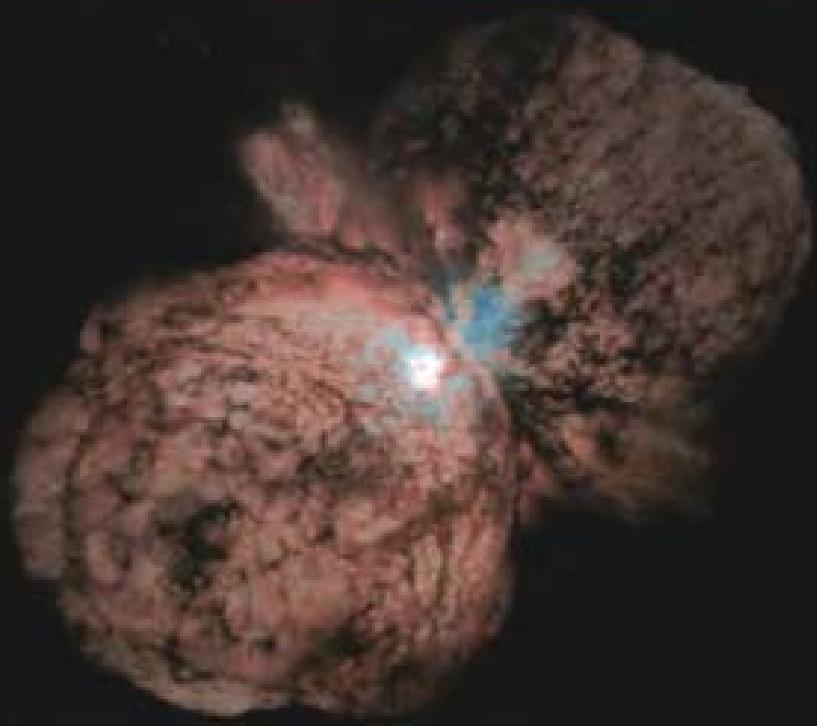


БЫСТРАЯ РАЗРАБОТКА ПРОГРАММ

Принципы, примеры, практика



Scanned by PPK

AGILE SOFTWARE DEVELOPMENT

Principles, Patterns, and Practices

ROBERT C. MARTIN

with contributions

by James W. Newkirk and Robert S. Koss



Pearson Education, Inc.
Upper Saddle River, NJ 07458

БЫСТРАЯ РАЗРАБОТКА ПРОГРАММ

Принципы, примеры, практика

РОБЕРТ К. МАРТИН

а также

Джеймс В. Ньюкирк и Роберт С. Косс



Москва • Санкт-Петербург
2004

ББК 32.973.26-018.2.75

М29

УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией С.Н. Тригуб

Перевод с английского А.П. Сергеева, Т.А. Шамренко

Под редакцией А.П. Сергеева

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:

info@williamspublishing.com, <http://www.williamspublishing.com>

Мартин, Роберт С.

М29 Быстрая разработка программ: принципы, примеры, практика. : Пер. с англ. — М. : Издательский дом "Вильямс", 2004. — 752 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0558-3 (рус.)

Роберт Мартин в соавторстве с Джеймсом Ньюкирком и Робертом Коссом предлагает вниманию читателей книгу о различных методиках быстрого (и даже экстремального) программирования. Изложение начинается с обзора основных понятий экстремального программирования и завершается готовыми программами, применяемыми на практике. В каждой главе приведены примеры кода на языках программирования Java и C++.

Книга будет полезной руководителям групп программистов, нацеленных на быструю разработку коммерческих программных проектов, характеризующихся высоким качеством и минимальной себестоимостью.

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 фоторепродукцию и запись на магнитный носитель, если на это нет письменного разрешения издательства Prentice Hall, Inc.

Authorized translation from the English language edition published by Prentice Hall, Ptr., Copyright © 2003 by Pearson Education, Inc.

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

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2004

ISBN 5-8459-0558-3 (рус.)

ISBN 0-13-597444-5 (англ.)

© Издательский дом "Вильямс", 2004

© Pearson Education, Inc., 2003

Оглавление

Часть I. Быстрая разработка ПО	35
Глава 1. Быстрая разработка ПО	37
Глава 2. Основы экстремального программирования	49
Глава 3. Планирование	62
Глава 4. Тестирование	70
Глава 5. Рефакторинг	81
Глава 6. Пример из практики программирования	97
Часть II. Быстрое проектирование	147
Глава 7. Быстрое проектирование. Краткое введение	149
Глава 8. Принцип персональной ответственности	164
Глава 9. Принцип открытия-закрытия	170
Глава 10. Принцип подстановки Лискоу	185
Глава 11. DIP: принцип инверсии зависимостей	206
Глава 12. ISP: принцип отделения интерфейса	218
Часть III. Практическое занятие: программа расчета зарплаты	233
Глава 13. Шаблоны Command и Active Object	239
Глава 14. Шаблоны Template Method и Strategy: наследование и делегирование	252
Глава 15. Шаблоны Facade и Mediator	268
Глава 16. Шаблоны Singleton и Monostate	273
Глава 17. Объект Null	287
Глава 18. Практическое занятие: программа по расчету зарплаты (первая итерация)	292
Глава 19. Практическое занятие: реализация программы по расчету зарплаты	309
Часть IV. Упаковка программы расчета зарплаты	365
Глава 20. Принципы упаковки программных проектов	366
Глава 21. Шаблон Factory	392
Глава 22. Практическое занятие: программа расчета зарплаты (часть 2)	401

Часть V. Практическое занятие: моделирование метеостанции	423
Глава 23. Шаблон Composite	424
Глава 24. Обратно к шаблонам: Observer	428
Глава 25. Некоторые примеры из практики: шаблоны Abstract Server, Adapter и Bridge	453
Глава 26. Шаблоны Proxy и Stairway to Heaven: управление API от независимых производителей	466
Глава 27. Практическое занятие: метеостанция	501
Часть VI. Практическое занятие: система ETS	543
Глава 28. Шаблон Visitor	544
Глава 29. Шаблон State	580
Глава 30. Схема ETS	609
Часть VII. Приложения	643
Приложение А. UML-запись I: CGI-пример	644
Приложение Б. UML-запись II: STATMUX	678
Приложение В. Сатира на деятельность двух компаний	705
Приложение Г. Исходный код — это уже проект	723
Предметный указатель	738

Содержание

Предисловие	20
Введение	22
Демонология: весь смысл — в деталях	23
Немного истории	23
Сотрудничество с Бучем	24
Экстремальное программирование и его влияние на практику разработки программных проектов	25
Участие Бека в проекте	25
Структура книги	27
Для кого предназначена эта книга	28
Если вы разработчик...	28
Если вы менеджер или бизнес-аналитик...	29
Если вы хотите изучить UML-язык...	29
Если вы хотите изучить шаблоны проектирования...	29
Если вы хотите изучить принципы объектно-ориентированного программирования...	29
Если вы хотите изучить методы быстрой разработки...	29
Если хочется повеселиться...	29
Благодарности	30
Источники	31
Об авторах	32
Роберт К. Мартин (Robert C. Martin)	32
Джеймс В. Ньюкирк (James W. Newkirk)	32
Роберт С. Косс (Robert S. Koss)	32
Перечень шаблонов проектирования	33
Часть I. Быстрая разработка ПО	35
Глава 1. Быстрая разработка ПО	37
Альянс специалистов по быстрой разработке ПО	39

Манифест альянса специалистов по быстрой разработке ПО	39
Принципы быстрой разработки ПО	44
Резюме	47
Литература	48
Глава 2. Основы экстремального программирования	49
Методика экстремального программирования	50
Заказчик — член команды разработчиков	50
Пользовательские истории	51
Короткие рабочие циклы	51
Приемочные тесты	52
Парное программирование	53
Методика пробных испытаний (тест-драйв)	53
Коллективное владение	54
Непрерывная интеграция	55
Равномерная работа	55
Открытая рабочая среда	56
Игра в планирование	56
Простая структура проекта	57
Рефакторинг	58
Метафора	59
Резюме	60
Литература	60
Глава 3. Планирование	62
Предварительное исследование	63
Соединение, разделение и скорость проектирования	63
Планирование выпусков программ	65
Планирование рабочих циклов	65
Планирование выполняемых задач	66
На полпути	67
Циклы	68
Резюме	68
Литература	68
Глава 4. Тестирование	70
Методика пробных испытаний (тест-драйв)	71
Пример проекта, основанного на результатах тестирования	72
Изоляция теста	73
Вероятностная изоляция	75
Приемочные тесты	76

Пример приемочного теста	77
Вероятностная архитектура	79
Резюме	80
Литература	80
Глава 5. Рефакторинг	81
Генерирование простых чисел: простой пример использования рефакторинга	82
Заключительный просмотр	91
Резюме	95
Литература	96
Глава 6. Пример из практики программирования	97
Игра в боулинг	98
Резюме	144
Часть II. Быстрое проектирование	147
Признаки плохого проекта	147
Принципы быстрой разработки ПО	148
Принципы и признаки плохого проекта	148
Литература	148
Глава 7. Быстрое проектирование. Краткое введение	149
Процесс разработки ПО	150
Признак плохого проекта	151
Чем вызвано “загнивание” программы?	154
“Загнивание” программ исключается в случае применения технологий быстрой разработки ПО	155
Пример “загнивающей” программы	155
Первоначальный проект	155
Изменение требований	157
Последний шаг, он трудный самый. . .	158
В ожидании изменений	159
Пример быстрого проектирования для программы Copy	160
Как разработчики программ, создаваемых в соответствии с технологией быстрого проектирования, узнали о том, что нужно делать?	161
Создание максимально “прозрачного” проекта	162
Резюме	162
Литература	163

Глава 8. Принцип персональной ответственности	164
Принцип персональной ответственности	165
Определение ответственности	167
Разделение спаренных ответственостей	168
Устойчивость	168
Резюме	169
Литература	169
Глава 9. Принцип открытия-закрытия	170
Принцип открытия-закрытия	171
Описание	171
Ключ к решению проблемы — абстракция	172
Приложение Shape	173
Нарушение принципа открытия-закрытия	173
Соответствие принципу открытия-закрытия	176
Не все так хорошо	178
Предположение и “естественная” структура	178
“Забрасывание удочек”	179
Использование абстракции в целях явного закрытия	180
Использование подхода, управляемого данными, в целях достижения завершения	182
Резюме	184
Литература	184
Глава 10. Принцип подстановки Лискоу	185
Принцип подстановки Лискоу	186
Простой пример нарушения принципа LSP	186
Классы Square и Rectangle: примеры менее заметных нарушений	188
Реальная проблема	191
Адекватность модели	192
Неправильное поведение объекта	193
Проектирование на контрактной основе	193
Спецификация контрактов при проведении модульных тестов	195
Реальный пример	195
Мотивация	195
Проблема	197
Решение, которое не согласуется с принципом LSP	199
Решение, не нарушающее принцип LSP	199
Факторинг вместо вычисления производных классов	200
Эвристика и соглашения	204
Вырожденные функции в производных классах	204

Удаление исключений из производных классов	204
Резюме	205
Литература	205
Глава 11. DIP: принцип инверсии зависимостей	206
Принцип DIP: принцип инверсии зависимостей	206
Разбиение на слои	207
Инверсия отношений собственности	209
Зависимость от абстракций	209
Простой пример	210
Поиск основополагающих абстракций	212
Пример с электроплиткой	213
Динамический и статический полиморфизм	215
Резюме	216
Литература	217
Глава 12. ISP: принцип отделения интерфейса	218
Проблемы, возникающие при формировании интерфейса	218
Отделение клиентов равнозначно изоляции интерфейсов	220
Обратное влияние клиентов на интерфейсы	220
ISP: принцип отделения интерфейса	222
Интерфейсы классов и объектов	222
Отделение путем делегирования	222
Отделение с помощью множественного наследования	223
Пример пользовательского интерфейса: ATM	224
Поливалентность и моновалентность	230
Резюме	232
Литература	232
Часть III. Практическое занятие: программа расчета зарплаты	233
Элементарная спецификация системы расчета зарплаты	234
Упражнение	235
Вариант использования 1: добавление сведений о новом сотруднике	235
Вариант использования 2: удаление записи, соответствующей сотруднику	236
Вариант использования 3: отсылка карточки табельного учета	236
Вариант использования 4: отсылка торговой квитанции	237
Вариант использования 5: отсылка сообщения о плате за членство в организации	237

Вариант использования 6: изменение сведений о работнике	237
Вариант использования 7: выполнение программы расчета зарплаты для одного платежного дня	238
Глава 13. Шаблоны Command и Active Object	239
Простые команды	240
Транзакции	242
Физическое и временное разделение связей	244
Временное разделение	244
Отмена действия (UNDO)	244
Шаблон Active Object	245
Резюме	251
Литература	251
Глава 14. Шаблоны Template Method и Strategy: наследование и делегирование	252
Шаблон Template Method	254
Не злоупотребляйте шаблонами	257
Сортировка методом “пузырька”	258
Шаблон Strategy	261
И снова сортировка	264
Резюме	267
Литература	267
Глава 15. Шаблоны Facade и Mediator	268
Шаблон Facade	268
Шаблон Mediator	270
Резюме	272
Литература	272
Глава 16. Шаблоны Singleton и Monostate	273
Шаблон Singleton	274
Преимущества шаблона Singleton	276
Недостатки шаблона Singleton	276
Пример использования шаблона Singleton	276
Шаблон Monostate	278
Преимущества шаблона Monostate	280
Недостатки шаблона Monostate	280
Пример использования шаблона Monostate	281
Резюме	286
Литература	286

Глава 17. Объект Null	287
Резюме	291
Литература	291
Глава 18. Практическое занятие: программа по расчету зарплаты (первая итерация)	292
Введение	293
Спецификация	293
Анализ по методу вариантов использования	294
Добавление записи о работнике	295
Удаление записи о работнике	297
Проводка карточки табельного учета	297
Проводка торговых квитанций	298
Проведение оплаты за членство в организации	299
Изменение сведений о работнике	300
Платежный день	302
Первые итоги	304
В поисках базовых абстракций	304
Абстракция, связанная с календарным планированием	305
Методы платежей	307
Взносы	307
Резюме	308
Литература	308
Глава 19. Практическое занятие: реализация программы по расчету зарплаты	309
Добавление записей о новых работниках	310
База данных Payroll	312
Использование шаблона Template Method при добавлении записей о работниках	314
Удаление записей о работниках	317
Глобальные переменные	319
Ведение карточек табельного учета, торговых квитанций и поддержка сведений об оплате услуг	319
Изменение записей о работниках	328
Внесение изменений в классификацию	332
Некоторые итоги	338
Оплата труда работников	344
Следует ли разработчикам принимать бизнес-решения?	345
Оплата труда штатных сотрудников	346
Оплата труда работников с почасовой ставкой	349

Периоды выплат: трудности проектирования	353
Основная программа	361
База данных	362
Резюме по проекту расчета зарплаты	363
Историческая справка	363
Источники	364
Литература	364
Часть IV. Упаковка программы расчета зарплаты	365
Глава 20. Принципы упаковки программных проектов	366
Пакеты и проектирование	367
Степень детализации: принципы сцепления пакетов	368
Принцип эквивалентности повторно применяемых выпусков (REP, Reuse-Release Equivalence Principle)	368
Принцип общего повторного использования (CRP, Common-Reuse Principle)	370
Принцип общего закрытия (CCP, Common-Closure Principle)	371
Сцепление пакетов: резюме	372
Устойчивость: принципы связывания пакетов	372
Принцип ациклических зависимостей (ADP, Acyclic-Dependencies Principle)	372
Еженедельные выпуски	373
Исключение циклов зависимостей	374
Проявление циклического эффекта в графе, моделирующем взаимосвязи между пакетами	376
Разбиение цикла	378
Флуктуации	378
Проектирование “сверху вниз”	379
Принцип устойчивых зависимостей (SDP, Stable-Dependencies Principle)	380
Устойчивость	380
Метрики устойчивости	381
Не все пакеты должны быть устойчивыми	383
Область применения высокоуровневого проектирования	385
Принцип устойчивости абстракций (SAP, Stable-Abstractions Principle)	386
Оценки абстракций	386
Главная последовательность	387
Оценка расстояния до главной последовательности	389
Резюме	391

Глава 21. Шаблон Factory	392
Цикл зависимости	395
Замещаемые фабрики	396
Применение фабрик для формирования схем тестов	398
Преимущества, связанные с использованием фабрик	400
Резюме	400
Литература	400
Глава 22. Практическое занятие: программа расчета зарплаты (часть 2)	401
Структура пакетов и запись	402
Применение принципа общего закрытия (CCP, Common-Closure Principle)	404
Принцип эквивалентности повторного использования и выпусков (REP, Reuse-Release Equivalency Principle)	406
Связывание и инкапсуляция	409
Метрики	411
Применение метрик в программе по расчету зарплаты	413
Фабрики объектов	416
Фабрики объектов для TransactionImplementation	417
Инициализация фабрик	417
Переосмысление ограничений на связывание	418
Заключительная структура пакетов	419
Резюме	422
Литература	422
Часть V. Практическое занятие: моделирование метеостанции	423
Глава 23. Шаблон Composite	424
Пример: составные команды	426
Множественное и единственное число	427
Глава 24. Обратно к шаблонам: Observer	428
Электронные часы	429
Резюме	449
Применение диаграмм в данной главе	449
Шаблон Observer	449
Каким образом шаблон Observer учитывает принципы ООП?	451
Литература	452

Глава 25. Некоторые примеры из практики: шаблоны Abstract Server, Adapter и Bridge	453
Шаблон Abstract Server	454
Кому принадлежит интерфейс?	455
Шаблон Adapter	456
Классовая форма шаблона Adapter	457
Модемная проблема, шаблон Adapter и принцип LSP	457
Шаблон Bridge	462
Резюме	464
Литература	465
Глава 26. Шаблоны Proxy и Stairway to Heaven: управление API от независимых производителей	466
Шаблон Proxy	467
Применение шаблона Proxy в приложении “тележка для закупок”	472
Резюме: шаблон Proxy	487
Обработка баз данных, промежуточного ПО и других интерфейсов от сторонних производителей	488
Шаблон Stairway to Heaven	490
Пример использования шаблона Stairway to Heaven	492
Другие шаблоны, применяемые при работе с базами данных	498
Резюме	500
Литература	500
Глава 27. Практическое занятие: метеостанция	501
Фирма Cloud Company	502
Программа WMS-LC	504
Разработка проекта Nimbus-LC	505
Резюме	535
Литература	535
Обзор требований к Nimbus-LC	536
Требования к применению	536
24-часовая история	536
Установка пользователем	536
Административные требования	537
Примеры применения Nimbus-LC	537
Исполнители	537
Варианты использования	538
История измерений	538
Установка	538
Администрирование	538

План выпуска версий Nimbus-LC	539
Введение	539
Версия I	539
Поставляемые модули	540
Версия II	540
Реализованные варианты использования	541
Риски	541
Поставляемые модули	541
Версия III	541
Реализованные варианты использования	541
Риски	542
Поставляемые модули	542
Часть VI. Практическое занятие: система ETS	543
Глава 28. Шаблон Visitor	544
Семейство Visitor в шаблонах проектирования	545
Шаблон Visitor	545
Представление шаблона Visitor в виде матрицы	549
Шаблон Acyclic Visitor	549
Шаблон Acyclic Visitor подобен разреженной матрице	554
Применение Visitor в генераторах отчетов	554
Другие применения шаблона Visitor	562
Шаблон Decorator	562
Множественные декораторы	567
Шаблон Extension Object	568
Резюме	579
Напоминание	579
Литература	579
Глава 29. Шаблон State	580
Автомат с конечным числом состояний	581
Методики реализации	583
Вложенные операторы Switch/Case	583
Интерпретация таблиц переходов	587
Шаблон State	589
Компилятор машины состояний (SMC, State-Machine Compiler)	593
Область применения машин состояний	596
Политики высокоуровневых приложений для GUI	597
Контроллеры, обеспечивающие взаимодействие GUI	598

Распределенная обработка	600
Резюме	600
Листинги	601
Применение табличной интерпретации: turnstile.java	601
Генерация с помощью SMC и другие вспомогательные файлы: turnstile.java	603
Глава 30. Схема ETS	609
Введение	610
Обзор проекта	610
Начало работы	612
Схема	613
Команда образца 1994 года	613
Срок завершения работ	614
Стратегия	614
Результаты	615
Разработка схемы	617
Общие требования к вычислительным приложениям	617
Разработка схемы оценивания	620
Использование шаблона Template Method	624
Однократная запись цикла	625
Общие требования к разрабатываемым приложениям	629
Разработка схемы поставки версий	630
Архитектура мастера задач (Taskmaster)	637
Резюме	642
Литература	642
Часть VII. Приложения	643
Приложение А. UML-запись I: CGI-пример	644
Система записи на курсы: описание проблемы	645
Исполнители	647
Варианты использования	647
Модель предметной области	653
Архитектура	659
Абстрактные классы и интерфейсы в последовательных диаграммах	674
Резюме	677
Литература	677

Приложение Б. UML-запись II: STATMUX	678
Определение статистического мультиплексора	678
Программная среда	679
Ограничения режима реального времени	680
Служба обработки прерываний входа	681
Стереотипы в окнах списка	682
Состояния и внутренние переходы	684
Переходы между состояниями	685
Вложенные состояния	685
Служба обработки прерываний выхода	687
Протокол связи	689
Диаграммы действий	693
Диаграммы объектов	697
Активные объекты	698
Интерфейсы	698
Диаграммы сотрудничества	700
Диаграммы последовательности сообщений	703
Резюме	704
Литература	704
Приложение В. Сатира на деятельность двух компаний	705
Крах проекта, фирма Rufus, Inc.	705
Проект Альфа, фирма Rupert Industries	715
Приложение Г. Исходный код — это уже проект	723
Определение программного проекта	724
Послесловие	735
Предметный указатель	738

Предисловие

Я пишу настоящее предисловие непосредственно после завершения основной версии проекта Eclipse, сопровождаемого открытым исходным кодом. Наш рецепт успеха прост. Ключевым фактором успешного производства программного продукта являются люди, а не процессы. Работайте с теми, кто всеми силами стремится создать хорошую программу, руководите разработкой, организуя нетрудоемкие процессы, которые можно адаптировать с учетом любых разработчиков.

После рассмотрения основных принципов работы моих разработчиков выяснилось, что к основному виду собственной деятельности они относят программирование. Разработчики не только создают код, но и непрерывно переосмысливают его значение, что позволяет получить корректное представление о разрабатываемой системе. Проверка проекта на основе исходного кода обеспечивает обратную связь, жизненно необходимую для гарантирования корректности проекта в целом. При этом программисты из моей команды осознают необходимость применения шаблонов, рефакторинга, тестирования, поэтапной разработки и поставки программы, частого формирования промежуточных версий программных модулей, а также использования других методов, которые относятся к экстремальному программированию (ЭП). Эти и другие подобные способы разработки программ способствовали изменению традиционных взглядов на суть деятельности программистов.

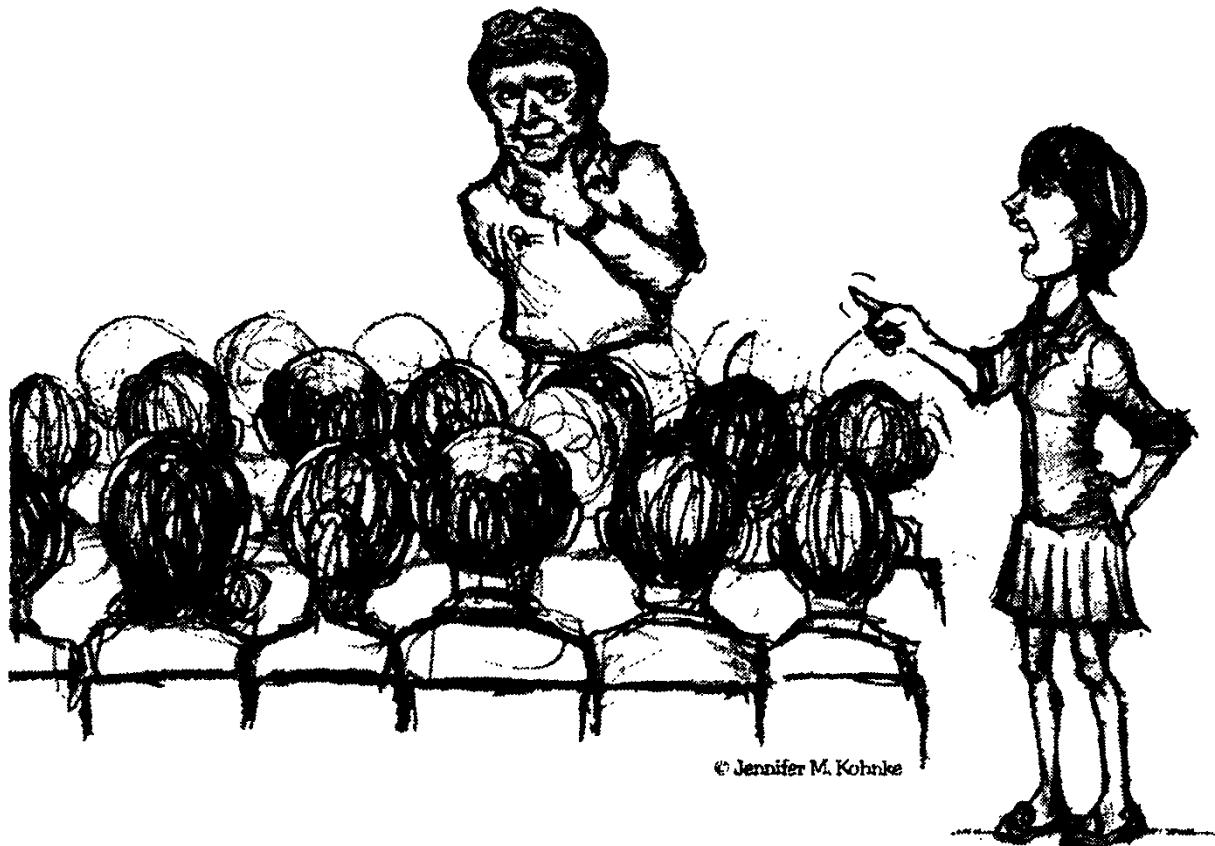
В процессе быстрой разработки программ отсутствуют какие-либо формальные приемы, а также не требуется обширная документация, сопровождающая проект. Но в то же время процесс подобной разработки является весьма насыщенным, предусматривающим интенсивное использование современных методов программирования. Именно подробному описанию этих методов посвящена настоящая книга.

Роберт уже давно и активно работает в среде приверженцев объектно-ориентированного программирования. Круг его интересов включает вопросы применения языка C++, шаблонов и принципов объектно-ориентированного проектирования в целом. Он был первопроходцем в деле пропаганды преимуществ экстремального программирования и методов быстрой разработки. В данной книге вы найдете описание полного спектра этих методов, а также конкретные примеры программного кода.

Эрих Гамма,
Object Technology International

Посвящается Энн Мэйн (Ann Mane), Анжеле (Angela), Мики (Micah), Джине (Gina), Джастину (Justin), Анжелике (Angelique), Мэтту (Matt) и Алексис (Alexis).

Введение



© Jennifer M. Kohnke

— Но Боб, вы же сказали, что книга будет готова еще в прошлом году.

Клаудия Фрес, UML World, 1999

Быстрая разработка предусматривает способность к скоростному созданию программ в условиях непрерывно изменяющихся требований. В связи с этим приходится использовать методы, гарантирующие необходимую организацию работы и обратную связь с разработчиком. При этом следует применять методы проектирования, которые обеспечивают гибкость и простоту дальнейшей эксплуатации ПО. Весьма желательно, чтобы разработчики имели представление о шаблонах проектирования, позволяющих использовать принципы быстрой разработки ПО в целях решения определенных проблем. В данной книге предпринята попытка (довольно успешная) связать все упомянутые концепции в единое целое.

Внимательный читатель найдет здесь описание принципов, шаблонов и методов быстрой разработки ПО, а также примеры их применения на практике. И что более важно, конкретные примеры представлены не как выполненные задачи, а скорее в виде разрабатываемых проектов. Благодаря подобному подходу вы наглядно увидите, как ошибаются разработчики проектов, каким образом происходит процесс идентификации ошибок и их дальнейшее исправление. Также

наглядно демонстрируется процесс решения сложных задач и процедуры анализа возникших противоречий и компромиссов. При этом вы сможете наблюдать сам процесс разработки и выполнения программных проектов “живьем”.

Демонология: весь смысл — в деталях

Эта книга включает *большой объем* кода на языках Java и C++. Я надеюсь, что вы внимательно прочтете этот код, поскольку в значительной степени именно он является *ключевым моментом* в изучении материала книги.

В тексте повторяется использование одного и того же шаблона. Его суть раскрывается в процессе рассмотрения нескольких конкретных примеров, причем соответствующий программный код может быть кратким или достаточно объемным. Рассмотрению каждого конкретного примера предшествует специальный материал, цель которого заключается в том, чтобы подготовить читателя к дальнейшему рассмотрению самого программного кода. Например, рассмотрению программы, реализующей расчет зарплаты, предшествуют главы, описывающие объектно-ориентированные принципы и шаблоны проектирования, которые используются в процессе создания программного кода.



Книга начинается с рассмотрения методов и процессов разработки программ. Данное обсуждение детализируется путем обсуждения небольших конкретных случаев и примеров. Затем в книге рассматривается проектирование и его принципы, некоторые из шаблонов проектирования, а потом — отдельные принципы проектирования, обеспечивающие создание программных пакетов, а также дополнительные шаблоны.

Данная книга носит сугубо технический характер, поэтому детали имеют определяющее значение.

Немного истории

Более шести лет тому назад я написал книгу *Проектирование объектно-ориентированных приложений на языке C++ с использованием метода Буча* (*Designing Object-oriented C++ Applications using the Booch Method*). Для меня эта книга стоила больших усилий, и я остался доволен полученными результатами, а также большими объемами продаж.

Данная книга изначально была задумана как второе издание “Проектирования...”, но итогом переработки явилось существенное изменение исходного издания. В частности, в данную книгу попали всего лишь три главы из предыдущей книги, которые были серьезно переработаны. Остались практически без изме-

нений содержание, концепция и большинство практических занятий из прежней книги. Но за прошедшие шесть лет автор существенно пополнил и расширил свои знания о проектировании и разработке ПО. Все это получило адекватное отображение в новой книге.

Да, много воды утекло с момента появления первого издания! *Проектирование...* появилось на свет накануне эпохи бурного развития Internet. С того времени количество аббревиатур, сопровождающих нас в повседневной жизни, удвоилось. Мы свободно оперируем терминами Design Patterns (шаблоны проектирования), Java, EJB, RMI, J2EE, XML, XSLT, HTML, ASP, JSP, Servlets (сервлеты), Application Servers (серверы приложений), ZOPE, SOAP, C#, .NET и т.д. Позвольте заметить, что “идти в ногу со временем” в процессе подготовки текста данной книги оказалось весьма непросто!

Сотрудничество с Бучем

В 1997 году ко мне обратился Гради Буч (Grady Booch) с просьбой помочь ему подготовить третье издание поразительно успешной книги *Объектно-ориентированный анализ и проектирование приложений* (*Object-Oriented Analysis and Design with Applications*). Прежде мне приходилось работать с Гради над несколькими проектами, и я с большим интересом знакомился с его различными работами, в том числе посвященными описанию UML-диаграмм. Поэтому я с радостью принял его предложение, попросив о помощи своего хорошего друга Джеймса Ньюкирка.

За последующие два года Джеймс и я совместно написали несколько глав для книги Буча. Конечно, можно было бы сделать и больше, но “нельзя объять необъятное...”. Более того, книга, которую вы сейчас держите в руках, была в то время, по сути, просто вторым изданием *Проектирования...*, и мне не слишком хотелось продолжать работу над ней. Всегда хочется сказать нечто новое, носящее революционный характер.

К сожалению, этой книге Буча не суждено было увидеть свет. Трудно выкроить время для написания книги в череде будней. В горячие дни работы над проектом “.com” это было практически невозможно. С течением времени Гради переориентировался на работу с новыми проектами. Поэтому наш совместный проект попал в “долгий ящик”. В конце концов, я попросил у Гради и руководства издательства Addison-Wesley разрешение включить главы, написанные мною и Джеймсом, в эту книгу. Они любезно согласились. Поэтому несколько примеров и глав, посвященных описанию UML-диаграмм, берут начало именно из этого источника.

Экстремальное программирование и его влияние на практику разработки программных проектов

Идея экстремального программирования (XP, Extreme Programming), подвергшая сомнению наши изначальные представления о разработке ПО, возникла в конце 1998 года. Следует ли разрабатывать множество UML-диаграмм, прежде чем приступить к написанию какого-либо программного кода, или же лучше воздержаться от этого и начинать реальную работу? Нужно ли создавать множество документов с комментариями, описывающими проект, или же имеет смысл попытаться придать выразительность самому коду, в результате чего будет устранена необходимость в разработке вспомогательных документов? Следует ли практиковать метод парного программирования? Нужно ли разрабатывать тесты перед созданием программного кода? Как вообще следует поступать в тех или иных ситуациях?

Как по мне, так революция в программировании произошла вовремя. Во второй половине 90-х годов прошлого столетия компания Object Mentor предоставила услуги нескольким компаниям при решении вопросов, возникающих в процессе объектно-ориентированного (ОО) проектирования и управления проектами. Мы же всего лишь помогали компаниям *довести* работу над проектами *до конца*. При этом мы делились с членами команд разработчиков своими взглядами на процесс разработки, а также внедряли собственные методы. К сожалению, эти взгляды и методы не были зафиксированы в письменном виде. Заказчики получали все необходимые сведения в устной форме.

В 1998 году я осознал, что следует в письменном виде сформулировать применяемый процесс и методы разработки, что позволит в более эффективной форме сформулировать их для наших заказчиков. Это побудило меня к написанию большого количества статей, посвященных рассмотрению процесса разработки проектов, которые были опубликованы в журнале *C++ Report*¹. Но этим статьям не хватало широты взглядов, а также отсутствовала интересная подача материала. Они были информативны и даже занимателны, но вместо того, чтобы систематизировать методы и взгляды, которые применялись при разработке проектов, они выполняли функции “пропаганды” ценностей, которые навязывались мне на протяжении десятилетий. На этот недостаток и указал мне Кен Бек (Kent Beck).

Участие Бека в проекте

В конце 1998 года, в процессе усердной работы над кодированием процесса Object-Mentor, я случайно наткнулся на работу Кена по экстремальному программированию (XP). Фрагменты статьи были размещены на Web-узле Уорда

¹Эти статьи можно найти в разделе “publications” на Web-узле <http://www.objectmentor.com>. Первые три из них называются “Iterative and Incremental Development” (I. II. III). Последняя статья называется “C.O.D.E Cuddled Object Development procEss”.

Каннингема (Ward Cunningham) под названием *wiki*², где они были представлены вперемешку с работами других авторов. С некоторым трудом мне удалось понять суть того, о чем говорил Кен. Я был заинтригован, но настроен весьма скептически. Некоторые из аспектов, которые были отображены в идеологии экстремального программирования, в точности совпадали с моей концепцией процесса разработки. Другие аспекты, такие как отсутствие четкого описания этапов проектирования, напротив, привели меня в некоторое замешательство.

Более сумасшедших обстоятельств разработки ПО, чем те, с которыми приходилось сталкиваться мне и Кену, трудно было бы придумать. Кен был признанным консультантом в области языка Smalltalk, а я — консультантом по языку C++. И этим двум мирам было затруднительно найти общие точки пересечения. Между ними находилась пропасть парадигм Кана (Kuhn)³.

При других обстоятельствах я никогда бы не попросил Кена написать статью для журнала *C++ Report*. Но наши представления о процессе разработки совпали, и именно это помогло преодолеть определенный барьер. В феврале 1999 года я встретился с Кеном в Мюнхене на конференции, посвященной объектно-ориентированному программированию. Он читал лекцию на тему XP-программирования в зале как раз напротив аудитории, в которой я читал лекцию о принципах объектно-ориентированной разработки. Поскольку я не имел возможности послушать лекцию Кена, я разыскал его во время обеда. Мы обсуждали проблемы экстремального программирования, и я попросил его написать статью для журнала *C++ Report*. Статья получилась великолепной. В ней описывался случай, в котором Кену и его сотруднику удалось коренным образом изменить ранее разработанный проект за считанные часы.

За несколько месяцев мне удалось преодолеть собственные страхи в отношении экстремального программирования. Больше всего я опасался реализации процесса, когда отсутствует четко выраженный этап предварительного проектирования. Я обнаружил, что именно из-за этого существенно тормозилась разработка программ. Чувство ответственности перед заказчиками заставляет меня убеждать всех заинтересованных лиц в том, что важность проектирования заслуживает потраченного на него времени?

Со временем я осознал, что сам не слишком часто использовал эту методику на практике. Даже в моих статьях и книгах, посвященных проектированию, диаграмм Буча и UML-диаграммам, я всегда использовал код в качестве метода подтверждения смысловой значимости диаграмм. Во всех своих консультациях с заказчиками я обычно тратил час или два на то, чтобы помочь им начертить

²<http://c2.com/cgi/wiki>. Этот Web-узел включает множество статей, посвященных самым различным темам. Количество пишущих авторов измеряется тысячами. Именно здесь было заявлено, что Уорд Каннингем осуществил социальную революцию с помощью нескольких строк кода на Perl.

³Thomas S. Kuhn. *The Structure of Scientific Revolutions*. The University of Chicago Press, 1962.

диаграммы, а затем рекомендовал протестировать их с помощью разработанного кода. Я начал понимать, что, несмотря на то, что концепция экстремального программирования была мне чужда (по принципу Кана⁴), методы, лежащие в ее основе, были знакомы и давно применялись на практике.

Если ранее я предпочитал работать в одиночку, то экстремальное программирование позволило мне осознать все преимущества парного программирования. Рефакторинг, непрерывную интеграцию и присутствие заказчика в период разработки — все было воспринято мною вполне естественно.

Один из методов, практикуемых в экстремальном программировании, оказался для меня открытием — методика пробных испытаний (*test-driven development*). Этот метод подразумевает разработку тестовых случаев перед созданием фактического кода. При написании производственного кода первоочередной задачей становится успешный прогон сбойных тестов. Я не был готов к появлению множества вариантов выполнения кода, которые возникают в результате применения подобных методов программирования. Это полностью изменило мой собственный метод разработки программ, причем в лучшую сторону (что можно увидеть в этой книге). Некоторый объем кода, представленного в данной книге, был разработан в 1999 году. Здесь отсутствуют тестовые случаи. С другой стороны, все примеры кода, написанные после 1999 года, сопровождаются тестовыми случаями, и, как правило, эти тестовые случаи рассматриваются в первую очередь. Я уверен в том, что вы обратите внимание на данное отличие.

Итак, к осени 1999 года я имел возможность убедиться, что экстремальное программирование следует включить в модель компании Object Mentor в качестве альтернативного процесса, а также то, что мне следует отказаться от желания описать разработанный мною процесс. Сформулировав методы и описание процесса экстремального программирования, Кен выполнил отличную работу, на фоне которой мои попытки сделать нечто аналогичное выглядели явно неадекватно.

Структура книги

- Часть I. *Быстрая разработка ПО*. Приводится описание концепции быстрой разработки и принципов экстремального программирования (XP), причем особое внимание удалено тем из них, которые оказывают влияние на выбор метода проектирования и написания кода.
- Часть II. *Быстрое проектирование*. Здесь идет речь об объектно-ориентированном проектировании ПО, в частности о том, что представляет собой проектирование, какие существуют методы преодоления различных проблем. Здесь же вы найдете описание принципов проектирования на уровне объектно-ориентированных классов.

⁴Дважды упомянутое имя Кана говорит о многом...

- Часть III. *Практическое занятие: программа расчета зарплаты.* Здесь вы найдете самое объемное и полное практическое занятие из представленных в данной книге. Описываются методы объектно-ориентированного программирования и реализации на языке C++ простой пакетной системы, реализующей расчет зарплаты.
- Часть IV. *Упаковка программы расчета зарплаты.* Эта часть начинается с описания *принципов объектно-ориентированного проектирования пакетов*. Затем эти принципы иллюстрируются на примере инкрементной упаковки классов, приведенном в предыдущей части.
- Часть V. *Практическое занятие: моделирование метеостанции.* В этой части рассматривается одно из практических занятий, которое изначально планировалось поместить в книге Буча. В данной части идет речь о компании, принявшей важное деловое решение, и о том, каким образом команда разработчиков ПО на языке Java отреагировала на такое решение. Как обычно, часть начинается описанием шаблонов проектирования, используемых в процессе разработки ПО, а завершается — рассмотрением методов разработки и реализации проекта.
- Часть VI. *Практическое занятие: система ETS.* Приводится описание реального проекта, в котором принимал участие автор. Речь идет об автоматизированной системе тестирования, используемой для подсчета количества баллов, набранных на экзаменах, и внедренной в Национальном совете регистрации коллегии по архитектуре (National Council of Architectural Registration Boards).
- Приложения, описывающие стандарты UML-записи. Здесь вы найдете краткое описание UML-языка, построенное на рассмотрении небольших практических примеров.
- Приложения, посвященные рассмотрению различных вопросов. Здесь вы найдете материалы, “разбавляющие” основную тематику данной книги.

Для кого предназначена эта книга

Если вы разработчик . . .

Прочтите эту книгу “от корки до корки”. Она была написана в первую очередь для разработчиков и содержит информацию, которая необходима вам для создания ПО с помощью методов быстрой разработки. Прочитав эту книгу полностью, вы получите представление о методах, принципах и шаблонах, а также ознакомитесь с конкретными примерами, которые связывают их в одно целое. Воспользовавшись всеми полученными знаниями, вы сможете *довести* работу над своими проектами *до конца*.

Если вы менеджер или бизнес-аналитик...

Прочтите часть I, “Быстрая разработка ПО”. Главы этой части содержат подробное описание принципов и методов быстрой разработки, требования к планированию, тестированию, рефакторингу и программированию. Вы получите инструкции о том, каким образом следует организовывать команды разработчиков и управлять проектами. Этот материал поможет вам *довести* работу над своими проектами *до конца*.

Если вы хотите изучить UML-язык...

Сначала прочтите приложение А. Затем обратитесь к материалу приложения Б. После этого прочтите все главы из части III. В результате вы получите представление об основах синтаксиса и использования UML-языка. Вы сможете осуществлять преобразование UML-записи в программный код на языках Java и C++.

Если вы хотите изучить шаблоны проектирования...

Чтобы найти определенный шаблон, воспользуйтесь “Перечнем шаблонов проектирования”.

Чтобы ознакомиться с шаблонами в общих чертах, прочтите часть II, чтобы получить представление о принципах проектирования, а затем прочтайте часть III, часть IV, часть V и часть VI. В материале этих частей определены все шаблоны и продемонстрирован принцип их использования в типичных ситуациях.

Если вы хотите изучить принципы объектно-ориентированного программирования...

Прочтайте часть II, часть III и часть IV. Здесь приводится описание принципов объектно-ориентированного проектирования, а также продемонстрирован принцип их использования.

Если вы хотите изучить методы быстрой разработки...

Прочтайте часть II. Здесь описываются принципы быстрой разработки, включая требования, планирование, тестирование, рефакторинг и программирование.

Если хочется повеселиться...

Прочтите приложение В.

Благодарности

Мы выражаем сердечную благодарность следующим людям:

Лоуэллу Линдстрому (Lowell Lindstrom), Брайану Баттону (Brian Button), Эрику Миду (Erik Meade), Майку Хиллу (Mike Hill), Майклу Фезерсу (Michael Feathers), Джиму Ньюкирку (Jim Newkirk), Мику Мартину (Micah Martin), Анжелике Товенин Мартин (Angelique Thouvenin Martin), Сьюзанне Рocco (Susan Rosso), Талишу Джейферсону (Talisha Jefferson), Рону Джейфрису (Ron Jeffries), Кену Беку (Kent Beck), Джейфу Лангр (Jeff Langr), Дэвиду Фарберу (David Farber), Бобу Коссу (Bob Koss), Джеймсу Греннингу (James Grenning), Лансу Уэлтеру (Lance Welter), Паскалю Рою (Pascal Roy), Мартину Фаулеру (Martin Fowler), Джону Гудсену (John Goodsen), Алану Апту (Alan Apt), Полу Ходжеттсу (Paul Hodgetts), Филу Маркграфу (Phil Markgraf), Питу Мак-Брину (Pete McBreen), Х.С. Ламэну (H. S. Lahman), Дэйву Харрису (Dave Harris), Джеймсу Кэйнзу (James Kanze), Марку Вэбстеру (Mark Webster), Крису Бигей (Chris Biegay), Алану Френсису (Alan Francis), Фрэн Даниэл (Fran Daniele), Патрику Линдеру (Patrick Lindner), Джейку Уарду (Jake Warde), Ами Todd (Amy Todd), Лауре Стил (Laura Steele), Уильяму Пьетр (William Pietr), Камилле Трентакоста (Camille Trentacoste), Винсу О'Брайану (Vince O'Brien), Грегори Дуллесу (Gregory Dulles), Линде Кастилло (Lynda Castillo), Крейгу Лармэну (Craig Larman), Тиму Оттингеру (Tim Ottinger), Крису Лопесу (Chris Lopez), Филу Гудвину (Phil Goodwin), Чарльзу Тоулэнду (Charles Toland), Роберту Эвансу (Robert Evans), Джону Роту (John Roth), Дебби Утьев (Debbie Utley), Джону Брюэр (John Brewer), Руссу Рутер (Russ Ruter), Дэвиду Выдре (David Vydra), Лану Смиту (Lan Smith), Эрику Эвансу (Eric Evans), а также всем членам группы “The Silicon Valley Patterns group”, Питу Бриттингэму (Pete Brittingham), Грэхему Перкинсу (Graham Perkins), Филипу (Phlip) и Ричарду Мак-Дональду (Richard MacDonald).

