнивания *57*
стратегического планирования *57*
управления информационными ресурсами *57*
ускоренный *45*
Пул *134*, *183*

Р

Развертывание ПО *77*, *82, 503*
Разделение потока *211*
Разработка ПО
на заказ *139, 183*
пакетная *182*
покомпонентная *139*, *182*
ускоренная *95*
через тестирование *95*
посредством тестирования *664*, *691*
Ракурс
системы *260*
взаимодействий *223*
деятельности *208*
реализации *233*
конечных автоматов *229*
статический *270*
прецедентов использования *201*
структуры *213*
Рамка *535*
Раскадровка *569*
работы пользователя *569*
Расширение ПО *112*
Реализация
базы данных *82*
системы *82*
Регистрационный журнал *636*, *643*
Рефакторинг *96*
Риск *91*, *158*, *183*
политический *159*
связанный с изменчивостью *159*
связанный с нарушением норм безопасности *158*
связанный с нарушением целостности баз данных *158*
связанный с процессом разработки *158*
снижения производительности *158*
технический *158*
юридический *159*
Роль *202*, *223*, *241*, *486*
ассоциации *292*

С

Свойства класса
  закрытые *363*
  защищенные *364*
  открытые *364*
Связанность *331*, *503*
  классов *471*, *472*
Связность *332*
  классов *471*
Связывание *442*
Связь
  ассоциативная *770*
  временная *757*
  декомпозиции *130*
  персистентная *756*
Семантика
  декларативная *52*
Сервер *81*, *431*, *503*
  Web *434*, *502*
  приложения *433*, *503*
Сервис *41*, *112*
  приложения *433*
  системный *177*
Сервлет *399*, *580*
Сигнал *318*, *517*
Сигнатура *299*, *332*, *780*
  операции *323*
Система
  EPR *103*
  ER *644*
  OLAP *111*
  OLTP *111*
  адаптивная *440*
  мультимедийная *75*
  обработки знаний *70*
  планирования корпоративных ресурсов, ERP *110*
  распределенная *431*
  управления базами данных *68*
Сквозной контроль *88*, *662*, *691*
Сложность *503*
  алгоритмическая *438*
  задачи *438*
  когнитивная *439*
  структурная *439*
    иерархий *441*
    сетей *440*
Служба
  JMS *264*, *332*
Событие *132*, *223*, *229*, *241*, *273*
  временное *327*
  вызова *327*
  изменения *327*
  сигнальное *327*
Согласованность ПО *40*
Соединительное звено *486*
Создание прототипов *151*
Сообщение *223*, *241*, *318*, *397*, *755*
  асинхронное *397*
  о событии *167*
  о создани объекта *397*
  ответное *398*
  синхронное *397*
Сопровождение ПО
  адаптивное *84*
  совершенствующее *84*
Состояние *229*, *241*, *270*
  составное *231*
Спецификация *79*
  агрегаций *297*
  ассоциаций *292*
  атрибутов классов *282*
  выполнения *223*
  действий *316*
  изменения состояний *325*, *329*
  интерфейса *303*
  классов *280*
  композиций *297*
  обобщений *300*
  объектов *305*
  операций классов *323*
  поведения *270*, *299*, *306*, *329*
  последовательностей сообщений *318*
  потока событий *206*
  прецедента использования *206*
  прецедентов *308*
  состояний *270*, *299*, *326*, *329*

тестовая *665*, *691*
требований *178*, *261*, *332*
Список
выпадающий *542*
Сплетение аспектов *97*
динамическое *98*
статическое *98*
Среда
архитектурная *54*
интегрированная *54*
Срез точек соединения *98*
Ссылка
скрытая *772*
Стандарт
CMM *46*
COBIT *50*, *110*
ISO 9000 *47*
ITIL *48*
JDBC *644*
ODBC *644*
продукции *50*
процесса *50*
Степень ассоциации *768*
Стереотип *357*, *409*
Столбец *644*
Страница
клиентская *548*, *579*
серверная *580*
Стратегия
реализации *139*
Строка *644*
Структура
архитектурная *262*, *330*
композитная *485*, *502*
Субъект *202*
Суперкласс *299*, *379*
Сущность
внешняя *167*
внутренняя *168*
Схема
идентификации требований *160*
логическая *601*
решения *139*
физическая *601*
Сценарий
тестовый *665*, *691*