Книгу рецензировали:

Пит Мак-Брин (Pete McBreen)/McBreen Consulting

Степан Дж. Меллор (Stephen J. Mellor)/Projtech.com

Брайан Баттон (Brian Button)/Object Mentor Inc.

Бьерн Страуструп (Bjarne Stroustrup)/AT&T Research

Микей Мартин (Micah Martin)/Object Mentor Inc.

Джеймс Греннинг (James Grenning)/Object Mentor Inc.

Особая благодарность выражается Гради Бучу и Полу Бекеру за то, что они разрешили мне включить в эту книгу главы, которые изначально предназначались для 3-го издания книги Гради *Объектно-ориентированный анализ и проектирование приложений*.

Особая благодарность выражается Джеку Ривзу за то, что он любезно разрешил мне воспользоваться материалом его статьи “Что такое проектирование?” (What is Design?).

Особую благодарность также выражаю Эриху Гамма за написание предисловия к этой книге. Я надеюсь, Эрих, что в этот раз шрифты для книги подобраны лучше!

Превосходные иллюстрации, приведенные в начале каждой главы, принадлежат Дженифер Конке (Jennifer Kohnke). Декоративные иллюстрации, приведенные в главах, — это восхитительная работа Анжелы Дон Мартин Брукс (Angela Dawn Martin Brooks), моей дочери, которая составляет одну из наибольших радостей моей жизни.

Источники

Все примеры исходного кода, приведенные в этой книге, находятся на Web-узле www.objectmentor.com/PPP.

Об авторах

Роберт К. Мартин (Robert C. Martin)

Роберт К. Мартин (Дядя Боб) является профессионалом в области разработки программ и международным консультантом по вопросам разработки ПО. Он является основателем и президентом компании Object Mentor Inc. — команды опытных консультантов, которые помогают своим клиентам в таких областях, как C++, Java, .NET, ОО, шаблоны проектирования, UML-язык, методика быстрой разработки ПО и экстремальное программирование. В 1995 году Роберт написал книгу *Проектирование объектно-ориентированных приложений на языке C++* (*Designing Object Oriented C++ Applications using the Booch Method*), которая стала бестселлером. В 1997 году он стал главным редактором книги *Языки шаблонов программного проектирования 3* (*Pattern Languages of Program Design 3*), которая была выпущена издательством Addison-Wesley. В 1999 году он стал редактором книги *Дополнительно о драгоценных камнях C++* (*More C++ Gems*), выпущенной издательством Cambridge Press. Вместе с Джеймсом Ньюкирком он принимал участие в написании книги *Экстремальное программирование на практике (XP in Practice)*, издательство Addison-Wesley, 2001. В 2002 году он написал долгожданную книгу *Быстрая разработка программ: принципы, примеры, практика (Agile Software Development: Principles, Patterns, and Practices)*, которая была выпущена издательством Prentice Hall в 2002 году. Его перу принадлежат десятки статей в различных отраслевых журналах, он также постоянно выступает на международных конференциях.

Джеймс В. Ньюкирк (James W. Newkirk)

Джеймс Ньюкирк является менеджером по разработке программ, а также действующим программистом. Его восемнадцатилетний опыт в этой области включает различные аспекты разработки ПО, начиная с микроконтроллеров, которые функционируют в режиме реального времени, и заканчивая службами в сети. Он является одним из авторов книги *Экстремальное программирование на практике (Extreme Programming in Practice)*, которая была опубликована издательством Addison-Wesley в 2001 году. С августа 2000 года он работает с инструментальным средством .NET Framework и внес немалый вклад в разработку NUnit — инструментального средства поэлементного тестирования для .NET.

Роберт С. Косс (Robert S. Koss)

Роберт С. Косс, доктор философии, разрабатывает программы уже на протяжении 29 лет. Во многих проектах, в которых он выполнял функции различного уровня — от программиста до старшего разработчика, применялись принципы объектно-ориентированного программирования. Доктор Косс читал сотни курсо-

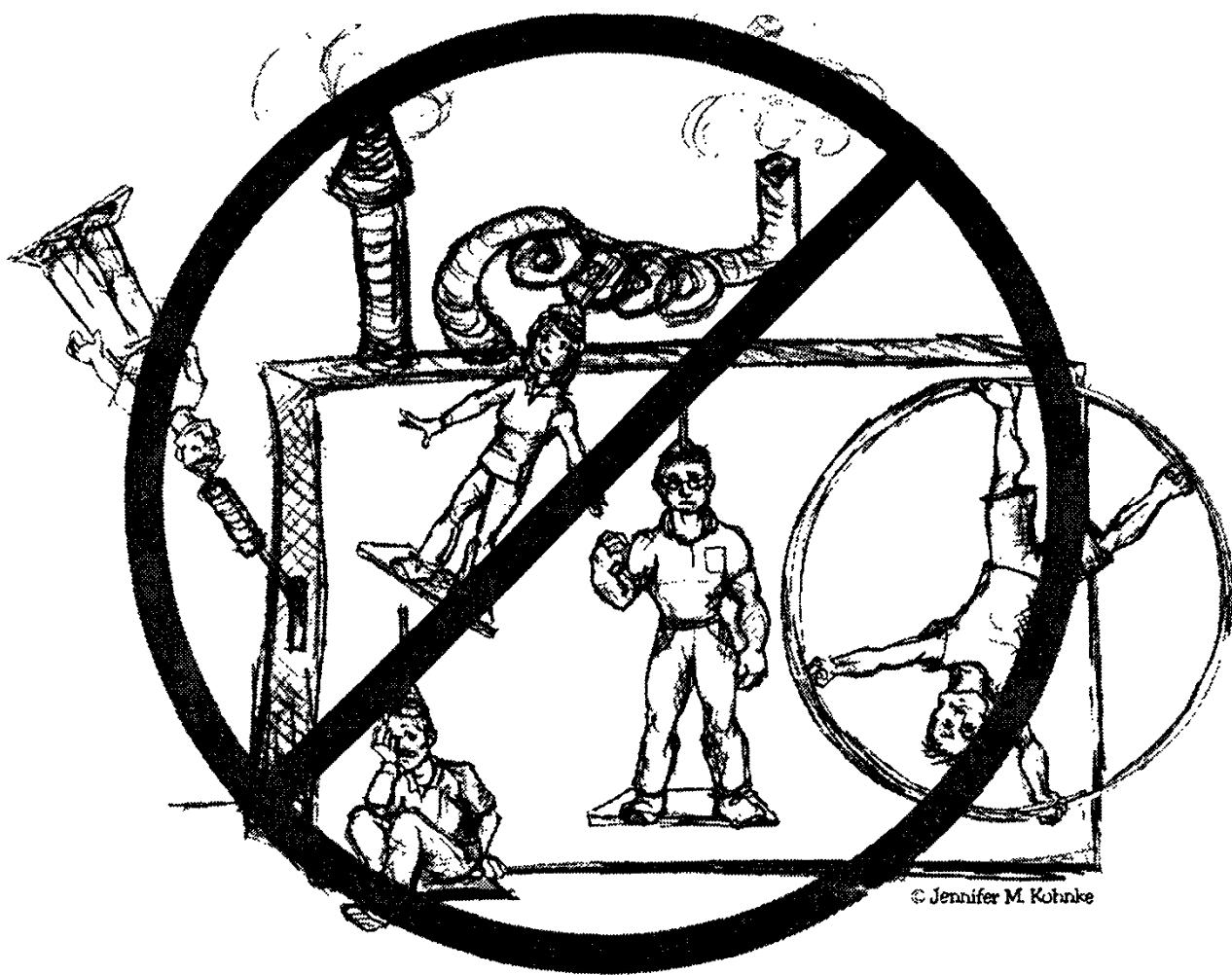
по объектно-ориентированной разработке и языкам программирования для тысяч студентов по всему миру. В настоящее время он работает старшим консультантом в компании Object Mentor, Inc.

Перечень шаблонов проектирования

Abstract Server	454
Active Object	245
Acyclic Visitor	549
Adapter	456
Bridge	462
Command	239
Composite	424
Decorator	562
Extension Object	568
Facade	268
Factory	392
Mediator	270
Monostate	278
Null Object	287
Observer	449
Proxy	467
Singleton	274
Stairway to Heaven	490
State	589
Strategy	261
Taskmaster	637
Template Method	254
Visitor	545

ЧАСТЬ I

Быстрая разработка ПО



Взаимодействие между людьми носит сложный характер, а различные вызванные им эффекты всегда “прозрачны”, но оно имеет наибольшее значение среди других рабочих аспектов.

Том Де-Марко и Тимоти Листер, *Peopleware*, с. 5

Принципы, шаблоны и применяемые практики имеют значение, но вся работа выполняется людьми. Согласно Алистеру Кокберну (Alistair Cockburn)⁵, “процессы и технологии оказывают второстепенное влияние на результат выполняемого проекта. Первостепенное влияние оказывают люди”.

Конечно, мы не можем управлять группами программистов подобно тому, как они управляют сформированными из компонентов системами, управляемыми процессами. Люди не являются “подключаемыми программными модулями”⁶. Если проекты выполняются успешно, следовательно, группы исполнителей являются самоуправляемыми, а между их членами хорошо налажено сотрудничество.

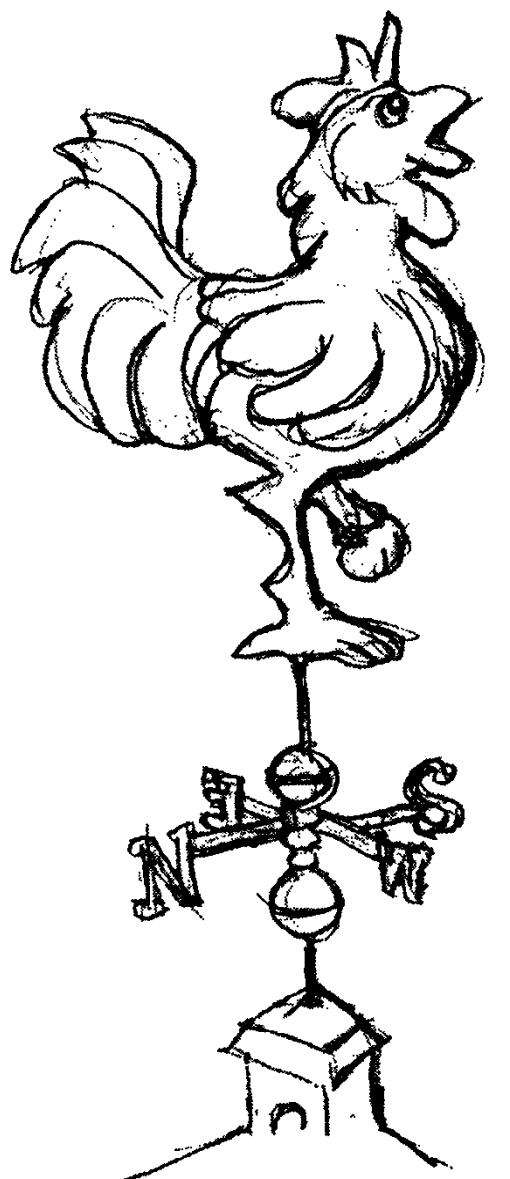
Компании, придерживающиеся подобных принципов при формировании команд программистов, получают *огромное* преимущество в конкурентной борьбе по сравнению с теми фирмами, которые рассматривают программистов как некую “серую массу”. На самом деле сплоченная группа программистов — наиболее мощная сила, влияющая на выполнение задач по разработке ПО.

⁵Частные коммуникации.

⁶Термин, введенный Кеном Беком (Kent Beck).

1

Быстрая разработка ПО



© Jennifer M. Kohnke

Флюгер на церковном шпиле быстро выйдет из строя под напором штормового ветра, если не обучится благоразумному искусству уклоняться от любого ветра.

Генрих Гейне

Многим из нас приходилось испытывать на себе трудности работы с проектом, когда отсутствуют четкие инструкции по его исполнению. В этом случае возможны непредсказуемые последствия, повторяются ошибки, а также напрасно затрачиваются усилия. В результате заказчики выражают недовольство из-за смещения графиков проекта и ухудшения его качества. Разработчики впадают в уныние, наблюдая обратную зависимость между объемом трудозатрат и качеством разрабатываемого программного обеспечения.

После такого фиаско возникает ощущение возможности его повторения. Именно это обстоятельство стало причиной разработки *процесса*, ограничивающего действия разработчиков и требующего получения определенных артефактов и результатов. Ограничения и результаты такого рода “извлекаются” из предыдущего опыта, “выбираются” положительные моменты, связанные с выполнением предыдущих проектов. Авторы книги выражают надежду, что вы все же снова приступите к работе, преодолев синдром боязни повторных ошибок.

В силу сложности отдельных проектов, несколько простых ограничений и артефактов не смогут гарантировать отсутствие ошибок. Дефекты и ошибки неизбежны, поэтому задача разработчика заключается в том, чтобы выполнить необходимую диагностику, а также определить дополнительные артефакты и ограничения, позволяющие предотвратить появление подобных ошибок в будущем. Итогом проведения подобного анализа в случае большого количества проектов может являться столь громоздкий процесс, который в значительной мере затруднит выполнение любого проекта.

Большой и громоздкий процесс может послужить причиной возникновения проблем, которые он должен устранять. В результате замедляется работа команды разработчиков, что, в свою очередь, приводит к смещению рабочего графика, а также к раздуванию бюджета. При этом способность команды к обратному реагированию может “приближаться к нулю”, из-за чего создаваемый программный продукт не будет отвечать поставленным требованиям. К сожалению, это приводит к тому, что у многих команд разработчиков создается впечатление о недостаточной продуманности самого процесса. Мотивируя неудачи громоздкостью и недостаточной продуманностью процесса, тем самым они способствуют “раздуванию” этого процесса.

Громоздкость и непродуманность процесса — это удачно подобранный термин, описывающий положение дел в большинстве компаний, занимающихся разработкой программного обеспечения 2–3 года назад. Хотя в то же самое время многие

разработчики вообще не использовали какой-либо определенный процесс, реализация на практике громадных процессов становится все более распространенной, и особенно это справедливо в случае больших корпораций (см. приложение С.)

Альянс специалистов по быстрой разработке ПО

В начале 2001 года группа экспертов в области разработки ПО обратила внимание на тот факт, что команды разработчиков во многих корпорациях попали в ловушку постоянно разбухающего процесса. Поэтому они организовали встречу, чтобы сформулировать оценку и принципы, позволяющие командам разработчиков оптимизировать свою работу и своевременно реагировать на изменения. Эта группа придумала себе запоминающееся название: “*Альянс специалистов по быстрой разработке ПО*” (*Agile Alliance*)¹. Работа над выработкой применяемых на практике оценочных принципов продолжалась на протяжении нескольких последующих месяцев. В результате появился “*Манифест альянса специалистов по быстрой разработке ПО*” (*The Manifesto of the Agile Alliance*).

Манифест альянса специалистов по быстрой разработке ПО

Мы находимся в процессе поиска более эффективных методов разработки ПО, а также помогаем делать это другим. В качестве предмета оценочного анализа выступает следующее:

- индивиды и взаимодействия, связанные с процессами и инструментальными средствами разработки;
- рабочий программный продукт и полный комплект документации;
- совместная работа с заказчиком и обсуждение условий контракта;
- реакция на происходящие изменения и соблюдение плана.

При этом отдаётся должное компонентам, перечисленным в левой части, но мы все же считаем, что правосторонние компоненты имеют большее значение.

Кен Бек (Kent Beck), Майк Бидль (Mike Beedle), Ари Ван-Биннекум (Arie van Bennekum), Элистер Кокберн (Alistair Cockburn), Уорд Каннингем (Ward Cunningham), Мартин Фаулер (Martin Fowler), Джеймс Гриннинг (James Grenning), Джим Хайсмит (Jim Highsmith), Эндрю Хант (Andrew Hunt), Рон Джейффрис (Ron Jeffries), Джон Керн (Jon Kern), Брайан Мэрик (Brian Marick), Роберт К. Мартин (Robert C. Martin), Стив Мэллор (Steve Mellor), Кен Шваубер (Ken Schwaber), Джейф Сатерленд (Jeff Sutherland), Дэйв Томас (Dave Thomas)

¹Web-узел www.agilealliance.org.

Индивиды в аспекте взаимодействия с процессами и инструментальными средствами

Успех в любом деле определяется людьми. Хорошо продуманный процесс не спасет проект от провала, если в команде отсутствуют сильные игроки. С другой стороны, плохо организованный процесс может привести к тому, что деятельность группы профессиональных разработчиков завершится крахом в случае, если они не смогут работать вместе.

Сильный игрок вовсе не обязательно является первоклассным программистом. Его успехи в программировании могут быть чрезвычайно скромными, главное — это умение успешно работать с другими членами команды. На самом деле, эффективное взаимодействие с сотрудниками, коммуникации и общение имеют большее значение, чем талант программирования сам по себе. Команда, состоящая из посредственных программистов, между которыми на должном уровне налажено взаимодействие, имеет больше шансов на успех, чем группа высококлассных программистов, которым не удается наладить работу в составе команды.

Правильно подобранные инструментальные средства также могут стать определяющими в достижении успеха. Компиляторы, интегрированные среды разработки (IDE, Integrated Development Environment), системы проверки исходного кода и некоторые другие компоненты играют жизненно важную роль, обеспечивая команде разработчиков надлежащие условия для работы. Однако значение инструментальных средств может быть преувеличено. Избыток больших и громоздких инструментов так же плохо, как и их недостаток.

Воспользуйтесь тактикой “малых шагов”. Не стоит делать вывод о том, что вы переросли возможности какого-либо инструментального средства до тех пор, пока не опробуете его в деле, после чего сделаете вывод относительно его непригодности для дальнейшей работы. Вместо того чтобы приобретать наиболее мощную и дорогую систему проверки исходного кода, следует найти ее бесплатно распространяемую версию и пользоваться ею до тех пор, пока не выявится несоответствие заявленным требованиям. Перед приобретением лицензии на использование лучших CASE-инструментов, используйте “белые доски” и миллиметровую бумагу в целях обоснования своих аргументов. Прежде чем отдать предпочтение сложнейшей системе баз данных, попробуйте использовать в работе однородные файлы. Не следует думать о том, что более громоздкие и совершенные инструментальные средства оптимизируют вашу работу автоматически. Чаще всего в этом случае проявляются недостатки, а не какие-либо преимущества.

Помните о том, что формирование команды намного важнее, чем создание среды разработки. Многие команды и менеджеры допускают ошибку, формируя изначально среду разработки и ожидая, что сплоченная команда образуется авто-

матически. Сначала следует сформировать команду, а затем позволить участникам команды сконфигурировать среду разработки, исходя из их собственных потребностей.

Рабочий программный продукт и исчерпывающая документация

Поставка программы без сопровождающей документации может привести к негативным последствиям. Программный код сам по себе не является идеальным средством, позволяющим взаимодействовать по законам логически непротиворечивой и структурированной системы. Настоятельно рекомендуется разработать документацию, удобную для восприятия человеком, в которой описывается система, а также приводится логическое обоснование принимаемых в ходе осуществления проекта решений.

Следует учитывать тот момент, что избыток документов далеко не всегда благо. Огромный объем сопровождающей документации потребует очень много времени на разработку и еще больше времени и трудозатрат на последующую поддержку, отображающую изменения в коде. Если документы не синхронизированы с кодом, они превращаются в громоздкие логически противоречивые источники неправдивой информации и вряд ли смогут применяться на практике.

В любом случае команде следует сформировать и поддерживать логически непротиворечивую и структурированную документацию, сопровождающую программный продукт. Этот документ должен быть *кратким и четко сформулированным*. Термин “краткий” подразумевает то, что документ должен состоять максимум из двух десятков страниц. Термин “четко сформулированный” означает, что документ включает полное логическое обоснование проекта и структуры высшего уровня, входящей в состав системы.

Если в нашем распоряжении оказывается краткий логически непротиворечивый и структурированный документ, каким же образом можно обучать новых членов команды работе с системой? Для этого следует хорошо поработать с новыми людьми. Новичков потребуется превратить в полноценных членов команды путем непосредственного обучения и взаимодействия.

Документы, которые являются наиболее подходящими в учебных целях, — это программный код и сама команда. Код не может являться источником ложной информации относительно выполняемых им функций. Выборка логического обоснования и цели из кода может быть связано с определенными трудностями, но все же код сам по себе — это единственный однозначный источник информации. Все члены команды хранят в памяти постоянно изменяющуюся карту-схему системы. Наиболее быстрый и эффективный способ передачи сведений о подобной карте другим членам команды заключается в непосредственном общении и взаимодействии.

Многие команды разработчиков идут по ошибочному пути, сосредотачиваясь на разработке подробной документации вместо того, чтобы нацелить все силы

и средства на разработку самого программного продукта. Зачастую эта ошибка является роковой. На сей счет существует простое правило, именуемое “Первый закон Мартина о документации”.

Не создавайте документацию до тех пор, пока потребность в ней не приобретет первостепенную важность.

Совместная работа с заказчиком и обсуждение условий контракта

Процедура заказа программ весьма отличается от действий, предпринимаемых в процессе заказа обычного товара. Недостаточно просто составить описание требуемого программного продукта, а затем попросить кого-либо разработать его в рамках четко определенного графика за фиксированную цену. Во многих случаях попытки трактовать подобным образом программные проекты терпят полный крах. Иногда эти провалы являются весьма болезненными.

Менеджеры компаний зачастую ограничиваются изложением команде разработчиков своих собственных требований, а затем ожидают от них поставки готовой системы, соответствующей поставленным требованиям. Именно подобный метод работы может привести к созданию низкокачественных программ, что эквивалентно краху проекта.

Успешные проекты предполагают регулярную и повторяющуюся обратную связь со стороны заказчиков. Вместо того чтобы полагаться на контракт или отчет о проделанной работе, заказчик программного продукта тесно работает с командой разработчиков, обеспечивая тесную обратную связь, которая позволяет согласовывать достигнутые ими результаты.

Контракт, определяющий требования, график и затраты на проект, как правило, включает массу недостатков. В большинстве случаев определяемые контрактом условия становятся бессмысленными прежде, чем проект будет выполнен². Наилучшие контракты лишь обусловливают способ, с помощью которого команда разработчиков и заказчик будут совместно работать над проектом.

В качестве примера можно привести контракт, который был заключен автором в 1994 году. Этот контракт предполагал выполнение большого проекта, рассчитанного на много лет и включающего до полумиллиона строк программного кода. Условиями контракта предусматривалось, что команда разработчиков получала относительно небольшую ежемесячную ставку. Большие денежные суммы (премии) выплачивались лишь после готовности больших программных блоков заданного размера, реализующих определенные функции системы. Эти блоки детально не описывались в контракте. Здесь просто указывалось, что выдача премий за готовый программный блок производилась лишь после того, когда он проходил через сито приемочных испытаний у заказчика. Детали проведения подобных испытаний в контракте не оговаривались.

²Иногда это происходит еще до подписания контракта!

В ходе выполнения проекта разработчики постоянно взаимодействовали с заказчиком. Отдельные программные блоки сдавались практически еженедельно (по пятницам). До понедельника или вторника последующей недели у заказчика накапливался для нас перечень изменений, которые было необходимо внести в программный продукт. Эти изменения распределялись в приоритетном порядке, и их внесение в продукт планировалось на последующие недели. Сотрудничество с заказчиком было настолько тесным, что никогда не возникал вопрос о проведении приемочных испытаний. Заказчик четко представлял, в каком случае блок, кодирующий функции системы, соответствовал сформулированным им требованиям, поскольку имел возможность наблюдать за процессом его разработки еженнощельно.

Требования, предъявляемые к данному проекту, непрерывного изменялись. Большинство изменений не носили исключительный характер. Некоторые блоки, кодирующие функции системы, удалялись и замещались новыми программными модулями. Но и контракт, и проект не потерпели крах и оказались успешными. Ключ к такому успеху заключался в тесном сотрудничестве с клиентом, а также в контракте, который обусловливал такое сотрудничество, но не предполагал определение деталей, связанных с областью действия и графиком в рамках выделенного бюджета.

Реакция на изменения и соблюдение плана

Очень часто успех или неудача программного проекта определяется способностью реагировать на изменения. В процессе формирования планов следует удостовериться в том, что эти документы обладают достаточной гибкостью и могут адаптироваться в соответствии с изменениями, вносимыми в процесс работы и технологию.

Ход выполнения программного проекта невозможно запланировать “с прицелом на будущее”. Это связано с тем, что непрерывно изменяется деловое окружение, в результате чего изменяются и требования. Во-вторых, заказчики могут изменить требования непосредственно после начала эксплуатации системы. И, наконец, даже если требования известны, и мы уверены в том, что они не изменятся в дальнейшем, вряд ли удастся точно определить время, требуемое на их формулирование.

Обычно начинающие менеджеры увлечены идеями создания PERT-диаграмм или диаграмм Ганта в проекте в целом, а затем их использованием в качестве наглядного пособия. Создается ошибочное впечатление, что подобная диаграмма может контролировать ход выполнения проекта. Появляется возможность отслеживать индивидуальные задания и по факту их выполнения вычеркивать их из диаграммы. Можно сравнивать реальные данные с тем, что запланировано, и реагировать на обнаруженные расхождения.

Но на самом деле это приводит к разрушению структуры самой диаграммы. По мере приобретения командой знаний о системе, а клиентами — знаний о своих потребностях, некоторые задачи, предусмотренные диаграммой, становятся ненужными. При этом обнаруживаются новые задачи, которые необходимо внести в диаграмму. Вообще говоря, изменяется не только запланированные данные, но и сама *форма*.

Более эффективная стратегия планирования заключается в составлении детальных планов на последующие две недели, очень приблизительных планов на последующие три месяца и крайне обобщенных планов на более длительные сроки. Требуется заранее знать, какие задачи нам предстоит выполнить на протяжении последующих двух недель. Можно лишь приблизительно знать требования, над которыми мы будем работать последующие три месяца. И у нас должно быть лишь обобщенное представление о том, какие функциональные возможности должна иметь система через год.

Подобное распределение плана по убывающей шкале означает, что мы акцентируем внимание на разработке детального плана, учитывающего выполнение только срочных задач. Как только составлен детальный план, его тяжело изменить, поскольку команда обладает известным “моментом инерции” и некоторой долей неповоротливости. Однако поскольку такой план рассчитан только на четырехнедельный период, остальная часть плана становится более “гибкой”.

Принципы быстрой разработки ПО

Вышеприведенные положения послужили причиной появления 12 принципов быстрой разработки ПО, заменяющих традиционный громоздкий процесс разработки.

- *Наша первостепенная задача — достичь соответствия требованиям заказчика, непрерывно поставляя ему в оптимальные сроки высококачественное ПО.* Журнал “MIT Sloan Management Review” опубликовал анализ правил разработки ПО, которые помогают компаниям разрабатывать продукты высокого качества³. Одно из правил акцентировано на взаимосвязи между качеством и поставкой частично функционирующей системы на раннем этапе разработки. В статье сообщалось, что чем меньшими функциональными возможностями обладает программный продукт на раннем этапе разработки, тем выше качество окончательной версии. Следующее положение, рассматриваемое в статье, говорит о наличии тесной взаимосвязи между качеством конечного продукта и частыми поставками отдельных модулей с различными функциональными возможностями. Чем чаще происходит поставка

³Product-Development Practices That Work: How Internet Companies Build Software, MIT Sloan Management Review, Winter 2001, Reprint number 4226.

ка частей программного продукта, чем выше качество конечного продукта. Совокупность методик быстрой разработки ПО предусматривает доставку разработанного компонента проекта на раннем этапе разработки и последующие частые поставки модулей по мере выполнения проекта. Наша цель — поставить элементарную систему на протяжении первых нескольких недель после начала выполнения проекта. Затем каждые две недели мы стремимся поставлять системные модули с растущими функциональными возможностями. Заказчики могут принять решение начать эксплуатацию системных модулей, если они считают, что они обладают достаточным уровнем функциональных возможностей. Или же они могут решить просто пересмотреть имеющиеся функциональные возможности и сообщить разработчикам об изменениях, которые необходимо внести в программный продукт.

- Следует приветствовать изменения требований, даже если они происходят на позднем этапе разработки. Правила быстрой разработки ПО рассматривают изменения как преимущество в конкурентной борьбе. Этот принцип отражает сложившуюся систему отношений. Участники процесса быстрой разработки ПО не опасаются изменений. Они рассматривают модификацию требований как положительный момент, поскольку при этом предполагается, что команда получила дополнительную информацию о том, каким образом следует соответствовать требованиям потребительского рынка. Команда, участвующая в быстрой разработке ПО, усердно и настойчиво трудится над сохранением гибкости структуры ПО. Благодаря этому в случае изменения требований влияние на систему будет минимальным. В последующих главах этой книги вы ознакомитесь с принципами и шаблонами объектно-ориентированного проектирования, которые позволят сохранять гибкость подобного рода.
- Следует часто поставлять рабочее ПО: от одного раза в две недели до одного раза в два месяца, стремясь к соблюдению более сжатых сроков. Команда разработчиков должна поставлять рабочий программный продукт, причем в максимально сжатые сроки (по истечению нескольких первых недель) и часто (каждые последующие несколько недель). При этом не следует довольствоваться поставками пачек документов или планов. Наше внимание сосредоточено на задаче поставить ПО, соответствующее требованиям заказчика.
- Бизнесмены и разработчики обязаны ежедневно совместно трудиться над проектом на протяжении всего процесса его выполнения. Чтобы обеспечить быстрое выполнение проекта, между заказчиками, разработчиками и организаторами проекта следует установить тесное и постоянное взаимодействие. Программный проект не должен соответствовать принципу “сделал и забыл”, поэтому проект должен находиться под постоянным руководством.

- *Выполняйте проекты с привлечением мотивированных индивидов. Представьте им соответствующую среду разработки и необходимую поддержку, причем доверьте им выполнение поставленных задач.* В проекте быстрой разработки ПО люди рассматриваются в качестве важнейшего фактора успеха. Все другие факторы — процесс, среда разработки, менеджмент и прочие — считаются второстепенными и могут изменяться, если они оказывают на людей неблагоприятное воздействие. Например, если среда разработки в офисе препятствует работе команды, ее следует изменить. Если определенные этапы процесса препятствуют работе команде, их следует изменить.
- *Наиболее результативный и эффективный метод передачи информации команде, а также в рамках самой команды заключается в интерактивном общении.* В проекте быстрой разработки ПО люди общаются друг с другом. Основной метод коммуникации — это общение. Документы можно создавать, но невозможно охватить всю информацию в письменном виде. Команде, работающей над проектом быстрой разработки ПО, не требуются спецификации, планы и проекты, выраженные в письменной форме. Эти документы могут быть сформированы, если возникнет безотлагательная и существенная потребность в этом, но они не являются абсолютной необходимости. Главное — это общение.
- *Рабочее программное обеспечение — это первостепенный критерий процесса.* Прогресс в проектах быстрой разработки ПО отслеживается путем оценки части ПО, которая в текущий момент времени удовлетворяет требования заказчика. Прогресс подобных проектов не связан с текущим этапом их выполнения, а также не может оцениваться на основе объема выпущенной документации или кода инфраструктуры, разработанного в результате их выполнения. Программный проект считается выполненным на 30%, если получен на 30% функционирующий программный продукт.
- *Проекты быстрой разработки ПО способствуют непрерывному развитию. Спонсоры, разработчики и пользователи должны уметь поддерживать постоянный темп работы.* Проект быстрой разработки ПО — это не спринтерский забег, а марафон. Команда не срывается с места на полной скорости и не пытается удержать эту скорость на протяжении всего проекта. Вместо этого она продвигается быстрым, но равномерным шагом. Слишком быстрое продвижение ведет к истощению сил, появлению недостатков и, в конечном счете, — к неудаче. Команды по быстрой разработке ПО сами поддерживают свой темп. Они не позволяют себе слишком устать. Они не заимствуют частичку завтрашней энергии для того, чтобы успеть сделать чуть больше сегодня. Они работают на скорости, позволяющей поддерживать высочайшие стандарты качества на протяжении всего проекта.

- *Постоянный акцент на высоком техническом уровне качества и хорошем проекте повышает быстроту разработки.* Высокое качество — это ключ к достижению высокой скорости разработки. Быстрого продвижения можно достичь путем поддержки максимального уровня качества и прочности программного продукта. Следовательно, все члены команды по быстрой разработке ПО обязаны создавать код с качественными характеристиками высочайшего качества. Следует избегать формирования “неразберихи”, обещая устранить ее при первой же возможности. Если происходит какая-либо путаница, устраняйте ее до конца текущего рабочего дня.
- *Необходимость в простоте.* Команды по быстрой разработке ПО не пытаются построить грандиозную суперсистему. Вместо этого они всегда избирают простейший путь, который согласуется с поставленными целями. Они не придают большого значения прогнозированию завтраших проблем и не пытаются оградить себя от них уже сегодня. Вместо этого сегодня они делают самую простую и высококачественную работу, будучи уверенными в том, что им удастся легко внести необходимые изменения, если в будущем возникнут какие-либо проблемы.
- *Наилучшие архитектуры, требования и проекты возникают у самоорганизующихся команд.* Команда по быстрой разработке ПО является самоорганизующейся. Обязанности не передаются отдельным членам команды, а учитывается лишь команда в целом. Лишь сама команда определяет наилучший способ их реализации. Члены команды по быстрой разработке ПО совместно работают над всеми аспектами проекта. Каждый из них имеет право внести свой вклад в общее дело. Ни один из членов команды не несет ответственность за архитектуру, требования или тесты. Команда разделяет эти обязанности, и каждый член команды может влиять на этот процесс.
- *Периодически команда делает выводы об эффективности своей работы и соответственно модифицирует и адаптирует свою линию поведения.* Команда по быстрой разработке ПО непрерывно подстраивает собственную организацию, правила, обычаи, взаимосвязи и т.д. Команда по быстрой разработке ПО осознает, что среда разработки постоянно изменяется и что она сама должна меняться вместе со средой, чтобы сохранить быстрый темп разработки.

Резюме

Профессиональная цель каждого разработчика ПО и команды в целом заключается в поставке своим работодателям и заказчикам продукта высочайшего качества. И все же, в удручающем числе случаев наши проекты терпят неудачу, или в результате их выполнения не удается произвести качественный продукт. Да-

же будучи хорошо спланированной, восходящая спираль “раздувания” процесса “виновна”, по меньшей мере, в некоторых из таких неудач. Принципы и критерии быстрой разработки ПО были сформированы с целью помочь команде разорвать замкнутый круг “раздувания” процесса и сфокусироваться на простых средствах для достижения поставленных целей.

На время написания этой книги уже существовало большое количество процессов быстрой разработки ПО. Сюда входят такие процессы, как SCRUM⁴, Crystal⁵, процесс разработки, управляемый свойствами (Feature Driven Development)⁶, адаптивная разработка программного обеспечения (Adaptive Software Development (ADP))⁷ и, самое важное, экстремальное программирование (Extreme Programming)⁸.

Литература

1. Beck K. *Extreme Programming Explained: Embracing Change*. Reading, MA: Addison-Wesley, 1999.
2. Newkirk J. and Martin R. *Extreme Programming in Practice*. Upper Saddle River, NJ: Addison-Wesley, 2001.
3. Highsmith J. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. NY: Dorset House, 2000.

⁴Web-узел www.controlchaos.com.

⁵Web-узел crystalmethodologies.org.

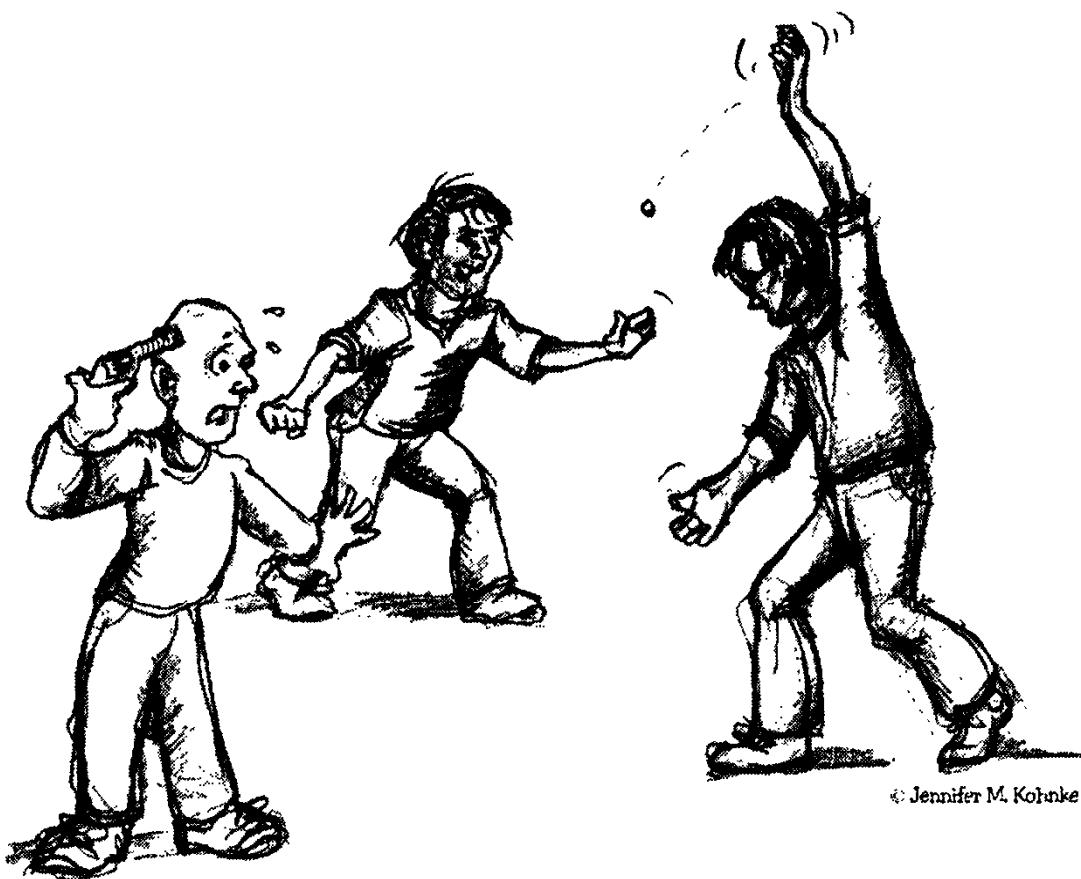
⁶Java Modeling In Color With UML: Enterprise Components and Process, Peter Goad, Eric Lefebvre, and Jeff De Luca, Prentice Hall, 1999.

⁷[Highsmith2000].

⁸[Beck1999].[Newkirk2001].

2

Основы экстремального программирования



© Jennifer M. Kohnke

Мы, разработчики, должны помнить о том, что экстремальное программирование — не самый лучший вариант среди всех возможных.

Пит Мак-Брин

В предыдущей главе были рассмотрены основные принципы быстрой разработки ПО. Однако там напрочь отсутствовали конкретные рекомендации. Несколько общих фраз и описание некоторых задач, но ничего определенного. Пришло время исправить сложившуюся ситуацию.

Методика экстремального программирования

Экстремальное программирование — один из наиболее известных методов быстрой разработки ПО. Он включает несколько простых взаимосвязанных методик, причем сумма целого превышает результат простого суммирования отдельных составляющих компонентов. В этой главе будут вкратце рассмотрены методики в целом, а в последующих главах подробнее рассматриваются отдельные составляющие компоненты.

Заказчик — член команды разработчиков

Крайне желательно, чтобы заказчики и разработчики поддерживали тесные контакты друг с другом, чтобы они были осведомлены о возникающих проблемах и вместе работали над их разрешением.

Кто в данном случае является заказчиком? Клиентом для команды разработчиков ПО является тот человек или группа лиц, которые определяют особенности и свойства будущего продукта, а также расставляют все приоритеты. Иногда в этой роли может выступать группа экономистов, занимающихся вопросами конъюнктуры рынка, или маркетологи, представляющие подобную компанию, занимающуюся вопросами разработки ПО. Также заказчиком может быть представитель группы пользователей конечного продукта или даже отдельный потребитель. Однако в любом подобном проекте, кем бы на самом деле не являлись заказчики, они должны трудиться “в одной упряжке” с разработчиками.

Наилучшим решением является размещение заказчика (или его представителя) в одной комнате с разработчиками. Отличным вариантом будет и расположение всех участников проекта не далее, чем метрах в тридцати друг от друга. Чем больше расстояние между участниками команды разработчиков, тем сложнее заказчику быть полноценным членом команды. Если он будет находиться в другом здании или в другом регионе страны, то вряд ли сможет без проблем влиться в общий коллектив.

Что делать, если клиент физически не может находиться в нужном месте? Оптимальный вариант — найти человека, который сможет его заменить (представителя), работая совместно с остальными членами команды.

Пользовательские истории

Для того чтобы приступить к планированию проекта, необходимо иметь общее представление о требованиях, предъявляемых к конечному результату работы, но совсем не обязательно заранее вдаваться в детали. Достаточно четко представлять себе все необходимые условия. Об отдельных деталях на данном этапе можно знать только то, что они существуют.

Мелкие детали, как правило, меняются со временем, особенно когда заказчик присутствует при процессе создания системы. Наблюдение за рождением и развитием системы лучше всего помогает формулированию и уточнению технических требований. Поэтому точное, но преждевременное описание различных второстепенных компонентов (до того, как они близки к завершению) зачастую может привести к напрасной трате времени и сил.

Когда работа организована в соответствии с принципами экстремального программирования, разработчики постоянно находятся в курсе всех требований благодаря их обсуждению с заказчиком, однако не следует фиксировать мельчайшие подробности. Вполне достаточно, если заказчик напишет на карточке несколько слов, которые будут напоминать ему и разработчику о прошедшей беседе. Разработчик в свою очередь запишет на карточке свои краткие выводы или заметки. Они будут основываться на деталях, оговоренных в беседе с клиентом.

Таким образом, связь с пользователями осуществляется путем проведения бесед, посвященных различным аспектам разработки программного продукта.

Пользовательские истории выступают в качестве инструмента планирования заказчика, с помощью которого он может отслеживать соблюдение технических требований на всех этапах разработки проекта.

Короткие рабочие циклы

При работе по методике экстремального программирования каждые две недели производится работоспособный программный продукт. Каждый из таких двухнедельных циклов позволяет получать ПО, отвечающее определенным требованиям заинтересованных сторон. В конце каждого цикла полученные результаты демонстрируются всем участникам проекта в целях получения отзывов.

План рабочего цикла

Отдельный рабочий цикл обычно длится две недели. За это время в программный продукт могут быть внесены некоторые (обычно второстепенные) изменения. При этом используются пользовательские истории и сведения о размере предоставляемого бюджетного финансирования.

Бюджет отдельного цикла устанавливается путем определения соотношения расходов и объема работы, проделанной на предыдущем этапе. Заказчик может сам выбрать необходимые усовершенствования ПО в пределах доступных средств.

После начала выполнения рабочего цикла заказчик уже не может изменять свои требования или устанавливать новые приоритеты. Разработчики же вольны сами распределять требования клиента на отдельные задачи и выполнять их в оптимальном порядке.

План выпуска версий

Коллектив разработчиков часто создает так называемый план выпуска версий, в котором описываются шесть ближайших циклов разработки программного продукта. Выпуск новой версии программы обычно занимает три месяца. План выпуска включает отобранный заказчиком список требований, которые расположены согласно установленным приоритетам, при этом их количество должно соответствовать размеру выделенного бюджета.

Размер бюджета устанавливается разработчиками, исходя из расходов на выпуск предыдущей версии ПО. Заказчик может сам отбирать все необходимые требования, предъявляемые к программе, а также усовершенствования в пределах установленной сметы. Также клиент может определить порядок внесения изменений в программный продукт. При общем согласии заинтересованные стороны могут полностью расписать несколько начальных циклов, подробно указав порядок действий разработчиков.

Отдельные версии программ также не являются чем-то неизменным. Заказчик в любой момент может изменить состав конечного продукта. Клиент может отменить некоторые усовершенствования, указать новые требования или по-новому расставить приоритеты.

Приемочные тесты

Соответствие конечного продукта запросам заказчика проверяется путем проведения приемочных тестов, форма которых определяется самим клиентом. Как правило, такие тесты предшествуют окончательному внесению изменений в программный продукт, иногда они проводятся параллельно с его обновлением. Подобные испытания формируются в форме своеобразных сценариев, что позволяет реализовывать автоматическое их выполнение. Заодно можно выяснить, работает ли система в целом так, как того желает заказчик.

Язык приемочных испытаний растет и развивается вместе со всей системой. Клиенты могут привлечь дополнительных сотрудников для разработки системы сценариев или просто сформировать специальный отдел контроля качества. Многие заказчики обращаются за помощью к таким отделам, чтобы создать специализированные инструменты контроля качества продукции.

После того как приемочный тест пройден, его результаты добавляются в общую базу данных, причем не допускаются какие-либо “сбои” в дальнейшем. Тесты из общей базы запускаются несколько раз в день, при каждой сборке системы.

Если хотя бы один тест завершается неудачей, вся компоновка признается неудачной. Благодаря этому система поддерживается в работоспособном состоянии, а все внесенные изменения сохраняются.

Парное программирование

Весь *производственный* программный код пишется отдельными парами программистов, каждая из которых работает вместе на одном отдельном компьютере. Один сотрудник набирает код на клавиатуре, а другой смотрит на экран, отслеживая ошибки и изменения в программе¹. При работе в таком режиме оба программиста участвуют в работе, а эффективность их труда повышается.

Периодически они меняются ролями. Ведущий программист может устать или временно иссякнуть, при этом его место займет партнер, который будет вводить текст программы. Обычно клавиатура переходит из рук в руки несколько раз в час. Авторские права на получившуюся программу принадлежат обоим сотрудникам.

Состав пар также меняется — как минимум, один раз в день. В результате каждый программист работает каждый день с двумя напарниками. На протяжении отдельного цикла разработки каждый программист в идеале должен успеть поработать со всеми остальными членами команды разработчиков, что способствует максимально интенсивному обмену мнениями и творческими идеями.

Подобная организация труда обеспечивает широкое распространение любой информации между всеми сотрудниками и повышает их квалификацию. Несмотря на то, что специализацию при такой системе работы никто не отменяет, и существуют задания, требующие специальных навыков, такие эксперты в узких областях успеют поработать со всеми другими сотрудниками. В большинстве случаев по прошествии некоторого времени рядовые сотрудники в большинстве случаев смогут при необходимости заменить “узких” специалистов.

Исследования, проведенные Лори Уильямсом (Laurie Williams)² и Дж. Носеком (Nosek)³, подтвердили, что использование методики парного программирования ничуть не снижает эффективности производства, при этом число ошибок значительно уменьшается.

Методика пробных испытаний (тест-драйв)

Глава 4, посвященная вопросам тестирования, подробно освещает все вопросы подготовки и проведения пробных испытаний. Здесь будет дан лишь краткий обзор.

¹Бывают такие пары, когда один участник контролирует ввод данных с клавиатуры, а другой — отслеживает перемещения мыши.

²[Williams2000], [Cockburn2001].

³[Nosek]

В процессе создания программы все ее отдельные элементы (блоки) подвергаются тестированию. Тестирование проводится неоднократно, что позволяет исключить вероятность появления ошибки. Сам тест создается раньше, чем пишется часть программы, ответственная за исполнение тестируемых функций.

Временной промежуток между написанием теста и части самой программы обычно очень мал и может составлять всего несколько минут. Тесты развиваются и усложняются вместе с эволюцией конечного программного продукта, разрыв между ними всегда остается минимальным (глава 6).

В результате постепенно создается сложнейший комплекс тестов. Он позволяет программистам в любой момент проверить функционирование программы. Когда пара разработчиков вносит изменения в код, появляется возможность сразу проверить их на наличие ошибок. Принятие подобных мер снижает вероятность тестирования “задним числом” на предмет поиска ошибок, допущенных в прошлых циклах разработки.

Когда вы пишете программный код, который тестируется надлежащим образом, то в результате получаете, как минимум, контролепригодный продукт. Кроме того, появляется дополнительный стимул к созданию блочной структуры программы (блоки легко тестируются независимо друг от друга). Блочная структура кроме того, что она позволяет уменьшить время тестирования, дает возможность не переписывать весь код заново при внесении изменений, а ограничиваться заменой отдельных элементов программы⁴.

Коллективное владение

Отдельная пара разработчиков может изменять *любые* части программы по своему усмотрению. Никто из программистов не отвечает лично за какой-то отдельный блок программного кода. Каждый сотрудник несет бремя обязанностей по разработке графического пользовательского интерфейса⁵. Каждый из них вносит свой вклад в разработку промежуточного ПО и программирование баз данных. Ничье мнение не является более важным, чем мнения других.

Все вышеизложенное совсем не означает, что в технологии экстремального программирования не используется специализация. Если вы всю сознательную жизнь занимались разработкой графических интерфейсов, конечно, имеет смысл сосредоточить свои усилия именно на разработке этого компонента программы, однако в любом случае придется принять участие и в работе над другими фрагментами кода и базами данных. Если вам захочется получить новую специальность,

⁴См. раздел II.

⁵Здесь я не отстаиваю трехуровневую архитектуру. Я просто выбрал три распространенных раздела технологии разработки ПО.

можно целенаправленно записаться в пару со специалистами, которые обучат вас необходимым навыкам. Здесь вы не ограничены рамками когда-то выбранной сферы деятельности.

Непрерывная интеграция

Программисты проверяют код и встраивают его в общую программу несколько раз в день. Здесь действует простое правило — кто первый придумал, тот и победил. Остальные просто копируют ранее созданные коды.

Коллективы разработчиков, работающих в рамках технологии экстремального программирования, применяют неблокируемый контроль источников. Это означает, что программисты могут тестировать программный модуль в любое время, вне зависимости от того, кто еще может в данный момент проводить тестирование. При этом нужно быть готовым к постоянному внесению изменений, сделанных коллегами.

Пара программистов работает над отдельной задачей час или два. Они разрабатывают программный код и соответствующую ему процедуру тестирования. Предположим, что по прошествии некоторого времени они решили изменить данную часть программы. В первую очередь программисты должны убедиться, что все тесты по-прежнему работают. Затем они встраивают свой фрагмент кода в общую систему, советуясь при необходимости с другими сотрудниками. После успешной интеграции система вновь проходит процесс сборки. Нужно снова выполнить все тесты, чтобы проверить все функции программы после добавления в нее нового модуля. В случае обнаружения новых ошибок они устраняются, а процедура тестирования запускается с самого начала.

Таким образом, разработчики собирают систему заново по несколько раз в день. Они снова и снова проходят все стадии сборки от начала до конца⁶. Если результат записывается на компакт-диске, каждый раз формируется новый диск. Если же результатом деятельности команды является, к примеру, Web-сайт, то заново проводится процесс его установки (как правило, на тестовом сервере).

Равномерная работа

Процесс разработки ПО — это не спринтерский рывок, а скорее — забег на марафонскую дистанцию. Команда, взявшая чересчур резвый старт, рискует выдохнуться задолго до финиша. Чтобы прийти к финишу вовремя, необходимо двигаться в постоянном темпе, что позволит сохранить энергию и поддерживать внимательность на должном уровне. Наиболее подходящим эпитетом здесь будет “медленно, но уверенно”.

⁶Однажды Рон Джэффрис (Ron Jeffries) отметил: “Завершение всегда происходит позже, чем вы

Одним из принципов экстремального программирования (несмотря на его название) является *запрет* сверхурочной работы. Единственным исключением может стать последняя неделя перед выпуском финальной версии продукта, по аналогии с рывком, который спортсмены делают перед финишной чертой.

Открытая рабочая среда

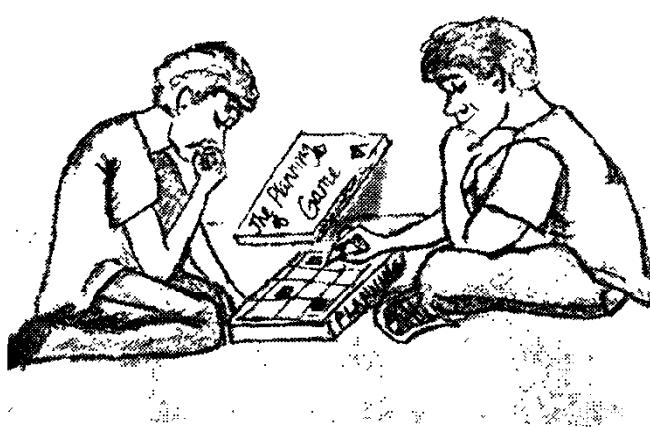
Команда разработчиков работает в общей открытой комнате. В помещении устанавливаются столы с компьютерами на них (два или три на каждом столе). Перед каждым компьютером должно стоять два стула для пары программистов. На стенах развешиваются необходимые для работы графики, диаграммы, таблицы и так далее.

Наилучший звуковой фон — негромкий гул от разговоров. Каждая пара должна находиться в пределах слышимости всех остальных. При этом каждый имеет возможность услышать о проблемах коллег и прийти к ним на помощь. Каждый знает о состоянии дел у своих соседей. Такая организация рабочего пространства способствует интенсивной коммуникации.

Кому-то такая обстановка может показаться слишком отвлекающей. Постоянный гул голосов может раздражать и не давать сосредоточиться. Но в действительности это не так. Более того, исследования, проведенные в Мичиганском университете показали, что подобная обстановка приводит к повышению эффективности работы примерно в два раза⁷.

Игра в планирование

В следующей главе процесс планирования процесса разработки ПО, выполняемый командой, которая работает по принципам экстремального программирования, будет рассмотрен подробнее. Здесь же ограничимся кратким описанием.



Сущность планирования заключается в распределении ответственности между заказчиками и разработчиками. Деловые люди (заказчики) принимают решения

⁷Web-узел <http://www.sciencedaily.com/releases/2000/12/001206144705.htm>.

о важности того или иного элемента программы, а разработчики рассчитывают стоимость его проектирования и внедрения.

Прежде чем приступить к работе над новой версией программного продукта, в начале каждого цикла разработчики знакомят клиентов со сметой, которая зависит от затрат, понесенных на предыдущих этапах. Заказчики сами решают, какие изменения и усовершенствования должны быть внесены в программу, не выходя за рамки существующего бюджета.

Руководствуясь этими простыми правилами, используя короткие производственные циклы и частое обновление версий программы, заказчики и разработчики быстро могут войти в ритм выполняемого проекта. Клиенты почувствуют темп работы программистов, что позволит им достаточно точно спрогнозировать сроки окончания работ и общую стоимость разработки ПО.

Простая структура проекта

Большинство команд, работающих в соответствии с методикой экстремального программирования, старается создавать максимально простые в использовании и выразительные программные продукты. Кроме того, они фокусируют свое внимание только на текущих требованиях заказчиков, которые необходимо реализовать на данном цикле разработки. О будущих изменениях речь в данном случае не идет — всему свое время. Вместо того чтобы пытаться прогнозировать будущие запросы клиентов, разработчики при необходимости изменяют общий дизайн всей системы “на лету”, подстраиваясь под требования заказчика.

Все вышеизложенное означает, что разработчики не начинают свою работу с создания какой-либо инфраструктуры системы. Они также не начинают свою деятельность с создания базы данных или промежуточного ПО. В первую очередь програмисты воплощают в жизнь первый пакет требований заказчика *максимально простым из всех возможных способов*. Инфраструктура наращивается постепенно и усложняется только по мере надобности.

Ниже следуют три основных правила, применяемые разработчиками в рамках технологии экстремального программирования.

Выбирайте простейшие решения

Разработчики всегда стараются найти максимально простой способ реализации всех пожеланий заказчика. Если все требования можно воплотить, используя однородные файлы, не следует использовать базы данных или другие сложные структуры. Если для передачи данных достаточно обычного двунаправленного канала, то не стоит использовать более передовые технологии типа ORB или RMI. Многопоточный режим также надо использовать лишь в случае необходимости. Имеет смысл стремиться к максимальному упрощению. Только тогда мы сможем приблизиться к реализации простой и надежной системы на практике.

Вам это не понадобится

Предположим, что мы твердо уверены в том, что именно рано или поздно понадобится нам в системе баз данных. Мы точно знаем, что впоследствии придется встраивать поддержку технологии ORB. Заранее известно, что придется реализовывать поддержку нескольких пользователей. Так может быть, имеет смысл заранее предусмотреть дополнительные блоки?

Разработчики достаточно серьезно относятся к оценке последствий создания инфраструктуры на ранних этапах проекта. За основу всегда берется предположение, что такая инфраструктура может не понадобиться впоследствии. Поэтому к ее созданию приступают только в случае, когда имеются неопровергимые доказательства того, что в дальнейшем без этих компонентов никак не обойтись. Также учитывается, что поздняя перестройка всей системы может повлечь дополнительные расходы.

Раз и навсегда

Разработчики, работающие в рамках технологии экстремального программирования, не приветствуют дублирование программного кода. Подобные фрагменты программы сразу удаляются. Источников такой ситуации может быть множество. Одной из самых популярных причин являются банальные операции копирования и вставки одного фрагмента кода сразу в несколько разделов программы. При обнаружении такие фрагменты заменяются кодом функции или базового класса. Иногда может оказаться, что два или более алгоритмов сходны друг с другом, отличаясь лишь незначительными деталями. Их можно преобразовать в отдельные функции, воспользовавшись шаблоном TEMPLATE METHOD⁸. В любом случае, что бы ни являлось причиной дублирования кода, такие участки должны быть удалены.

Наилучший способ избежать избыточности состоит в применении различных процедур абстракции, или обобщения. Если два предмета сходны между собой, всегда можно выделить их общую составляющую. Таким образом, процесс удаления избыточного кода заключается в создании массы обобщенных данных и в дальнейшем уменьшении их количества.

Рефакторинг⁹

Эта тема подробно рассматривается в главе 5. Здесь можно найти общее описание процесса.

Код имеет тенденцию к ухудшению качества по мере усложнения всей системы. Каждый раз, когда в программу вносятся усовершенствования или исправляются ошибки, ее работа ухудшается. Если этот процесс оставить без внимания, то,

⁸См. главу 14.

⁹[Fowler99]

в конце концов, это может привести к непоправимым последствиям, неразберихе и полному хаосу.

Коллективы, практикующие методику экстремального программирования, стараются избежать этого, применяя частый рефакторинг. *Рефакторинг* — это внесение серии малозаметных изменений, улучшающих структуру всей системы, но не влияющих на ее работу. Каждая отдельная трансформация крайне проста и незначительна, однако вместе они оказывают значительное влияние на программную архитектуру.

После каждой трансформации проводится тестирование данного блока, дабы убедиться в том, что изменения не привели к появлению ошибок. Затем вносится следующее изменение, проводится тест и т.д. Благодаря такому алгоритму работы системы всегда находится в полностью работоспособном состоянии.

Лучше проводить процесс рефакторинга постоянно, чем использовать его по завершении проекта, при выпуске новой версии продукта или в конце каждого цикла/дня. Имеет смысл пользоваться этой технологией один раз в час или даже каждые полчаса. Чем чаще это делать, тем “чище” и проще будет выглядеть программный код, а его работа будет более надежной.

Метафора

Метафора или модельное представление — одно из самых сложных для восприятия понятий экстремального программирования. Большинство программистов — прагматики, поэтому они не любят отсутствия четких определений. Действительно, многие сторонники технологии экстремального программирования предлагают прекратить использовать метафору или модельное представление в качестве стандартной методики. Однако пока — это одна из самых важных составляющих всего процесса разработки ПО.

Представьте себе обычную мозаику. Как вы узнаете, в каком порядке должны быть расположены ее составляющие? Конечно, каждая “деталь” впритык соединяется с другими, и ее форма должна идеально им соответствовать. Если бы вы были слепым и обладали хорошим осязанием, то могли бы решить эту задачу путем простого перебора, пытаясь соединить каждую деталь с каждой.

Однако существует и более важный признак соответствия деталей мозаики друг другу. Это — картинка, нанесенная на их поверхность. Это настолько важный признак, что когда вы *видите*, что две подходящие по цвету детали не подходят друг другу по форме, то это означает только то, что создатель головоломки допустил ошибку.

Это и есть метафора — общая картина, отображающая всю систему в целом. Это такое видение системы, которое делает очевидным расположение и функционирование каждого отдельного модуля. Если “форма” модуля кажется неподходящей, значит, что-то с ним не так.

Зачастую метафора сводится к правильному именованию элементов программы. Такая система позволяет не только определить по названию назначение отдельного модуля, но и видеть его связь с другими компонентами.

Например, однажды я работал с системой, которая выводила на экран текст со скоростью 60 символов в секунду. При этом заполнение всего экрана занимало некоторое время. Поэтому допускалось, чтобы программа, генерирующая текст, заполняла им буфер. Когда буфер заполнялся целиком, программа выгружалась на диск. После очистки буфера программа снова загружалась в память и продолжала работу.

В работе использовались термины, связанные с грузовиками-мусоровозами. Буфера представлялись в виде маленьких грузовиков. Экран монитора был “мусорной свалкой”. Сама программа была “производителем мусора”. Такие названия элементов позволяли нам легко обмениваться информацией при обсуждении и четко видеть всю картину в целом.

В другом случае я работал над системой, которая анализировала сетевой трафик. Каждые тридцать минут она опрашивала несколько десятков сетевых адаптеров и принимала от них данные. Каждый отдельный адаптер передавал небольшой блок данных, состоящий из нескольких переменных. Такие блоки назывались “ломтиками”. Программа-анализатор “готовила” эти ломтики, поэтому ее назвали “Тостером”. Отдельные переменные внутри ломтиков назывались “крошками”. В общем, это оказалось очень полезной и забавной метафорой.

Резюме

Экстремальное программирование — это набор простых и конкретных методик, предназначенных для быстрой разработки ПО. Множество команд программистов успешно используют эту технологию на практике.

Экстремальное программирование — отличный универсальный метод разработки ПО. Многие коллективы используют его в исходном виде, без малейших изменений. Другие же адаптируют его для достижения своих целей и задач.

Литература

1. Dahl D. *Structured Programming*. NY: Hoare. Academic Press, 1972.
2. Conner D. *Leading at the Edge of Chaos*. Wiley, 1998.
3. Cockburn A. *The Metliodology Space*. Humans and Technology technical report HaT TR.97.03 (dated 97.10.03), <http://members.aol.com/acockburn/papers/methyspace.methysoace.htm>.
4. Beck K. *Extreme Programming Explained: Embracing Change*. Reading. MA: Addison-Wesley, 1999.

5. Newkirk J. and Martin R. *Extreme Programming in Practice*. Upper Saddle River. NJ: Addison-Wesley, 2001.
6. Williams L., Kessler R., Cunningham W. and Jeffries R. *Strengthening the Case for Pair Programming*. IEEE Software, July-Aug. 2000.
7. Cockburn A. and Williams L. *The Costs and Benefits of Pair Programming*. XP2000 Conference in Sardinia, reproduced in *Extreme Programming Examined*. Addison-Wesley, 2001.
8. Nosek J. T. *The Case for Collaborative Programming*. Communications of the ACM, 1998.
9. Fowler M. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

3

Планирование



Если вы можете измерить то, о чем говорите, и выразить результаты в цифрах, значит, вы кое-что знаете о предмете разговора; если же вы не можете провести измерения, то ваши данные крайне эфемерны и недостаточны.

Лорд Кельвин

В этой главе описывается процесс планирования, применяемый в ходе экстремального программирования (иногда такое планирование называют игровым)¹. Сама методика планирования сходна с теми, которые используются совместно с другими технологиями быстрого программирования², такими как SCRUM (www.controlchaos.com)³, Crystal (www.crystalmethodologies.org)⁴, разработка, основанная на свойствах программы⁵ и адаптивная разработка программного обеспечения (ADP)⁶. Однако в данной главе не производится детальное описание всех перечисленных методик, поскольку задача этой книги является совершенно другой.

Предварительное исследование

На начальной стадии проекта разработчики и заказчики стараются выбрать действительно важные для программы требования, формулируемые пользователями. При этом они не пытаются определить все пожелания клиентов. В процессе создания программы исходные данные и требования заказчиков могут изменяться (а также появляться новые). При этом поток замечаний и пожеланий не прекращается даже после выпуска финальной версии программного продукта.

Разработчики производят оценку важности сформулированных требований. Естественно, их оценка является относительной, а не абсолютной. На карточках с пользовательскими историями заказчиков указывается их относительная стоимость в условных пунктах. Стоимость одного пункта не всегда можно точно выразить в единицах времени, но понятно, что выполнение требований, записанных на карточке с 8 пунктами, займет в два раза больше времени, чем пользовательская формулировка, оцененная в 4 пункта.

Соединение, разделение и скорость проектирования

“Слишком большие” или “слишком маленькие” пожелания пользователей сложно оценивать корректно. Разработчики стремятся разделять на отдельные задачи большие формулировки и объединять вместе компактные пожелания. В принципе, любое громоздкое пользовательское пожелание может быть разбито на несколько отдельных задач, и наоборот — всегда можно объединить вместе несколько маленьких заданий.

¹[Beck99], [Newkirk2001].

²Web-узел www.AgileAlliance.org.

³Web-узел www.controlchaos.com.

⁴Web-узел www.crystalmethodologies.com.

⁵*Java Modeling In Color With UML: Enterprise Components and Process* by Peter Goad, Eric Lefebvre, and Jeff De Luca, Prentice Hall, 1999.

⁶[Highsmith2000].

Предположим, что пользовательская история заказчика звучит следующим образом: “Пользователи должны иметь возможность безопасно вносить деньги на счет, снимать их и переводить с одного счета на другой”. Эта история относится к категории больших. Ее важность трудно оценить, и вряд ли оценка будет корректной. Однако можно разделить это утверждение на простые отдельные компоненты, оперировать которыми гораздо проще:

- пользователи могут регистрироваться в системе;
- пользователи могут выходить из системы;
- пользователи могут переводить деньги на свой счет;
- пользователи могут снимать деньги со своего счета;
- пользователи могут переводить деньги с одного счета (своего) на другой.

Когда пользовательские истории разделяются на части (или наоборот — несколько историй объединяются в одну), необходимо выполнить переоценку. При этом не стоит просто складывать или вычитать пункты, указанные на карточках. Основная причина, в силу которой производится разбиение или слияние отдельных формулировок, заключается в их приведении к удобному для оценки объему. Нередки ситуации, когда большая задача “стоимостью в 5 пунктов” разбивается на составляющие подзадачи, которые в сумме оцениваются в 10 пунктов. В данном случае 10 пунктов — более точная оценка.

Относительные оценки ничего не говорят нам об абсолютном размере пользовательских формулировок и не могут помочь в принятии решения о разбиении на части или “склеивании”. Для того чтобы надлежащим образом определить правильный размер пользовательской формулировки, требуется использовать новый фактор — *скорость*. Если мы знаем точное значение скорости, то можем умножить значение оценки каждой отдельной истории на величину скорости, чтобы получить фактическое время, необходимое для выполнения данных требований. Например, если скорость выполнения проекта можно обозначить как “1 пункт за два дня”, а оценка пользовательской истории соответствует 4 пунктам, выполнение всех требований, изложенных в историях из данного раздела, займет восемь дней.

В процессе выполнения проекта оценка скорости становится более точной, поскольку появляется возможность подсчета количества пунктов, выполняемых за один цикл. Впрочем, в самом начале проектирования разработчики не всегда четко представляют себе будущую скорость работы. Обычно они в этом случае ограничиваются предположениями, уточняя их по мере возможности. Зачастую имеет смысл потратить несколько дней на моделирование рабочей ситуации, чтобы оценить будущую скорость выполнения заказа. Этот процесс получил название *пробного, или временного решения*.

Планирование выпусков программ

Получив информацию о скорости работы, заказчики могут оценить стоимость внедрения каждой пользовательской истории. Также они осведомлены об их ценности и о расстановке приоритетов. Все это позволяет заказчикам выбирать истории, в выполнении которых они заинтересованы в первую очередь. Расстановка приоритетов здесь не всегда играет главенствующую роль. Иногда что-то достаточно важное, но слишком дорогостоящее откладывается “на потом”, а вначале внедряются более дешевые функции. Такие решения называются *деловыми*, они основываются на соотношении затрат и ожидаемой прибыли.

Разработчики и клиенты совместно устанавливают дату первого выпуска проекта. Обычно на это отводится от двух до четырех месяцев. Заказчики отбирают пользовательские истории, в которых описаны функции, присутствующие в данном выпуске программы, и устанавливают примерный порядок их внедрения. Количество учтенных историй зависит от скорости выполнения проекта. Так как изначально скорость определяется лишь приблизительно, то и выбор набора требований может быть лишь примерным. Впрочем, на этом этапе излишняя точность и необязательна. Во время работы над проектом скорость будет определена более точно, соответственно, можно будет внести корректиды и в другие области.

Планирование рабочих циклов

Далее разработчики совместно с заказчиками определяют рабочий (производственный) цикл. Обычно за основу берется двухнедельный период. Клиенты выбирают пользовательские истории, которые будут внедрены в течение первого цикла. При этом опять-таки нельзя выбрать больше историй, чем допускает текущая скорость работы.

Порядок выполнения пользовательских историй заказчика в пределах одного цикла является чисто техническим решением. Разработчики проводят их внедрение в наиболее удобном для них порядке. Они могут реализовать заказы один за другим, а могут одновременно работать сразу над несколькими. Такие решения целиком предоставляются на усмотрение программистов.

Заказчики не могут изменять набор пользовательских историй после начала выполнения производственного цикла. Они свободно могут выбирать любые требования или менять их порядок, за исключением тех формулировок, над которыми в данный момент идет работа.

Цикл заканчивается в заранее определенное время, *даже в том случае, если не все задания были выполнены*. Затем оценивается объем выполненной работы и рассчитывается скорость работы в пределах цикла. Эта скорость используется для планирования следующего цикла. Здесь действует одно простое правило. Планируемая скорость выполнения следующего цикла должна соответствовать скорости работы в предыдущем цикле. Если команда реализовала в прошедшем

производственном цикле 31 пункт, то совокупная стоимость требований заказчика для следующего цикла также должна составлять 31 пункт. Иными словами, скорость работы будет составлять 31 пункт за цикл.

Подобная взаимосвязь помогает синхронизировать планирование в внутри всей команды. Если сотрудники будут со временем приобретать опыт работы, то их скорость работы будет возрастать соразмерно. Если кто-то покинет команду, то скорость выполнения будет снижена. Если развитие архитектуры системы способствует повышению удобства и скорости разработки, то общая скорость работы всей команды возрастет.

Планирование выполняемых задач

В начале каждого производственного цикла разработчики и клиенты вместе составляют план. После этого программисты разделяют пользовательские истории на производственные задания — задачи, на выполнение которых программист затрачивает от 4 до 16 часов. Все пользовательские истории анализируются с помощью заказчика, а также нумеруются выполняемые задачи.

Список задач оформляется в виде плаката или записывается на “белой” доске (на которой пишут фломастерами). Затем разработчики последовательно выбирают задачи, за решение которых они готовы взяться. Как только разработчик берется за задание, он производит оценку, используя произвольную систему пунктов (часто в этом случае используют количество рабочих часов)⁷.

Каждый отдельный разработчик может приняться за решение любой проблемы. Специалисты по базам данных не обязаны все время заниматься их разработкой. В свою очередь, проектированием баз данных могут заниматься сотрудники, которые до этого разрабатывали пользовательский программный интерфейс. На первый взгляд подобная методика работы кажется неэффективной, однако, как вы увидите в дальнейшем, существует механизм, управляющий этим процессом. Выгода очевидна — чем больше каждый сотрудник будет знать обо всем проекте, тем здоровее и сплоченнее будет обстановка в коллективе, выше будет и общий уровень информированности. Информация о проекте должна распространяться среди сотрудников свободно, независимо от их специализации.

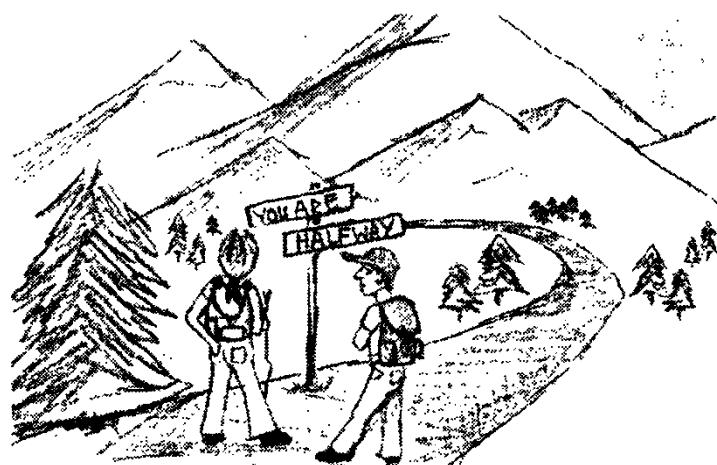
Каждый разработчик знает, сколько произвольных пунктов он выполнил за прошедший производственный цикл. Их количество образует персональный бюджет сотрудника. При распределении задач на следующий цикл сотрудник не может брать на себя больше работы, чем он выполнил на протяжении предыдущего цикла (то есть, не способен выходить за рамки своего бюджета).

Распределение задач продолжается до момента завершения этого процесса, или пока все сотрудники не исчерпают свои бюджеты. Если при этом еще

⁷Многие разработчики для обозначения пунктов задачий используют также термин “наилучшие часы программирования”.

остаются свободные задачи, то разработчики распределяют их согласно навыкам и специализации. В случае, когда и после этого еще остаются задания, разработчики предлагают клиентам удалить некоторые из них из общего списка для данного цикла. Если все задачи распределены, а у разработчиков еще остался избыток бюджета, заказчики могут добавить свои требования к списку.

На полпути



В середине производственного цикла вся команда собирается на планерку. К этому моменту половина пользовательских историй, содержащихся в списке, должна быть обработана. Если же этого не произошло, команда разработчиков перераспределяет задания и ответственность за их выполнение между своими членами таким образом, чтобы к концу цикла все они были завершены. В случае, когда этот способ неприемлем, необходимо провести переговоры с заказчиками на предмет сокращения списка пользовательских историй. В крайнем случае можно присвоить некоторым историй низкий приоритет — за их выполнение разработчики возьмутся только в том случае, если позволит время.

Предположим, например, что заказчики выбрали восемь пользовательских историй, суммарная стоимость которых составляет 24 пункта. Предположим также, что разработчики разделили их на 42 задачи. Следовательно, в середине цикла должна быть выполнена 21 задача (общей стоимостью в 12 пунктов). При этом данные 12 пунктов должны представлять полностью выполненные пользовательские истории клиентов, поскольку конечной целью команды является выполнение историй заказчиков, а не отдельных задач. В противном случае может сложиться крайне неприятная ситуация, когда к концу цикла будет завершено 90% всех задач, но не будет реализована ни одна пользовательская история. Поэтому в середине производственного цикла оценка проделанной работы проводится на основе завершенных пользовательских историй.

Циклы

Каждые две недели текущий производственный цикл подходит к концу и начинается следующий цикл. В конце цикла проделанная работа демонстрируется заказчикам, которые оценивают “общий вид”, состояние и качество функционирования всей системы. Реакция пользователей выражается в новых требованиях и историях, которые отличаются от прежних историй.

Заказчики постоянно следят за разработкой проекта, а также контролируют скорость работы команды. Благодаря этому появляется возможность долгосрочного планирования и детальной расстановки приоритетов на относительно ранних этапах. Короче говоря, клиенты владеют полной информацией о проекте и могут влиять на него на всех этапах разработки.

Резюме

Цикл за циклом, выпуск за выпуском — проект постоянно развивается в предсказуемом и удобном для всех ритме. Каждый его участник знает, что и когда можно ожидать. Организаторы следят за выполнением проекта непрерывно. Разнообразным диаграммам и графикам выполнения работ они предпочитают действующее программное обеспечение, с которым они могут поработать и оценить его достоинства и недостатки.

Программисты работают по плану, разработанному на основе их собственных оценок и расчетной скорости проектирования. Они сами выбирают для себя задания, ориентируясь на комфортабельную работу и возможность достижения качественных результатов.

Менеджеры получают новые данные после завершения каждого производственного цикла. Эти данные они используют для контроля и управления проектом. Нет никакой необходимости использовать давление на персонал, угрозы или уговоры, пытаясь заставить разработчиков завершить проект пораньше.

Конечно, “в каждой бочке меда есть своя ложка дегтя”. Организаторы проекта далеко не всегда бывают довольны результатами, особенно в начале работы над программой. Использование быстрых методов создания программ еще не означает, что все мечты пользователей исполняются в кратчайшие сроки. Заказчики смогут контролировать производственный процесс, и у них будет шанс получить максимальную выгоду при наименьших затратах, но не более того.

Литература

1. Beck K. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley, 1999.

2. Newkirk J. and Martin R. *Extreme Programming in Practice*. Upper Saddle River. NJ: Addison-Wesley, 2001.
3. Hiehsmith J. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. NY: Dorset House, 2000.

4

Тестирование



© Jennifer M. Kohnke

В огне проверяется золото, а трудности выявляют сильных духом.

Сенека

Процесс создания модульных тестов представляет собой в большей степени процесс проектирования, чем верификации. В данном случае мы имеем дело с процессом документирования. В ходе создания модульных тестов включается множество цепей обратной связи, одна из которых реализует функции верификации.

Методика пробных испытаний (тест-драйв)

Что бы произошло, если бы перед созданием программы сначала разрабатывались тесты? Что было бы, если бы мы отказывались от применения функций в программе до тех пор, пока получаются неудовлетворительные результаты тестирования (по причине отсутствия этой функции)? Что было бы, если бы мы отказывались добавить *одну единственную строку* кода в программу из-за неудовлетворительного результата теста (по причине отсутствия этой строки)? Что, если бы мы постепенно добавляли в программы функциональные возможности методом первоочередной разработки тестов, давших неудовлетворительные результаты, которые подтверждали бы существование таких функциональных свойств, а затем проводили успешное тестирование? Как это повлияло бы на текущий программный проект? Какие преимущества вытекают из существования такого набора тестов?

Первое и самое очевидное влияние такой методики заключается в том, что возможности каждой функции в программе подтверждают тесты. Подобный тестовый набор является основой для дальнейшей разработки программы. Благодаря тестам мы узнаем о том, случайно или нет нарушено существующее функциональное свойство. Кроме того, можно добавлять функции в программу или изменять структуру последней, не боясь нарушить что-то важное в выполняемом процессе. Благодаря тестам мы удостоверяемся в нормальной работе программы. Таким образом, тесты упрощают процесс внесения изменений и поправок в программу.

Более важное, но менее очевидное влияние методики первоочередного создания тестов заключается в сосредоточении нашего внимания на другой точке зрения. Создаваемая программа должна рассматриваться с точки зрения модуля, вызывающего эту программу. Поэтому мы сразу же сталкиваемся как с интерфейсом программы, так и с ее функцией. При первоочередном создании тестов программа разрабатывается таким образом, чтобы быть *удобным образом вызываемой на выполнение*.

Более того, первоочередное создание теста является стимулом для разработки наилучшим образом *тестируемой* программы. Разработка вызываемой и тестируемой программы имеет важное значение. Чтобы быть более вызываемым и тестируемым, программное обеспечение должно быть изолировано от среды. Таким образом, при создании тестов необходимо вначале *изолировать программу*.

Другая важная цель первоочередной разработки тестов заключается в том, что тесты являются бесценной формой документирования. Для того чтобы узнать, как вызвать функцию или создать объект, можно воспользоваться соответствующим тестом. Тесты представляют собой набор примеров, благодаря которым другие программисты могут определить, как работать с кодом. Этот комплект документации должен быть полным и удобным в применении. Эта документация всегда остается подлинной и должна соответствовать действительности.

Пример проекта, основанного на результатах тестирования

Не столь давно автор, развлекаясь на досуге, написал игру *Охота на чудовище* (*Hunt the Wumpus*). Это простая приключенческая игра, в которой игрок пробирается через пещеру, пытаясь убить чудовище, прежде чем оно съест его. Пещера представляет собой ряд залов, соединенных друг с другом проходами. Каждый зал имеет выходы на север, юг, запад и восток. Игрок передвигается по пещере, задавая компьютеру то или иное направление.

Одним из первых тестов, созданных для этой программы, был модуль `testMove`, код которого приводится в листинге 4.1. Эта функция применяется для создания нового лабиринта `WumpusGame`, где зал 4 соединяется с залом 5 через восточный проход, затем игрок переходит в зал 4, выдается команда перемещения на восток, затем сообщается, что игрок должен быть в зале 5.

Листинг 4.1. Тестовая программа

```
public void testMove()
{
    WumpusGame g = new WumpusGame();
    g.connect(4, 5, "E");
    g.setPlayerRoom(4);
    g.east();
    assertEquals(5, g.getPlayerRoom());
}
```

Этот код разработан до этапа завершения разработки игры `WumpusGame`. Автор прислушался к совету Уорда Канингэма (Ward Cunningham) и создал легко воспринимаемый тест. Он был уверен в том, что тест даст положительные результаты, если создать код, который соответствовал бы структуре, предполагаемой тестом. Такой подход называется *агрессивным программированием*. Это подразумевает, что перед применением теста необходимо определить его цели в простом для восприятия виде. Можно надеяться, что эта простота и четкость свидетельствуют о хорошей структуре программы.

Агрессивное программирование сразу же подтолкнуло меня к интересному решению вопроса, возникшего в ходе выполнения проекта. В teste не используется класс `Room`. Этот класс вряд ли понадобится для передачи сообщений о перемещении по коридорам. Вместо этого будут использоваться целые числа для представления залов.

Вам это может показаться алогичным. В конечном итоге может показаться, что все содержание программы сводится только к залам: перемещение по залам, обнаружение содержимого залов и так далее. Можно ли полагать проект, задуманный автором, ущербным по причине отсутствия класса `Room`?

Нетрудно показать, что для игры Wumpus понятие коридоров имеет большее значение, чем понятие зала. Можно доказать, что этот первоначальный тест обозначил неплохой способ разрешения проблемы. Действительно, думаю, что именно так оно и есть, но, тем не менее, это не то, что я пытаюсь доказать. Дело в том, что тест выявил главный момент проекта на самой ранней стадии. *Первоочередное создание тестов — это умение находить решение вопросов, возникающих в ходе выполнения проекта.*

Обратите внимание, что с помощью тестов можно узнать о работе программы. Для многих из нас не представляет особого труда создать четыре именованных метода игры WumpusGame, исходя из этого простого определения. Можно также без особых усилий спроектировать и написать команды, реализующие перемещение в трех других направлениях. Если захочется узнать, как соединить два зала или переместиться в том или ином направлении, то результаты этого теста позволят нам определить свое поведение в условиях неопределенности. Этот тест является полным и удобным в применении документом, который описывает программу.

Изоляция теста

При разработке тестов до момента разработки кода часто выявляются такие фрагменты программного кода, которые необходимо изолировать. Например, на рис. 4.1 представлена простая UML-диаграмма¹, моделирующая процесс формирования платежных ведомостей. Класс Payroll использует класс EmployeeDatabase в целях приведения в соответствие с объектом Employee. Объекту Employee передается команда о начислении оплаты. Затем вычисленная величина передается объекту CheckWriter в целях выписки чека. В конечном итоге полученное значение пересыпается объекту Employee, после чего объект снова возвращается в базу данных.

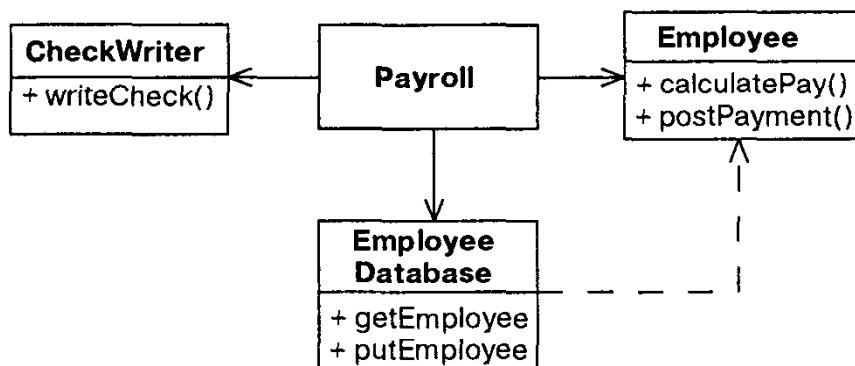


Рис. 4.1. Модель двойного формирования платежной ведомости

Предположим, что код еще не запрограммирован. Поэтому эта диаграмма остается пока на доске, она была нарисована в процессе сеанса быстрой разработки

¹ Подробности о работе с UML-диаграммами можно найти в приложениях А и Б.

проекта². Теперь необходимо разработать тесты, определяющие поведение объекта *Payroll*. С написанием таких тестов связано много проблем. Во-первых, какую базу данных следует использовать? Записи платежной ведомости должны считываться из той или иной базы данных. Нужно ли создавать полнофункциональную базу данных, прежде чем будет протестирован класс *Payroll*? Какие сведения необходимо загружать в эту базу данных? Во-вторых, как можно удостовериться в том, что тот или иной чек распечатан? Невозможно создать автоматизированный тест, который проверял бы наличие отпечатанного чека, подтверждая при этом указанную в нем сумму!

Решение этих проблем реализуется с помощью шаблона *Mock Object*³. Можно включить интерфейсы, определяющие взаимодействие между разработчиками программы формирования платежных ведомостей, а также сформировать тестовые заглушки, которые реализуют эти интерфейсы.

На рис. 4.2 представлена описанная структура. Класс *Payroll* использует интерфейсы для взаимодействия с объектами *EmployeeDatabase*, *CheckWriter* и *Employee*. Для использования этих интерфейсов было создано три шаблона *Mock Object*. Объект *PayrollTest* запрашивает шаблоны *Mock Object* с тем, чтобы определить, правильно ли работает с ними объект *Payroll*.

В листинге 4.2 демонстрируется цель теста. Здесь создаются соответствующие имитационные объекты, которые передаются объекту *Payroll*. Последнему объекту дается команда о начислении оплаты всем служащим, а у имитационных объектов запрашивается подтверждение правильности выписки всех чеков и передачи сведений обо всех выплатах.

Листинг 4.2. TestPayroll

```
public void testPayroll()
{
    MockEmployeeDatabase db = new MockEmployeeDatabase();
    MockCheckWriter w = new MockCheckWriter();
    Payroll p = new Payroll (db, w);
    p.payEmployees();
    assert(w.checksWereWrittenCorrectly());
    assert(db.paymentsWerePostedCorrectly());
}
```

Назначение этого теста заключается в проверке того, вызвал ли объект *Payroll* все требуемые функции с корректными исходными данными. Фактически в процессе выполнения теста не проверяется, выписаны ли чеки и то, что именно требуемая база данных была надлежащим образом обновлена. Вместо этого проверяется, что класс *Payroll* ведет себя таким образом, как и должен вести в случае изоляции.

²[Jeftnes2001].

³[Mackinnon2000).

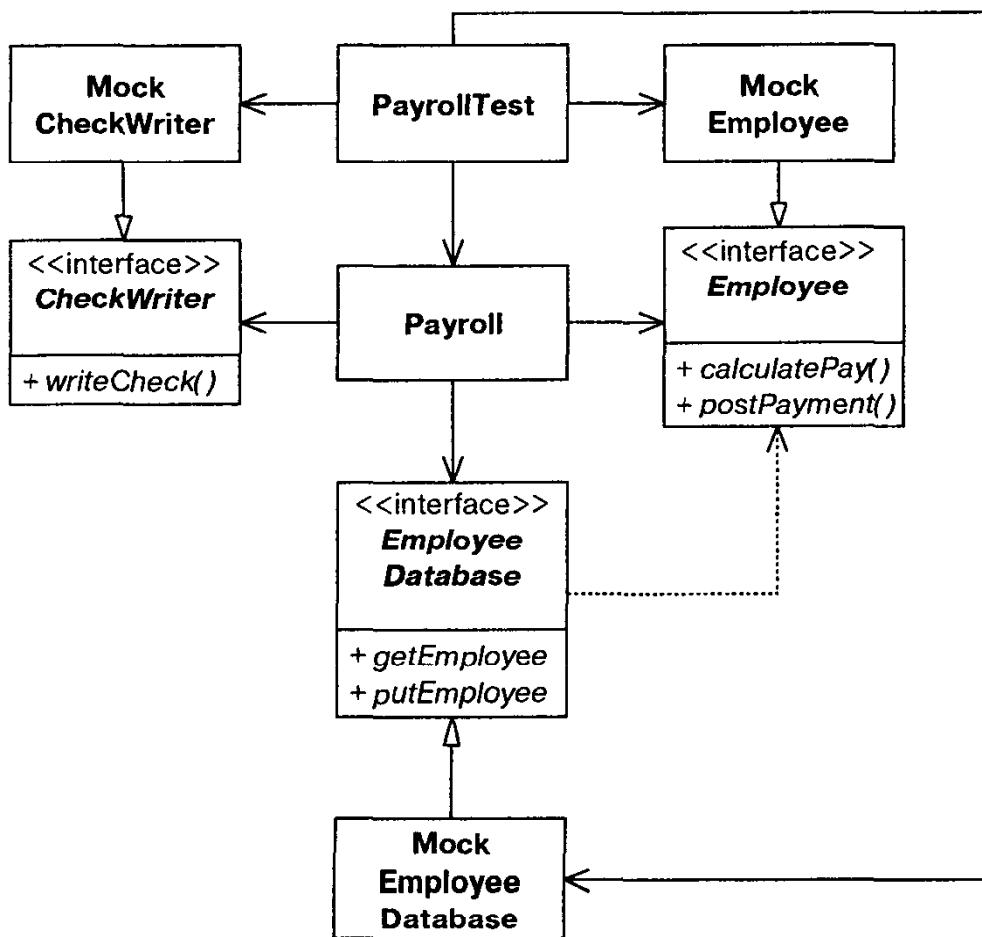


Рис. 4.2. Изолированный класс Payroll, использующий Mock Object в процессе тестирования

Может возникнуть вопрос о необходимости существования объекта **MockEmployee**. Вполне вероятно, что настоящий класс **Employee** может применяться вместо копии. Если бы это было так, можно было бы допустить, что класс **Employee** является более сложным, чем требовалось бы для проверки функции **Payroll**.

Вероятностная изоляция

Неплохо было бы произвести изоляцию объекта **Payroll**. Благодаря этому можно переключаться между различными базами данных и системами контроля документов как в целях тестирования, так и для расширения возможностей приложения. Думаю, будет интересно узнать, что такая изоляция производится в случае необходимости тестирования. Очевидно, что необходимость в изоляции тестируемого модуля побуждает нас производить изоляцию способами, наиболее приемлемыми для всей структуры программы. *Создавая сначала тесты, а затем код, мы корректируем проекты.*

Большая часть этой книги посвящена принципам проектирования, определяющим управление зависимостями. Благодаря этому вырабатываются определенные направления и методики изоляции классов и пакетов. Эти принципы могут

оказаться более полезными, если используются как часть стратегии модульного тестирования. Именно модульное тестирование служит начальным толчком и указателем направления для изоляции.

Приемочные тесты

Модульное тестирование необходимо, но не является столь существенным фактором для выполнения верификации. Результаты модульного тестирования подтверждают ожидаемый уровень функционирования небольших элементов системы, однако они не подтверждают корректность работы системы в целом. Модульные тесты функционируют в соответствии с принципом “белого ящика”⁴, подтверждая работу индивидуальных механизмов системы. Приемочные тесты функционируют по принципу “черного ящика”⁵, подтверждая выполнение требований заказчика.