T

Таблица
реляционная *604*, *606*, *643*
Теория классификации *272*
Тестирование *87*
авторизации *675*
баз данных *674*
методом прозрачного ящика *88*
методом черного ящика *88*
основанное на выполнении программы *88*
по коду *672*
пользовательского интерфейса *673*
по методу “прозрачного ящика” *672*
по методу “черного ящика” *671*
по программному коду *88*
по спецификации *88*, *671*, *691*
поступательное *89*
регрессионное *89*, *671*, *691*
системных ограничений *672*
Техническое задание *175*
Технология
JavaBeans *399*
JSP *399*
Struts *580*
компонентного программного обеспечения *94*
ориентированная на сервисы, SOA *41*
Тип *486*
элементарный *604*, *644*
Точка
верификации *665*
контрольная *636*, *643*
соединения *98*
сохранения *638*, *644*
Траектория
подозрительная *162*
соподчинения *87*
Транзакция *67*, *634*, *644*
Трассируемость *682*, *691*

Требование *112*
к безопасности *178*
к данным *175*, *177*
к интерфейсу *178*
к производительности *178*
к удобству *178*
к функциям *175*
нефункциональное *143*, *144*, *175*, *182*
политическое *178*
тестовое *667*, *691*
функциональное *143*, *175*, *184*
эксплуатационное *178*
юридическое *178*
Требования
перекрывающиеся *158*
противоречивые *158*
Триггер *435*, *503*, *611*, *644*

У

Уведомление *98*
заблаговременное *98*
сквозное *98*
Удобство системы *144*
Узел *210*, *237*, *242*, *431*, *468*, *503*
равноправный *432*
Уникальный идентификатор *160*
Упаковка объектов *75*
Управление
взаимоотношениями с заказчиками *103*
знаниями *70*
изменениями *677*
качеством *47*, *661*
параллельное *68*
программным обеспечением *661*
проектом *661*
решением *48*
требованиями *159*
Уровень *437*, *503*
абстракции *161*
компонентов *267*
контроллера *267*
оперативный *66*
посредника *267*
представления *267*
ресурсов *267*
стратегический *66*
сущностей *267*
Условие
сторожевое *231*, *241*
Усовершенствование *691*
Участник проекта *42*, *112*, *176*

Ф

Формальная экспертиза *88*
Форма *580*
Формулировка
ограничений *143*
ограничения *78*, *184*
сервиса *143*, *184*
услуги *78*
Фрагмент
взаимодействия *404*, *409*
комбинированный *404*
Фрейм *580*
Функция приложения *436*

Х

Холархия *395*, *441*
Холон *394*
Хранилище данных *67*, *69*, *112*

Ц

Целостность
базы данных *67*
ссылочная *604*, *608*, *644*
Цепочка
навигационная *559*
начисления стоимости *102*
Цикл
ассоциаций *292*
сообщений *292*

Ш

Шаблон *175*, *503*
РЕАА *644*
анализа *484*
архитектурный *444*, *484*
проектирования *444*, *484*

цепочка обязанностей *451*
Единица работы *626*
Загрузка по требованию *626*, *643*
Издатель-Подписчик *453*
Коллекция объектов *625*, *643*
Наблюдатель *453*, *503*
Преобразователь данных *626*, *643*
Посредник *458*, *503*
Абстрактная фабрика *448*, *503*
Фасад *504*
Цепочка обязанностей *504*
технического задания *175*
Шлюз *133*

Э

Элемент
UX *572*
Этап
анализа *76*
определения требований *163*
планирования *84*
проектирования *77*
реализации *77*
спецификации требований *163*
тестирования *84*
эксплуатации и сопровождения *77*, *83*
Эффективность системы *145*

Я

Язык программирования
прототипный *754*
Язык
BPEL *132*, *181*
BPMN *129*, *181*
Level1 SQL *632*
Level2 SQL *632*
Level3 SQL *633*
Level4 SQL *633*
Level5 SQL *634*
UML *53*
Ярус *433*, *504*
Web *264*
бизнеса *264*
интеграции *264*
информационной системы предприятия *264*
клиента *264*
представления *264*
со стороны сервера *264*

Научно-популярное издание

Лешек А. Мацяшек

АНАЛИЗ И ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ С ПОМОЩЬЮ UML 2.0, 3-Е ИЗД.

Литературный редактор	*И.А. Попова*
Верстка	*Л.В. Чернокозинская*
Художественный редактор	*В.Г. Павлютин*
Корректор	*Л.А. Гордиенко*

ООО "И.Д. ВИЛЬЯМС"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 03.06.2008. Формат 70x100/16.
Гарнитура Minion. Печать офсетная.
Усл. печ. л. 65,79. Уч.-изд. л. 46,75.
Тираж 1000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

req_analysis_system design.indb 816 05.06.2008 14:43:49