Приемочные тесты создаются программистами, которые незнакомы с внутренними механизмами системы. Такие тесты могут создаваться непосредственно заказчиком или техническим персоналом со стороны заказчика, возможно, например, контролерами качества (QA, quality assurance). Поскольку приемочные тесты реализованы на программном уровне, они удобны в применении. Обычно тесты разрабатываются с помощью специального языка подготовки сценариев, разрабатываемого для заказчиков приложения.

Приемочные тесты полностью документируют свойства. Если заказчик разработал приемочные тесты, подтверждающие корректность свойства, программисты могут считывать эти тесты с тем, чтобы иметь более полное представление об этом свойстве. Итак, в то время как модульные тесты служат полным и удобным в применении документированием внутренних компонентов системы, тесты приемлемости служат в качестве полного и удобного в применении документа, описывающего свойства системы.

Более того, первоочередное формирование приемочных тестов имеет огромное влияние на архитектуру системы. Чтобы система могла быть тестируемой, ее необходимо изолировать на самом высоком архитектурном уровне. Например, интерфейс пользователя может быть отделен от деловых правил, благодаря чему деловые правила могут тестироваться на “предмет приемлемости”, не получая доступ к самому интерфейсу.

На самых ранних стадиях разработки проекта появляется соблазн составлять приемочные тесты вручную. Этого делать не рекомендуется, поскольку отпадет



потребность в изоляции на ранних стадиях из-за необходимости автоматизации такого тестирования. На самой первой стадии разработки проекта, при условии информирования о необходимости автоматизации приемочных тестов, достигаются важные компромиссы, связанные с архитектурой разрабатываемого приложения. Аналогично тому, что в результате выполнения модульных тестов принимаются важные проектные решения на локальном уровне, благодаря приемочным тестам принимаются архитектурные решения на глобальном уровне.

Создание схемы приемочного тестирования может представлять значительную сложность. Тем не менее, если учесть ценность свойств, присущих только одной итерации и создать только ту часть схемы, которая необходима для нескольких тестов приемлемости, то обнаружится, что создание таких схем не настолько сложно, как может показаться. Как правило, затраченные в этом случае средства себя окупают.

Пример приемочного теста

Рассмотрим еще раз приложение, реализующее формирование платежной ведомости. На первом этапе необходимо уметь добавлять и удалять фамилии служащих из базы данных. Следует также научиться составлять платежные чеки для служащих, сведения о которых находятся в базе данных. К счастью, мы имеем дело только со служащими, получающими фиксированный оклад. Другие категории служащих рассматриваются на следующих стадиях выполняемого проекта.

До сих пор еще не был составлен ни один фрагмент кода, и мы не приступали к разработке проекта. Теперь наступило время для разработки приемочных тестов. Еще раз напомню, что агрессивное программирование является весьма полезным инструментом. Приемочные тесты следует составлять так, как по нашему мнению они должны выглядеть, а затем структурировать язык сценариев и систему платежной ведомости в соответствии с данной структурой.

Хотелось бы разработать такие приемочные тесты, которые были бы удобны для написания, и обеспечивали простое внесение изменений. Следует их сохранить таким образом, чтобы можно было бы запускать в любое время. Поэтому есть смысл в том, чтобы приемочные тесты записывались в виде простых текстовых файлов.

Ниже приводится пример сценария теста приемлемости.

```
AddEmp 1429 "Robert Martin" 3215.88  
Payday  
Verify Paycheck Empid 1429 GrossPay 3215.88
```

В этом примере в базу данных добавляется информация о служащем под номером 1429. Его зовут Роберт Мартин (Robert Martin), а его месячная зарплата составляет 3215,88 долларов. Затем системе сообщается о том, что сегодня день выдачи зарплаты и что всем служащим необходимо ее выплатить. В конечном

итоге система подтверждает, что для служащего 1429 был выписан чек на зарплату в размере 3215,88 долларов.

Понятно, что написание сценария такого рода не составит особого труда для заказчика. Кроме того, в такой сценарий без особых усилий можно добавить новое функциональное свойство. Тем не менее, стоит подумать о том, что все это означает для структуры системы.

Первые две строки сценария реализуют функции приложения по формированию платежных ведомостей. Эти строки представляют транзакции, выполняемые при формировании платежной ведомости. Эти функции предназначены для пользователей платежных ведомостей. Тем не менее, строка `Verify` (подтвердить) — это не та транзакция, которая нужна пользователям платежных ведомостей. Эта строка является директивой, специфичной для приемочного теста.

Таким образом, рассматриваемая схема приемочного тестирования предусматривает анализ данного текстового файла, отделяя транзакции по формированию платежных ведомостей от директив тестирования приемлемости. Она должна отсылать транзакции по формированию платежной ведомости приложению, ответственного за составление этой ведомости, а затем для подтверждения данных использовать директивы приемочного тестирования для запроса приложения по составлению платежной ведомости.

При этом бремя издержек архитектуры ложится на программу по формированию платежных ведомостей. Эта программа ответственна за прием данных, вводимых пользователями, и также схемой приемочного тестирования. Необходимо объединить оба перечисленных способа ввода как можно раньше. Поэтому, скорее всего, программа по формированию платежных ведомостей должна сопровождаться обработчиком транзакций. Этот модуль обрабатывает транзакции в форме `AddEmp` и `Payday`, приходящие из более чем одного источника. Необходимо найти некоторую общую форму для этих транзакций с тем, чтобы свести к минимуму число специализированных кодов.

Одним из решений этой проблемы могла бы быть передача транзакции приложению по формированию платежной ведомости в виде XML-кода. Схема приемочного тестирования могла бы, конечно, генерировать XML-код, и вполне возможно, что пользовательский интерфейс системы по формированию платежных ведомостей также мог бы генерировать XML-код. Поэтому транзакции могли бы выглядеть следующим образом.

```
<AddEmp PayType=Salaried>
<EmpId>1429</EmpId>
<Name>Robert Martin</Name>
<Salary>3215.88</Salary>
</AddEmp>
```

Эти транзакции должны передаваться приложению по формированию платежных ведомостей посредством вызова подпрограммы, сокета или даже файла

пакетного ввода. На самом деле обычным делом будет переход от одного к другому в ходе разработки. Поэтому на ранних этапах принимается решение о чтении транзакций из файла, переход к API или сокетам выполняется позже.

Каким образом схема приемочного тестирования активизирует директивы верификации? Понятно, что эта схема должна получить доступ к данным, полученным из приложения по составлению платежных ведомостей. Еще раз обратите внимание на то, что схема приемочного тестирования не предусматривает считывание информации, находящейся на распечатанном чеке. Для этого можно прибегнуть к более традиционному решению.

Можно сделать так, чтобы приложение по формированию платежных ведомостей генерировало платежные чеки, используя возможности XML. Схема приемочного тестирования может передать запрос сформированному XML-коду относительно требуемых данных. Заключительный этап распечатки чека из XML-кода достаточно прост, поэтому может реализовываться ручными тестами приемлемости.

Приложение по формированию платежных ведомостей может создавать документ в формате XML, в котором содержатся все платежные чеки. Этот код записывается следующим образом.

```
<Paycheck>
  <EmpId>1429</EmpId>
  <Name>Robert Martin</Name>
  <GrossPay>3215.88</GrossPay>
</Paycheck>
```

Понятно, что схема приемочного тестирования может выполнить директиву *Verify* только при наличии данного XML-кода. Напомню еще раз, что можно разбить XML-код, воспользовавшись сокетами, библиотекой API или произвести запись в файл. На начальном этапе вариант с применением файла будет самым простым. Поэтому приложение по формированию платежной ведомости начнет выполнять чтения XML-транзакций из файла и последующей передачи платежных чеков в файл. Схема приемочного тестирования будет считывать транзакции в текстовом формате, переводя их в XML-код и записывая в файл. Затем она вызовет программу по составлению платежных ведомостей. В конечном итоге будет считан из программы сгенерированный XML-код, а также активизирована директива *Verify*.

Вероятностная архитектура

Обратите внимание на влияние, оказываемое тестами приемлемости на архитектуру системы формирования платежных ведомостей. Сам факт первоочередного обращения к тестам очень быстро привел нас к осознанию важности задачи по вводу/выводу XML-кода. Эта архитектура изолировала источники транзакции от приложения по формированию платежных ведомостей. Она также отделила

механизм распечатки платежных чеков от приложения по формированию платежных ведомостей. В этом кроются преимущества применяемых архитектурных решений.

Резюме

Чем проще выполняются тестовые наборы, тем чаще эти тесты будут запускаться на выполнение. Чем чаще выполняются тесты, тем быстрее обнаружится любое отклонение от нормы. При неоднократном выполнении тестов в течение дня система не будет давать сбой более чем на несколько минут. Это весьма благоразумная цель. Мы просто не позволяем системе выходить из строя.

Тем не менее, верификация является одним из преимуществ, являющихся результатом создания тестов. Как модульные тесты, так и приемочные тесты являются формой документирования. Это документирование полное и простое в применении, поэтому является точным и надежным. Более того, эти тесты создаются на точных языках, благодаря чему они легко воспринимаются аудиторией. Программисты могут читать модульные тесты, поскольку те написаны на используемом ими языке программирования. Пользователи могут читать приемочные тесты, поскольку они написаны на понятном для пользователей языке.

Возможно, самым важным преимуществом такого рода тестирования является его воздействие на архитектуру и проект. Чтобы протестировать модуль или приложение, его необходимо изолировать. Чем в большей степени является тестируемым модуль, тем больше он изолирован. Использование полных модульных тестов и тестов приемлемости весьма положительно влияет на структуру программы.

Литература

1. Mackinnon T., Craig P. Endo-Testing: Unit Testing with Mock Objects. *Extreme Programming Examined*. Addison-Wesley, 2001.
2. Jeffries R. *Extreme Programming Installed*. Upper Saddle River, NJ: Addison-Wesley, 2001.

5

Рефакторинг



Единственное, в чем ощущается недостаток в нашем Пресытившемся мире, — это человеческое внимание.

Кевин Келли

Эта небольшая глава целиком посвящается человеческому вниманию. Следует обращать самое пристальное внимание на то, чем ты занят, имея твердую уверенность в том, что все выполняется наилучшим образом. Иногда мы наблюдаем существенную разницу между тем, что реализуется, и тем, что должно быть реализовано. И большое значение в этом случае имеет структура разрабатываемого кода.

В своем классическом труде *Refactoring* Мартин Фаулер (Martin Fowler) определяет внимание как “... процесс, приводящий к такому изменению программной системы, когда для совершенствования внутренней структуры системы не требуется изменение внешнего поведения кода”¹. Но зачем улучшать структуру функционирующего кода? Недаром народная мудрость гласит: “Лучшее — враг хорошего”.

Каждый программный модуль выполняет три функции. Первая функция состоит в исполнении при запуске. Именно с этой целью и создается модуль. Вторая функция модуля — изменяемость структуры. Практически все модули время от времени модифицируются, и от умения разработчиков зависит, чтобы изменения были простейшими из возможных вариантов. Если модуль нуждается в корректировании, а его модификация весьма затруднена, разрабатывается новая версия модуля. Третья функция модуля заключается в поддержке связей с пользователями. Даже разработчики, которые незнакомы с модульной структурой, должны без особых усилий представить ее. Если модуль не соответствует этому требованию, вместо него создается новая версия.

Каким же образом можно разработать модуль, удобный для просмотра и внесения изменений? Большая часть этой книги посвящена изложению основных положений и примеров шаблонов, которые и должны помочь при создании гибких и легко адаптируемых модулей. Кроме основных принципов и шаблонов, в процессе разработки подобных модулей большое значение играют такие качества программиста, как внимание, дисциплина и наличие творческого подхода.

Генерирование простых чисел: простой пример использования рефакторинга²

Рассмотрим код из листинга 5.1. Здесь представлен код программы по генерированию простых чисел. В программе используется одна функция многих переменных (переменные обозначены отдельными буквами) и содержится ком-

¹[Flower99], с. xvi.

²Первоначально эта программа создавалась для XP Immersion и в ней использовались тесты, написанные Дж. Ньюкирком (Jim Newkirk), Кен Beck (Kent Beck) и Дж. Ньюкирк с помощью этой программы демонстрировали студентам возможности рефакторинга. Автор воссоздает здесь процесс рефакторинга.

ментарий, который призван помочь в процессе просмотра кода потенциальными пользователями.

Листинг 5.1. GeneratePrimes.java, версия 1

```
/**  
 * Этот класс генерирует простые числа вплоть  
 * до указанного пользователем предела. Алгоритм  
 * использует решето Эратосфена.  
 * <p>  
 * Эратосфен первым вычислил длину окружности Земли.  
 * Также известны его работы по разработке календаря,  
 * учитывающего високосные годы,  
 * руководил библиотекой в Александрии.  
 * <p>  
 * Алгоритм довольно прост. Рассмотрим массив целых  
 * чисел, начиная с 2. Вычеркнем все числа, кратные  
 * 2. Найдем следующее невычеркнутое число и  
 * вычеркнем все кратные ему числа.  
 * Продолжаем так до значения, равного корню  
 * квадратному из максимального числа  
 *  
 * @author Robert C. Martin  
 * @version 9 Dec 1999 rcm  
 */  
import java.util.*;  
  
public class GeneratePrimes  
{  
    /**  
     * @param maxValue предел генерирования.  
     */  
    public static int[] generatePrimes(int maxValue)  
    {  
        if (maxValue >= 2) // и только в этом случае  
        {  
            // объявления  
            int s = maxValue +1; // размер массива  
            boolean[] f = new boolean[s];  
            int i;  
  
            // инициализируйте массив значением true.  
            for (i = 0; i < s; i++)  
                f[i] = true;  
  
            // удалите известные числа, не являющиеся  
            // простыми  
            f[0] = f[1] = false;  
  
            // решето  
            int j;
```

```

for (i = 2; i < Math.sqrt(s) + 1; i++)
{
    if (f[i]) // если i не вычеркнуто,
        // вычеркните кратные ему числа.
    {

        for (j=2*i;j<s;j+=i)
        f[j] = false; // кратное не является
                    // простым
    }
}

// сколько простых чисел мы получим?
int count = 0;
for (i = 0; i < s; i++)
{
    if (f[i])
        count++; // подсчет прерывается.
}

int[] primes = new int[count];
// запишем простые числа, полученные в
// результате
for (i=0, j=0; i<s; i++)
{
    if (f[i]) // если простое
        primes[j++] = i;
}

return primes; // возвращаются простые числа
}
else // maxValue < 2
return new int[0]; // если начальное значение
// выбрано неудачно, возвращается нулевой массив.
}
}

```

Первоначальный тестовый модуль для программы `GeneratePrimes` представлен кодом из листинга 5.2. Предлагается статистический подход, позволяющий узнать, будут ли генерироваться простые числа в диапазоне от 0, 2, 3 до 100. Число 0 не является простым. Во втором случае получим одно простое число 2. В третьем случае имеем два простых числа, 2 и 3. В последнем случае получаем 25 простых чисел, последнее из них — 97. Если тестирование пройдет удачно, можно предположить, что генератор простых чисел функционирует успешно. Сомнения всегда имеют место, но трудно представить, чтобы при условии успешного прохождения тестов функция работала бы со сбоями.

Листинг 5.2. TestGeneratePrimes.java

```
import junit.framework.*;
import java.util.*;

public class TestGeneratePrimes extends TestCase
{
    public static void main(String args[])
    {
        junit.swingui.TestRunner.main(
            new String[] {"TestGeneratePrimes"});
    }
    public TestGeneratePrimes(String name)
    {
        super(name);
    }

    public void testPrimes()
    {
        int[] nullArray = GeneratePrimes.generatePrimes(0);
        assertEquals(nullArray.length, 0);

        int[] minArray = GeneratePrimes.generatePrimes(2);
        assertEquals(minArray.length, 1);
        assertEquals(minArray[0], 2);

        int[] threeArray = GeneratePrimes.generatePrimes(3);
        assertEquals(threeArray.length, 2);
        assertEquals(threeArray[0], 2);
        assertEquals(threeArray[1], 3);

        int[] centArray = GeneratePrimes.generatePrimes(100);
        assertEquals(centArray.length, 25);
        assertEquals(centArray[24], 97);
    }
}
```

Перейдем к рефакторингу данной программы. В данном случае применяется броузер рефакторинга *Idea* от *IntelliJ*. Этот инструмент значительно облегчает процесс извлечения и переименования переменных и классов.

Несложно заметить, что основную функцию можно разбить на три отдельные функции. Первая функция инициализирует все переменные и устанавливает начальные параметры решета. Вторая функция на практике реализует этот принцип, а третья — загружает отобранные числа в целочисленный массив. Эта структура представлена в коде из листинга 5.3, где функции скомпонованы в виде трех отдельных методов. Также удалено несколько необязательных комментариев, название класса изменено на *PrimeGenerator*. Тестирование, как и ранее, подтверждает правильность выполняемых действий.

В процессе разработки функций возникла идея превратить некоторые функциональные переменные в статические поля класса. Именно благодаря этому подходу можно более четко выделить локальные переменные, а также определить те из них, которые оказывают наибольшее влияние на ход выполнения.

Листинг 5.3. PrimeGenerator.java, версия 2

```
/**  
 * Этот класс генерирует простые числа до указанного  
 * пользователем максимального числа. Алгоритм  
 * использует решето Эратосфена. Данный массив целых  
 * чисел начинается с 2:  
 * Найдите первое невычеркнутое целое число и  
 * вычеркните все кратные ему числа.  
 * Повторяйте процесс до тех пор, пока первое  
 * невычеркнутое число не превысит квадратный  
 * корень из максимального числа.  
 */  
import java.util.*;  
public class PrimeGenerator  
{  
    private static int s;  
    private static boolean[] f;  
    private static int[] primes;  
  
    public static int[] generatePrimes(int maxValue)  
    {  
        if (maxValue < 2)  
            return new int[0];  
        else  
        {  
            initializeSieve(maxValue);  
            sieve();  
            loadPrimes();  
            return primes; // возвращает простые числа  
        }  
    }  
  
    private static void loadPrimes()  
    {  
        int i;  
        int j;  
  
        // сколько найдено простых чисел?  
        int count = 0;  
        for (i = 0; i < s; i++)  
        {  
            if (f[i])  
                count++; // подсчет прерывается.  
        }  
    }  
}
```

```
primes = new int[count];

// запишем полученные в результате простые числа
for (i = 0, j = 0; i < s; i++)
{
    if (f[i]) // если число простое
        primes[j++] = i;
}

private static void sieve()
{
    int i;
    int j;
    for (i = 2; i < Math.sqrt(s) + 1; i++)
    {
        if (f[i]) // если i не вычеркнуто,
            // вычеркните все кратные ему числа.
        {
            for (j = 2 * i; j < s; j += i)
                f[j] = false; // кратное не является простым
        }
    }
}

private static void initializeSieve(int maxValue)
{
    // объявления
    s = maxValue + 1; // размер массива
    f = new boolean[s];
    int i;
    // инициализируйте массив значением true.
    for (i = 0; i < s; i++)
        f[i] = true;
    // устраним известные числа, не являющиеся простыми
    f[0] = f[1] = false;
}
```

Функция `initializeSieve` несколько неупорядочена, поэтому в коде из листинга 5.4 она немного модифицируется. Во-первых, переменная `s` заменяется на `f.length`. Затем названия трех функций заменены более выразительными. Наконец, для удобства просмотра модифицирована внутренняя структура `initializeArrayOfIntegers` (первоначально `initializeSieve`). Тестирование по-прежнему подтверждает корректность выполняемых действий.

Листинг 5.4. PrimeGenerator.java, версия 3 (частичная реализация)

```

public class PrimeGenerator
{
    private static boolean[] f;
    private static int[] result;

    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            initializeArrayOfIntegers(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult();
            return result;
        }
    }

    private static void initializeArrayOfIntegers(int maxValue)
    {
        f = new boolean[maxValue + 1];
        f[0] = f[1] = false; //нет ни простых, ни кратных чисел.
        for (int i = 2; i < f.length; i++)
            f[i] = true;
    }
}

```

Затем попробуем модифицировать `crossOutMultiples`. Эта функция содержит несколько операторов, в том числе `if (f[i] == true)`. Этот оператор проверяет, не вычеркнуто ли число `i`, поэтому название `f` изменено на `unCrossed`. Но тогда возникает путаница с операторами типа `unCrossed[i] = false`. Здесь можно усмотреть даже два момента, приводящих к путанице. Поэтому название массива изменено на `isCrossed`, а также обновлено содержание всех булевых переменных. Тестирование по-прежнему подтверждает правильность выполняемых действий.

Устранена инициализация, при которой `isCrossed [0]` и `isCrossed [1]` присваивается значение `true`, и теперь можно с уверенностью заключить, что ни в одной части функции не используется массив `isCrossed` для индексов, не превышающих 2. Из функции `crossOutMultiples` устраняется внутренний цикл, и эта функция получает название `crossOutMultiplesOf`. Полагая, что оператор `if(isCrossed[i] == false)` выписан нечетко, вводим функцию `notCrossed`, а оператор `if` изменяем на `if(notCrossed(i))`. Тестирование по-прежнему подтверждает правильность выполняемых действий.

Довольно много времени затрачено на то, чтобы объяснить, почему итерация продолжается до извлечения квадратного корня из величины, соответствующей размеру массива. Чтобы включить поясняющий комментарий, вычисления были

перенесены в тело функции. Комментарий подчеркивает тот факт, что квадратный корень из величины, равной размеру массива, является максимальным простым сомножителем для любого целого числа, содержащегося в массиве. Также подобраны названия всех переменных и функций. В результате проведенного рефакторинга получим код, приведенный в листинге 5.5. Тестирование по-прежнему подтверждает правильность выполняемых действий.

Листинг 5.5. PrimeGenerator.java версия 4 (частичная реализация)

```
public class PrimeGenerator
{
    private static boolean[] isCrossed;
    private static int[] result;

    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            initializeArrayOfIntegers(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult();
            return result;
        }
    }

    private static void initializeArrayOfIntegers(int maxValue)
    {
        isCrossed = new boolean[maxValue + 1];
        for (int i = 2; i < isCrossed.length; i++)
            isCrossed[i] = false;
    }

    private static void crossOutMultiples()
    {
        int maxPrimeFactor = calcMaxPrimeFactor();
        for (int i = 2; i <= maxPrimeFactor; i++)
            if (notCrossed(i))
                crossOutMultiplesOf(i);
    }

    private static int calcMaxPrimeFactor()
    {
        // Вычеркиваются все числа, кратные p, где p
        // является простым числом.
        // Поэтому все вычеркнутые кратные имеют в
        // качестве сомножителей p и q.
        // Если p > sqrt для размера массива, тогда
        // q не превысит значения 1. Таким образом, p
```

```

// является наибольшим простым сомножителем в
// массиве, а также пределом итерации.

double maxPrimeFactor = Math.sqrt(isCrossed.length)+1;
return (int) maxPrimeFactor;
}

private static void crossOutMultiplesOf(int i)
{
    for (int multiple = 2*i;
        multiple < isCrossed.length;
        multiple += i)
        isCrossed[multiple] = true;
}

private static boolean notCrossed(int i)
{
    return isCrossed[i] == false;
}

```

Последней функцией, подвергаемой рефакторингу, является `putUncrossedIntegersIntoResult`. Она состоит из двух компонентов. Первый компонент реализует подсчет количества невычеркнутых целых чисел в массиве, после чего создается результирующий массив аналогичного размера. Затем невычеркнутые целые числа передаются в результирующий массив. Первая часть задания теперь включена в отдельную функцию, проделана также определенная дополнительная работа. Тестирование по-прежнему подтверждает правильность выполняемых действий.

Листинг 5.6. PrimeGenerator.java, версия 5 (частичная реализация)

```

private static void putUncrossedIntegersIntoResult()
{
    result = new int[numberOfUncrossedIntegers()];
    for (int j=0, i=2; i < isCrossed.length; i++)
        if (notCrossed(i))
            result[j++] = i;
}

private static int numberOfUncrossedIntegers()
{
    int count = 0;
    for (int i = 2; i < isCrossed.length; i++)
        if (notCrossed(i))
            count++;

    return count;
}

```

Заключительный просмотр



Просмотрим программу еще раз, от начала и до конца, аналогично тому, как изучаются геометрические доказательства. Это довольно ответственный момент. При разработке программы использовались элементы рефакторинга. Теперь нужно проверить, будет ли программа иметь цельный вид, *удобный для просмотра*.

Во-первых, неудачным выглядит название `initializeArrayOfIntegers`. Фактически, в данном случае идет речь не о массиве целых чисел, а о массиве булевых значений. Функция `initializeArrayOfBooleans` не является улучшенной версией предыдущей функции. В процессе ее выполнения отыскиваются все невычеркнутые целые числа, в результате чего обеспечивается вычеркивание всех кратных им чисел. Исходя из этих соображений, название функции изменено на `uncrossIntegersUpTo`. Также можно прийти к выводам о том, что название `isCrossed` не подходит для массива, содержащего булевые значения. Поэтому было использовано новое название, `crossedOut`. Тестирование по-прежнему подтверждает правильность выполняемых программой действий.

Может сложиться впечатление, что изменение названий функций не столь важно, поскольку при наличии броузера рефакторинг выполнение подобных модификаций не составит особого труда, а также не потребует больших затрат средств. Даже в случае отсутствия подобного броузера, простой поиск и замена обеспечивают все необходимые изменения при минимальных затратах. А благодаря проведенным тестам сводится к минимуму риск, связанный с неосторожным изменением кода.

Автор вряд ли помнит все обстоятельства, приведшие к написанию модуля `maxPrimeFactor`. Значение квадратного корня из величины размера массива необязательно образует простое число. Этот метод *не* позволяет вычислить наибольший простой сомножитель. Сопроводительный комментарий, мягко говоря, просто *некорректен*. Поэтому комментарий был изменен в целях лучшего объяс-

нения понятий, связанных с операцией извлечения квадратного корня³, а также соответствующим образом были переименованы все переменные. Тесты по-прежнему не выявили каких-либо ошибок.

Какую роль играет в этом случае $+1$? Автор весьма опасался того, что дробная часть квадратного корня в результате преобразования в целочисленное число будет слишком малой, чтобы служить в качестве предельного значения, приводящего к завершению итераций. Ведь это просто глупо! Фактический предел итерации — это наибольшее простое число, которое меньше или равно квадратному корню из числа, определяющего размер массива. Лучше вообще исключить $+1$.

Все тесты выполнены, но именно последние изменения могут “вывести из равновесия”. Нетрудно уяснить математический смысл квадратного корня, но создается впечатление, что существует нечто важное, что было упущено из рассмотрения. Поэтому будет разработан следующий тест, выполняющий проверку на предмет отсутствия кратных чисел в списке, содержащем простые числа в диапазоне от 2 до 500. (Обратите внимание на код функции `testExhaustive` в листинге 5.8.) После прогона новых тестов имеющие место опасения в значительной степени “испарились”.

Остальная часть программного кода не требует дополнительных объяснений. Окончательная версия кода приведена в листингах 5.7 и 5.8.

Листинг 5.7. PrimeGenerator.java (заключительная версия)

```
/**  
 * Этот класс генерирует простые числа до указанного  
 * пользователем максимального числа. Алгоритм  
 * использует решето Эратосфена. Данный массив целых  
 * чисел начинается с 2:  
 * Найдите первое невычеркнутое целое число и  
 * вычеркните все кратные ему числа.  
 * Повторяйте процесс до тех пор, пока в массиве  
 * не останутся одни лишь простые числа  
 */  
  
public class PrimeGenerator  
{  
    private static boolean[] crossedOut;  
    private static int[] result;  
  
    public static int[] generatePrimes(int maxValue)  
    {
```

³Когда Кен Бек и Джим Ньюкирк выполняли рефакторинг этой программы, они вообще исключили операцию извлечения квадратного корня. Кен обосновал это тем, что понятие квадратного корня трудно для понимания, а также не существует теста, который завершился бы сбоем в случае достижения границы массива (в процессе последовательных итераций). Я полагаю, что причина этого заключается в том, что в прошлом я программировал на ассемблере.

```
if (maxValue < 2)
    return new int[0];
else
{
    uncrossIntegersUpTo(maxValue);
    crossOutMultiples();
    putUncrossedIntegersIntoResult();
    return result;
}
}

private static void uncrossIntegersUpTo(int maxValue)
{
    crossedOut = new boolean[maxValue + 1];
    for (int i = 2; i < crossedOut.length; i++)
        crossedOut[i] = false;
}

private static void crossOutMultiples()
{
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
        if (notCrossed(i))
            crossOutMultiplesOf(i);
}

private static int determineIterationLimit()
{
    // Каждое число в массиве имеет кратный
    // ему простой делитель, который меньше или
    // равен квадратному корню из размера массива,
    // поэтому не следует вычеркивать делители
    // чисел, величина которых превышает значение
    // этого корня.
    double iterationLimit = Math.sqrt(crossedOut.length)
    return (int) iterationLimit;
}

private static void crossOutMultiplesOf(int i)
{
    for (int multiple = 2*i;
        multiple < crossedOut.length;
        multiple += i)
        crossedOut[multiple] = true;
}

private static boolean notCrossed(int i)
{
    return crossedOut[i] == false;
```

```
private static void putUncrossedIntegersIntoResult()
{
    result = new int[numberOfUncrossedIntegers()];
    for (int j=0,i=2;i< crossedOut.length; i++)
        if (notCrossed(i))
            result[j++] = i;
}

private static int numberOfUncrossedIntegers()
{
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
            count++;

    return count;
}
}
```

Листинг 5.8. TestGeneratePrimes.java (заключительная версия тестового модуля)

```
import junit.framework.*;

public class TestGeneratePrimes extends TestCase
{
    public static void main(String args[])
    {
        junit.swingui.TestRunner.main(
            new String[] {"TestGeneratePrimes"});
    }
    public TestGeneratePrimes(String name)
    {
        super(name);
    }

    public void testPrimes()
    {
        int[] nullArray = PrimeGenerator.generatePrimes(0);
        assertEquals(nullArray.length, 0);

        int[] minArray = PrimeGenerator.generatePrimes(2);
        assertEquals(minArray.length, 1);
        assertEquals(minArray[0], 2);

        int[] threeArray = PrimeGenerator.generatePrimes(3);
        assertEquals(threeArray.length, 2);
        assertEquals(threeArray[0], 2);
        assertEquals(threeArray[1], 3);
    }
}
```

```
int[] centArray = PrimeGenerator.generatePrimes(100);
assertEquals(centArray.length, 25);
assertEquals(centArray[24], 97);
}

public void testExhaustive()
{
    for (int i = 2; i<500; i++)
        verifyPrimeList(PrimeGenerator.generatePrimes(i));
}

private void verifyPrimeList(int[] list)
{
    for (int i=0; i<list.length; i++)
        verifyPrime(list[i]);
}

private void verifyPrime(int n)
{
    for (int factor=2; factor<n; factor++)
        assert(n%factor != 0);
}
}
```

Резюме

Конечный результат наших усилий выглядит значительно лучше, чем первые робкие попытки. В процессе работы был существенно усовершенствован программный алгоритм. Автор вполне удовлетворен достигнутыми результатами. Программа значительно упростилась для восприятия, следовательно, ее стало проще модифицировать. Также структура программы способствует изоляции ее отдельных компонентов. Это обстоятельство облегчает изменение программного кода в будущем.

Может возникнуть потребность в исключении функций, вызываемых однократно. Подобные функции могут оказывать негативное влияние на быстродействие программы. Также в большинстве случаев можно улучшить читабельность кода, пожертвовав несколькими наносекундами производительности. Причем подобная “жертва” начинает сказываться только в случае наличия нескольких вложенных циклов. Я полагаю, что потери в этом случае незначительны и ожидаю от вас доказательств обратного утверждения.

Каким же образом можно воспользоваться временем, сэкономленным благодаря повышению производительности? Помимо всего прочего, польза от применения какой-либо функции выявляется в процессе ее применения. Настоятельно рекомендуется *всегда* применять процедуру рефакторинга к *каждому* написан-

ному пользователем модулю, а также ко всем сопровождаемым вами модулям. Затраты времени будут поистине смехотворными по сравнению с теми трудозатратами, которых избежите вы и ваши компаньоны в ближайшем будущем.

Процедура рефакторинга напоминает уборку в кухне после обеда. Первое время можно избегать этого неприятного процесса, благодаря чему уменьшится суммарное время, выделяемое на проведение обеда. Однако недостаток чистых тарелок и рабочего пространства приведет к тому, что к обеду следующего дня придется дольше готовиться. И снова вам захочется исключить процедуру уборки кухни. Несомненно, если пропустите этап уборки, то сможете завершить обед быстрее *сегодня*, но при этом беспорядок будет постоянно накапливаться. И наступит время, когда вам придется буквально “охотиться” за чистой посудой, “вырубая зубилом” остатки засохшей пищи с тарелок, а также прилагая “героические усилия” для мытья остальной посуды. В результате, обед может растянуться до бесконечности. Пропуск этапа уборки не приводит к реальному уменьшению времени, затрачиваемого на обед.

В этой главе уже упоминалось о том, что целью рефакторинга является ежедневная “уборка” кода. Не следует допускать накопления беспорядка. Необходимо сохранить возможности для расширения и изменения системы с минимальными усилиями. Во главу угла всегда следует ставить цель, заключающуюся в обеспечении “чистоты” кода.

Автору, имеющему большой опыт работы в качестве программиста, надоели бесконечные стрессы. Все принципы и шаблоны, описанные в этой книге, превращаются в ничто, если код остается неупорядоченным. Поэтому прежде всего стремитесь к достижению “чистоты” исходного кода.

Литература

1. Fowler M. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

6

Пример из практики программирования



Разработка проектов и программирование относятся к сфере человеческой деятельности. Стоит об этом забыть, как все будет потеряно.

Бьери Страуструп

А теперь перейдем к рассмотрению практики экстремального программирования. Боб Косс (Bob Koss, БК) и Боб Мартин (Bob Martin, БМ) на ваших глазах разработают весьма простую программу. Процесс создания этого приложения реализуется на основе методики пробных испытаний (тест-драйв). Также используется рефакторинг. В основе этого примера лежит реальная разработка программы, выполненная двумя Бобами в номере гостиницы, где они останавливались в конце 2000 года.

В процессе разработки было допущено достаточно много ошибок. Некоторые из них относятся к программному коду, остальные связаны с логикой, проектированием, а также с некорректным формулированием требований. В этой главе демонстрируется процесс идентификации, локализации и устранения ошибок, а также некорректно сформированных концепций в самых различных областях. Описанный процесс является в достаточной степени хаотичным, как и все остальные процессы, осуществляемые людьми. Полученный результат примечателен тем, что демонстрирует способ “выделения” из среды подобных беспорядочных процессов.

Программа предназначена для подсчета количества очков, заработанных в боулинге, поэтому сначала следует ознакомиться с правилами этой игры. Если вы еще незнакомы с этой игрой, прочтите правила, помещенные в соответствующем примечании в конце этой главы.

Игра в боулинг

- БМ. Не хотите ли мне помочь в написании компактной программы, предназначенной для подсчета количества очков в боулинге?
- БК. (Мысленно: практика экстремального программирования рекомендует ни в коем случае не отвечать “нет”, услышав просьбу о помощи. Это особенно верно в том случае, когда об этом просит ваш босс.) Хорошо, Боб, я рад помочь тебе.
- БМ. Отлично! Мне бы хотелось написать программу, которая отслеживает результаты чемпионата по игре в боулинг. Программа будет предназначена для фиксации результатов всех игр, определения рейтинга команд согласно турнирной таблице, еженедельного определения победителей и проигравших, а также точного подсчета результатов каждой игры.

БК. Отлично. Я стану хорошим игроком в боулинг. Эта идея мне очень нравится. Ты рассказал мне несколько пользовательских историй, с какой же следует начать?

БМ. Давай начнем с разработки алгоритма подсчета результатов простой игры.

БК. Хорошо. Что это может означать на практике? Каковы входные и выходные данные для этой истории?

БМ. Мне кажется, что в качестве входных данных выступает некая последовательность бросков. Сам бросок моделируется целым числом, которое соответствует количеству кеглей, сбитых шаром. Выходные данные — количество очков, заработанных в каждой игре.

БК. Поскольку вы являетесь заказчиком этой программы, определите форму, в которой будут выражаться входные/выходные данные?

БМ. Хорошо, я согласен быть заказчиком. Для начала следует написать функцию, которая предназначается для вычисления суммы всех бросков, а также другую функцию, которая передает результаты бросков. В данном случае идет речь о следующем коде.

```
throwBall(6);
throwBall(3);
assertEquals(9, getScore());
```

БК. Хорошо, теперь нам потребуется набор тестовых данных. Позволь мне набросать эскиз карточки для записи результатов игры (рис. 6.1).

1	4	4	5	6		5		0	1	7		6		2	6
5		14		29		49		60		61		77		97	

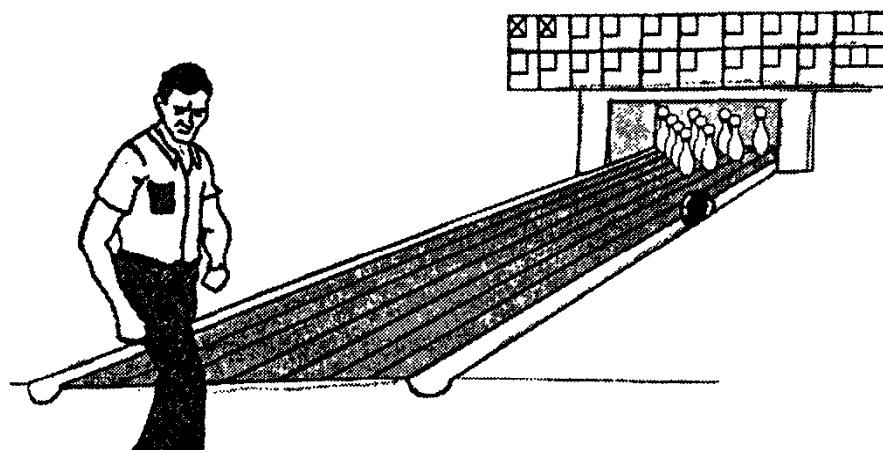
Рис. 6.1. Типичная карточка для записи очков, полученных при игре в боулинг

БМ. Автор этой карточки ведет себя достаточно странно.

БК. Или просто пьян, хотя на самом деле эта карточка может служить неплохим приемочным тестом.

БМ. Нам потребуются и другие тесты, но поговорим о них немного позже. С чего же следует начать? Может быть, с разработки проекта системы?

БК. Можно начать с разработки UML-диаграммы, наглядно демонстрирующей концепцию проблемной области, иллюстрируемой примером с карточкой. При этом появляются некие объекты, которые можно исследовать с помощью программного кода.



БМ. (Снимает шляпу перед опытным дизайнером объектов.) Хорошо, ясно, что объект Game состоит из последовательности, образуемой десятью фреймами (подходами) игры. Каждый объект Frame предусматривает один, два или три броска.

БК. Гениально. Я тоже придерживаюсь подобной стратегии. Позвольте мне проиллюстрировать сказанное (рис. 6.2.).

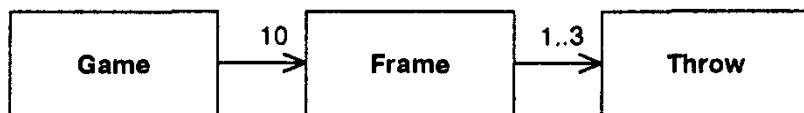


Рис. 6.2. UML-диаграмма, моделирующая карточку для записи результатов игры в боулинг

БК. Хорошо, а теперь возьмем класс... произвольный класс. Давай начнем с конца цепи зависимостей и будем продвигаться в обратном направлении. Благодаря этому облегчается процесс тестирования.

БМ. Хорошо, почему бы и нет. Давай разработаем тестовый случай для класса Throw.

БК. Начинает вводить следующий код.

```

//TestThrow.java-----
import junit.framework.*;

public class TestThrow extends TestCase
{
    public TestThrow(String name)
    {
        super(name);
    }
    // public void test?????
}
  
```

БК. Есть ли у тебя ключ к определению поведения объекта Throw?

- БМ. Я сохраняю сведения о количестве кеглей, сбитых игроком.
- БК. Хорошо, ты только что сказал в нескольких словах, что этот объект ничего особого не делает. Может, есть смысл вернуться назад и сосредоточить внимание на объекте, который фактически моделирует поведение вместо того, чтобы выполнять функции простого хранилища данных.
- БМ. Любопытно. Ты полагаешь, что класс Throw на самом деле не существует?
- БК. Да, но если он не включает поведение, какую ценность он может представлять? Я не знаю, существует ли этот класс в настоящее время. Я считаю, что будет гораздо лучше, если мы будем работать с объектом, который представляет собой нечто большее, чем хранилище для функций установки и получения значений методов. Но если у тебя есть кое-что интересное, милости просим... (передает клавиатуру БМ).
- БМ. Хорошо, переместим цепь зависимости в класс Frame и посмотрим, можно ли разработать тестовые случаи, которые позволят завершить формирование класса Throw. (Передаем клавиатуру обратно БК.)
- БК. (Удивляюсь, заводит ли БМ меня в тупик в процессе обучения или на самом деле он соглашается со мной.) Хорошо, новый файл, новый тестовый случай.

```
//TestFrame.Java-----
import junit.framework.*;
public class TestFrame extends TestCase
{
    public TestFrame( String name )
    {
        super( name );
    }
    //public void test???
}
```

- БМ. Хорошо, во второй раз мы воспользовались этим кодом. Теперь следует подумать о тестовом случае для класса Frame?
- БК. Класс Frame хранит результаты, количество сбитых кеглей...
- БМ. Хорошо, покажи мне этот код.

```
//TestFrame.Java-----
import junit.framework.*;

public class TestFrame extends TestCase
{
    public TestFrame( String name )
    {
        super( name );
    }
    public void testScoreNoThrows()
```

```

{
    Frame f = new Frame();
    assertEquals( 0, f.getScore() );
}
}

//Frame.java-----
public class Frame
{
    public int getScore()
    {
        return 0;
    }
}

```

- БМ. Хорошо, тестовый случай вполне подходящий, но функция `getScore` на самом деле не выдерживает никакой критики. Она отказывается работать в случае, если в класс `Frame` добавляется информация о бросках. Давай разрабатаем тестовый случай, обеспечивающий добавление информации о бросках, а также позволяющий выполнить подсчет количества очков.

```

//TestFrame.java-----
public void testAddOneThrow()
{
    Frame f = new Frame();
    f.add(5);
    assertEquals(5, f.getScore());
}

```

- БМ. Этот код не следует компилировать. Здесь отсутствует метод, реализующий добавление значений в класс `Frame`.

- БК. Будет лучше, если ты определишь этот метод, а я выполню компиляцию:-)

```

//Frame.java-----
public class Frame
{
    public int getScore()
    {
        return 0;
    }

    public void add(Throw t)
    {
    }
}

```

- БМ. Вряд ли этот код можно скомпилировать, ведь мы до сих пор не определили класс `Throw`.

- БК. Объясни мне, Боб, следующее. Тестовый случай передает целочисленное значение, а метод ожидает объект `Throw`. Невозможно определить его двумя

способами. Прежде чем перейти к Throw снова, можешь ли ты описать его поведение?

- БМ. Фантастика! Вряд ли мне удастся сказать что-либо хорошее даже о f.add(5). Следовало бы описать конструкцию written f.add(new Throw (5)), но она мне совершенно не нравится. Неужели кому-то придется по душе это безобразное чудовище: f.add(5).
- БК. Нравится или нет, оставим эстетические соображения, здесь они неуместны. Можешь ли ты, Боб, определить поведение объекта Throw (в виде двоичного кода)?
- БМ. 101101011010100101. Я не знаю, относится ли это к поведению объекта Throw. Я начинаю думать об объекте Throw в “целочисленном” стиле. Тем более что целочисленный подход может быть реализован с применением метода Frame.add.
- БК. Тем более, что самое главное — придерживаться максимальной простоты. Если нас что-то не будет устраивать, можно немного “ усложнить” картину.
- БМ. Согласен.

```
//Frame.java-----
public class Frame
{
    public int getScore()
    {
        return 0;
    }

    public void add(int pins)
    {
    }
}
```

- БМ. Хорошо, выполним компиляцию и успешно “провалим” тест. Давай выполним тестовый прогон.

```
//Frame.Java-----
public class Frame
{
    public int getScore
    {
        return itsScore;
    }

    public void add(int pins)
    {
        itsScore += pins;
    }
    private int itsScore =0;
```

БМ. Компиляция и прогон тестов были выполнены успешно, но начальные условия были слишком просты. Как насчет еще одного тестового случая?

БК. Может, сначала немного передохнем?

-----Перерыв-----

БМ. Теперь лучше. Название `Frame.add` не очень подходяще. Может, лучше назовем функцию `11`?

БК. Это может привести к возникновению исключительной ситуации. Следует подумать о пользователях этой функции. Выполняет ли она роль схемы приложения, используемой тысячами людей, и вследствие этого нуждается в дополнительной защите. Или же эта функция предназначена для “внутреннего применения”? В последнем случае не следует использовать название `11 (chuckle)`.

БМ. Хорошее начало, при тестировании других компонентов системы будут перехватываться некорректные аргументы. Если же возникнут какие-либо проблемы, можно будет реализовать дополнительную проверку в дальнейшем. В настоящее время функция `add` не может обрабатывать результаты страйков (`strike`) или спайра (`spare`). Давай разработаем тестовый случай, который устраняет подобную проблему.

БК. Гмм... Если функция `add(10)` будет моделировать страйк, какой результат должна возвращать функция `getScore()`? Мне неизвестно, каким образом можно сформулировать подходящее утверждение. Может, это связано с тем, что мы пытаемся ответить на некорректно сформулированный вопрос. Или сформулирован правильный вопрос о некорректно определенном объекте.

БМ. Если выполнить вызов `add(10)` или `add(3)`, а затем — `add(7)`, беспомысленно затем вызывать функцию `getScore` для объекта `Frame`. Объект `Frame` должен рассматриваться дальнейшими экземплярами `Frame` в целях подсчета количества очков. Если упомянутые экземпляры `Frame` не существуют, могут возвращаться непредвиденные результаты, например, `-1`, что недопустимо.

БК. Да, мне тоже не нравятся значения типа `-1`. Ты предложил идею относительно объектов `Frames`, которым известно относительно других объектов `Frames`. Где будут храниться различные объекты `Frame`?

БМ. Для этого предназначается объект `Game`.

БК. Да, объект `Game` зависит от объекта `Frame`; объект `Frame`, в свою очередь, зависит от объекта `Game`. Мне это не нравится.

БМ. Объекты `Frames` не зависят от объекта `Game`; в данном случае можно воспользоваться сортировкой в форме связанного списка. Каждый объект `Frame` может хранить указатели на предыдущий и следующий объекты `Frames`.

Объект `Frame` может выглядеть “отстающим” в целях получения сведений о количестве очков из предыдущего `Frame`, или “опережающим” — при необходимости получения сведений о результатах, связанных со страйками или спайрами (в случае необходимости).

- БК. Хорошо, но я не могу представить код, соответствующий описанной ситуации. Продемонстрируй мне соответствующий пример.
- БМ. Хорошо, но для начала следует обратиться к тестовому случаю. Этот случай будет предназначаться для объекта `Game` или `Frame`?
- БМ. Мне кажется, что тест должен быть ориентирован на `Game`, поскольку именно этот объект инициирует формирование объектов `Frames` с последующим установлением связей между ними.
- БК. Хочешь ли ты прервать работу с `Frame` и перейти к объекту `Game`? Или просто воспользоваться объектом `MockGame`, который требуется для работы с объектом `Frame`?
- БМ. Нет, давай прекратим работу с объектом `Frame` и перейдем к объекту `Game`. Тестовые случаи, предназначенные для объекта `Game`, предназначены для осуществления проверки того, что мы нуждаемся в связанном списке объектов `Frames`.
- БК. Я не понимаю, каким образом можно продемонстрировать необходимость использования списка. Мне нужен конкретный пример кода.

БМ вводит следующий код.

```
//TestGame.java-----
import junit.framework.*;

public class TestGame extends TestCase
{
    public TestGame(String name)
    {
        super(name);
    }

    public void testOneThrow()
    {
        Game g = new Game();
        g.add(5);
        assertEquals(5, g.score());
    }
}
```

- БМ. Теперь вроде как все приемлемо?
- БК. Согласен, но я по-прежнему рассматриваю возможность проверки перечня объектов `Frames`.

БМ. Согласен с тобой. Давай обратимся к следующим тестовым случаям, а также рассмотрим результаты их выполнения.

```
//Game.java-----
public class Game
{
    public int score()
    {
        return 0;
    }

    public void add(int pins)
    {
    }
}
```

БМ. Хорошо, этот код компилируется, а результаты прогона теста будут неудачными. Давай выполним прогон теста.

```
//Game.Java-----
public class Game
{
    public int score()
    {
        return itsScore;
    }

    public void add(int pins)
    {
        itsScore += pins;
    }
    private int itsScore = 0;
}
```

БМ. Прогон теста прошел успешно.

БК. Не согласен с этим мнением, поскольку по-прежнему рассматриваю этот пример в качестве доказательства необходимости связанных объектов Frame. Поэтому на первом месте должен находиться объект Game.

БМ. Да, я не возражаю. Я более чем уверен в том, что как только мы сформируем тестовые случаи, позволяющие оценить результаты страйков и спайров, мы образуем объекты Frames, а затем построим на их основе связанный список. Но я не собираюсь заниматься этим до тех пор, пока не возникнет настоятельная потребность.

БК. Согласен с этой точкой зрения. Давай продвигаться небольшими шагами в направлении объекта Game. Что можешь сказать относительно следующего теста, который предусматривает два броска, но не учитывает спайр (spare)?

БМ. Хорошо, давай выполним прогон этого теста прямо сейчас.

```
//TestGame.Java-----  
  
public void testTwoThrowsNoMark()  
{  
    Game g = new Game();  
    g.add(5);  
    g.add(4);  
    assertEquals(9, g.score());  
}
```

- БМ. Да, прогон был удачным. Теперь выполним тест с применением четырех шаров, которые не являются целевыми.
- БК. Хорошо, и в этом случае тест завершился удачно. Я не ожидал этого. Мы можем сохранять результаты бросков, но до сих пор не использовали объект Frame. Но мы до сих пор не занимались обработкой результатов спайра и страйков. Не пора ли приступить к этому прямо сейчас.
- БМ. Я учту эти соображения. Обрати внимание на следующий тестовый случай.

```
//TestGame.Java-----  
public void testFourThrowsNoMark()  
{  
    Game g = new Game();  
    g.add(5);  
    g.add(4);  
    g.add(7);  
    g.add(2);  
    assertEquals(18, g.score());  
    assertEquals(9, g.scoreForFrame(1));  
    assertEquals(18, g.scoreForFrame(2));  
}
```

- БМ. Достаточно ли убедительно это выглядит?
- БК. Думаю, что вполне нормально. Я забыл о том, что следует предусмотреть возможность подсчета очков для каждого фрейма. Да, этот эскиз карточки для подсчета очков можно использовать разве что в качестве подставки под стакан с кока-колой. Да, склероз — вещь неприятная.
- БМ. Хорошо, мы добьемся того, что тестовый случай завершится неудачей путем включения метода scoreForFrame в объект Game.

```
//Game.Java-----  
public int scoreForFrame(int frame)  
{  
    return 0;  
}
```

- БМ. Результат компиляции был неудачным. Каким же образом можно выполнить прогон?

- БК. Мы можем начать с создания объектов Frame, но в этом случае будет выполненен прогон лишь простейших тестов, не так ли?
- БМ. Нет, фактически мы прямо сейчас можем создать целочисленный массив, включив его в состав объекта Game. При каждом вызове метода add в массив добавляется целое число. В результате вызова функции scoreForFrame производится опережающий просмотр массива и вычисление очков.

```
//Game.Java-----
public class Game
{
    public int score()
    {
        return itsScore;
    }
    public void add(int pins)
    {
        itsThrows[itsCurrentThrow++]=pins;
        itsScore += pins;
    }

    public int scoreForFrame(int frame)
    {
        int score = 0;
        for ( int ball = 0;
              frame > 0 && (ball < itsCurrentThrow);
              ball+=2, frame--)
        {
            score += itsThrows[ball] + itsThrows[ball+1];
        }
        return score;
    }
    private int itsScore = 0;
    private int[] itsThrows = new int[21];
    private int itsCurrentThrow =0;
}
```

- БМ. Все работает.
- БК. Почему появилось магическое число 21?
- БМ. Это число выражает максимально возможное количество бросков в игре.
- БК. Да уж. Мне кажется, что когда-то ты был хакером в среде Unix. Эти люди считают особым шиком закодировать целое приложение в виде одной конструкции, которую никто не сможет декодировать.
- К функции scoreForFrame() потребуется применить рефакторинг, после чего степень ее коммуникативности возрастет. Но прежде давай попробуем ответить на следующий вопрос. Является ли объект Game лучшим местом для размещения этого метода? Мне кажется, что Game нарушает принцип

персональной ответственности¹. В этом случае принимаются броски, а также определяется способ подсчета количества очков для каждого фрейма. Что ты думаешь относительно объекта Scorer?

- БМ. Я не знаю, в каком месте определяются функции; на самом деле я заинтересован исключительно в том, чтобы запустить в работу механизм подсчета количества очков. Как только мы сделаем все необходимое, можно будет поговорить о ценности принципа SRP.

А теперь давай попытаемся упростить цикл, написанный в традициях Unix-хакеров.

```
public int scoreForFrame(int theFrame)
{
    int ball = 0;
    int score=0;
    for (int currentFrame = 0;
        currentFrame < theFrame;
        currentFrame++)
    {
        score += itsThrows[ball++] + itsThrows[ball++];
    }

    return score;
}
```

- БМ. Выглядит немного лучше, но я замечаю некие “побочные эффекты” в выражении `score+=`. В данном случае это не имеет особого значения, поскольку порядок двух сравниваемых значений несущественен. Или все же порядок имеет значение? Возможно ли, чтобы два приращения выполнялись прежде любой операции над элементами массива?

- БК. Мы можем провести эксперимент, в процессе которого определим возможное влияние “побочных эффектов”, но рассматриваемая функция не может применяться для обработки результатов спайров и страйков. Следует ли нам придать коду этой функции “читабельный” характер, или следует сделать упор на дальнейшее расширение функциональных возможностей?

- БМ. Результаты эксперимента будут достоверными только в случае применения определенных компиляторов. В случае с другими компиляторами возможно применение другого порядка оценивания. Не знаю, имеет ли это значение, но давай все же избавимся от потенциальной зависимости от порядка оценивания и перейдем к рассмотрению дополнительных тестовых случаев.

```
public int scoreForFrame(int theFrame)
{
    int ball = 0;
```

```

int score=0;
for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
{
    int firstThrow = itsThrows[ball++];
    int secondThrow = itsThrows[ball++];
    score += firstThrow + secondThrow;
}

return score;
}

```

БМ. Хорошо, перейдем к рассмотрению следующего тестового случая. Попытаемся обработать результаты спайра.

```

public void testSimpleSpare()
{
    Game g = new Game ();
}

```

БМ. Я устал вводить такой объем кода. Давай применим рефакторинг к тестовой программе и передадим обязанности по разработке игры функции `setUp`.

```

//TestGame.Java -----
import junit.framework.*;

public class TestGame extends TestCase
{
    public TestGame(String name)
    {
        super(name);
    }

    private Game g;

    public void setup()
    {
        g = new Game();
    }
    public void testOneThrow()
    {
        g.add(5);
        assertEquals(5, g.score());
    }

    public void testTwoThrowsNoMark()
    {
        g.add(5);
        g.add(4);
        assertEquals (9, g.score());
    }
}

```

```
}

public void testFourThrowsNoMark()
{
    g.add(5);
    g.add(4);
    g.add(7);
    g.add(2);
    assertEquals(18, g.score ());
    assertEquals(9, g.scoreForFrame(1)
    assertEquals(18, g.scoreForFrame(2)
}

public void testSimpleSpare()
{
}
}
```

БМ. Это гораздо лучше, теперь займемся тестовым случаем, реализующим подсчет количества очков в случае спайра.

```
Public void testSimpleSpare()
{
    g.add(3);
    g.add(7);
    g.add(3);
    assertEquals(13, g.scoreForFrame(1));
}
```

БМ. Хорошо, этот тестовый случай привел к сбою. Что нужно сделать для достижения успеха.

БК. Я попытаюсь исправить ситуацию.

```
public int scoreForFrame(int theFrame)
{
    int ball = 0;
    int score=0;
    for (int currentFrame = 0;
        currentFrame < theFrame;
        currentFrame++)
    {
        int firstThrow = itsThrows[ball++];
        int secondThrow = itsThrows[ball++];

        int frameScore = firstThrow + secondThrow;
        // спайр требует переход к следующим
        // фреймам перед броском
        if ( frameScore == 10 )
            score += frameScore + itsThrows[ball++];
        else
```

```

        score += frameScore;
    }
    return score;
}

```

БК. Ура! Все работает!

БМ. Хорошо, но думаю, что вычисление приращения движения мяча в конструкции `frameScore==10` не совсем уместно. Вот тестовый случай, подтверждающий мою точку зрения.

```

public void testSimpleFrameAfterSpare()
{
    g.add(3);
    g.add(7);
    g.add(3);
    g.add(2);
    assertEquals(13, g.scoreForFrame(1));
    assertEquals(18, g.score());
}

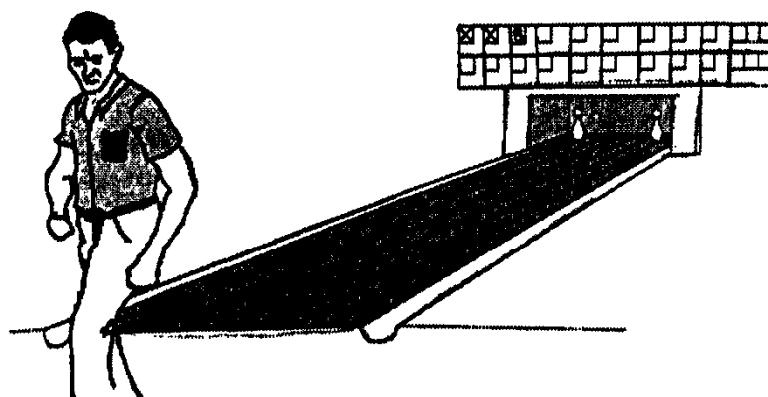
```

БМ. Ха! Видишь, все закончилось сбоем. Давай избавимся от этого “надоедливого” приращения...

```

if ( frameScore == 10 )
    score += frameScore + itsThrows[ball];

```



БМ. Снова ничего не получается... Может быть неверен применяемый метод подсчета количества очков? Давай проведем повторное тестирование, воспользовавшись функцией `scoreForFrame(2)`.

```

public void testSimpleFrameAfterSpare()
{
    g.add(3);
    g.add(7);
    g.add(3);
    g.add(2);
    assertEquals(13, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
}

```

БМ. Интересно... Прогон прошел удачно. Метод подсчета количества очков должен быть изменен. Обрати внимание на следующий код.

```
public int score()
{
    return itsScore;
}

public void add(int pins)
{
    itsThrows[itsCurrentThrow++]=pins;
    itsScore += pins;
}
```

БМ. Ты допустил ошибку. Это вариант метода подсчета количества очков просто возвращает суммарное количество кеглей, которое не имеет отношения к корректной сумме очков. Для подсчета количества очков следует вызвать функцию `scoreForFrame()` в текущем фрейме.

БК. Мы не можем определить текущий фрейм. Давай хотя бы добавим сообщение к каждому из наших текущих текстов, отображая одно из подобных сообщений за единицу времени.

БМ. Хорошо.

```
//TestGame.Java-----
public void testOneThrow()
{
    g.add(5);
    assertEquals(5, g.score());
    assertEquals(1, g.getCurrentFrame());
}

//Game.java-----
public int getCurrentFrame()
{
    return 1;
}
```

БМ. Хорошо, мы получили вполне работоспособный код, но он далек от изящества. Давай обратим внимание на следующий тестовый случай.

```
public void testTwoThrowsNoMark()
{
    g.add(5);
    g.add(4);
    assertEquals(9, g.score ());
    assertEquals(1, g.getCurrentFrame());
}
```

БМ. Этот вариант не столь интересен. Лучше испытать еще один вариант.

```

public void testFourThrowsNoMark()
{
    g.add(5);
    g.add(4);
    g.add(7);
    g.add(2);
    assertEquals(18, g.score());
    assertEquals(9, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
    assertEquals(2, g.getCurrentFrame ());
}

```

БМ. Здесь произошел сбой. А теперь выполним прогон теста.

БК. Я полагаю, что используемый в этом случае алгоритм будет тривиальным. Следует просто разделить количество бросков на два, как только количество бросков превысит два из расчета на один фрейм. До сих пор мы рассматривали случай со страйком... хотя на самом деле страйков как таковых не было (поэтому этот вариант может быть спокойно проигнорирован).

БМ применяет операции сложения и вычитания до тех пор, пока это возможно².

```

public int getCurrentFrame()
{
    return 1 + (itsCurrentThrow-1)/2;
}

```

БМ. Полученный результат трудно назвать блестящим.

БК. А что если мы не будем выполнять вычисления постоянно? Если просто изменять значение переменной-члена `currentFrame` после каждого броска?

БМ. Хорошо, давай попробуем.

```

//Game.java-----
public int getCurrentFrame()
{
    return itsCurrentFrame;
}

public void add(int pins)
{
    itsThrows[itsCurrentThrow++]=pins;
    itsScore += pins;
    if (firstThrow == true)
    {
        firstThrow = false;
        itsCurrentFrame++;
    }
}

```

²Дэйв Томас (Dave Thomas) и Энди Хант (Andy Hunt) назвали эту методику “синхронным программированием”.

```
        }
    else
    {
        firstThrow=true;;
    }
}
private int itsCurrentFrame =0;
private boolean firstThrow = true;
```

- БМ. Хорошо, все работает. Но в этом случае следует учитывать то, что в качестве текущего рассматривается фрейм, во время которого был брошен последний шар. Здесь не имеется в виду фрейм, во время которого будет брошен следующий шар. Если учитывать это обстоятельство, все будет в порядке.
- БК. К сожалению, я не обладаю хорошей памятью, поэтому лучше придать коду читабельный характер. Однако перед тем как наводить окончательный лоск, давай исключим код из области действия функции `add()` и поместим его в частную функцию-член `adjustCurrentFrame()` либо в любой другой подобный объект.
- БМ. Хорошо, звучит весьма заманчиво.

```
public void add(int pins)
{
    itsThrows[itsCurrentThrow++]=pins;
    itsScore += pins;
    adjustCurrentFrame();
}

private void adjustCurrentFrame()
{
    if (firstThrow == true)
    {
        firstThrow == false;
        itsCurrentFrame++;
    }
    else
    {
        firstThrow=true;;
    }
}
```

- БМ. Ну а теперь изменим названия переменной и функции, придав им более “читабельный” характер. Что скажешь по поводу названия `itsCurrentFrame`?
- БК. Мне нравятся подобные названия. Я не думаю, что в разработанном нами коде приращение значений выполняется в нужном месте. Текущий фрейм, как мне кажется, определяется номером (во время этого фрейма осуществляются броски). Поэтому приращение значений следует выполнять после того, как был выполнен последний бросок во фрейме.

БМ. Согласен. Изменим тестовые случаи таким образом, чтобы учесть это замечание. В результате будут устранены недоработки функции `adjustCurrentFrame`.

```
//TestGame.Java-----
public void testTwoThrowsNoMark()
{
    g.add(5);
    g.add(4);
    assertEquals(9, g.score());
    assertEquals(2, g.getCurrentFrame());
}

public void testPourThrowsNoMark()
{
    g.add(5);
    g.add(4);
    g.add(7);
    g.add(2);
    assertEquals(18, g.score());
    assertEquals(9, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
    assertEquals(3, g.getCurrentFrame());
}

//Game.j ava-----
private void adjustCurrentFrame()
{
    if (firstThrow == true)
    {
        firstThrow = false;
    }
    else
    {
        firstThrow=true;
        itsCurrentFrame++;
    }
}

private int itsCurrentFrame = 1;
}
```

БМ. Отлично, все работает. А теперь протестируем функцию `getCurrentFrame` на базе двух случаев спайра.

```
public void testSimpleSpare()
{
    g.add(3);
    g.add(7);
    g.add(3);
    assertEquals(13, g.scoreForFrame(1));
```

```

        assertEquals(2, g.getCurrentFrame ());
    }

public void testSimpleFrameAfterSpare()
{
    g.add(3);
    g.add(7);
    g.add(3);
    g.add(2);
    assertEquals(13, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
    assertEquals(3, g.getCurrentFrame());
}

```

БМ. Все работает. Теперь вернемся к исходной задаче. Нам потребуется запустить в работу функцию **score**. Эта функция будет вызывать на выполнение **scoreForFrame(getCurrentFrame() - 1)**.

```

public void testSimpleFrameAfterSpare()
{
    g.add(3);
    g.add(7);
    g.add(3);
    g.add(2);
    assertEquals(13, g.scoreForFrame(1));
    assertEquals(18, g.scoreForFrame(2));
    assertEquals(18, g.score());
    assertEquals(3, g.getCurrentFrame());
}

//Game.Java -----
public int score()
{
    return scoreForFrame(getCurrentFrame()-1);
}

```

БМ. Тестовый случай **TestOneThrow** привел к сбою. Давай попробуем исследовать возможные причины.

```

public void testOneThrow()
{
    g.add(5);
    assertEquals(5, g.score());
    assertEquals(1, g.getCurrentFrame());
}

```

БМ. В случае одного броска первый фрейм не будет завершен. Метод **score** вызывает функцию **scoreForFrame(0)**. Весьма неплохо.

БК. Может быть да, а может быть нет. Поставь себя на место заказчика программы,зывающего функцию **score()**? Вполне логично предположить, что не всегда возникает желание вызывать незавершенный фрейм.

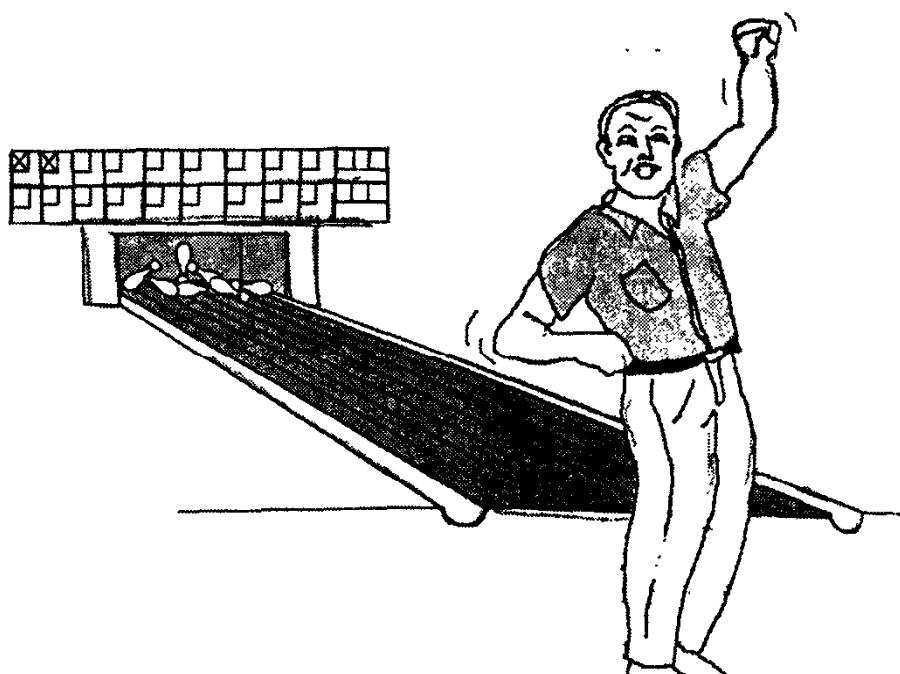
БМ. Да, этот вопрос меня тоже беспокоит. Во избежание этого давай вынесем функцию `score` за пределы тестового случая `testOneThrow`. Возможно ли это в принципе?

БК. Возможно. Мы можем исключить даже весь тестовый случай `testOneThrow`. Для этого сначала нужно ответить на вопрос, нужен ли он нам вообще в данный момент времени? Может быть, лучше ограничиться другими тестовыми случаями.

БМ. Хорошо, я разделяю твою точку зрения.

БМ исключает код тестового случая (редактирует исходный код, запускает на выполнение тест, получая изображение зеленой дорожки).

БМ. Да, теперь все выглядит гораздо лучше. А теперь перейдем к тестовому случаю, обрабатывающему результаты страйка. После всего сделанного следует заключить все объекты `Frame` в состав связанного списка, не так ли? (хихиканье)



```
public void testSimpleStrike()
{
    g.add(10);
    g.add(3);
    g.add(6);
    assertEquals(19, g.scoreForFrame(1));
    assertEquals(28, g.score());
    assertEquals(3, g.getCurrentFrame());
}
```

БМ. Хорошо, этот код проходит этап компиляции, а его выполнение завершается сбоем (как и предвиделось). Предпримем действия, обеспечивающие успешное выполнение прогона теста.

```
//Game.Java-----
public class Game
{
    public void add(int pins)
    {
        itsThrows[itsCurrentThrow++]=pins;
        itsScore += pins;
        adjustCurrentFrame(pins);
    }

    private void adjustCurrentFrame(int pins)
    {
        if (firstThrow == true)
        {
            if(pins == 10) // тестирование страйка
                itsCurrentFrame++;
            else
                firstThrow = false;
        }
        else
        {
            firstThrow=true;
            itsCurrentFrame++;
        }
    }

    public int scoreForFrame(int theFrame)
    {
        int ball = 0;
        int score=0;
        for (int currentFrame = 0;
             currentFrame < theFrame;
             currentFrame++)
        {
            int firstThrow = itsThrows[ball++];
            if (firstThrow == 10)
            {
                score += 10 + itsThrows[ball] + itsThrows[ball+1];
            }
            else
            {
                int secondThrow = itsThrows[ball++];

                int frameScore = firstThrow + secondThrow;
                // спайр сначала требует броска в
                // следующих фреймах
            }
        }
    }
}
```

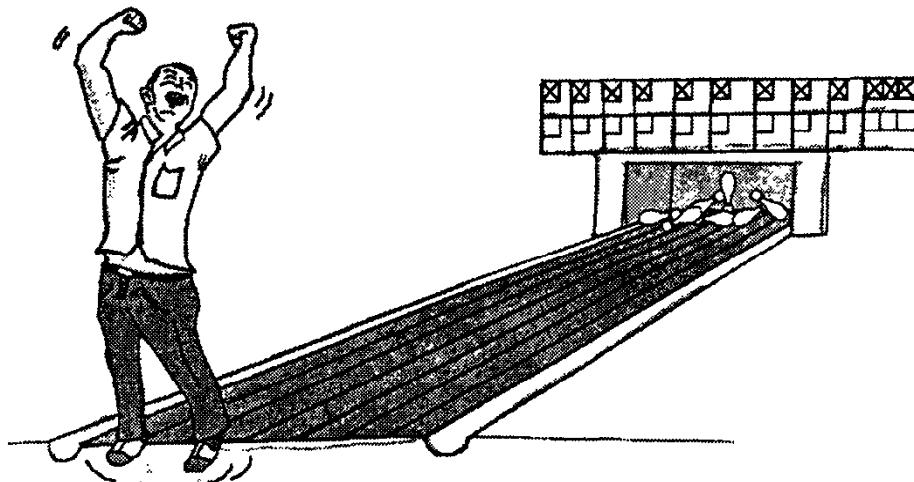
```

        if ( frameScore == 10 )
            score += frameScore + itsThrows[ball];
        else
            score += frameScore;
    }

    return score;
}
private int itsScore = 0;
private int[] itsThrows = new int[21];
private int itsCurrentThrow = 0;
private int itsCurrentFrame = 1;
private boolean firstThrow = true;
}

```

БМ. Пока что все понятно. Давай посмотрим, можно ли подсчитать количество очков в “идеальной игре”.



```

public void testPerfectGame()
{
    for (int i=0; i<12; i++)
    {
        g.add(10);
    }
    assertEquals(300, g.score());
    assertEquals(10, g.getCurFrame());
}

```

БМ. Да, получилось 330 очков. Откуда такой результат?

БК. Это связано с тем, что номер текущего фрейма увеличился до 12.

БМ. Давай ограничим его значением 10.

```
private void adjustCurFrame(int pins)
```

```
if (firstThrow == true)
{
    if( pins == 10 ) // страйк
        itsCurrentFrame++;
    else
        firstThrow = false;
}
else
{
    firstThrow=true;
    itsCurrentFrame++;
}
itsCurrentFrame = Math.min(10, itsCurrentFrame);
}
```

- БМ. Да, теперь получилось 270 очков. В чем причина?
- БК. Боб, функция подсчета количества очков вычитает значения из результата, полученного функцией `getCurrentFrame`. Поэтому вычисляется количество очков для девятого, но не для десятого фрейма.
- БМ. Что? Это означает, что я должен ограничивать текущий фрейм номером 11, а не 10? Я попытаюсь сделать это.
- ```
itsCurrentFrame = Math.min(11, itsCurrentFrame);
```
- БМ. Хорошо, теперь очки подсчитываются верно, но работа программы завершается сбоем. Причина появления проблем заключается в том, что текущий фрейм имеет номер 11, а не 10. Да уж! Всегда в бочке меда найдется ложка дегтя. Мы хотим, чтобы текущим считался тот фрейм, в котором игрок совершает бросок, но что все это будет означать к концу игры?
- БК. Может, следует вернуться к идеи о том, что текущим называется тот фрейм, в котором был брошен последний шар.
- БМ. А может имеет смысл “вооружиться” концепцией о последнем *завершенном* фрейме? В конце концов, сумма очков игры в любой рассмотренный промежуток времени представляет собой сумму очков, заработанных в последнем завершенном фрейме.
- БК. А завершенный фрейм характеризуется тем, что производится фиксация суммы заработанных очков, не так ли?
- БМ. Да, а фрейм со спайром рассматривается в качестве завершенного после броска следующего шара. Фрейм со страйком рассматривается в качестве завершенного после броска следующих двух шаров. Фрейм без особенностей считается завершенным после броска второго шара. Подожди минутку... Мы должны попытаться запустить в работу метод `score()`, верно? Все, что нужно сделать для этого, — заставить метод `score()` вызывать функцию `scoreForFrame(10)` после завершения игры.

- БК. Каковы признаки завершения игры?
- БМ. Функция `adjustCurrentFrame` выполняет приращение значения `itsCurrentFrame` до тех пор, пока не будет достигнут десятый фрейм, после чего игра считается завершенной.
- БК. Минутку. До этого ты говорил, что если `getCurrentFrame` возвращает 11, игра считается завершенной. Каков же алгоритм функционирования кода в настоящее время!
- БМ. Да, придется изменить тестовый случай в целях соответствующей адаптации кода.

```
public void testPerfectGame()
{
 for (int i=0; i<12; i++)
 {
 g.add(10);
 }
 assertEquals (300, g.score());
 assertEquals(11, g.getCurrentFrame());
}
```

БМ. Хорошо, теперь все работает. Думаю, что не будет особых проблем, если `getMonth` возвращает значение 0 для января. Но я по-прежнему чувствую себя неуютно.

БК. Возможно, эта проблема коснется нас в будущем. Ну а сейчас я вижу ошибку. Можно мне попытаться ее исправить?

```
public void testEndOfArray()
{
 for (int i=0; i<9; i++)
 {
 g.add(0);
 g.add(0);
 }
 g.add(2);
 g.add(8); // 10-й фрейм со спайром
 g.add(10); // страйк в последней позиции массива.
 assertEquals (20, g.score());
}
```

- БК. Гм-м. Это не может привести к сбою. Я думаю, что если 21-ая позиция массива включает сведения о страйке, блок подсчета количества очков должен попытаться включить 22-ю и 23-ю позиции. Но я не понимаю, как это можно реализовать на практике.
- БМ. Гмм, ты по-прежнему думаешь о модуле подсчета количества очков, как о чем то абстрактном. В любом случае я вижу, что, поскольку `score` никогда не вызывает `scoreForFrame` с числовым параметром, большим 10,

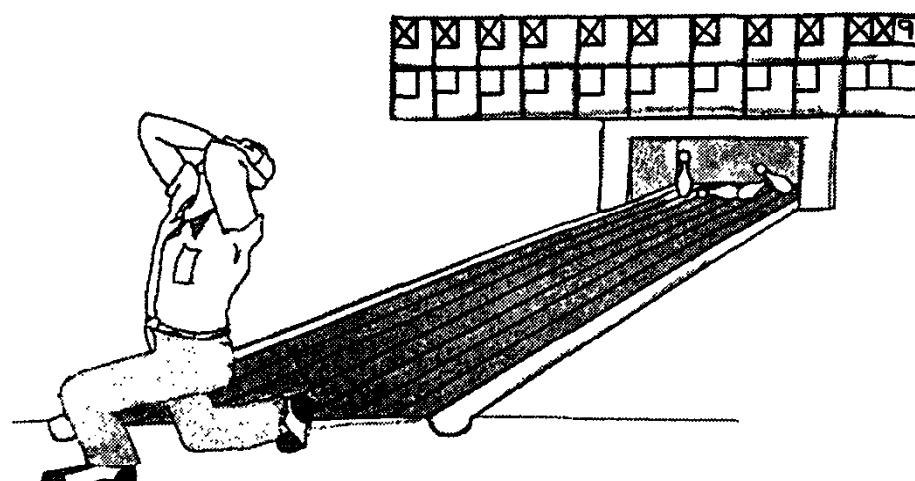
последний страйк вовсе не рассматривается в качестве такого. Он просто засчитывается в качестве 10-го в целях завершения последнего спайра. При этом мы никогда не выходим за пределы массива.

- БК. Хорошо, давай реализуем исходную карточку для подсчета количества очков в программе.

```
public void testSampleGame()
{
 g.add(1);
 g.add(4);
 g.add(4);
 g.add(5);
 g.add(6);
 g.add(4);
 g.add(5);
 g.add(5);
 g.add(10);
 g.add(0);
 g.add(1);
 g.add(7);
 g.add(3);
 g.add(6);
 g.add(4);
 g.add(10);
 g.add(2);
 g.add(8);
 g.add(6);
 assertEquals(133, g.score());
}
```

- БК. Хорошо, все работает. А что ты думаешь по поводу других тестовых случаев?

- БМ. Давай попробуем протестировать дополнительные граничные условия — как ты посмотришь на то, что кое-кому захочется выполнить 11 страйков, а затем — 9?



```
public void testHeartBreak()
{
 for (int i=0; i<11; i++)
 g.add(10);
 g.add(9);
 assertEquals (299, g.score());
}
```

БМ. Все отлично работает, а как быть в случае спайра для десятого фрейма?

```
public void testTenthFrameSpare()
{
 for (int i=0; i<9; i++)
 g.add(10);
 g.add(9);
 g.add(1);
 g.add(1);
 assertEquals (270, g.score());
}
```

БМ. Все отлично работает. О лучшем можно и не мечтать, не так ли?

БК. Да, я думаю, что мы сделали все. Хотя на самом деле потребуется выполнить рефакторинг и “привести все в порядок”. Кроме того, следует обратить внимание на объект `scorer`.

БМ. Действительно, функция `scoreForFrame` выглядит достаточно хаотично. Следует обратить на нее особое внимание.

```
public int scoreForFrame(int theFrame)
{
 int ball = 0;
 int score=0;
 for (int currentFrame = 0;
 currentFrame < theFrame;
 currentFrame++)
 {
 int firstThrow = itsThrows[ball++];
 if (firstThrow == 10)
 {
 score += 10 + itsThrows[ball] + itsThrows[ball+1];
 }
 else
 {
 int secondThrow = itsThrows[ball++];

 int frameScore = firstThrow + secondThrow;
 // в случае спайра первый бросок
 // выполняется в следующих фреймах
 if (frameScore == 10)
 score += frameScore + itsThrows[ball];
 else
```

```
 score += frameScore;
 }
}
return score;
}
```

БМ. На самом деле мне бы хотелось выделить конструкцию `else` в виде функции `handleSecondThrow`, но в данный момент этого сделать нельзя, поскольку используются локальные переменные `ball`, `firstThrow` и `secondThrow`.

БК. Эти локальные переменные можно включить в состав переменных-членов.

БМ. Хорошо, давай реализуем твою идею по поводу того, чтобы вычисление количества очков выполнять внутри выделенного для этого объекта `scorer`. Перейдем к практике.

БК забирает клавиатуру.

```
private void adjustCurrentFrame(int pins)
{
 if (firstThrowInFrame == true)
 {
 if(pins == 10) // страйк
 itsCurrentFrame++;
 else
 firstThrowInFrame = false;
 }
 else
 {
 firstThrowInFrame = true;
 itsCurrentFrame++;
 }
 itsCurrentFrame = Math.min(11, itsCurrentFrame);
}

public int scoreForFrame(int theFrame)
{
 ball = 0;
 int score=0;
 for (int currentFrame = 0;
 currentFrame < theFrame;
 currentFrame++)
 {
 firstThrow = itsThrows[ball++];
 if (firstThrow == 10)
 {
 score += 10 + itsThrows[ball] + itsThrows[ball+1];
 }
 else
 {
 secondThrow = itsThrows[ball++];
 score += firstThrow + secondThrow;
 }
 }
 return score;
}
```

```

 int frameScore = firstThrow + secondThrow;
 // для спайра требуется бросок в следующих
 // фреймах
 if (frameScore == 10)
 score += frameScore + itsThrows[ball];
 else
 score += frameScore;
 }

}

return score;
}

private int ball;
private int firstThrow;
private int secondThrow;

private int itsScore = 0;
private int[] itsThrows = new int[21];
private int itsCurrentThrow = 0;
private int itsCurrentFrame = 1;
private boolean firstThrowInframe = true;

```

- БК. Я не ожидал увидеть здесь конфликт имен. В нашем распоряжении уже имеется экземпляр переменной `firstThrow`. Но все же лучше ее именовать как `firstThrowInframe`. В любом случае это не оказывает влияние на работоспособность. Теперь можно выделить конструкцию `else` в виде отдельной функции.

```

public int scoreForFrame(int theFrame)
{
 ball = 0;
 int score=0;
 for (int currentFrame = 0;
 currentFrame < theFrame;
 currentFrame++)
 {
 firstThrow = itsThrows[ball++];
 if (firstThrow == 10)
 {
 score += 10 + itsThrows[ball] + itsThrows[ball+1];
 }
 else
 {
 score += handleSecondThrow();
 }
 }
 return score;
}

private int handleSecondThrow()

```

```

{
 int score =0;
 secondThrow = itsThrows[ball++];

 int frameScore = firstThrow + secondThrow;
 // спайр требует броска для следующих фреймов
 if (frameScore == 10)
 score += frameScore + itsThrows[ball];
 else
 score += frameScore;
 return score;
}

```

БМ. Обрати внимание на структуру функции `scoreForFrame!` С применением псевдокода она записывается следующим образом.

```

if strike
 score += 10 + nextTwoBalls();
else
 handleSecondThrow.

```

БМ. А что если внести небольшие изменения?

```

if strike
 score += 10 + nextTwoBalls();
else if spare
 score += 10 + nextBall();
else
 score += twoBallsInFrame()

```

БК. Великолепно! Сформированный код практически отвечает правилам подсчетам количества очков в боулинге, не так ли? Хорошо, давай посмотрим, каким образом можно реализовать эту структуру в виде реальной функции. Сначала изменим способ приращения значения переменной `ball` так, чтобы этой переменной могли манипулировать независимым образом три тестовых случая.

```

public int scoreForFrame(int theFrame)
{
 ball = 0;
 int score=0;
 for (int currentFrame = 0;
 currentFrame < theFrame;
 currentFrame++)
 {
 firstThrow = itsThrows[ball];
 if (firstThrow == 10)
 {
 ball++;
 score += 10 + itsThrows[ball] + itsThrows[ball+1];
 }
 else
 score += firstThrow;
 }
}

```

```

 else
 {
 score += handleSecondThrow();
 }
 }

 return score;
}
private int handleSecondThrow()
{
 int score = 0;
 secondThrow = itsThrows[ball+1];

 int frameScore = firstThrow + secondThrow;
 // для спайра нужно выполнить первый бросок в
 // в следующих фреймах
 if (frameScore == 10)
 {
 ball+=2;
 score += frameScore + itsThrows[ball];
 }
 else
 {
 ball+=2;
 score += frameScore;
 }
 return score;
}

```

БМ. Хорошо, теперь настало время избавиться от переменных `firstThrow` и `secondThrow`, заменив их на соответствующие функции.

```

public int scoreForFrame(int theFrame)
{
 ball = 0;
 int score = 0;
 for (int currentFrame = 0;
 currentFrame < theFrame;
 currentFrame++)
 {
 firstThrow = itsThrows[ball];
 if (strike())
 {
 ball++;
 score += 10 + nextTwoBalls();
 }
 else
 {
 score += handleSecondThrow();
 }
 }
}

```

```

 return score;
 }

private boolean strike()
{
 return itsThrows[ball] == 10;
}
private int nextTwoBalls()
{
 return itsThrows[ball] + itsThrows[ball+1];
}

```

БМ. Все в порядке, продолжаем двигаться дальше.

```

private int handleSecondThrow()
{
 int score = 0;
 secondThrow = itsThrows[ball+1];

 int frameScore = firstThrow + secondThrow;
 // спайр вынуждает выполнить первый бросок
 // в следующих фреймах
 if (spare())
 {
 ball+=2;
 score += 10 + nextBall();
 }
 else
 {
 ball+=2;
 score += frameScore;
 }
 return score;
}

private boolean spare()
{
 return (itsThrows[ball] + itsThrows[ball+1]) == 10;
}

private int nextBall()
{
 return itsThrows[ball];
}

```

БМ. Хорошо, этот модуль тоже вполне работоспособен. Теперь поработаем `frameScore`.

```

private int handleSecondThrow()
{
 int score = 0;

```

```

secondThrow = itsThrows[ball+1];

int frameScore = firstThrow + secondThrow;
// для спайра требуется первый бросок в
// следующих фреймах
if (spare ())
{
 ball+=2;
 score += 10 + nextBall();
}
else
{
 score += twoBallsInFrame();
 ball+=2;
}
return score;
}

private int twoBallsInFrame()
{
 return itsThrows[ball] + itsThrows[ball+1];
}

```

- БК. Боб, твой алгоритм приращения количества шаров не выдерживает критики. В случае спайра и страйка ты выполняешь приращение до того, как подсчитываешь количество очков. В случае с `twoBallsInFrame` ты выполняешь приращение *после* вычисления количества очков. Получается, что поведение кода зависит от выбранного порядка! Что же здесь неправильно?
- БМ. Прошу прощения, я должен привести некоторые объяснения. Я планирую перенести вычисление приращений в модули `strike`, `spare` и `twoBallsInFrame`. В результате эти модули будут исключены из функции `scoreForFrame`, которая после этого приобретет сходство с упомянутым псевдокодом.
- БК. Хорошо, я доверю тебе выполнение некоторых дополнительных шагов, но помни о том, что я буду тебя контролировать.
- БМ. Хорошо, теперь, когда функции `firstThrow`, `second/Throw` и `frameScore` не используются, пришло время избавиться от них.

```

public int scoreForFrame(int theFrame)
{
 ball = 0;
 int score=0;
 for (int currentFrame = 0;
 currentFrame < theFrame;
 currentFrame++)
 {
 if (strike())

```

```
{
 ball++;
 score += 10 + nextTwoBalls();
}
else
{
 score += handleSecondThrow();
}
}
return score;
}

private int handleSecondThrow()
{
 int score = 0;
 // спайр требует первый бросок в следующих
 // фреймах
 if (spare())
 {
 ball+=2;
 score += 10 + nextBall();
 }
 else
 {
 score += twoBallsInFrame();
 ball+=2;
 }
 return score;
}
```

БМ. С тех пор, как все три случая объединены с помощью одной переменной ball, а эта переменная обрабатывается независимым образом в каждом случае, можно объединить все три случая.

```
public int scoreForFrame(int theFrame)
{
 ball = 0;
 int score=0;
 for (int currentFrame = 0;
 currentFrame < theFrame;
 currentFrame++)
 {
 if (strike())
 {
 ball++;
 score += 10 + nextTwoBalls();
 }
 else if (spare())
 {
 ball+=2;
 }
 }
}
```

```

 score += 10 + nextBall();
 }
 else
 {
 score += twoBallsInFrame();
 ball+=2;
 }
}
return score;
}

```

- БК. Хорошо, теперь устраним неоднозначность из функции вычисления приращения и переименуем остальные функции в целях улучшения их “читабельности”.

```

public int scoreForFrame(int theFrame)
{
 ball = 0;
 int score=0;
 for (int currentFrame = 0;
 currentFrame < theFrame;
 currentFrame++)
 {
 if (strike())
 {
 score += 10 + nextTwoBallsForStrike();
 ball++;
 }
 else if (spare())
 {
 score += 10 + nextBallForSpare();
 ball+=2;
 }
 else
 {
 score += twoBallsInFrame();
 ball+=2;
 }
 }

 return score;
}

private int nextTwoBallsForStrike()
{
 return itsThrows[ball+1] + itsThrows[ball+2];
}

private int nextBallForSpare()
{

```

```
 return itsThrows[ball+2];
}
```

БМ. Обрати внимание на функцию `scoreForFrame!` Эта функция в краткой форме определяет основные правила боулинга.

БК. Боб, а что же случилось со связанным списком из объектов `Frame`?

БМ. Нас мучают “демоны грамматической избыточности”. О, эти три небольших квадратика, нарисованных на обратной стороне салфетки, `Game`, `Frame` и `Throw`, по-прежнему остаются достаточно сложными и в чем-то даже неправильно сформированными.

БК. Мы допустили ошибку, начав с класса `Throw`. Следовало бы начать с класса `Game`!

БМ. Несомненно! В следующий раз попытаемся начать с наивысшего уровня и попытаться спуститься вниз.

БК. Проектирование “сверху вниз”!??!?!?

БМ. Точнее, проектирование “сверху вниз”, *управляемое тестами*. Откровенно говоря, я не знаю, хорошо это или нет. Просто эта технология может оказаться полезной. Поэтому в следующий раз я попытаюсь применить ее и посмотрю, к чему это приведет.

БК. Хорошо. В любом случае придется выполнить некоторый рефакторинг. Переменная `ball` выступает в качестве частного итератора для функции `scoreForFrame` и ее “фаворитов”. Эти объекты были перемещены в другой объект.

БМ. О, да, что же с нашим объектом `Scorer`? Не пора ли заняться им вплотную?

БК тянется к клавиатуре и выполняет несколько небольших шагов, очерченных тестами...

```
//Game.Java-----
public class Game
{
 public int score()
 {
 return scoreForFrame(getCurrentFrame()-1);
 }

 public int getCurrentFrame()
 {
 return itsCurrentFrame;
 }

 public void add(int pins)
 {
 itsScorer.addThrow(pins);
```

```
 itsScore += pins;
 adjustCurrentFrame(pins);
}

private void adjustCurrentFrame(int pins)
{
 if (firstThrowInFrame == true)
 {
 if(pins == 10) // страйк
 itsCurrentPrame++;
 else
 firstThrowInFrame = false;
 }
 else
 {
 firstThrowInFrame=true;
 itsCurrentFrame++;
 }
 itsCurrentFrame = Math.min(11, itsCurrentFrame);
}

public int scoreForFrame(int theFrame)
{
 return itsScorer.scoreForFrame(theFrame);
}

private int itsScore = 0;
private int itsCurrentFrame = 1;
private boolean firstThrowInFrame = true;
private Scorer itsScorer = new Scorer();
}

//Scorer.java-----
public class Scorer
{
 public void addThrow(int pins)
 {
 itsThrows[itsCurrentThrow++] = pins;
 }

 public int scoreForFrame(int theFrame)
 {
 ball = 0;
 int score=0;
 for (int currentFrame = 0;
 currentFrame < theFrame;
 currentFrame++)
 {
 if (strike())
 .
```

```
 score += 10 + nextTwoBallsForStrike();
 ball++;
 }
 else if (spare())
 {
 score += 10 + nextBallForSpare();
 ball+=2;
 }
 else
 {
 score += twoBallsInFrame();
 ball+=2;
 }
}
return score;
}

private boolean strike()
{
 return itsThrows[ball] == 10;
}

private boolean spare()
{
 return (itsThrows[ball] + itsThrows[ball+1]) == 10;
}

private int nextTwoBallsForStrike()
{
 return itsThrows[ball+1] + itsThrows[ball+2];
}

private int nextBallForSpare()
{
 return itsThrows[ball+2];
}

private int twoBallsInFrame()
{
 return itsThrows[ball] + itsThrows[ball+1];
}

private int ball;
private int[] itsThrows = new int[21];
private int itsCurrentThrow = 0;
}
```

- БК. Теперь намного лучше. Объект Game отслеживает фреймы, а объект Scorer занят подсчетом количества очков. Работает принцип персональной ответственности!

БМ. Как бы там ни было, но этот вариант вполне хороши. Ты уже заметил, что переменная `itsScore` уже не используется?

БК. Ха! Ты прав. Давай “убьем ее”.

```
public void add(int pins)
{
 itsScorer.addThrow(pins);
 adjustCurrentFrame(pins);
}
```

БК. Неплохо. Теперь приступим к “санитарной очистке” функции `adjustCurrentFrame`?

БМ. Хорошо, давай приступим к рассмотрению этого процесса.

```
private void adjustCurrentFrame(int pins)
{
 if (firstThrowInFrame == true)
 {
 if(pins == 10) // страйк
 itsCurrentFrame++;
 else
 firstThrowInFrame = false;
 }
 else
 {
 firstThrowInFrame=true;
 itsCurrentFrame++;
 }
 itsCurrentFrame = Math.min(11, itsCurrentFrame);
}
```

БМ. А теперь выделим модуль вычисления приращения, образовав отдельную функцию, которая также ограничивает номер фрейма 11. (Вообще говоря, мне не нравится число 11.)

БК. Боб, в данном случае 11 означает конец игры.

БМ забирает клавиатуру и выполняет пару изменений в тестах.

```
private void adjustCurrentFrame(int pins)
{
 if (firstThrowInFrame == true)
 {
 if(pins == 10) // страйк
 advanceFrame();
 else
 firstThrowInFrame = false;
 }
 else
 {
 firstThrowInFrame=true;
```

```
 advanceFrame();
}
}

private void advanceFrame()
{
 itsCurrentFrame = Math.min(11, itsCurrentFrame +1);
}
```

БМ. Хорошо, уже немного лучше. Теперь выделим случай, тестирующий страйк, в виде отдельной функции. (Эта операция потребует нескольких небольших шагов и тестов.)

```
private void adjustCurrentFrame(int pins)
{
 if (firstThrowInFrame == true)
 {
 if (adjustFrameForStrike(pins) == false)
 firstThrowInFrame = false;
 }
 else
 {
 firstThrowInFrame=true;
 advanceFrame();
 }
}

private boolean adjustFrameForStrike(int pins)
{
 if (pins == 10)
 {
 advanceFrame();
 return true;
 }
 return false;
}
```

БМ. Сделано достаточно хорошо. Что ты скажешь относительно 11.

БК. Ты ненавидишь это число?

БМ. Да, рассмотрим функцию `score()`.

```
public int score ()
{
 return scoreForFrame(getCurrentFrame()-1);
}
```

БМ. Число -1 нечетное. Именно в этом единственном случае используется функция `getCurrentFrame`, а теперь придется подобрать возвращаемое значение.

- БК. Да, ты прав. Сколько раз мы возвращались обратно?
- БМ. Достаточно часто. Вот один из таких случаев. Код ожидает функцию `itsCurrentFrame` в целях представления фрейма, в котором был брошен последний шар, а не текущий фрейм.
- БК. Неплохо бы упростить имеющиеся тестовые случаи.
- БМ. Мне кажется, что из всех тестовых случаев следует удалить функцию `getCurrentFrame`. На самом деле ее никто не использует.
- БК. (забирая клавиатуру). Хорошо, постараюсь учесть твои пожелания. Придется впрячь “ломовую лошадь” в эту телегу,

```
//Game.Java-----
 public int score()
 {
 return scoreForFrame(itsCurrentFrame);
 }
 private void advanceFrame()
 {
 itsCurrentFrame = Math.min(10, itsCurrentFrame + 1);
 }
```

- БМ. О, я громко плачу... Мы проделали такой громадный объем работы. И все, что мы сделали, — изменили предельное значение с 11 на 10 и удалили -1. Фантастика!

- БК. Дядя Боб, на самом деле все не так страшно, как тебе кажется.
- БМ. Мне не нравятся побочные эффекты, связанные с настройкой функции `adjustFrameForStrike()`. Я хочу от нее избавиться. Что ты думаешь по поводу следующего кода?

```
private void adjustCurrentFrame(int pins)
{
 if ((firstThrowInFrame && pins == 10) ||
 (!firstThrowInFrame))
 advanceFrame();
 else
 firstThrowInFrame = false;
}
```

- БК. Мне нравится эта идея, тем более, что она выдержала проверку тестами, но я недолюблю длинные конструкции `if`. Что ты скажешь по поводу следующего кода?

```
private void adjustCurrentFrame(int pins)
{
 if (strike(pins) || !firstThrowInFrame)
 advanceFrame();
 else
```

```
 firstThrowInFrame = false;
}

private boolean strike(int pins)
{
 return (firstThrowInFrame && pins == 10);
}
```

БМ. Достаточно не плохо. Мы можем даже продвинуться на один шаг вперед.

```
private void adjustCurrentFrame(int pins)
{
 if (lastBallInFrame(pins))
 advanceFrame();
 else
 firstThrowInFrame = false;
}

private boolean lastBallInFrame(int pins)
{
 return strike(pins) || !firstThrowInFrame;
}
```

БК. Великолепно!

БМ. Да, похоже, что мы подходим к завершению работы. Давай еще раз просмотрим всю программу на предмет простоты и “читабельности” ее кода.

```
//Game.Java-----
public class Game
{
 public int score()
 {
 return scoreForFrame(itsCurrentFrame);
 }

 public void add(int pins)
 {
 itsScorer.addThrow(pins);
 adjustCurrentFrame(pins);
 }

 private void adjustCurrentFrame(int pins)
 {
 if (lastBallInFrame(pins))
 advanceFrame();
 else
 firstThrowInFrame = false;
 }

 private boolean lastBallInFrame(int pins)
```

```
 return strike(pins) || !firstThrowInFrame;
 }

private boolean strike(int pins)
{
 return (firstThrowInFrame && pins == 10);
}

private void advanceFrame()
{
 itsCurrentFrame = Math.min(10, itsCurrentFrame + 1);
}

public int scoreForFrame(int theFrame)
{
 return itsScorer.scoreForFrame(theFrame);
}

private int itsCurrentFrame = 0;
private boolean firstThrowInFrame = true;
private Scorer itsScorer = new Scorer();
}

//Scorer.java-----
public class Scorer
{
 public void addThrow(int pins)
 {
 itsThrows[itsCurrentThrow++] = pins;
 }

 public int scoreForFrame(int theFrame)
 {
 ball = 0;
 int score=0;
 for (int currentFrame = 0;
 currentFrame < theFrame;
 currentFrame++)
 {
 if (strike())
 {
 score += 10 + nextTwoBallsForStrike();
 ball++;
 }
 else if (spare())
 {
 score += 10 + nextBallForSpare();
 ball+=2;
 }
 else

```

```
 {
 score += twoBallsInFrame();
 ball+=2;
 }
 }
 return score;
}

private boolean strike()
{
 return itsThrows[ball] == 10;
}

private boolean spare()
{
 return (itsThrows[ball] + itsThrows[ball+1]) == 10;
}

private int nextTwoBallsForStrike()
{
 return itsThrows[ball+1] + itsThrows[ball+2];
}

private int nextBallForSpare()
{
 return itsThrows[ball+2];
}

private int twoBallsInFrame()
{
 return itsThrows[ball] + itsThrows[ball+1];
}
private int ball;
private int[] itsThrows = new int[21];
private int itsCurrentThrow = 0;
}
```

БМ. ОК, все выглядит достаточно хорошо. Мне кажется, что все, что нужно, уже сделано.

БК. Да, все нормально. Давай еще посмотрим на тесты, позволяющие выполнить хорошую оценку.

```
//TestGame.Java-----
import junit.framework.*;

public class TestGame extends TestCase
{
 public TestGame(String name)
 {
 super(name);
```

```
}

private Game g;

public void setUp()
{
 g = new Game();
}

public void testTwoThrowsNoMark()
{
 g.add(5);
 g.add(4);
 assertEquals(9, g.score());
}

public void testFourThrowsNoMark()
{
 g.add(5);
 g.add(4);
 g.add(7);
 g.add(2);
 assertEquals(18, g.score());
 assertEquals(9, g.scoreForFrame(1));
 assertEquals(18, g.scoreForFrame(2));
}

public void testSimpleSpare()
{
 g.add(3);
 g.add(7);
 g.add(3);
 assertEquals(13, g.scoreForFrame(1));
}

public void testSimpleFrameAfterSpare()
{
 g.add(3);
 g.add(7);
 g.add(3);
 g.add(2);
 assertEquals(13, g.scoreForFrame(1));
 assertEquals(18, g.scoreForFrame(2));
 assertEquals(18, g.score());
}

public void testSimpleStrike()
{
 g.add(10);
 g.add(3);
 g.add(6);
```

```
 assertEquals(19, g.scoreForFrame(1));
 assertEquals(28, g.score());
 }

 public void testPerfectGame()
 {
 for (int i=0; i<12; i++)
 {
 g.add(10);
 }
 assertEquals (300, g.score());
 }

 public void testEndOfArray()
 {
 for (int i=0; i<9; i++)
 {
 g.add(0);
 g.add(0);
 }
 g.add(2);
 g.add(8); // спайр 10-го фрейма
 g.add(10); // страйк в последней позиции !
 assertEquals(20, g.score());
 }

 public void testSampleGame()
 {
 g.add(1);
 g.add(4);
 g.add(4);
 g.add(5);
 g.add(6);
 g.add(4);
 g.add(5);
 g.add(5);
 g.add(10);
 g.add(0);
 g.add(1);
 g.add(7);
 g.add(3);
 g.add(6);
 g.add(4);
 g.add(10);
 g.add(2);
 g.add(8);
 g.add(6);
 assertEquals(133, g.score());
```

```
public void testHeartBreak()
{
 for (int i=0; i<11; i++)
 g.add(10);
 g.add(9);
 assertEquals(299, g.score());
}

public void testTenthFrameSpare()
{
 for (int i=0; i<9; i++)
 g.add(10);
 g.add(9);
 g.add(1);
 g.add(1);
 assertEquals(270, g.score());
}
```

БК. Может, есть смысл разработать дополнительные тесты?

БМ. Не думаю, существующий набор вполне достаточен.

БК. Тогда работа завершена.

БМ. И я так думаю. Спасибо за помощь.

БК. Пожалуйста, мне было приятно с тобой работать.

## Резюме

Продолжая работать над этой главой, мы публиковали ее на Web-узле Object Mentor<sup>3</sup>. Многие прочли ее и снабдили собственными комментариями. Некоторые читатели высказались, что в данной работе практически не применялись принципы объектно-ориентированного проектирования. Некоторые из этих отзывов показались мне интересными. Следует ли применять методы объектно-ориентированного проектирования при разработке любого приложения и программы? В данном случае программа является достаточно простой и не требует применения подобных методов. Реальным примером признания принципов объектно-ориентированного программирования является класс Scorer, хотя в этом случае применяется упрощенная схема по сравнению с истинным объектно-ориентированным проектированием.

Другие видят признаки объектно-ориентированного программирования в классе Frame. Один из таких программистовшел так далеко, что создал свою версию программы, включающую класс Frame. В результате получился более сложный и громоздкий код.

<sup>3</sup><http://www.objectmentor.com>.

Некоторые комментаторы заявили, что мы избегали использования UML-диаграмм. На самом деле это связано с тем, что перед началом работы не был разработан завершенный проект. Нельзя же рассматривать в качестве проекта небольшую и симпатичную UML-диаграмму, нарисованную на обратной стороне салфетки (рис. 6.2). Здесь отсутствует последовательность диаграмм. Лично для меня этот аргумент звучит несколько странно. Вряд ли стоит включать в состав схемы на рис. 6.2 последовательность диаграмм и отказываться от классов Throw и Frame. Надеюсь, мы убедили вас в том, что эти классы жизненно необходимы.

Пытаюсь ли я утверждать, что подобные диаграммы бесполезны? Конечно, нет. Хотя некоторая доля истины в этом утверждении имеется. Именно для этой программы диаграммы не являются важными и полезными. Скорее, в данном случае они отвлекают внимание. Если вы воспользуетесь ими, то получите программу, сложность которой значительно “превышает допустимую степень сложности”. Вы можете возразить мне, что диаграммы позволяют облегчить дальнейшее сопровождение программы, хотя я не согласен с этим мнением. Рассматриваемая нами программа несложна по своей структуре, вследствие чего ее сопровождение не вызывает особых проблем. Не существует каких-либо неуправляемых зависимостей, в силу которых программа будет неустойчивой или “закрепощенной”.

Итак, применение диаграмм не всегда целесообразно. Когда же их не следует применять? Наверное в том случае, если они создаются без сопутствующего кода для проверки последнего, *а затем им намереваются следовать*. Нет ничего плохого в том, когда диаграмма рисуется в целях иллюстрации какой-либо идеи. Однако, располагая диаграммой, не рассчитывайте на то, что она представляет лучший вариант проекта, предназначенного для решения конкретной задачи. При наличии необходимого количества тестов можно найти лучший вариант проекта, отвечающий поставленным целям.

### Правила игры в боулинг

Цель игры в боулинг состоит в том, чтобы сбить максимальное количество кеглей (из 10 возможных), пустив шар по узкой дорожке. Как правило, кегли изготавливаются из дерева.

Продолжительность игры — десять фреймов. В начале каждого фрейма расставляются все десять кеглей. Каждому игроку предоставляются две попытки. Если игрок сбивает все кегли за один раз, значит, мы имеем дело со страйком (strike), после которого фрейм завершается.

Если игрок не сбивает все кегли с первого раза, а добивается успеха со второй попытки, подобная ситуация называется спайром (spare).

После второго броска фрейм завершается, даже если не все кегли были сбиты. Количество очков фрейма подсчитывается путем сложения количества кеглей, сбитых двумя шарами во фрейме, и результата предыдущего фрейма.

Если страйк был получен в десятом фрейме, игрок должен метнуть два дополнительных шара для завершения количества очков страйка.

Если в десятом фрейме получился спайр, игрок должен метнуть один дополнительный шар для завершения количества очков спайра.

Таким образом, в десятом фрейме нужно метнуть три шара вместо традиционных двух.

|   |   |    |   |    |   |    |   |    |   |    |   |     |
|---|---|----|---|----|---|----|---|----|---|----|---|-----|
| 1 | 4 | 4  | 5 | 6  | 5 | 5  | 0 | 1  | 7 | 6  | 2 | 6   |
| 5 |   | 14 |   | 29 |   | 49 |   | 60 |   | 61 |   | 133 |

Карточка регистрации количества очков иллюстрирует типичную игру, которая завершилась с посредственным результатом.

В первом фрейме игрок сбил одну кеглю первым шаром и еще четыре кегли — вторым шаром. Количество очков, заработанных в этом фрейме, равно пяти.

Во втором фрейме игрок сбил четыре кегли первым шаром и пять кеглей — вторым шаром. Суммарное количество сбитых кеглей равно девяти, вместе с результатом первого фрейма получилось четырнадцать очков.

В третьем фрейме игрок сбил шесть кеглей первым шаром и все остальные — вторым шаром (спайр). Очки за этот фрейм не насчитываются до тех пор, пока не будет брошен следующий шар.

В четвертом фрейме игрок сбил пять кеглей первым шаром. Благодаря этому можно подсчитать результат спайра третьего фрейма. Количество очков за третий фрейм равно десяти, плюс количество очков за второй фрейм (14), плюс количество кеглей, сбитых первым шаром в четвертом фрейме (5). Итого получилось 29. Последний шар в четвертом фрейме привел к спайру.

Пятый фрейм завершился страйком. В результате количество очков, полученных в четвертом фрейме, равно  $29 + 10 + 10 = 49$ .

Результаты шестого фрейма печальны. Первый шар ушел в канавку и не сбил ни одной кегли. Второй шар сбил только одну кеглю. Результат страйка в пятом фрейме рассчитывается по формуле  $49 + 10 + 0 + 1 = 60$ .

Остальные записи в карточке читатель сможет без труда расшифровать сам.

# ЧАСТЬ II

## Быстрое проектирование

В предыдущих главах уже говорилось о том, что быстрая разработка ПО — это процесс, разбитый на множество мелких шагов. Как же в таком случае формируется весь программный *проект* в целом? Каким образом можно убедиться в том, что создаваемая программа обладает структурой, которая обеспечивает гибкость, способность к сопровождению и повторному применению? Можно ли в рамках применяемой модели выделить этап, на котором осуществляется “уборка мусора” и повторная переработка кода?

В команде, работающей согласно принципам быстрого проектирования, большая картина формируется по мере разработки ПО. На каждом шаге происходит совершенствование системного проекта, благодаря чему его потребительские качества непрерывно улучшаются. На самом деле команда разработчиков “живет сегодняшним днем”. Они не пытаются создавать инфраструктуру, которая нацелена на поддержку свойств, которые будут реализованы в будущем. Внимание разработчиков сосредоточено на *текущей* структуре системы, а также на том, чтобы в максимальной степени совершенствовать ее возможности.

### Признаки плохого проекта

Каким же образом определяется качество разрабатываемого проекта? В первой главе этой части перечисляются признаки, присущие плохому проекту. Здесь же демонстрируется, каким образом происходит накопление этих признаков, а также даются советы, позволяющие избежать появления излишних проблем.

Вот краткий перечень признаков плохого проекта:

- закрепощенность;
- неустойчивость;
- неподвижность;
- вязкость;
- неоправданная сложность;
- неоправданные повторения;
- неопределенность.

Перечисленные признаки напоминают аналогичные признаки для программного кода<sup>4</sup>, но в данном случае затрагивается структура ПО в целом, а не маленький раздел программного кода.

## Принципы быстрой разработки ПО

В следующих главах этой части книги описываются принципы объектно-ориентированного проектирования, позволяющие разработчикам исключить недостатки проекта, сформировав наилучший проект на основе имеющегося набора свойств.

Ниже приводится перечень этих принципов:

- принцип персональной ответственности (SRP, Single Responsibility Principle);
- принцип открытия-закрытия (OCP, Open-Closed Principle);
- принцип подстановки Лискоу (LSP, Liskov Substitution Principle);
- принцип инверсии зависимостей (DIP, Dependency Inversion Principle);
- принцип отделения интерфейса (ISP, Interface Segregation Principle).

Эти принципы воплощают опыт десятилетий программного инжиниринга. И хотя в данном случае идет речь о законах объектно-ориентированного проектирования, на самом деле мы имеем дело с “реинкарнацией” принципов, характерных для программного инжиниринга.

## Принципы и признаки плохого проекта

Признак плохого проекта часто определяется на субъективной основе. Довольно часто проявляющаяся в данном случае проблема является следствием нарушения одного или большего количества принципов. Например, признак закрепощенности часто является следствием недостаточного внимания к принципу открытия-закрытия (OCP).

Устранение признаков плохого проекта производится путем применения описанных принципов командами разработчиков ПО. Не следует увлекаться безоглядным применением этих принципов, поскольку это может привести к неоправданной сложности.

## Литература

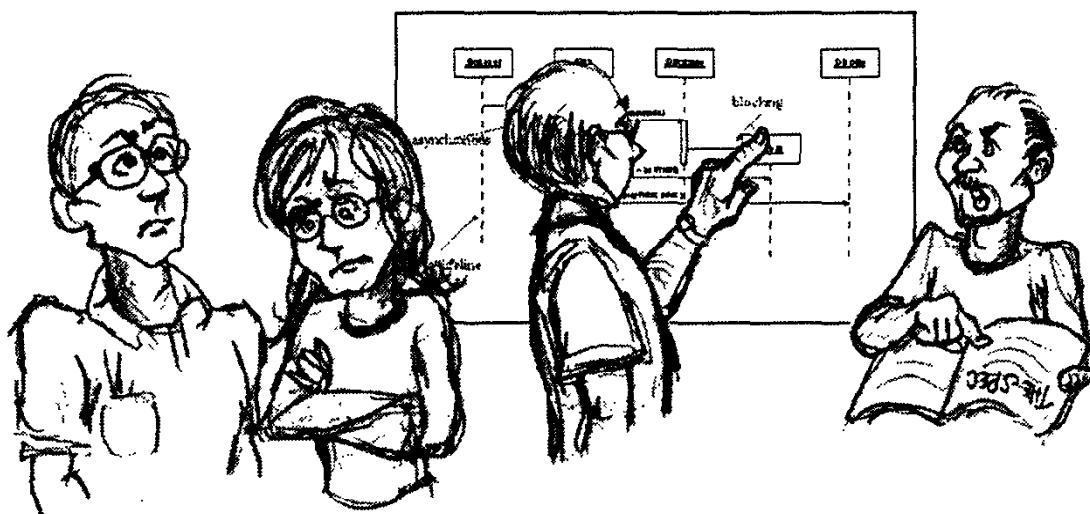
1. Fowler M. Refactoring. Addison-Wesley, 1999.

---

<sup>4</sup>[Fowler99].

# 7

## Быстрое проектирование. Краткое введение



© Jennifer M. Kohnke

Ознакомившись с жизненным циклом разработки ПО, я пришел к выводу, что единственной программной документацией, которая может фактически соответствовать критериям разработки проекта, являются листинги исходного кода.

---

Джек Ривз

В 1992 году Джек Ривз (Jack Reeves) опубликовал содержательную статью в журнале C++ Journal, “What is Software Design” (“Определение программного проекта”)<sup>1</sup>. В этой статье Ривз развивает идею о том, что первоначальное документирование программного проекта реализуется в форме его исходного кода. Диаграммы, представляющие исходный код, являются дополнительной частью проекта. В связи с этим статью Джека можно рассматривать в качестве предвестника эры быстрой разработки программ.

В главе часто упоминается понятие “проект”. Не следует его воспринимать в качестве набора UML-диаграмм, существующих отдельно от кода. Этот набор может представлять отдельные компоненты проекта, а не сам *проект* в целом. Разработка программного проекта является абстрактным понятием. Именно с его помощью конструируется форма и структура программы в целом, а также определяется подробная схема каждого модуля, класса и метода. Проект может быть представлен многими различными способами, однако его окончательное воплощение — это исходный код.

## Процесс разработки ПО

В идеальном случае разработка программы начинается с формирования четкого представления о системе. Системный проект — это некий живой образ, сформированный в вашем воображении. В идеальном случае четкое осознание проекта приводит к появлению первой версии программы.

Дальнейшие события могут принять нежелательный характер. Программа начинает “издавать запах”, т.е. неаккуратно написанная программа приводит к значительному усложнению процесса технического сопровождения.

В конечном итоге явные усилия, требуемые для выполнения даже простейших изменений, становятся настолько обременительными, что разработчики и ведущие менеджеры требуют выполнение повторной разработки программного проекта.

Подобные действия редко приводят к успеху. Несмотря на то, что разработчики проекта руководствуются наилучшими намерениями, они четко сознают, что “стреляют по движущейся мишени”. Старая система непрерывно развивается и изменяется, а новый проект должен развиваться “в русле” этих изменений. В связи с этим в новом проекте накапливаются различные недостатки еще задолго до выпуска первой рабочей версии программы.

<sup>1</sup>[Reeves92]. Настоятельно рекомендуется прочесть эту замечательную статью. Она включена в приложение Г этой книги.

## Признак плохого проекта

Диагноз “загнивания” программы устанавливается в случае обнаружения одного из следующих признаков плохого проекта.

1. **Закрепощенность:** система с трудом поддается изменениям, поскольку любое минимальное изменение вызывает эффект “снежного кома”, затрагивающего другие компоненты системы.
2. **Неустойчивость:** в результате осуществляемых изменений система разрушается в тех местах, которые не имеют прямого отношения к непосредственно изменяемому компоненту.
3. **Неподвижность:** достаточно трудно разделить систему на компоненты, которые могли бы повторно использоваться в других системах.
4. **Вязкость:** сделать что-то правильно намного сложнее, чем выполнить какие-либо некорректные действия.
5. **Неоправданная сложность:** проект включает инфраструктуру, применение которой не влечет непосредственной выгоды.
6. **Неоправданные повторения:** проект содержит повторяющиеся структуры, которые могут унифицироваться с применением простой абстракции.
7. **Неопределенность:** проект трудно читать и понимать. Недостаточно четко выражено содержимое проекта.

### Закрепощенность

Закрепощенность проявляется в том случае, когда программа с трудом поддается изменениям, производимым с помощью простых методов. Проект становится жестким, если одно изменение влечет за собой каскад последующих изменений в зависимых модулях. Чем больше модулей подвержено изменениям, тем более жестким считается проект.

Большинство разработчиков, так или иначе, сталкиваются с этой проблемой. Их просят сделать простые на первый взгляд изменения. Они внимательно изучают характер будущих изменений, а затем выполняют обоснованную оценку требуемого в этом случае объема работы. Позднее, в процессе реализации изменений, разработчики приходят к выводу о непредсказуемых последствиях возможных изменений. В частности, подвергаются переработке огромные блоки кода, причем в процесс модификации вовлекается намного больше модулей, чем говорят результаты первоначальной оценки. Как правило, внедрение изменений занимает намного больше времени, чем планировалось изначально. Если спросить разработчиков о том, почему не оправдались их расчеты, они будут жаловаться на то, что задача оказалась намного сложнее, чем предполагалось изначально.

## Неустойчивость

Неустойчивость при внесении одного изменения программа “разрушается” во многих местах. Очень часто новые проблемы возникают в тех областях, которые не связаны с изменяемым компонентом. В процессе исправления этих ошибок возникают новые ошибки, в результате чего команда разработчиков начинает походить на собаку, гоняющуюся за своим хвостом.

По мере возрастания неустойчивости программного модуля вероятность появления непредвиденных проблем приближается к 100%. Несмотря на всю абсурдность подобного утверждения, подобные модули встречаются довольно часто. Они могут занимать первые места в перечне ошибок, и, кроме того, по мнению разработчиков, должны перерабатываться повторно (но никто не желает иметь с этим дело). В данном случае идет речь о модулях, которые *ухудшаются* по мере их корректирования.

## Неподвижность

Проект считается неподвижным, если он содержит компоненты, которые могут применяться в других системах, однако усилия и степень риска, связанные с изоляцией этих компонентов от первоначальной системы, слишком велики. К сожалению, эта тенденция проявляется весьма часто.

## Вязкость

Вязкость может проявляться в двух формах: по отношению к ПО и к среде.

В случае необходимости внесения изменений разработчики, как правило, применяют различные методы. Некоторые из них способствуют сохранению исходного проекта, а другие — нет (поскольку относятся к разряду хакерских приемов). Если предлагаемые проектом методы сложнее в применении, чем хакерские приемы, то говорят, что вязкость проекта чрезвычайно высока. В этом случае легко допустить ошибку, а нужные и корректные действия выполнить не так уж и просто. В идеале требуется разработка программы, допускающих внесение изменений, сохраняющих структуру проекта.

Симптом вязкости наблюдается в случае, если среда разработки характеризуется словами “медленный” и “неэффективный”. Например, если количество повторных компиляций чрезвычайно высоко, может возникнуть желание изменять программный код таким образом, чтобы избежать подобной ситуации (даже если нарушается структура проекта). Если системе управления исходным кодом требуется несколько часов, чтобы проверить всего несколько файлов, то разработчики захотят сделать изменения, требующие как можно меньше регистраций, независимо от того, сохранен ли проект.

В обоих случаях вязкий проект представляет собой проект, в котором трудно сохраняется разработка программы. Мы хотим создавать такие системы и среды проектов, благодаря которым можно без особых усилий сохранить проект.

## Неоправданная сложность

Проект имеет неоправданную сложность, если содержит элементы, не использующиеся в настоящий момент времени. Это часто происходит в том случае, когда разработчики предвидят изменения в требованиях и проводят мероприятия, направленные на то, чтобы справиться с этими потенциальными изменениями. На первый взгляд кажется, что это неплохо. В конце концов, подготовка к предстоящим изменениям должна сохранить наш код гибким и предотвратить кошмарные изменения, которые могут возникнуть впоследствии.

К сожалению, эффект от таких мероприятий может быть совсем противоположным. При подготовке к наступлению множества дополнительных непредвиденных обстоятельств проект начинает изобиловать никогда не использующимися конструкциями. Некоторые из таких подготовительных мероприятий могут окунуться, а другие — нет. А между тем проект несет на себе груз неиспользуемых элементов проекта. В силу этих причин программа получается сложной и трудно понимаемой.

## Неоправданные повторения

Операции “вырезки” и “вставки” могут быть полезными при редактировании текста, но в то же время они могут быть опасными операциями в случае редактирования кода. Слишком часто программные системы выстраиваются на десятках или сотнях повторяющихся элементов кода. Это происходит примерно следующим образом.

Предположим, что Ральфу необходимо написать код, выполняющий некие функции. Он просматривает другие части кода, где, по его мнению, выполнялись именно эти функции, и обнаруживает подходящий фрагмент. Он вырезает и вставляет этот код в свой модуль и производит необходимые изменения.

Без ведома Ральфа фрагмент кода был позаимствован Тодом из модуля, написанного Лили. Причем Лили было нужно, чтобы ее код выполнял поиск натуральных чисел. Она быстро обнаружила, что код, выполняющий поиск целых чисел, тоже может применяться в этой ситуации. Затем она просто вырезала последний фрагмент кода и включила его в свой модуль, изменив соответствующим образом последний.

Если один и тот же код появляется несколько раз в нескольких формах, разработчики теряют абстрактный образ. Поиск всех повторений, а также их сокращение с применением соответствующих абстракций может и не включаться в первые пункты списка приоритетов, но следует учитывать то, что в противном случае затрачивается немало усилий и времени для того, чтобы система получилась доступной для понимания и управления в дальнейшем.

Слишком громоздкий системный код усложняет работу по изменению системы. Ошибки, обнаруженные в модуле, содержащем повторяющийся код, требуют исправления при каждом повторении. Тем не менее, поскольку каждое повторение

лишь в незначительной степени отличается от всех остальных, то исправления не всегда могут носить одинаковый характер.

## Неопределенность

Неопределенность программного модуля проявляется в сложности его понимания. Код может быть написан либо в четкой и выразительной манере, либо быть неопределенным и трудным для понимания. Код, эволюционирующий с течением времени, становится все более неопределенным. Необходимо постоянно следить за тем, чтобы код оставался “прозрачным” и выразительным, сводя возможную неопределенность к минимуму.

На начальном этапе составления модуля код разработчикам может показаться достаточно “прозрачным”. Это происходит в силу того, что они, погружаясь в проблему с головой, воспринимают код на своем внутреннем уровне. Позднее, когда это состояние проходит, они могут вернуться к модулю и удивиться, что могли составить нечто подобное. Чтобы предотвратить такие случаи, разработчики должны представить себя на месте своих читателей и предпринять согласованные усилия для разложения кода на элементарные операции, чтобы читатели могли его понять. Кроме того, желательно, чтобы кто-то еще просмотрел составленный ими код.

## Чем вызвано “загнивание” программы?

В средах, отличных от сред быстрой разработки программ, может происходить разрушение проектов из-за непредусмотренных изменений требований. Очень часто такие изменения необходимо производить быстро, к тому же их могут выполнять разработчики, незнакомые с философией исходного проекта. Поэтому, хотя и происходят изменения проекта, они все же каким-то образом разрушают оригинальный проект. Постепенно, по мере продолжения внесения изменений, эти нарушения накапливаются, и проект начинает “загнивать”.

Тем не менее, не следует причину ухудшения проекта усматривать в изменении требований. Мы, как разработчики, достаточно хорошо знаем, что требования изменяются. Действительно, большинство из нас понимает, что требования — это самые непостоянные элементы проекта. Если формирование проектов затруднено по причине постоянного изменения требований, то винить в этом следует только наши проекты и нашу деятельность. Необходимо найти какой-нибудь способ разработки проектов, более устойчивых к таким изменениям, и осуществлять меры по предупреждению их “загнивания”.

## “Загнивание” программ исключается в случае применения технологий быстрой разработки ПО

Разработчики программ, использующие технологии быстрого проектирования, преуспевают в случае необходимости выполнения изменений. Группа практически не выполняет авансовых инвестиций, поэтому не происходит вложение средств в изначально устаревшие проекты. Вместо этого сохраняется максимально “прозрачный” и простой проект системы, который поддерживается таковым посредством многих модульных тестов и тестов приемлемости. При этом также обеспечивается гибкость проекта и легкость его изменения в дальнейшем. Преимущества, связанные с подобной гибкостью, применяются группой разработчиков в целях постоянного совершенствования проекта, чтобы к концу каждого периода получалась система, проект которой как можно больше соответствовал бы требованиям этого периода.

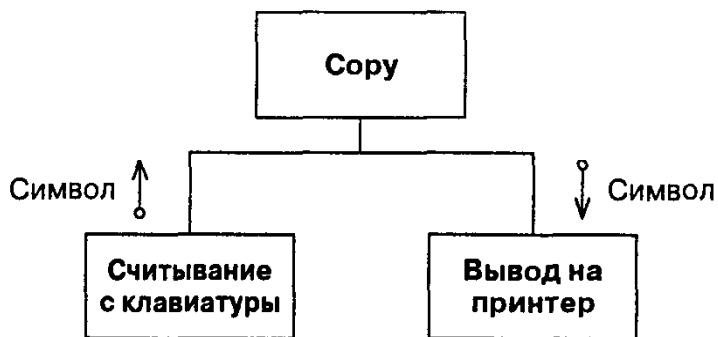
## Пример “загнивающей” программы

Отслеживание процесса “загнивания” проекта может помочь в понимании изложенных выше моментов. Предположим, что ваш начальник приходит в понедельник утром и просит составить программу, копирующую символы клавиатуры на принтер. Проделав в уме необходимые вычислительные операции, вы приходите к выводу, что эта процедура займет менее десяти строк кода. Время, потраченное на проектирование и кодирование, составит не больше одного часа. А если еще учесть многофункциональные групповые совещания, совещания, посвященные изучению качества, ежедневные групповые совещания по состоянию выполнения работ, и три кризисные ситуации, возникшие в настоящий момент времени в региональных офисах... С учетом перечисленных моментов составление программы займет около недели (сюда включены часы сверхурочной работы). Тем не менее, вы всегда можете умножить результаты своих расчетов на три. “Три недели”, — говорите вы своему начальнику. Он одобрительно кивает и уходит, оставляя вас наедине со своим заданием.

## Первоначальный проект

До начала совещания по обзору процесса у вас есть немного времени, поэтому вы решаете спланировать программный проект. Используя структурированный подход к проектированию, получаете структурную диаграмму, представленную на рис. 7.1.

В разрабатываемом приложении используется три модуля (или подпрограммы). Модуль *Copy* вызывает другие два модуля. Программа копирования выбирает символы из модуля *Read Keyboard* и направляет их модулю *Write Printer*.



**Рис. 7.1. Структурная диаграмма программы Copy**

Внимательно изучив свой проект, вы видите, что он неплох. Улыбнувшись, вы выходите из кабинета и идете на совещание. Наконец-то там можно будет немного вздремнуть.

Во вторник вы приходите пораньше, чтобы завершить работу над программой Copy. К несчастью, за истекшую ночь “созрел” очередной кризис. Поэтому вам необходимо пойти в лабораторию и помочь справиться с проблемой. Во время ланча, за который вы наконец-то принялись в 3 часа дня, вам удалось ввести код программы Copy. Результатом этого является листинг 7.1.

---

#### Листинг 7.1. Программа Copy

---

```

void Copy()
{
 int c;
 while ((c=RdKbd()) != EOF)
 WrtPrt(c);
}

```

---

Едва успев сохранить текущую версию кода, вы поняли, что уже опоздали на совещание, посвященное борьбе за качество. Вы понимаете, что это очень важное совещание, поскольку на нем будет рассматриваться амплитуда нулевых дефектов. Поэтому, с жадностью проглотив свой гамбургер и запив его кока-колой, вы стремглав мчитесь на совещание.

В среду вы снова приходите рано, и на этот раз, кажется, ничего непредвиденного не должно произойти. Поэтому вы загружаете исходный код программы Copy и начинаете его компилировать. Процесс компиляции прошел без ошибок. Неплохо, поскольку начальник вызывает вас на внеочередное совещание, на котором обговаривается необходимость экономии тонера для лазерного принтера.

В четверг вы проговорили четыре часа по телефону с техническим консультантом из сервисной службы, который помог вам справиться с командами удаленной отладки и регистрации ошибок в одном из самых “скрытых” системных компонентов. После этого вы тестируете программу Copy. Работает! С первого раза! Неплохо, поскольку вы новый студент-практикант только что удалил

с сервера каталог рабочего программного кода, в результате чего вам необходимо найти последние резервные ленты и восстановить его. Конечно, последнее полное резервное копирование проводилось три месяца назад, и, кроме того, для восстановления каталога рабочего программного кода у вас есть девяносто четыре резервные ленты, сформированные в соответствии с технологией добавочного копирования.

В пятницу вы полностью свободны. Тоже неплохо, поскольку целый день уйдет на успешную загрузку программы Copy в систему управления исходным кодом.

Конечно, программа имеет потрясающий успех, в результате чего ее начинают применять во всех отделах компании. Еще раз подтверждена ваша репутация первоклассного программиста, и вы “почиваете на лаврах”. При удачном стечении обстоятельств вы могли бы ограничиться тридцатью строками кода в этом году.

## Изменение требований

Спустя несколько месяцев ваш начальник говорит о том, что было бы неплохо, если бы программа Copy могла бы производить чтение данных из устройства считывания перфоленты. Стиснув зубы и закатив глаза, вы думаете о том, почему люди всегда меняют свои требования. Ведь ваша программа не предназначена для работы с устройствами считывания перфоленты. Вы предупреждаете начальника о том, что подобного рода изменения могут плохо сказаться на “прозрачности” проекта. Все равно начальник непреклонен, его аргументы сводятся к тому, что пользователям действительно иногда необходимо считывать символы из устройства считывания перфоленты.

Итак, вы вздыхаете и начинаете планировать изменения. Вы хотели бы добавить булев аргумент в функцию Copy. Если его значение истинно, будет производиться считывание с перфоленты; если значение ложно — считывание производится с клавиатуры, как и раньше. К сожалению, поскольку теперь множество других программ используют программу Copy, интерфейс изменить невозможно. Модификация интерфейса займет несколько недель повторного компилирования и тестирования. За это время разработчики системных тестов расправятся с вами, не говоря уже о семи сотрудниках из группы контроля конфигурации. В рамках принятой методики предусматривается просмотр любого фрагмента кода, который назывался Copy!

Нет, изменение интерфейса невозможно. Как же тогда дать понять программе Copy, что она должна производить считывание данных с перфоленты? Конечно же, следует воспользоваться глобальным оператором. Вы можете также использовать наилучшее и самое полезное свойство из набора языков С: оператор ?: . Код из листинга 7.2. демонстрирует результат ваших усилий.

---

**Листинг 7.2. Первая модификация программы Copy**

---

```
bool ptFlag = false;
// напоминание о переустановке флага
void Copy()
{
 int c;
 while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
 WrtPrt(c);
}
```

---

Пользователи,зывающие программу Copy и желающие считывать данные с перфоленты, должны сначала присвоить `ptFlag` значение “true”. Затем они могут вызывать программу Copy, которая с успехом будет производить считывание данных с перфоленты. Как только модуль Copy возвращает результат,зывающий модуль должен восстановить значение `ptFlag`, иначе следующая вызывающая процедура может по ошибке произвести чтение с устройства считывания перфоленты, а не с клавиатуры. Чтобы напомнить программистам о необходимости переустановки флагка, следует добавить соответствующий комментарий.

Вы снова выставляете свою программу на критическое рассмотрение. Ее успех в этот раз еще больше, о чем свидетельствуют толпы программистов, жаждущих поскорее воспользоваться ею. Да, жизнь хороша.

## **Последний шаг, он трудный самый...**

Несколько недель спустя, ваш начальник сообщает вам, что заказчикам иногда необходимо, чтобы программа Copy выводила данные на перфоленту.

Эти заказчики вечно стремятся разрушить ваши проекты! *Программирование проходило бы намного проще, если бы программы не создавались для заказчиков.*

Вы объясняете начальнику, что все эти непрекращающиеся изменения весьма негативно сказываются на внешнем виде (целостности) вашего проекта. Вы предупреждаете его, что в случае, если изменения будут вноситься с такой невероятной скоростью, то программой невозможно будет управлять уже к концу года. Ваш начальник понимающе кивает, но, тем не менее, настаивает на внесении изменений в любом случае.

Это изменение проекта аналогично предыдущему. Все, что от вас требуется, — еще одна глобальная переменная и еще один оператор ?: . В листинге 7.3 представлены результаты ваших трудов.

---

**Листинг 7.3. Новая версия программы**

```
bool ptFlag = false;
bool punchFlag = false;
// напоминание о переустановке флага
void Copy()
{
 int c;
 while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
 punchFlag ? WrtPunch(c) : WrtPrt(c);
}
```

---

Вы особо гордитесь тем, что не забыли изменить комментарий. Все же вы опасаетесь, что структура программы начинает разрушаться. Любые другие изменения, вносимые в устройство ввода, конечно же, вынудят вас полностью реструктуризовать условный цикл `while`. Возможно, как раз самое время встряхнуть пыль со своего резюме и кое-что вспомнить...

## В ожидании изменений

Предоставляю вам право самим судить, насколько все вышесказанное можно считать сатирическим преувеличением. Смысл этой истории в том, чтобы наглядно продемонстрировать, в какой степени может разрушиться проект программы при внесении изменений. Исходный проект программы `Copy` был прост и в то же время превосходен. Однако всего лишь после двух изменений он начал проявлять признаки негибкости, неустойчивости, сложности, избыточности и неопределенности. В случае продолжения проявления этих признаков программа может превратиться в нечто кошмарное.

Винить во всем можно было бы изменения. Можно было бы возразить, что программа была хорошо спроектирована с учетом исходной спецификации, и что последующие изменения этой спецификации спровоцировали разрушение проекта. Тем не менее, в данном случае был проигнорирован один из основных постулатов разработки ПО: *требования всегда изменяются*.

Не следует забывать о том, что самый непостоянный фактор почти во всех проектах разработки ПО – это требования. Требования находятся в состоянии постоянного изменения. Это факт, которым мы, разработчики, не должны пренебрегать! *Мы живем в мире изменяющихся требований, поэтому наша задача состоит в том, чтобы гарантировать выживание наших программ в потоке всех изменений*. Если проект программ разрушается из-за изменяющихся требований, значит, мы недостаточно проворны.

## Пример быстрого проектирования для программы Copy

Быстрая разработка должна начинаться точно таким же образом с кода, представленного в листинге 7.1.<sup>2</sup> Когда начальник попросил разработчиков программ, создаваемых в соответствии с технологией быстрого проектирования, составить программу, производящую чтение с перфоленты, их ответным действием могло бы быть такое изменение, в результате которого проект становится более устойчивым к такому виду изменений. Результатом таких усилий может служить код из листинга 7.4.

---

### Листинг 7.4. Быстрая вторая версия программы Copy

---

```
class Reader
{
public:
 virtual int read() = 0;

class KeyboardReader : public Reader
{
public:
 virtual int read() {return RdKbd();}
};

KeyboardReader GdefaultReader;

void Copy(Reader& reader = GdefaultReader)
{
 int c;
 while ((c=reader.read()) != EOF)
 WrtPrt(c);
}
```

---

Вместо попыток восстановления проекта в целях выполнения новых требований, команда хватается за возможность исправить проект таким образом, чтобы он был устойчив к такого рода изменениям в будущем. С этого момента, когда бы начальник ни попросил создать новое устройство ввода, команда разработчиков может создать проект, не влияющий на качество работы программы Copy.

Команда разработчиков применила принцип “*открытия–закрытия*”, который подробнее рассматривается в главе 9. Благодаря этому принципу создаются модули, которые могут дополняться, не прибегая к модификациям. Это именно то, чего добилась команда разработчиков. Каждое новое устройство ввода, порученное разработчикам, может быть создано без изменения программы Copy.

Тем не менее, следует отметить, что команда даже не пыталась предусмотреть изменения в программе, происходящие при первичном проектировании модуля.

---

<sup>2</sup>Фактически практика процесса разработки, управляемого с помощью результатов тестирования, приведет к созданию в достаточной степени гибкого проекта. Тем не менее, в данном примере мы не будем на этом останавливаться.

Вместо этого они разработали простейший модуль. Когда в конечном итоге изменялись требования, только тогда разработчики модифицировали проект модуля таким образом, чтобы он был устойчив к такого рода изменениям.

Можно было бы поспорить, что это была только половина работы программистов. В то время, как команда защищает себя от устройств ввода, возможна защита и от устройств вывода. Тем не менее, команда на самом деле и не представляет, что устройства вывода могут когда-нибудь измениться. Добавление дополнительной защиты сейчас отнюдь не будет относится работой, не выполняющей ни к одной из ближайших целей. Понятно, что если понадобится такая защита, ее можно будет ввести позже. Поэтому нет смысла добавлять ее сейчас.

## Как разработчики программ, создаваемых в соответствии с технологией быстрого проектирования, узнали о том, что нужно делать?

Разработчики программ, создаваемых в соответствии с технологией быстрого проектирования, из вышеизложенного примера создали абстрактный класс, чтобы защитить себя от изменений в устройстве ввода. Как они узнали, что именно это нужно сделать? Это связано с одним из основополагающих принципов объектно-ориентированного программирования.

Исходный проект программы Copy является негибким из-за *направленности* его зависимостей. Еще раз взгляните на рис. 7.1. Обратите внимание, что модуль Copy непосредственно зависит от модулей KeyboardReader и PrinterWriter. В этом приложении модуль Copy является модулем верхнего уровня. Он определяет политику приложения. Он знает, как следует копировать символы. К сожалению, его сделали зависимым и от деталей нижнего уровня клавиатуры и принтера. Поэтому изменение деталей нижнего уровня влияет на политику верхнего уровня.

Выявив признаки негибкости, разработчики программ, создаваемых в соответствии с технологией быстрого проектирования, поняли, что зависимость модуля Copy от устройства ввода должна быть инвертирована<sup>3</sup>, после чего этот модуль не будет связан с устройством ввода. Затем они использовали шаблон **STRATEGY**<sup>4</sup> для создания желаемой инверсии.

Итак, если говорить вкратце, разработчики программ, создаваемых в соответствии с технологией быстрого проектирования, знают, что делать, потому что:

- они обнаружили проблему, следуя практике быстрого проектирования;
- они диагностировали проблему, применив принципы быстрого проектирования;

<sup>3</sup> Подробнее о принципе инвертирования зависимости см. в гл. 11.

<sup>4</sup> Подробнее о шаблоне **STRATEGY** см. в гл. 14.

- они разрешили проблему, применив соответствующий шаблон проекта;
- взаимосвязь между этими тремя аспектами разработки ПО осуществляется в рамках проектирования.

## **Создание максимально “прозрачного” проекта**

Разработчики программ, создаваемых в соответствии с технологией быстрого проектирования, заинтересованы в создании подходящего и в максимальной степени “прозрачного” проекта. Здесь не идет речь о случайных действиях. Эти разработчики не “вычищают” проект каждые несколько недель. Вместо этого они стараются создавать максимально простые, выразительные и “прозрачные” программы”, которые могли бы оставаться такими каждый день, час или минуту. Они никогда не говорят: “Мы потом вернемся к этому и все уладим”. Они никогда не допустят, чтобы программа начала “гнить”.

Отношение разработчиков программ, создаваемых в соответствии с технологией быстрого проектирования, к проекту программ можно сравнить с отношением хирургов к стерильности, без которой риск инфицирования был бы слишком высок. Такое же отношение к своим проектам выражают и разработчики программ, создаваемых в соответствии с технологией быстрого проектирования.

Проект должен оставаться “прозрачным”, и поскольку исходный код является самым важным выражением проекта, он также должен оставаться “прозрачным”. Профессионализм не позволяет нам, разработчикам ПО, допускать “загнивания” кода.

## **Резюме**

Итак, что же такое быстрое проектирование? Быстрое проектирование – это процесс, а не событие. Это непрерывное применение принципов, шаблонов и практик, направленных на совершенствование структуры и надежности программных продуктов. Его предназначение состоит в том, чтобы всегда сохранять проект системы как можно более простым, выразительным и “прозрачным”.

Из следующих глав читатель узнает о принципах и шаблонах проектирования ПО. Изучая эти главы, помните, что разработчики программ, создаваемых в соответствии с технологией быстрого проектирования, не применяют подобные принципы и шаблоны в рамках больших проектов. Они применяются на каждой итерации в целях сохранения “прозрачности” кода и реализуемого проекта.

## Литература

1. J. Reeves What Is Software Design? *C++ Journal*. Vol. 2. No. 2, 1992. Эта статья находится на Web-узле [http://www.bleading-edge.com/  
Publications/C++Journal/Cpjourn2.htm](http://www.bleading-edge.com/Publications/C++Journal/Cpjourn2.htm).

# 8

## Принцип персональной ответственности



© Jennifer M. Kohnke

Никто, кроме самого Будды, не несет ответственность за передачу нам мистических тайн и секретов. . .

---

Э. Кобхэм Бруэр, словарь выражений и мифов, 1898

Этот принцип был описан в работе Тома Де-Марко (Tom DeMarco)<sup>1</sup> и Мейлира Пейдж-Джонса (Meilir Page-Jones)<sup>2</sup>. Эту закономерность они назвали *связностью*. Причем связность определяется в качестве функционального соотношения, установленного между элементами модуля. В дальнейшем значение этого термина частично изменяется таким образом, что связность начинает определять изменение модуля или класса.

## Принцип персональной ответственности

*Существует лишь одна причина, приводящая к изменению класса.*

Обратимся к примеру игры в боулинг, рассматриваемому в главе 6. В процессе создания кода этой игры классу `Game` были назначены две различные ответственности, а именно — отслеживание текущего фрейма и подсчет количества очков. В конце концов, БМ и БК передали эти две обязанности различным классам. На класс `Game` была возложена функция по отслеживанию фреймов, ну а классу `Scorer` были переданы функции по подсчету количества очков.

В чем же смысл передачи двух ответственостей различным классам? Скорее всего, это связано с тем, что каждая ответственность представляет собой своего рода “ось изменения”. В случае изменения требований производится изменение ответственностей, распределенных между классами. Если на класс возложено несколько ответственостей, возникает ряд причин для их изменения.

Если все же классу соответствует несколько ответственостей, производится их спаривание. В случае изменения одной ответственности затрудняется или становится невозможным соответствие класса другим, возложенным на него ответственостям. В результате формируется “хрупкий” проект, который может разрушиться непредвиденным образом после выполнения каких-либо изменений.

Например, обратите внимание на проект, изображенный на рис 8.1. Очевидно, что класс `Rectangle` включает два метода. Один из этих методов рисует на экране прямоугольник, а второй служит для вычисления площади нарисованной фигуры.

Как видите, два различных приложения используют класс `Rectangle`. Одно из этих приложений реализует вычислительную геометрию. При этом все математические вычисления, производимые над геометрическими фигурами, выполняются с помощью класса `Rectangle`. Причем это приложение “не способно” к рисованию прямоугольников на экране. Другое приложение по своей природе является графическим. Оно может также реализовывать некоторую вычислительную геометрию, но главное его предназначение заключается в рисовании прямоугольников на экране.

<sup>1</sup>[DeMarco79]. с. 310.

<sup>2</sup>[Page-Jones88]. Глава 6. с. 82.

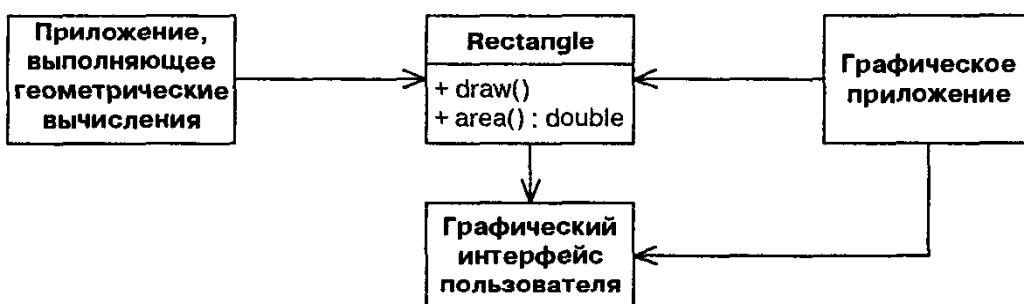


Рис. 8.1. Набор ответственостей

В описанном проекте нарушается принцип персональной ответственности (SRP). На класс `Rectangle` возлагаются две ответственности. Смысл первой ответственности заключается в реализации геометрии прямоугольника. Суть второй ответственности заключается в представлении прямоугольника средствами графического интерфейса пользователя.

Нетрудно заметить, что в данном случае нарушается принцип персональной ответственности (SRP). В результате появляется ряд проблем. Во-первых, придется включать класс GUI в приложение, реализующее геометрические вычисления. Если вы программируете на C++, с приложением потребуется связать класс GUI. Это приводит к дополнительным затратам времени на этапе связывания, компиляции, а также к потреблению дополнительной памяти. В случае формирования Java-приложений на целевой платформе развертываются файлы типа `.class`, применяемые для формирования класса GUI.

Во-вторых, в случае, если изменение `GraphicalApplication` приводит к модификации `Rectangle`, потребуется повторно сформировать, протестировать и развернуть `ComputationalGeometryApplication`. Если эти этапы пропущены, это может привести к непредсказуемому сбою в работе приложения.

В наилучших проектах реализуются две ответственности различными классами, как показано на рис. 8.2. В этом проекте вычислительные операции модуля `Rectangle` выполняются классом `GeometricRectangle`. Теперь изменения, влияющие на способ отображения прямоугольника, не затрагивают `ComputationalGeometryApplication`.

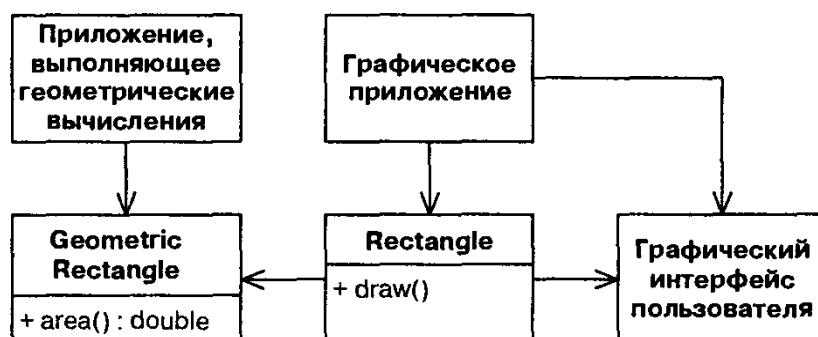


Рис. 8.2. Распределенные ответственности

## Определение ответственности

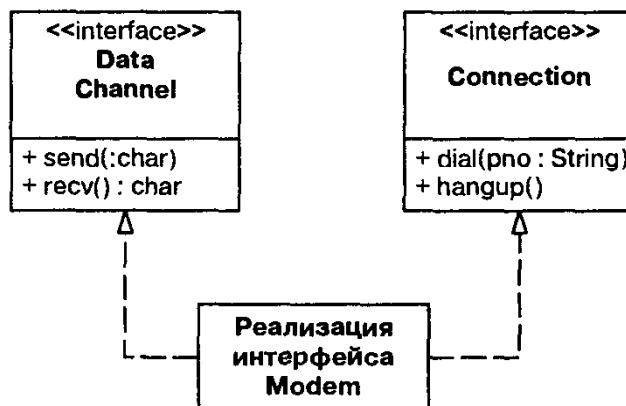
В контексте SRP-принципа ответственность определяется в качестве “причины для изменения”. Если существует несколько мотивов для изменения класса, ему соответствует более одной ответственности. Порой этот факт не так уж просто распознать. Чаще имеют в виду ответственность, распределяемую в группах. Обратите внимание на интерфейс Modem, код которого приводится в листинге 8.1. Большинство из нас согласятся с мыслью о том, что это код выглядит совершенно естественно. Четыре объявляемых здесь функции предназначены для практической реализации различных свойств, присущих модему.

**Листинг 8.1. Modem.java --- нарушение принципа SRP**

```
interface Modem
{
 public void dial(String pno);
 public void hangup();
 public void send(char c);
 public char recv();
}
```

Здесь представлены две ответственности. Суть первой ответственности заключается в управлении соединением. Вторая ответственность заключается в осуществлении передачи данных. Функции `dial` и `hangup` управляют соединением, устанавливаемым модемом, а функции `send` и `recv` обеспечивают передачу данных.

Следует ли разделять две упомянутые ответственности? Все зависит от того, каким образом изменяется приложение. Если применяемые при этом методы влияют на сигнатуру функций, устанавливающих соединение, проект станет излишне “закрепощенным”. Это связано с тем, что классы `send` и `recv` придется компилировать и повторно использовать гораздо чаще, чем следовало бы. В этом случае следует разделять две ответственности, как показано на рис. 8.3. Благодаря этому предотвращается спаривание двух ответственостей клиентскими приложениями.



**Рис. 8.3. Отделенный интерфейс модема**

С другой стороны, если при изменении приложения не применяются способы, приводящие к изменению двух ответственостей в различное время, разделять их вовсе не требуется. В противном случае мы имели бы дело с избыточной сложностью.

Из всего сказанного напрашивается вполне логичный вывод. *Ось изменения является таковой лишь в том случае, если изменения действительно происходят.* Следует применять принцип SRP только в том случае, когда это оправдано.

## Разделение спаренных ответственостей

Обратите внимание, что на рис. 8.3 рассматриваются две ответственности, спаренные в классе `ModemImplementation`. Это не всегда желательно, но иногда может быть необходимо. Весьма часто возникают причины, связанные с аппаратным обеспечением или операционной системой, принуждающие к выполнению спаривания. В этом случае благодаря разделению интерфейсов можно разделять концепции таким образом, чтобы не затрагивать остальные компоненты приложения.

Класс `ModemImplementation` можно рассматривать как некое искусственное образование. Обратите внимание на то, что все зависимости протекают *вне* этого класса. Зависимость от этого класса не является обязательной. В этом случае никакой другой модуль, исключая `main`, не нуждается в знании относительно существования этого класса. В результате возможные неприятности ограждаются своеобразным “барьером”. Благодаря этому не “загрязняется” остальной код приложения.

## Устойчивость

На рис. 8.4 демонстрируется общий случай нарушения принципа SRP. Класс `Employee` включает деловые правила, а также обеспечивает контроль устойчивости. Эти две ответственности практически никогда не смешиваются. Деловые правила могут достаточно часто изменяться, и хотя то же самое нельзя сказать относительно устойчивости, изменение последней происходит в силу совершенно иных причин. Связывание деловых правил с подсистемой обеспечения устойчивости может привести к появлению определенных проблем.

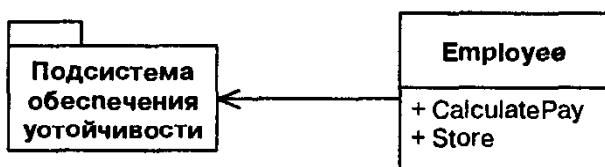


Рис. 8.4. Связывание устойчивости

К счастью, практика разработки, руководимой тестами (глава 4), обычно приводит к разделению этих двух ответственостей еще задолго до того, как начнет “загнивать” проект. Однако в случаях, когда тесты не привели к выполнению разделения, а симптомы закрепощенности и неустойчивости становятся сильными, к проекту применяется рефакторинг. При этом используются шаблоны *Facade* или *Proxy*, позволяющие разделять ответственности.

## Резюме

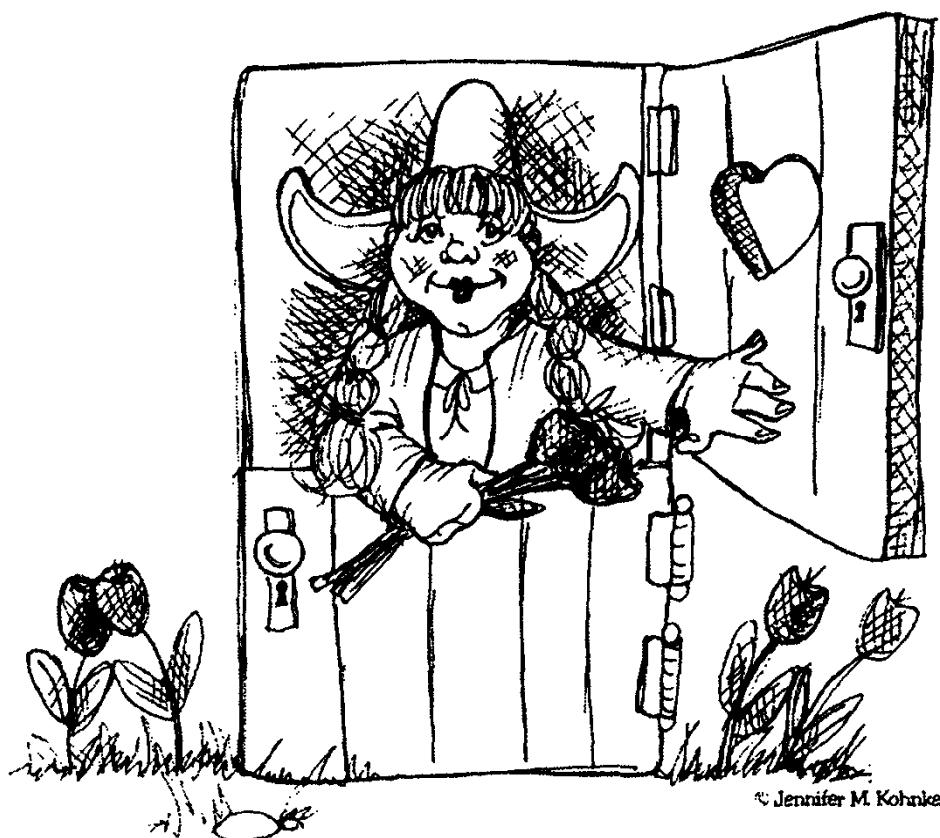
Принцип SRP является одним простейших, но в то же время одним из труднейших в плане применения. Объединение ответственостей является общепринятой практикой. Поиск и отделение этих ответственостей во многом определяют функции программного проекта. На самом деле остальные принципы, которые будут рассматриваться в дальнейшем, часто возвращаются связанны с принципом SRP.

## Литература

1. T. DeMarco *Structured Analysis and System Specification*. Yourdon Press Computing Series. Englewood Cliff, NJ, 1979.
2. Page-Jones, Meilir. *The Practical Guide to Structured Systems Design*. 2d ed. Englewood Cliff. NJ: Yourdon Press Computing Series, 1988.

# 9

## Принцип открытия-закрытия



© Jennifer M. Kohnke

**Голландская дверь (Dutch Door)** — дверь, разделенная горизонтально на две части таким образом, что любая из частей может быть либо открытой, либо закрытой.

---

Словарь американского наследия английского языка,  
четвертое издание, 2000

Как отметил Ивар Джекобсон (Ivar Jacobson), “все системы изменяются на протяжении своих жизненных циклов. Этот фактор необходимо учитывать, если существование разрабатываемых систем рассчитано на период, более продолжи-

тельный, чем существование первой версии”<sup>1</sup>. Каким же образом можно разработать проекты, устойчивые к изменениям, срок жизни которых превышает срок существования первой версии? В 1988 году Берtrand Мейер (Bertrand Meyer) ответил на этот вопрос, предложив принцип открытия-закрытия (OCP)<sup>2</sup>.

## Принцип открытия-закрытия

*Программные объекты (классы, модули, функции и т.д.) должны быть открыты для расширения, но в то же время закрыты для модификации.*

Когда одно изменение в программе вызывает каскад изменений в зависимых модулях, проект начинает проявлять признаки закрепощенности. Используя принцип открытия-закрытия, необходимо выполнить рефакторинг системы с тем, чтобы дальнейшие изменения такого рода не вызывали еще больших изменений. При правильном применении этого принципа дальнейшие изменения такого рода реализуются путем добавления нового кода, а не изменением старого, ныне применяемого кода.

## Описание

Модули, соответствующие принципу открытия-закрытия, имеют два основных атрибута (признака).

1. “Открыто для расширения”. Это означает, что поведение модуля может быть расширено. По мере изменения требований приложения можно расширить модуль за счет включения новых типов поведения, соответствующих этим изменениям. Другими словами, можно изменить то, что делает модуль.
2. “Закрыто для модификации”. В результате расширения поведения модуля изменения в исходном или двоичном коде модуля не производятся. Двоичная исполняемая версия модуля, будь то в связанной библиотеке DLL, или Java .jar, остается неизменной.

На первый взгляд может показаться, что два описанных атрибута находятся в неравном положении по отношению друг к другу. Обычно расширение поведения модуля имеет место в результате изменения исходного кода этого модуля. Модуль, который невозможно изменить, обычно считается модулем с фиксированным поведением.

Возможно ли изменение поведения модуля без изменения его исходного кода? Как можно изменить то, что делает модуль, не изменив сам модуль?

<sup>1</sup>[Jacobson92], с. 21.

<sup>2</sup>[Meyer97], с. 57.

## Ключ к решению проблемы — абстракция

В языках C++, Java или же любых других объектно-ориентированных языках программирования возможно создание абстракций, которые фиксированы и представляют возможные независимые типы поведения. Абстракции — это абстрактные основные классы, поэтому возможные независимые типы поведения представлены всеми возможными производными классами.

Модуль может манипулировать абстракцией. Такой модуль может быть закрыт для модификации, поскольку зависит от абстракции, которая является фиксированной. Но все же поведение этого модуля может быть расширено путем создания новых производных абстракций.

На рис. 9.1 представлен простой проект, не соответствующий принципу открытия-закрытия. Классы *Client* и *Server* определены. Класс *Client* использует класс *Server*. Если бы объекту *Client* потребовалось использовать другой объект сервера, то класс клиента нужно было бы изменить, чтобы называть новый класс сервера.



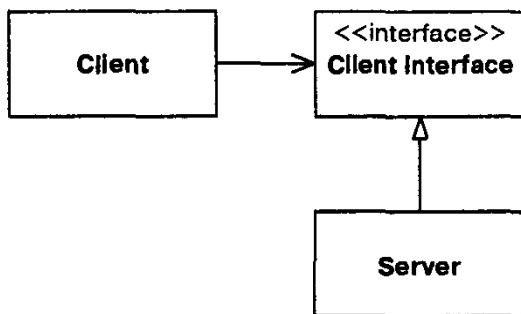
**Рис. 9.1.** Класс Client не открыт и не закрыт

На рис. 9.2 представлен подходящий проект, соответствующий принципу открытия-закрытия. В этом случае класс *ClientInterface* — это абстрактный класс с абстрактными функциями-членами. Клиентский класс использует эту абстракцию; тем не менее, объекты класса *Client* будут использовать объекты производного класса *Server*. Если необходимо, чтобы объекты *Client* использовали другой серверный класс, можно создать новый производный класс от класса *ClientInterface*. Класс *Client* может остаться неизмененным.

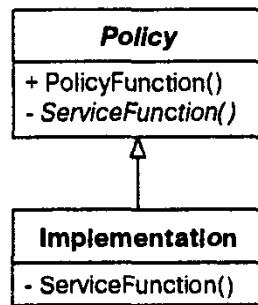
Клиенту необходимо выполнить определенную работу, и он может описать эту работу в рамках абстрактного интерфейса, представленного объектом *ClientInterface*. Подтипы *ClientInterface* могут использовать этот интерфейс любым выбранным им способом. Таким образом, поведение, определенное в клиенте, может быть расширено и модифицировано путем создания новых подтипов клиентского интерфейса.

У читателя может возникнуть вопрос, почему я назвал объект *ClientInterface* именно так. Почему я не назвал его *AbstractServer*? Как позже выяснится, причина состоит в том, что *абстрактные классы более тесно связаны со своими клиентами, чем с классами, которые их реализуют*.

На рис. 9.3 представлена альтернативная структура. Класс *Policy* включает набор базовых общедоступных функций, реализующих политику некоторого



**Рис. 9.2.** Шаблон Strategy: клиент открыт и закрыт одновременно



**Рис. 9.3.** Шаблон Template Method: основной класс открыт и закрыт

рода и аналогичных функциям класса *Client*, представленным на рис. 9.2. Как и прежде, эти функции политик описывают некоторую работу, которую необходимо сделать в рамках некоторых абстрактных интерфейсов. Тем не менее, в этом случае абстрактные интерфейсы являются частью самого класса *Policy*. В языке C++ это были бы чисто виртуальные функции, а в языке Java — абстрактные методы. Эти функции реализованы в подтипах политики. Таким образом, поведения, заданные в классе *Policy*, могут быть расширены или модифицированы путем создания новых производных класса *Policy*.

Эти два шаблона представляют самые обычные способы, соответствующие принципу открытия-закрытия. Они четко изолируют общие функциональные функции от детального воплощения этих функций.

## Приложение Shape

Следующий пример рассматривается во многих книгах, посвященных объектно-ориентированному проектированию. Этот пример широко известен и называется *Shape* (форма). Обычно он используется для демонстрации принципа действия полиморфизма. Однако сейчас мы используем его для разъяснения принципа открытия-закрытия.

Рассмотрим приложение, с помощью которого можно нарисовать окружности и квадраты, воспользовавшись стандартным графическим интерфейсом пользователя. Окружности и квадраты необходимо рисовать в определенном порядке. Список окружностей и квадратов создается соответствующим образом, а программа, рисуя каждую окружность или квадрат, должна придерживаться порядка, установленного этим списком.

## Нарушение принципа открытия-закрытия

В языке С при использовании процедурных методик, не соответствующих принципу открытия-закрытия, возникшая проблема решается способом, представ-

ленным в листинге 9.1. Здесь дается набор структур данных с одинаковым первым элементом, но все другие элементы отличаются друг от друга. Первый элемент каждого набора — это код типа, определяющий структуру данных как окружность либо как квадрат. Функция `DrawAllShapes` выполняет обход массива указателей для этих структур данных, проверяя тип кода, а также выполняя вызов соответствующей функции.

---

**Листинг 9.1. Процедурное решение проблемы ‘‘окружность/квадрат’’**

---

```
--shape.h-----
enum ShapeType {circle, square};

struct Shape
{
 ShapeType itsType;
};

--circle.h-----
struct Circle
{
 ShapeType itsType;
 double itsRadius;
 Point itsCenter;
};

void DrawCircle(struct Circle*);

--square.h-----
struct Square
{
 ShapeType itsType;
 double itsSide;
 Point itsTopLeft;
};

void DrawSquare(struct Square*);

--drawAllShapes.cc-----
typedef struct Shape *ShapePointer;
void DrawAllShapes(ShapePointer list[], int n)
{
 int i;
 for (i=0; i<n; i++)
 {
 struct Shape* s = list[i];
 switch (s->itsType)
 {
 case square:
 DrawSquare((struct Square*)s);
 break;
 }
 }
}
```

```
 case circle:
 DrawCircle((struct Circle*)s);
 break;
 }
}
```

Функция `DrawAllShapes` не соответствует принципу открытия-закрытия, поскольку не может быть закрыта для новых типов форм. Если бы потребовалось расширить эту функцию, чтобы она могла нарисовать список форм, включающих треугольники, то эту функцию пришлось бы изменить. По сути, понадобилось бы модифицировать функцию для любого другого типа формы, которую хотелось бы нарисовать.

Конечно, эта программа представляет собой довольно простой пример. В действительности же конструкция `switch` в функции `DrawAllShapes` повторяется несколько раз в различных функциях в ходе выполнения всего приложения. Причем каждая из них реализует нечто свое, отличное от других функций. Тут могут быть функции, выполняющие перетаскивание, растягивание, перемещение, удаление форм и т.д. Добавление новой формы подобного типа к приложению означает поиск тех мест, где находятся конструкции `switch` (или цепи `if/else`). Новые формы будут добавляться в каждой подобной точке.

Более того, мала вероятность того, что все конструкции `switch` и цепи `if/else` будут иметь такую же хорошо продуманную структуру, как и конструкции в функции `DrawAllShapes`. Наиболее вероятно, что предикаты утверждений `if` будут объединены с логическими операторами, или что выражения, осуществляющие выбор в конструкциях `switch`, будут объединены для облегчения принятия локального решения. В некоторых ситуациях могут применяться функции, выполняющие аналогичные операции с объектами `Squares` и `Circles`. Такие функции могут и не включать конструкции `switch/case` или цепи `if/else`. Таким образом, проблему нахождения и определения всех мест добавления новых форм можно считать несущественной.

Обратите также внимание на тип вносимых изменений. Необходимо добавить новый элемент в перечень `ShapeType`. Поскольку все различные формы зависят от объявления этого перечня, необходимо все эти формы перекомпилировать<sup>3</sup>. Кроме того, следует перекомпилировать все модули, которые зависят от `Shape`.

Поэтому следует не только изменить исходный код всех конструкций `switch/case` в цепи `if/else`, но необходимо также изменить двоичные файлы (посредством перекомпиляции) всех модулей, использующих любую из структур данных `Shape`. Изменение двоичных файлов означает, что следует повторно развернуть

<sup>3</sup>Изменения в `enums` может вызвать модификацию размера переменной, используемой для поддержания типа `enum`. Поэтому принимать решения о необходимости перекомпилирования других объявлений формы следует с большой осторожностью.

любые DLL, общие библиотеки либо другие подобные типы двоичных компонентов. Простое действие по добавлению новой формы в приложение вызывает каскад последующих изменений во многих исходных модулях и даже в еще большем количестве бинарных модулей и бинарных компонентов. Понятно, что эффект добавления новой формы весьма существенен.

### Плохой проект

Еще раз рассмотрим поставленный вопрос. Решение, представленное в листинге 9.1, негибкое, поскольку добавление функции `Triangle` вызывает перекомпилирование и повторное развертывание функций `Shape`, `Square`, `Circle`, и `DrawAllShapes`. Кроме того, этот проект неустойчив, поскольку может появиться множество других конструкций `switch/case` или `if/else`, сложных в обнаружении и расшифровке. Проекту присуща неподвижность, поскольку при попытке повторного использования `DrawAllShapes` в другой программе необходимо использовать `Square` и `Circle`, даже если эта новая программа в них не нуждается. Таким образом, листинг 9.1 демонстрирует множество признаков “сырого” проекта.



### Соответствие принципу открытия-закрытия

В листинге 9.2 демонстрируется код, решающий проблему `square/circle`, который соответствует принципу открытия-закрытия. В этом случае был создан абстрактный класс `Shape`. Этот класс включает абстрактный единственный метод `Draw`. Оба класса `Circle` и `Square` наследуются из класса `Shape`.

---

**Листинг 9.2. Решение проблемы "окружность/квадрат" с учетом принципа открытия-закрытия**

---

```
class Shape
{
public:
 virtual void Draw() const = 0;
};

class Square : public Shape
{
public:
 virtual void Draw() const;
};

class Circle : public Shape
{
public:
 virtual void Draw() const;
};

void DrawAllShapes(vector<Shape*>& list)
{
 vector<Shape*>::iterator i;
 for (i=list.begin(); i != list.end(); i++)
 (*i)->Draw();
```

---

Обратите внимание, что если мы хотим расширить поведение функции `DrawAllShapes` в листинге 9.2 с тем, чтобы нарисовать новую форму, все, что потребуется сделать, — добавить новый производный класс `Shape`. Функция `DrawAllShapes` может не изменяться. Таким образом, функция `DrawAllShapes` соответствует принципу открытия-закрытия. Ее поведение можно изменить, не прибегая к изменению ее самой по себе. Действительно, добавление класса `Triangle` не оказывает *абсолютно никакого влияния* на модули, представленные здесь. Ясно, что определенная часть системы должна измениться с тем, чтобы иметь дело с классом `Triangle`, однако весь код, представленный здесь, невосприимчив к изменениям.

В настоящем приложении класс `Shape` включает намного больше методов. Поэтому процесс добавления новой формы в приложение довольно прост, поскольку все, что при этом требуется — создать новый производный класс и применить все его функции. Нет необходимости искать во всем приложении фрагменты, требующие изменений. Это решение не является непрочным.

Это решение также не является закрепощенным. Ни один из существующих исходных модулей не требует изменений, и с одним исключением ни один из существующих двоичных модулей не требует перестроения. Модуль, который фактически создает экземпляры новых производных класса `Shape`, должен быть

изменен. Обычно эти действия должны выполняться главным модулем, некоторой функцией, вызываемой этим модулем, либо методом некоторого объекта, созданного модулем `main`<sup>4</sup>.

И наконец, это решение не является неизменным. Функция `DrawAllShapes` может повторно применяться любым приложением, причем вовсе не требуется вызывать методы `Square` или `Circle`. Таким образом, это решение не имеет ни одного из признаков “загнивающего” проекта, о которых говорилось выше.

Эта программа соответствует принципу открытия-закрытия. *Она изменяется посредством добавления нового кода, а не путем изменения уже существующего кода.* Поэтому она не подвержена каскаду изменений, которые обнаруживаются в несовместимых программах. В этом случае следует добавить лишь новый модуль, а также выполнить связанные с главным модулем изменения, которые позволяют создать новые экземпляры объектов.

## Не все так хорошо

Предыдущий пример соответствовал идеальной ситуации! Подумаем, что могло бы произойти с функцией `DrawAllShapes` из листинга 9.2, если бы мы решили, что *все* окружности *следует нарисовать прежде* квадратов. Функция `DrawAllShapes` не защищена от подобного рода изменений. В целях реализации этого изменения потребуется перейти к `DrawAllShapes`, а также просмотреть список на предмет обнаружения `Circles` и `Squares`.

## Предположение и “естественная” структура

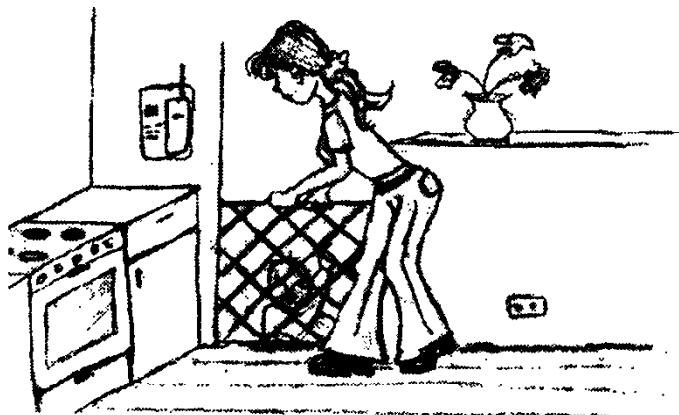
Если бы можно было предвидеть такое изменение, то можно было бы создать абстракцию, которая могла бы предотвратить это изменение. Абстракции, выбранные в листинге 9.2, представляют собой большей частью помехи, нежели что-то реально полезное. Это утверждение может звучать странно. В конце концов, что может быть более естественным, чем базовый класс `Shape` с производными классами `Square` и `Circle`? Почему бы ни использовать эту естественную наилучшую модель? Дело в том, что эта модель *не является* естественной для системы, в которой очередность играет большую роль, чем тип формы.

Это приводит нас к неутешительному заключению. Вообще говоря, не имеет значения то, насколько “закрыт” модуль, поскольку всегда найдется какое-нибудь изменение, от которого он не будет закрыт. *Нет такой модели, которая была бы естественной для всех контекстов.*

Поскольку завершение не может быть произведено, оно должно быть стратегическим. То есть, разработчик проекта должен выбрать такие виды изменений, от которых он должен защитить свой проект. Он должен предвидеть самые ве-

---

<sup>4</sup>Подобные объекты называются фабриками и будут рассматриваться в главе 21.



роятные типы изменений, а затем создать абстракции, чтобы защититься от этих изменений.

Такой шаг требует определенной доли предвидения, что приходит только с опытом. Опытный разработчик надеется, что он достаточно хорошо знает пользователей и производство, чтобы судить о вероятности различных видов изменений. Затем он применяет принцип открытия-закрытия в отношении самых вероятных изменений.

Но все это не так просто. Такая методика подразумевает выстраивание приходящих с опытом предположений о возможных видах изменений, которым со временем будет подвержено приложение. Конечно же, в большинстве случаев их предположения будут ошибочны.

Кроме того, соответствие принципу открытия-закрытия обходится недешево. Разработчики тратят немало времени и усилий, чтобы создать соответствующие абстракции. Эти абстракции также усложняют проект программы. Разработчики могут позволить себе лишь ограниченное количество абстракций. Понятно, что мы стремимся ограничить применение принципа открытия-закрытия по отношению к возможным изменениям.

Как можно узнать, какие из изменений наиболее вероятны? Мы проводим соответствующее исследование, мы задаем соответствующие вопросы и используем свой опыт и здравый смысл. И в конце концов, *мы ждем, пока не произойдут изменения!*

## “Забрасывание удочек”

Как защитить себя от изменений? В прошлом веке был популярным такой принцип — мы “поймаем на удочку” изменения, которые, по нашему мнению, должны произойти. Мы предполагали, что таким образом наше ПО может оставаться гибким.

Тем не менее, “забрасывание удочек” зачастую не приносило никаких результатов. Хуже того, программа начинала проявлять признаки неоправданной сложности, которые необходимо было поддерживать и управлять ими, даже если они

не использовались. Это уже плохой признак. Нет смысла загружать проект, имеющий множество ненужных абстракций. Вместо этого мы часто ждем, пока нам на самом деле не понадобится абстракция, и только затем загружаем ее в проект.

## Обманутый однажды...

Есть такая старая поговорка: “Обманутый однажды винит обманщика, а обманутый дважды винит себя”. Эта народная мудрость актуальна и для разработчиков ПО. Защищая себя от загрузки программ, содержащих неоправданные сложности, мы позволяем таким образом обмануть себя *однажды*. Это значит, что мы изначально создаем код, предполагая, что он не должен изменяться. Когда происходят изменения, мы прибегаем к абстракциям, которые защищают нас от будущих изменений *такого рода*. Короче говоря, мы “принимаем первую пулю”, успокаиваясь на том, что защитили себя от “других пуль, выпущенных из этого же ружья”.

## Стимулирование изменений

Решив “принять первую пулю”, нам теперь выгодно “принимать часто вылетающие пули”. Необходимо определить наиболее вероятные изменения, прежде чем зайти слишком далеко по “тропе разработок”. Чем дольше мы будем искать возможные изменения, тем сложнее будет создать соответствующие абстракции.

Поэтому возникает необходимость в стимулировании изменений, что достигается путем выполнения определенных действий, о которых подробнее говорилось в главе 2.

- В первую очередь мы создаем тесты. Тестирование — один из видов использования системы. При первоочередном создании тестов система становится тестируемой. Поэтому изменения тестов будут впоследствии уже ожидаемыми. Можно создать абстракции, благодаря которым система становится тестируемой. Вполне вероятно, что многие из этих абстракций в будущем защитят нас от изменений другого рода.
- Процесс разработки занимает очень короткий цикл — вместо нескольких недель всего лишь несколько дней.
- Мы прорабатываем свойства прежде, чем инфраструктуру и часто демонстрируем эти свойства заинтересованным сторонам.
- Мы разрабатываем самые важные свойства в первую очередь.
- Мы выпускаем ПО рано и часто. Мы как можно быстрее и чаще предоставляем его своим заказчикам и пользователям.

## Использование абстракции в целях явного закрытия

Итак, мы получили “первую пулю”. Пользователь желает, чтобы мы нарисовали все Circles прежде чем Squares. Теперь нам необходимо защитить себя от любых будущих изменений подобного рода.

Как можно защитить функцию `DrawAllShapes` от изменений в очередности рисования? Помните, что завершение основывается на абстракции. Таким образом, чтобы защитить функцию `DrawAllShapes` от очередности, необходима своего рода “абстракция очередности”. Благодаря такой абстракции обеспечивается абстрактный интерфейс, посредством которого могут осуществляться любые возможные политики очередности.

Политика очередности предполагает, что, имея два объекта, можно определить, который из них должен быть нарисован первым. Можно определить абстрактный метод `Shape` под именем `Precedes`. Эта функция использует другой метод `Shape` в качестве аргумента и возвращает результат булева типа. Значение результата верно, если объект `Shape`, получающий сообщение, будет нарисован перед объектом `Shape`, переданным в качестве аргумента.

В языке C++ эта функция может быть представлена посредством перегруженной функции `operator<`. В листинге 9.3 показано, как должен выглядеть класс `Shape` при использовании методов очередности.

Зная теперь, как определить относительную очередь двух объектов `Shape`, можно отсортировать их и нарисовать по порядку. В листинге 9.4 представлен код C++, с помощью которого выполняется эта процедура.

---

#### Листинг 9.3. Методы расстановки приоритетов для `Shape`

---

```
class Shape
{
public:
 virtual void Draw() const = 0;
 virtual bool Precedes(const Shape*) const = 0;

 bool operator<(const Shape& s) {return Precedes(s);}
};
```

---

---

#### Листинг 9.4. Методы расстановки приоритетов для `DrawAllShapes`

---

```
template <typename P>
class Lessp // утилита сортировки контейнеров указателей.
{
public:
 bool operator()(const P p, const P q) {return (*p) < (*q);}
};

void DrawAllShapes(vector<Shape*>& list)
{
 vector<Shape*> orderedList = list;
 sort(orderedList.begin(),
 orderedList.end(),
 Lessp<Shape*>());
```

---

```

vector<Shape*>::const_iterator i;
for (i=orderedList.begin(); i != orderedList.end(); i++)
 (*i)->Draw();
}

```

---

С помощью приведенного кода можно назначить приоритеты для объектов `Shape` и нарисовать их в соответствующем порядке. Однако у нас до сих пор нет подходящей абстракции очередности. Как полагается, методы, соответствующие отдельным формам, будут перекрыты методом `Precedes` в целях расстановки приоритетов. Как это сделать? Какой код необходимо создать в `Circle::Precedes`, чтобы удостовериться в том, что объекты `Circles` нарисованы прежде, чем объекты `Squares`? Рассмотрим листинг 9.5.

---

#### Листинг 9.5. Расстановка приоритетов для `Circle`

---

```

boot Circle::Precedes(const ShapeSc s) const
{
 if (dynamic_cast<Square*>(s))
 return true;
 else
 return false;
}

```

---

Должно быть понятно, что эта функция и все ее “близнецы” в других производных классах `Shape` не соответствуют принципу открытия-закрытия. Ни один из способов не может закрыть их от новых производных `Shape`. Каждый раз, создавая новый производный класс для `Shape`, необходимо изменять все функции `Precedes()`<sup>5</sup>.

Конечно, не важно, будут когда-либо созданы новые производные `Shape` или нет. С другой стороны, если они создаются часто, то проект в этом случае может давать значительную “пробуксовку”. Опять же, мы должны получить “ первую пулью”.

## Использование подхода, управляемого данными, в целях достижения завершения

Если необходимо, чтобы производные классы `Shape` “не знали” о существовании друг друга, можно использовать подход, управляемый табличными данными. В листинге 9.6 демонстрируется такая возможность.

---

<sup>5</sup>Эту проблему можно разрешить с помощью шаблона `acyclic visitor`, описанного в гл. 29. Рассмотрение этого решения сейчас было бы немного преждевременным. В конце данной главы я напомню об этом пункте.

---

**Листинг 9.6. Механизм расстановки приоритетов, управляемый табличными данными**

---

```
#include <typeinfo>
#include <string>
#include <iostream>

using namespace std;

class Shape
{
public:
 virtual void Draw() const = 0;
 bool Precedes(const Shapes) const;
 bool operator<(const Shape& s) const
 {return Precedes(s);}

private:
 static const char* typeOrderTable[];
};

const char* Shape::typeOrderTable[] =
{
 type id(Circle).name(),
 typeid(Square).name(),
 0
};

// Эта функция осуществляет поиск имен классов в
// таблице. Таблица определяет порядок, в котором
// будут нарисованы формы. Выбираются формы,
// для которых отсутствуют предшествующие формы
//
bool Shape::Precedes(const Shapes& s) const
{
 const char* thisType = typeid(*this).name();
 const char* argType = typeid(s).name();
 bool done = false;
 int thisOrd = -1;
 int argOrd = -1;
 for (int i=0; !done; i++)
 {
 const char* tableEntry = typeOrderTable[i];
 if (tableEntry != 0)
 {
 if (strcmp(tableEntry, thisType) == 0)
 thisOrd = i;
 if (strcmp(tableEntry, argType) == 0)
 argOrd = i;
 if ((argOrd >= 0) && (thisOrd >= 0))
 done = true;
 }
 }
}
```

```

 done = true;
 }
 else // table entry == 0
 done = true;
 }
 return thisOrd < argOrd;
}

```

---

Применив описанный подход, мы успешно закрыли функцию `DrawAllShapes` от вопросов расстановки приоритетов (в общем), а также каждый производный объект `Shape` от создания новой производной формы. Также была изменена политика, определяющая переупорядочивание объектов `Shape` в соответствии с их типами (например, изменение очередности таким образом, чтобы сначала были нарисованы объекты `Squares`).

Единственный элемент, который не закрыт от расстановки приоритетов различных `Shapes`, — это сама таблица. Эта таблица может быть размещена в своем собственном модуле, отдельно от всех остальных модулей, таким образом, чтобы вносимые в нее изменения не могли повлиять на любые другие модули. Действительно, в языке C++ можно выбрать таблицу, которую можно использовать во время компоновки.

## Резюме

Во многих отношениях принцип открытия-закрытия лежит в основе объектно-ориентированного проектирования. Соответствие этому принципу порождает большие преимущества, характерные для объектно-ориентированной технологии (т.е. гибкость, повторное использование и удобство эксплуатации). Но в то же время соответствие этому принципу не достигается лишь использованием объектно-ориентированного языка программирования. Не следует также прибегать к чрезмерным абстракциям для каждой части приложения. Вместо этого разработчики советуют применять абстракции только для тех частей программы, которые подвержены частым изменениям. *Вопрос предотвращения непродуманных абстракций имеет такое же значение, как и сама абстракция.*

## Литература

1. Jacobson I. *Object-Oriented Software Engineering*. Reading. MA: Addison-Wesley, 1992.
2. Meyer B. *Object-oriented Sofware Construction*. Upper Saddle River. NJ: Prentice Hall, 1997.

# 10

## Принцип подстановки Лискоу



Основными механизмами, следующими по значимости за принципом ОСР (Open-Closed Principle, принцип открытия-закрытия), являются абстракция и полиморфизм. В статически типизированных языках, таких как C++ и Java, один из ключевых принципов, поддерживающих абстракцию и полиморфизм, — это наследование. Благодаря наследованию можно создавать производные классы, используя абстрактные правила для базовых классов.

Какими основными принципами следует руководствоваться при использовании механизма наследования? Какими характеристиками должна обладать оптимальная структура наследования? Какие причины ведут к созданию иерархий, структура которых не согласуется с системой ОСР? Ответы на эти вопросы позволяет получить принцип подстановки Лискоу (LSP/ППЛ).

## Принцип подстановки Лискоу

Принцип LSP может быть сформулирован следующим образом.

**ПОДТИПЫ ДОЛЖНЫ БЫТЬ ЗАМЕНЯЕМЫ ИХ ИСХОДНЫМИ ТИПАМИ.**

Барбара Лискоу (Barbara Liskov) сформулировала это правило в 1988 году<sup>1</sup>.

*Необходимо четко усвоить следующее правило. Если каждому объекту  $O1$  типа  $S$  соответствует объект  $O2$  типа  $T$ . Таким образом, всех программ  $P$ , определенных на основе  $T$ , поведение  $P$  не меняется при замене  $O1$  на  $O2$ , причем  $S$  является подтипом  $T$ .*

Значимость этого правила становится очевидной в случае, если рассмотреть последствия его нарушений. Предположим, что у нас есть функция  $f$ , содержащая в качестве аргумента указатель или ссылку на базовый класс  $B$ . Также представим, что существует производная от  $B$  (сокращенно  $D$ ), которая при подстановке в функцию  $f$  под видом  $B$  вызывает изменения в поведении последней. В этом случае  $D$  игнорирует принцип LSP. Очевидно, что  $D$  представляет собой одну из “неустойчивостей”  $f$ .

Разработчики функции  $f$  могут подвергнуться искушению провести определенные тесты, которые покажут что поведение функции не изменяется при подстановке в нее производной функции  $D$ . Такое тестирование отрицает основные принципы ОСР, поскольку оно не охватывает весь диапазон значений производных от  $B$ . Обычно подобными вещами “грешат” неопытные разработчики (или, что гораздо хуже, решения принимаются в спешке).

## Простой пример нарушения принципа LSP

Нарушение принципа LSP зачастую приводит к тому, что информация о типах в процессе исполнения (RTTI) применяется в стиле, не соответствующем принципам ОСР. Обычно в таких случаях для определения подходящего типа объекта (не вызывающего изменения в поведении нужной функции) используются операторы `if` или последовательности операторов `if/else`. Обратите внимание на листинг 10.1.

---

**Листинг 10.1.** Нарушение принципа подстановки приводит к игнорированию правил ОСР

---

```
struct Point {double x, y;};

struct Shape {
 enum ShapeType {square, circle} itsType;
 Shape(ShapeType t) : itsType(t) {}
};
```

<sup>1</sup>[Liskov88].

```
struct Circle : public Shape
{
 Circle() : Shape(circle) {};
 void Draw() const;
 Point itsCenter;
 double itsRadius;
};

struct Square : public Shape
{
 Square() : Shape(square) {};
 void Draw() const;
 Point itsTopLeft;
 double itsSide;
};

void DrawShape(const Shapes& s)
{
 if (s.itsType == Shape::square)
 static_cast<const Square&>(s).Drawf()
 else if (s.itsType == Shape::circle)
 static_cast<const Circle&>(s).Draw();
}
```

Ясно, что функция `DrawShape`, код которой приведен в листинге 10.1, противоречит принципам OCP. Она должна работать с любыми производными классами `Shape` и изменяться при появлении новых таких производных классов. В действительности же пристальное рассмотрение данной функции сразу выявляет ее “слабые места”. Почему же программисты все же используют подобные функции?

Возьмем для примера среднестатистического инженера по имени Джо. Он в свое время изучал объектно-ориентированную технологию и пришел к выводу о том, что использование полиморфизма приводит к чрезмерному расходованию ресурсов системы<sup>2</sup>. Поэтому он определил класс `Shape`, не используя виртуальные функции. Классы (структуры) `Square` и `Circle` представляют собой производные от `Shape` и включают функции `Draw()`, однако они не подменяют собой функцию в классе `Shape`. Поскольку `Circle` и `Square` не могут заменяться в классе `Shape`, функция `DrawShape` должна проконтролировать входящее значение `Shape`, определить его тип, и затем вызвать подходящую функцию `Draw`.

Тот факт, что `Square` и `Circle` не являются взаимозаменяемыми с `Shape`, нарушает принцип LSP. Это приводит к нарушению основных принципов OCP при работе с функцией `DrawShape`. Таким образом, *нарушение принципа LSP – это скрытое нарушение всей структуры OCP*.

<sup>2</sup>На достаточно быстром компьютере задержка не превышает 1 наносекунды для каждого метода, поэтому аргументация Джо несостоятельна.

## Классы `Square` и `Rectangle`: примеры менее заметных нарушений

Существуют и менее явные случаи нарушения принципа LSP. Рассмотрим приложение, в котором используется класс `Rectangle`, код которого приводится в листинге 10.2.

---

### Листинг 10.2. Класс `Rectangle`

---

```
class Rectangle
{
public:
 void SetWidth(double w) {itsWidth=w;}
 void SetHeight(double h) {itsHeight=h;}
 double GetHeight() const {return itsHeight;}
 double GetWidth() const {return itsWidth;}
private:
 Point itsTopLeft;
 double itsWidth;
 double itsHeight;
};
```

---

Представим себе, что это приложение работает вполне корректно и установлено на многих Web-узлах. Как часто бывает при разработке популярного ПО, пользователи время от времени требуют изменить отдельные компоненты программы. Допустим, что в один прекрасный день пользователям потребовалась возможность манипулировать квадратами (название класса `Rectangle` на русский язык переводится как “прямоугольник”) в дополнение к возможности работы с прямоугольниками.

Часто говорят, что наследование представляет собой взаимосвязь типа *IS-A*. Иными словами, если новый и старый типы объекта взаимозависимы согласно условиям *IS-A*, то класс нового объекта будет производным от класса старого (объекта).

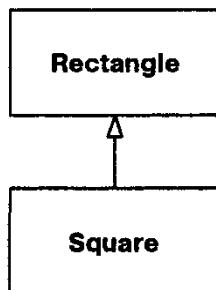
По всем правилам и во всех случаях квадрат представляет собой прямоугольник. Поэтому логично рассматривать класс `Square` как производный от класса `Rectangle`. (рис. 10.1.)

Такое использование взаимосвязи типа *IS-A* считается одной из основных техник объектно-ориентированного анализа<sup>3</sup>. Квадрат представляет собой частный случай прямоугольника, поэтому класс `Square` должен быть производным от класса `Rectangle`.

Однако такой способ мышления может привести к некоторым незаметным на первый взгляд, но довольно серьезным проблемам. Как правило, эти проблемы

---

<sup>3</sup>Часто применяемый, но редко определяемый термин.



**Рис. 10.1.** Класс *Square* наследует класс *Rectangle*

нельзя предусмотреть заранее, до тех пор, пока они не “проявятся” в программном коде.

Первым признаком того, что “что-то не так”, является факт, что класс *Square* не нуждается в переменных *itsHeight* и *itsWidth*. Он наследует их от класса *Rectangle*. Очевидно, что такой код не слишком “экономен”. В большинстве случаев подобное дублирование не имеет серьезных последствий. Однако если нам необходимо создавать сотни тысяч объектов типа *Square* (к примеру, при работе с системами автоматизированного проектирования типа CAD/CAE, где отдельные элементы представляются в виде маленьких квадратов), работа программы значительно замедлится.

Предположим, что нас не слишком волнует эффективность использования памяти нашей программой. Но существуют и другие проблемы, вытекающие в силу зависимости между классами *Square* и *Rectangle*. Квадрат будет наследовать функции *SetWidth* и *SetHeight*. Эти функции не могут применяться для *Square*, так как ширина и высота квадрата одинаковы. Сложившаяся ситуация непосредственно указывает на новую проблему. Однако есть способ ее обойти. Мы можем подменить функции *SetWidth* и *SetHeight*, как показано ниже.

```

void Square::SetWidth(double w)
{
 Rectangle::SetWidth(w);
 Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
 Rectangle::SetHeight(h);
 Rectangle::SetWidth(h);
}

```

Теперь при изменении ширины объекта *Square* его высота изменится автоматически. И наоборот, изменение высоты повлечет за собой соответственную коррекцию ширины объекта. Таким образом, инварианты<sup>4</sup> класса (утверждения,

<sup>4</sup>Эти свойства всегда истинны независимо от состояния.

которые должны быть истинными при создании каждого экземпляра объекта класса и сохранять свое значение в течение всего времени существования объекта) `Square` остаются нетронутыми. Объект `Square` будет оставаться правильным квадратом.

```
Square s;
s.SetWidth(1); // высота --- 1.
s.setHeight(2); // ширина и~высота совпадают и
// равны 2. Отлично.
```

Рассмотрим следующую функцию:

```
void f(Rectangles r)
r.setWidth(32); // вызывает метод Rectangle::setWidth
```

Если мы включим в эту функцию ссылку на объект `Square`, то структура объекта исказится, потому что его высота не будет изменяться. Это — явное нарушение принципа LSP. Функция `f` не работает с производными от ее аргументов. Причина сбоя заключается в том, что функции `setWidth` и `setHeight` не были объявлены как `virtual` в объекте `Rectangle`; следовательно, они не являются полиморфными.

Подобную ситуацию достаточно легко исправить. Однако, если создание производного класса вынуждает нас вносить изменения в базовый класс, это может означать, что в общей структуре кода есть недостатки. Несомненно, что при этом нарушаются принципы OCP. Можно, конечно, сказать, что `setWidth` и `setHeight` не получили признак `virtual` из-за простой забывчивости, а теперь эта ошибка уже исправлена. Но при этом достаточно сложно обосновать такие действия после того, как значения ширины и высоты прямоугольника устанавливались с помощью крайне примитивных операций. По этой причине имело смысл использовать виртуальные функции, если только заранее мы не предвидели появление объекта `Square`.

Тем не менее, давайте предположим, что мы приняли данный аргумент и внесли изменения в классы. При этом мы получим код, показанный в листинге 10.3.

---

#### Листинг 10.3. Самосогласованные объекты `Rectangle` и `Square`

---

```
class Rectangle
{
public:
 virtual void SetWidth(double w) {itsWidth=w; }
 virtual void SetHeight(double h) {itsHeight=h; }
 double GetHeight() const {return itsHeight; }
 double GetWidth() const {return itsWidth; }
private:
 Point itsTopLeft
 double itsHeight;
 double itsWidth;
```

```
class Square : public Rectangle
{
public:
 virtual void SetWidth(double w);
 virtual void SetHeight(double h);
};

void Square::SetWidth(double w)
{
 Rectangle::SetWidth(w);
 Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
 Rectangle::SetHeight(h);
 Rectangle::SetWidth(h);
}
```

## Реальная проблема

Итак, `Square` и `Rectangle` теперь работают как надо. Не имеет значения, что вы делаете с объектом `Square`, — он в любом случае остается математически правильным квадратом. И вне зависимости от того, какие действия вы производите над объектом `Rectangle`, он остается прямоугольником. Более того, вы можете включить `Square` в функцию, содержащую указатель или ссылку на `Rectangle`, и объект `Square` по-прежнему будет иметь вид квадрата.

Таким образом, мы можем сделать вывод о том, что структура программного кода является самосогласованной и правильной. Однако такое заключение будет ошибочным. Самосогласованная структура не обязательно согласуется и работает корректно со всеми пользователями! Обратите внимание на следующую функцию `g`.

```
void g(Rectangles r)
{
 r.setWidth(5);
 r.setHeight(4);
 assert(r.Area() == 20);
}
```

Эта функция вызывает функции `Setwidth` и `SetHeight`, относящиеся к объекту `Rectangle`. В случае прямоугольника все работает отлично, но при обработке квадрата программа выдаст ошибку. Вот в чем заключается настоящая проблема. *Автор функции `g` полагал, что изменение ширины прямоугольника не повлияет на его высоту.*

Вполне естественно предположить, что изменение ширины прямоугольника не влияет на его высоту. Однако далеко не все объекты, которые можно использо-

вать в данной функции, отвечают этим условиям. Если вы используете экземпляр класса `Square` в функции типа `g`, автор которой прибегнул к вышеизложенным допущениям, то эта функция даст сбой. Функция `g` является неустойчивой по отношению к иерархии `Square/Rectangle`.

Пример с функцией `g` показывает, что могут существовать функции, которые работают с указателями или ссылками на объекты `Rectangle`, но не могут оперировать с объектами `Square`. Таким образом, при использовании таких функций объект `Square` не является заменяемым для `Rectangle`, взаимосвязь между такими объектами нарушает принцип подстановки Лискоу.

Можно, конечно, утверждать, что вся проблема заключается в самой функции `g` — автор не имел права делать допущение, что ширина и высота не зависят друг от друга. Однако вряд ли сам автор согласится с этим. Функция `g` использует объект `Rectangle` в качестве аргумента. Существуют инварианты, истинные утверждения, которые явно относятся к классу под названием `Rectangle`, и одно из таких утверждений как раз гласит, что ширина и высота объекта не зависят друг от друга. Нет никаких причин, по которым автор функции `g` должен был избегать этого инварианта. Нарушил же это правило, тот, кто написал код объекта `Square`.

Стоит отметить, что разработчик класса `Square` не нарушил инвариант, относящийся к `Square`. Поскольку `Square` является производной от `Rectangle`, автор `Square` нарушил инвариант `Rectangle`!

## Адекватность модели

Принцип LSP подводит нас к очень важному заключению. *Модель, рассматриваемая отдельно от общей структуры, не может быть однозначно оценена в плане своей пригодности.* Пригодность конкретной модели может быть определена только в отношении ее клиентов/области использования. Например, при исследовании финальных версий классов `Square` и `Rectangle` (в “изолированном” состоянии), мы обнаружим, что они являются самосогласованными и вполне адекватными. Если же мы будем рассматривать их с точки зрения программиста, делающего допущения относительно базовых классов, то вся структура модели буквально “рассыплется на части”.



При оценке пригодности отдельных элементов проекта нельзя рассматривать их “в отрыве” от всей системы. Нужно проводить исследование в рамках разумных предположений и допущений, сделанных пользователями данного проекта<sup>5</sup>.

Но как узнать, какие разумные допущения будут сделаны пользователями проекта? Как правило, это нелегко предвидеть. Более того, если мы постараемся заранее предвосхитить все возможные пожелания пользователей, то весь процесс проектирования может пойти по пути неоправданного усложнения. Поэтому имеет смысл отложить воплощение этих свойств (за исключением коррекции наиболее явных нарушений принципа LSP) до тех пор, пока не возникнет реальный риск неустойчивости системы.

## Неправильное поведение объекта

Что же все-таки произошло? Почему корректная, на первый взгляд, модель `Square` и `Rectangle` оказалась неправильной? В конце концов, разве квадрат не является прямоугольником? Имеет ли здесь место нарушение зависимости между ними?

Все дело в разработчике функции `g!` Естественно, квадрат представляет собой частный случай прямоугольника, однако с точки зрения программиста объект `Square` определенно *не является* объектом `Rectangle`. Почему? Да потому, что *поведение* объекта `Square` отличается от поведения объекта `Rectangle`. С этой точки зрения “квадрат” не является “прямоугольником”, причем *поведение* объекта — единственный критерий оценки его программой. Принцип подстановки ясно дает понять, что в объектно-ориентированном проектировании (ООП) взаимосвязь типа *IS-A* имеет прямое отношение к *поведению* объектов, которое, в свою очередь, зависит от пожеланий клиентов.

## Проектирование на контрактной основе

Многие разработчики не очень хорошо себе представляют “разумное” и “рациональное” поведение объектов. Как узнать, чего клиенты ждут от вас? Существует технология, позволяющая это выяснить, не нарушая при этом принципа LSP. Эта методика называется проектированием по контракту (*design by contract*, DBC), а разработана она Берtrandом Мейером (Bertrand Meyer)<sup>6</sup>.

При использовании методики DBC автор отдельного класса составляет для него так называемый “контракт”. Данный контракт информирует авторов клиентского кода об ожидаемом поведении данного класса. Здесь указываются предварительные и выходные условия для каждого применяемого метода. Для успешного

<sup>5</sup>Достаточно часто отмечают, что эти законные утверждения формулируются в терминах тестовых модулей, написанных для базового класса. Это еще одна веская причина для выполнения разработки, управляемой тестами.

<sup>6</sup>[Meyer97], глава 11, с. 331.

применения технологии DBC во входных условиях не должно быть ошибок. По завершению работы DBC гарантирует отсутствие ошибок в выходных условиях.

Выходные условия для `Rectangle::SetWidth(double w)` можно посмотреть следующим образом.

```
assert((itsWidth == w) && (itsHeight == old.itsHeight));
```

В данном примере “old” — это старое значение высоты прямоугольника (до того, как была вызвана функция `SetWidth`). Теперь можно сформулировать правило Мейера для входных и выходных условий производных.

*Процедура повторного описания [при работе с производными классами]* может заменять оригинальное входное условие только равным ему или “более слабым”, при этом выходное условие заменяется эквивалентным или “сильнейшим”<sup>7</sup>.

“Слабое” или “сильное” условие означает следующее. X слабее Y в том случае, если X не выполняет все ограничения, налагаемые на Y. При этом количество поддерживаемых ограничений не имеет значения.

Иными словами, при работе с объектом через интерфейс базового класса, пользователь оперирует только входными или выходными условиями для базового класса. Таким образом, производные объекты не должны укладываться в более жесткие рамки условий, чем того требует базовый класс. То есть они должны отвечать всем требованиям, предъявляемым базовому классу. Также производные классы должны соответствовать выходным условиям для базового класса. То есть, их поведение и выходные данные не должны нарушать ограничения, установленные для базового класса. Пользователей базового класса не должны смущать результаты работы производных классов.

Понятно, что выходное условие `Square::SetWidth (double w)` “слабее”<sup>8</sup>, чем такое же условие для `Rectangle::SetWidth (double w)`, поскольку оно не поддерживает ограничение `(itsHeight == old.itsHeight)`. Таким образом, функция `SetWidth` объекта `Square` нарушает условия контракта для базового класса.

Определенные языки программирования, такие как Eiffel, обладают прямой поддержкой входных и выходных условий. Вы можете явно указать их и получить рабочую систему, которая будет проверять их выполнение. Ни C++, ни Java такими возможностями не обладают. В этих языках мы должны вручную обрабатывать входные и выходные условия каждого метода и проверять их на соответствие правилу Мейера. Кроме того, не лишним будет описать все условия и дать подробные комментарии.

---

<sup>7</sup>[Meyer97], с. 573, правило перенаправления утверждения (1).

<sup>8</sup>Определение “слабее” может вводить в заблуждение. X слабее Y, если X не устанавливает все ограничения для Y. Не имеет значения, какое количество новых ограничений устанавливается X.

## Спецификация контрактов при проведении модульных тестов

Контракты также могут описываться путем подбора модульных тестов. При всестороннем тестировании поведения класса его поведение становится вполне определенным. Благодаря анализу результатов тестирования авторы клиентского кода могут делать разумные допущения при работе с классами.

## Реальный пример

Хватит с нас квадратов и прямоугольников! Попробуем разобраться в том, каким образом принцип LSP используется при разработке реальных программ. Давайте рассмотрим пример, представляющий собой часть реального проекта (над которым автор книги работал несколько лет назад).

### Мотивация

В начале 90-х годов прошлого века я приобрел библиотеку классов от стороннего производителя, которая содержала несколько классов-контейнеров (объемлющих классов). Контейнеры были в некотором роде связаны с объектами *Bags* и *Sets* языка Smalltalk. Существовало два множества *Set* и два подобных множества *Bag*. Первое множество называлось “ограниченным” и представляло собой массив. Второе множество носило имя “неограниченного” и было создано на основе связанного списка.

Создатель *BoundedSet* указал максимальное число элементов, которые могут содержаться в этом множестве. Для них было зарезервировано место в массиве внутри *BoundedSet*. Таким образом, в случае успешного создания множества *BoundedSet* можно было не волноваться, что не хватит памяти. Поскольку массив представлял собой обычную матрицу, скорость обработки данных была очень высокой. При нормальной работе распределения памяти не производилось. При предварительном резервировании памяти можно было быть уверенным в том, что обработка массива *BoundedSet* не вызовет переполнения рабочей области памяти (так называемой “кучи”). С другой стороны, подобный стиль программирования приводит к неэкономному расходованию памяти, поскольку лишь в редкие моменты времени будет использоваться весь зарезервированный объем. С другой стороны, множество *UnboundedSet* не ограничено относительно количества входящих в него элементов. Пока будет оставаться свободная память, можно будет дополнять *UnboundedSet* новыми элементами. Следовательно, такая технология является более гибкой. К тому же она гораздо более экономична, поскольку использует столько памяти, сколько нужно для хранения данных, а не весь ее доступный/зарезервированный объем. При этом работает такая система медлен-

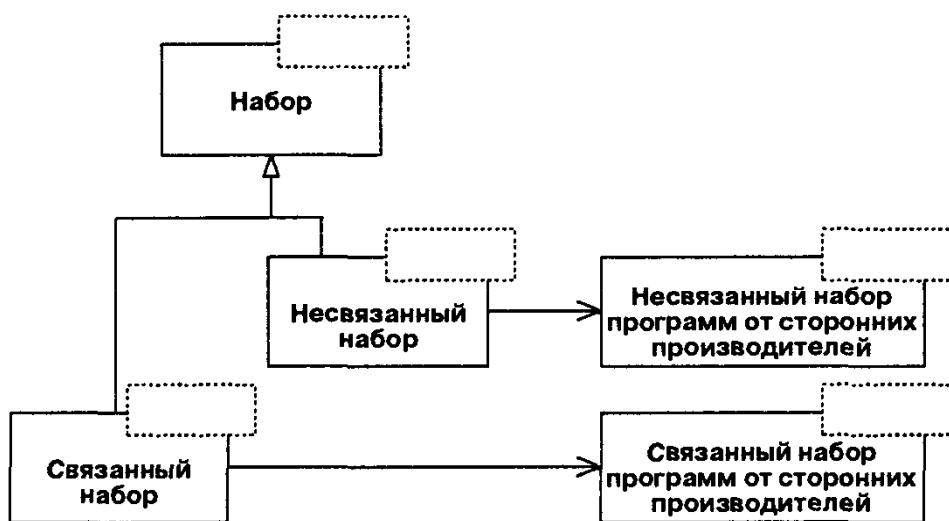


Рис. 10.2. Класс-контейнер на уровне адаптера

нее, так как происходит постоянное перераспределение выделяемой для работы программы памяти. Наконец, существует опасность переполнения памяти.

Поскольку автор не слишком любит работать с интерфейсами классов от сторонних производителей. Мне не хотелось, чтобы мой программный код зависел от них, потому что со временем планировалось заменить данные классы лучшими. Поэтому я вложил “контейнеры” от сторонних производителей в свой собственный абстрактный интерфейс, как показано на рис. 10.2.

Я создал абстрактный класс под названием **Set**, содержащий чисто виртуальные функции **Add**, **Delete** и **isMember**, как показано в листинге 10.4. Эта структура унифицировала связанные и несвязанные переменные из двух наборов от сторонних производителей и позволяла оперировать ими через общий интерфейс. Таким образом, любой клиент может принимать аргумент типа **Set<T>&**, не заботясь о том, с переменными из какого набора (ограниченного или неограниченного) работает в данный момент класс **Set**. (см. функцию **PrintSet** в листинге 10.5.)

---

#### Листинг 10.4. Абстрактный класс Set

```

template <class T>
class Set
{
public:
 virtual void Add(const T&) = 0;
 virtual void Delete (const T&) = 0;
 virtual bool IsMember(const T&) const = 0;

```

**Листинг 10.5. Функция PrintSet**

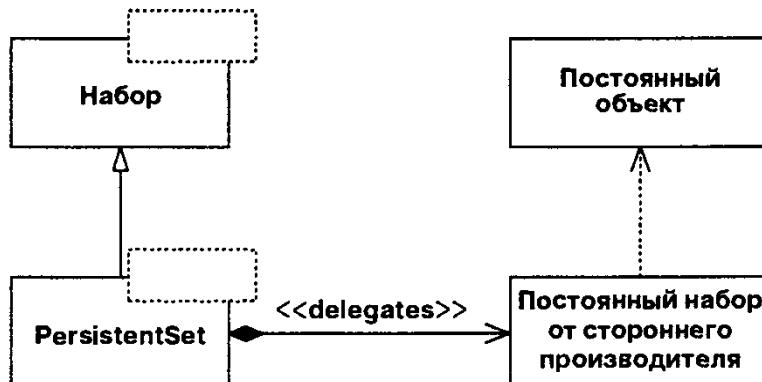
```
template <class T>
void PrintSet(const Set<T>& s)
{
 for (Iterator<T>i(s); i; i++)
 cout << (*i) << endl;
```

Большим преимуществом в рассматриваемом случае является то, что нам не нужно знать, какая разновидность класса *Set* используется. Это означает, что программист сам может решать, какой тип *Set* лучше подходит для каждого конкретного случая, а принятое решение не влияет ни на одну клиентскую функцию. Программист может предпочесть множество *UnboundedSet* в случае, когда объем доступной памяти ограничен, а скорость работы системы не слишком важна, или выбрать *BoundedSet*, когда памяти достаточно, а быстродействие системы “на высоте”. Клиентские функции будут манипулировать этими объектами через интерфейс базового класса *Set*, поэтому выбор одного из двух множеств не будет иметь значения.

## Проблема

Теперь добавим в существующую иерархию множество *PersistentSet*. Постоянное (*persistent*) множество — это множество, которое может быть записано в поток данных, а затем считано оттуда позже (возможно, другим приложением). К сожалению, возможность создания такого множества предоставлял только “контейнер” от стороннего производителя, к которому у меня был доступ, но он не был шаблонным классом. Вместо этого он мог принимать объекты, которые являлись производными от абстрактного базового класса *PersistentObject*. В результате была создана иерархия, показанная на рис. 10.3.

Обратите внимание, что *PersistentSet* содержит копию множества от стороннего производителя, которой и делегируются все методы объекта. Таким образом, если вы обратитесь к функции *Add* объекта *PersistentSet*, вызов бу-



**Рис. 10.3. Постоянный набор иерархий**

дет делегирован соответствующему элементу из множества от стороннего производителя.

На первый взгляд все кажется правильным и рациональным. Однако подобный подход имеет и свои минусы. Элементы, добавляемые в множество от независимого производителя, должны представлять собой производные от `PersistentObject`. Так как `PersistentSet` просто делегирует свои функции тому же набору (от сторонних изготовителей), то любой элемент, входящий в `PersistentSet`, должен быть производным от `PersistentObject`. Тем не менее, интерфейс класса `Set` не содержит такого ограничения.

Когда клиент добавляет объекты к базовому классу `Set`, он (клиент) не может проверить, является ли член класса `Set` также и членом `PersistentSet`. Таким образом, клиент не знает, представляют ли собой добавляемые им элементы производные от `PersistentObject`.

Рассмотрим код для `PersistentSet::Add()`, представленный в листинге 10.6.

---

#### Листинг 10.6. Шаблон <typename T>

---

```
void PersistentSet::Add(const T& t)
{
 PersistentObject& p =
 dynamic_cast<PersistentObject&>(t);
 itsThirdPartyPersistentSet.Add(p);
}
```

---

Здесь видно, что, если клиент попытается добавить к множеству `PersistentSet` объект, не являющийся производным от класса `PersistentObject`, то программа выдаст сообщение об ошибке. Осуществляется переход с `dynamic_cast` на `bad_cast`. Исключений для существующих клиентов абстрактного базового класса `Set` не предусмотрено. Ведь поскольку данные функции будут смешаны с производными от `Set`, то такое изменение иерархии приведет к нарушению правила подстановки LSP.

Столкнулись ли мы здесь с проблемой? Безусловно. Функции, которые ранее безупречно работали с производными от класса `Set`, могут вызвать ошибки при работе с `PersistentSet`. Устранение проблем подобного типа — довольно сложный процесс, так как ошибки во время работы редко связаны с неправильными логическими решениями. Логическая ошибка может являться следствием решения включить `PersistentSet` в функцию или добавить в `PersistentSet` объект, не являющийся производным от `PersistentObject`. В любом случае реальное решение проблемы может находиться через миллион операторов от вызова процедуры `Add`. Найти такую ошибку нелегко, а исправить еще сложнее.

## Решение, которое не согласуется с принципом LSP

Как можно устраниить эту проблему? Несколько лет назад я решил ее условно. Это означает, что я не воплотил решение в программном коде. Вернее, я создал соглашение, благодаря которому `PersistentSet` и `PersistentObject` не опознавались всем приложением целиком. Они распознавались при помощи отдельного модуля программы. Этот модуль отвечал за чтение и запись всех контейнеров в постоянное хранилище. При необходимости записи контейнера его содержимое копировалось в подходящие производные объекты для `PersistentObject`, а затем добавлялось в `PersistentSets`, копируемый затем в поток данных. При считывании контейнера из потока происходил обратный процесс. Вначале считывался `PersistentSet`, после этого `PersistentObjects` удалялся из `PersistentSet` и копировался в обычные (непостоянные) объекты, которые затем добавлялись в класс `Set`.

Такое решение может показаться слишком “ограничительным”, но это был единственный способ, придуманный мной, чтобы предотвратить появление объектов из `PersistentSet` в функциях, обрабатывающих “непостоянные” объекты. Кроме того, при этом нарушается зависимость остальной части приложения от общего принципа постоянства (способности ПО создавать и поддерживать permanentные объекты).

Работоспособно ли данное решение? Не совсем так, как хотелось бы. В некоторых модулях приложения данное соглашение было нарушено разработчиками, которые считали, что его важность преувеличена. В этом заключается главная проблема подобных соглашений — их нужно постоянно доводить до сведения каждого разработчика. Если же разработчик не ознакомился с соглашением или просто не согласен с ним, то, скорее всего, соглашение будет нарушено. А одного такого случая достаточно для нарушения работы всей структуры.

## Решение, не нарушающее принцип LSP

Как бы автор решил эту проблему сейчас? Очевидно, что `PersistentSet` не связана с `Set` зависимостью типа *IS-A*, иными словами, не является истинным производным классом от `Set`. Поэтому проще частично разделить иерархии. Множества `Set` и `PersistentSet` имеют общие особенности. Фактически только метод `Add` может вызвать конфликты с принципом LSP. Следовательно, имеет смысл создать иерархию, в которой `Set` и `PersistentSet` будут являться элементами одного уровня в абстрактном интерфейсе (см. рис. 10.4.). Такое решение позволило бы тестировать объекты `PersistentSet` на предмет принадлежности ко множеству и возможности использования в рабочих циклах. Причем объекты, не являющиеся производными от `PersistentObject`, нельзя будет добавить в множество `PersistentSet`.

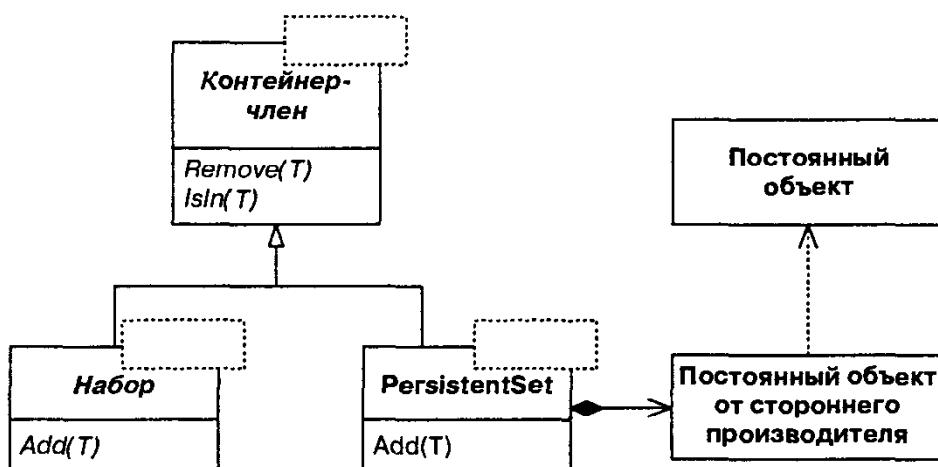


Рис. 10.4. Решение, не нарушающее принцип LSP

## Факторинг вместо вычисления производных классов

Другим интересным случаем наследования является работа с линией и ее сегментами (`Line` и `LineSegment`)<sup>9</sup>. Рассмотрим код из листингов 10.7 и 10.8. Каждая переменная и функция из `LineSegment` должна быть объявлена в `Line`. Кроме того, в `LineSegment` добавлена новая функция `GetLength` и перекрывается значение функции `isOn`. Но даже эти два класса нарушают правило LSP, хотя и не так явно, как в изложенных выше случаях.

---

### Листинг 10.7. `geometry/line.h`

---

```

#ifndef GEOMETRY_LINE_H
#define GEOMETRY_LINE_H
#include "geometry/point.h"

class Line
{
public:
 Line(const Point& p1, const Point& p2);

 double GetSlope() const;
 double GetIntercept() const; // Y Intercept
 Point GetP1() const {return itsP1;};
 Point GetP2() const {return itsP2;};
 virtual bool IsOn(const Point&) const;

private:
 Point itsP1;
 Point itsP2;
}

```

<sup>9</sup> Несмотря на схожесть этого примера с примером “квадрата/прямоугольника”, в данном случае мы имеем дело с реальным приложением, которое явилось “плодом” дискуссии, связанной с реальной проблемой.

```
};
#endif
```

---

**Листинг 10.8. geometry/lineseg.h**

---

```
#ifndef GEOMETRY_LINESEGMENT_H
#define GEOMETRY_LINESEGMENT_H
class LineSegment : public Line
{
public:
 LineSegment(const Point& p1, const Point& p2);
 double GetLength() const;
 virtual bool IsOn(const Points) const;
};
#endif
```

---

Пользователь `Line` вправе ожидать, что здесь содержатся все “точки пересечения” (коллинеарные точки). Например, в точке, возвращаемой функцией `Intercept`, линия пересекается с осью координат Y. Так как эта точка коллинеарна линии, пользователи `Line` вправе ожидать, что `IsOn(intercept()) == true`. Однако во многих случаях (при различных значениях `LineSegment`) данное утверждение не является верным.

Почему этому придается такое большое значение? Почему бы просто не вывести (дифференцировать) `LineSegment` из `Line` и не обращать внимания на мелкие неувязки? Ответственность за подобные решения целиком ложится на разработчиков. В некоторых *редких* случаях целесообразнее закрыть глаза на мелкие недочеты в полиморфной структуре, чем пытаться перестроить всю структуру так, чтобы безуказненно соблюдался принцип LSP. Здесь мы имеем дело с типичным случаем “инженерного компромисса”. Хороший инженер должен четко знать, когда компромисс *выгоднее*, чем совершенствование программного кода. Однако не стоит все время с легкостью жертвовать согласованностью с LSP во имя упрощения или удешевления процесса проектирования. Гарантия того, что производный класс всегда будет работать там, где использовались базовые классы — лучший способ справиться с возрастающей сложностью кода. Если данное соответствие не выполняется, нужно рассматривать каждый подкласс отдельно.

При работе с `Line` и `LineSegment` существует простое решение, которое иллюстрирует один из важных инструментов объектно-ориентированного проектирования (ООП). Если у нас есть доступ к обоим классам — `Line` и `LineSegment`, то мы можем выделить их общие элементы в абстрактный базовый класс. Листинги 10.9, 10.10 и 10.11 демонстрируют выделение из `Line` и `LineSegment` базового класса `LinearObject`.

---

**Листинг 10.9. geometry/linearobj.h**

---

```
#ifndef GEOMETRY_LINEAR_OBJECT_H
#define GEOMETRY_LINEAR_OBJECT_H

#include "geometry/point.h"

class LinearObject
{
public:
 LinearObject(const Point& p1, const Point& p2);
 double GetSlope() const;
 double GetIntercept() const;

 Point GetP1() const {return itsP1;};
 Point GetP2() const {return itsP2;};
 virtual int IsOn(const Point&) const =0; // абстрактный класс

private:
 Point itsP1;
 Point itsP2;
};

#endif
```

---



---

**Листинг 10.10. geometry/line.h**

---

```
#ifndef GEOMETRY_LINE_H
#define GEOMETRY_LINE_H
#include "geometry/linearobj.h"

class Line : public LinearObject
{
public:
 Line(const Point& p1, const Point& p2);
 virtual bool IsOn(const Point&) const;
};

#endif
```

---



---

**Листинг 10.11. geometry/lineseg.h**

---

```
#ifndef GEOMETRY_LINESEGMENT_H
#define GEOMETRY_LINESEGMENT_H
#include "geometry/linearobj.h"

class LineSegment : public LinearObject
{
public:
 LineSegment(const Point& p1, const Point& p2);
 double GetLength() const;
```

---

```

 virtual bool IsOn(const Point&) const;
};

#endif

```

---

Класс `LinearObject` представляет как `Line`, так и `LineSegment`. Этот класс обеспечивает практически полный набор функций и данных обоих подклассов, за исключением процедуры `IsOn`, которая является чисто виртуальной. Пользователи `LinearObject` не могут оперировать экстентом (*непрерывная область, резервируемая для определенного набора данных*) объекта, с которым они работают. Таким образом, они вполне могут принимать объект `Line` или `LineSegment`. Кроме того, пользователи `Line` не имеют дела с `LineSegment`.

Факторинг — это один из инструментов разработки, который лучше использовать до того, как написана большая часть программного кода. Безусловно, при работе с несколькими десятками клиентов класса `Line` (см. листинг 10.7) будет непросто заниматься факторингом класса `LinearObject`. Однако при наличии благоприятных условий факторинг является мощным инструментом. Если объекты могут быть разделены на два подкласса, производные классы найдут применение в дальнейшем. По поводу факторинга Ребекка Вирфс-Брок (Rebecca Wirfs-Brock), Брайан Вилкерсон (Brian Wilkerson) и Лорен Вайнер (Lauren Wiener) говорят следующее:

*“Можно смело утверждать, что если все классы из определенного набора обладают общим свойством, то они могут наследовать это свойство от общего для них суперкласса (базового класса).*

*Если общий базовый класс еще не существует, создайте его и переместите туда общие свойства других классов. Такой класс будет очень полезен с точки зрения наглядности структуры наследования. Вполне вероятна ситуация, когда при дальнейшем расширении системы появится новый подкласс, который будет обладать такими же свойствами. Он может быть и абстрактным”<sup>10</sup>.*

В листинге 10.12 показано, как атрибуты `LinearObject` могут использоваться новым, непредусмотренным ранее классом под названием `Ray`. Именно `Ray` является заменяемым для `LinearObject`, любой пользователь класса `LinearObject` может с ним работать.

---

#### Листинг 10.12. `geometry/ray.h`

```

#ifndef GEOMETRY_RAY_H
#define GEOMETRY_RAY_H

class Ray : public LinearObject
{

```

---

<sup>10</sup>[WirfsBrock90], с. 113.

---

```

public:
 Ray(const Point& p1, const Point& p2);
 virtual bool IsOn(const Point&) const;
};

#endif

```

---

## Эвристика и соглашения

Существует несколько простых эвристических механизмов, способных помочь в поиске нарушений принципа LSP. Все они работают с производными классами, которые каким-либо образом *удалили* свойства из собственных базовых классов. Такие производные классы (обладающие более низкой функциональностью, чем базовые) обычно не являются подставляемыми и, следовательно, нарушают правило LSP.

## Вырожденные функции в производных классах

Рассмотрим листинг 10.13. Функция `f` принадлежит базовому классу. Однако в классе `Derived` она является вырожденной. По-видимому, автор класса `Derived` обнаружил, что функция `f` не несет никакой полезной нагрузки в `Derived`. К сожалению, пользователи класса `Base` не знают, что они не могут вызывать данную функцию, поскольку это приведет к нарушению принципа подстановки.

---

**Листинг 10.13. Вырожденная функция в производном классе**

---

```

public class Base
{
 public void f() /*some code*/
}

public class Derived extends Base
{
 public void f() {}
}

```

---

Наличие вырожденных функций в производных классах не всегда ведет к нарушениям правила LSP, однако стоит относиться к ним с пристальным вниманием.

## Удаление исключений из производных классов

Еще одна форма нарушений — добавление исключений к методам производных классов в случае, если базовые классы не поддерживают их. Если пользователи базовых классов не предполагают наличия таких исключений, то они добавляют объекты, которые не могут быть подставляемыми. Поэтому должны

учитываться ожидания пользователей или же в производные классы не должны вноситься исключения.

## Резюме

Принцип открытия-закрытия (OCP) — один из краеугольных камней объектно-ориентированного программирования. Приложения, в которых он используется, являются более “ремонтопригодными” и устойчивыми в работе, а также допускают использование отдельных элементов в других программах. Принцип LSP — один из ведущих принципов OCP. Взаимозаменяемость подтипов позволяет расширять функциональные возможности модуля, основанного на базовом типе, не прибегая к его модификации. Разработчики должны доверять этому принципу без малейших колебаний. Таким образом, контракт базового типа не должен содержать ошибок и быть, по возможности, интуитивно понятным, если все его элементы сразу не содержатся в программном коде.

Толкование термина “IS-A” является слишком широким, чтобы он мог служить четким признаком подтипа. Правильным определением подтипа является “заменяемость” или “подставимость”, которая выступает в качестве явного или неявного ограничения.

## Литература

1. Meyer B. *Object-Oriented Software Construction*, 2d ed. Upper Saddle River, NJ: Prentice Hall, 1997.
2. Wirfs-Brock R. *Designing Object-oriented Software*. Englewood Cliffs, NJ: Prentice Hall, 1990.
3. Liskov B. Data Abstraction and Hierarchy. *SIGPLAN Notices*, 1988.

# 11

## DIP: принцип инверсии зависимостей



© Jennifer M. Kohnke

Никогда впредь интерес державный не должен зависеть от тысяч непредвиденных случайностей, от слабостей человеческих.

---

Сэр Томас Нун Тэлфорд

### Принцип DIP: принцип инверсии зависимостей

*Модули высокого уровня не должны зависеть от модулей низкого уровня. Оба типа модулей обязаны зависеть от абстракций. Абстракции не должны зависеть от подробностей. Подробностям следует зависеть от абстракций.*

Очень многих читателей интересует, почему в названии данного принципа проектирования используется слово “инверсия”. Причина в том, что более традиционные методы разработки ПО, такие как структурированный анализ и проектирование, обычно применяются для создания программных структур, в которых наблюдается зависимость модулей высокого уровня от модулей низкого уровня. Политика применения подобных методов зависит от имеющихся деталей. Более того, эти методы служат для определения подпрограммной иерархии, уточняющей принципы обращения модулей высокого уровня к модулям низкого уровня. Примером подобной иерархической структуры служит программа Сору, схематически изображенная на рис. 7.1 (см. главу 7). Структура зависимостей хорошо разработанной, объектно-ориентированной программы “инвертирована” по отношению к структуре зависимости, принятой в традиционных процедурных методах.

Рассмотрим зависимость модулей высокого уровня от низкоуровневых модулей. Именно модули высокого уровня включают важные политические решения и бизнес-модели приложения. Они описывают суть приложения. Но до тех пор, пока эти модули зависят от модулей низкого уровня, изменения последних могут непосредственно влиять на функционирование модулей более высокого уровня.

В итоге формируется пресловутый “замкнутый круг”! Существуют модули высокого уровня, устанавливающие определенные политики, но на них влияют в целом детали модули низкого уровня. Модули, содержащие описания бизнес-правил высокого уровня, должны иметь определенную независимость и приоритет перед модулями, включающими детали. Модули высокого уровня ни в коей мере не должны зависеть от модулей низкого уровня.

Более того, к высокоуровневым модулям, определяющим основные положения политик, приходится обращаться повторно. Модули низкого уровня достаточно удобно использовать повторно в виде библиотек подпрограмм. Если модули высокого уровня зависят от модулей низкого уровня, довольно сложно использовать высокоуровневые модули в различных контекстах. Но если модули высокого уровня не зависят от модулей низкого уровня, высокоуровневые модули легко применять повторно. Этот принципложен в основу схематического проектирования (framework design).

## Разбиение на слои

Согласно мнению Буча, “... любые хорошо структурированные объектно-ориентированные архитектуры имеют четко определенные слои, каждый из которых поддерживает некоторый компактный набор служб с помощью хорошо определенного и контролируемого интерфейса”<sup>1</sup>. Простая интерпретация этого утвержде-

<sup>1</sup>[Booch96], с. 54.

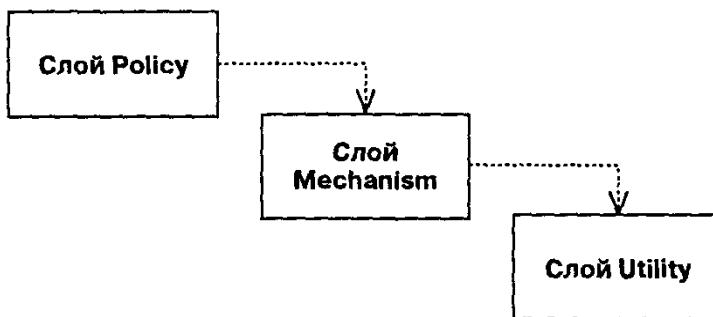


Рис. 11.1. Примитивная схема использования слоев

ния приводит к схеме, показанной на рис. 11.1. На этой диаграмме слой высокого уровня *Policy* использует слой низкого уровня *Mechanism*, который в свою очередь задействует слой на уровне деталей *Utility*. Сожалением приходится констатировать, что слой *Policy* сильно реагирует на любые изменения слоя *Utility*. *Свойство зависимости обладает транзитивностью*. Слой *Policy* зависит от некоторых понятий, относящихся к слою *Utility*; поэтому слой *Policy* транзитивно зависит от слоя *Utility*. Это довольно печально.

На рис. 11.2 показана более совершенная модель. Каждый слой верхнего уровня объявляет абстрактный интерфейс для необходимых служб. Затем на основе этих абстрактных интерфейсов реализуются слои нижних уровней. Каждый класс

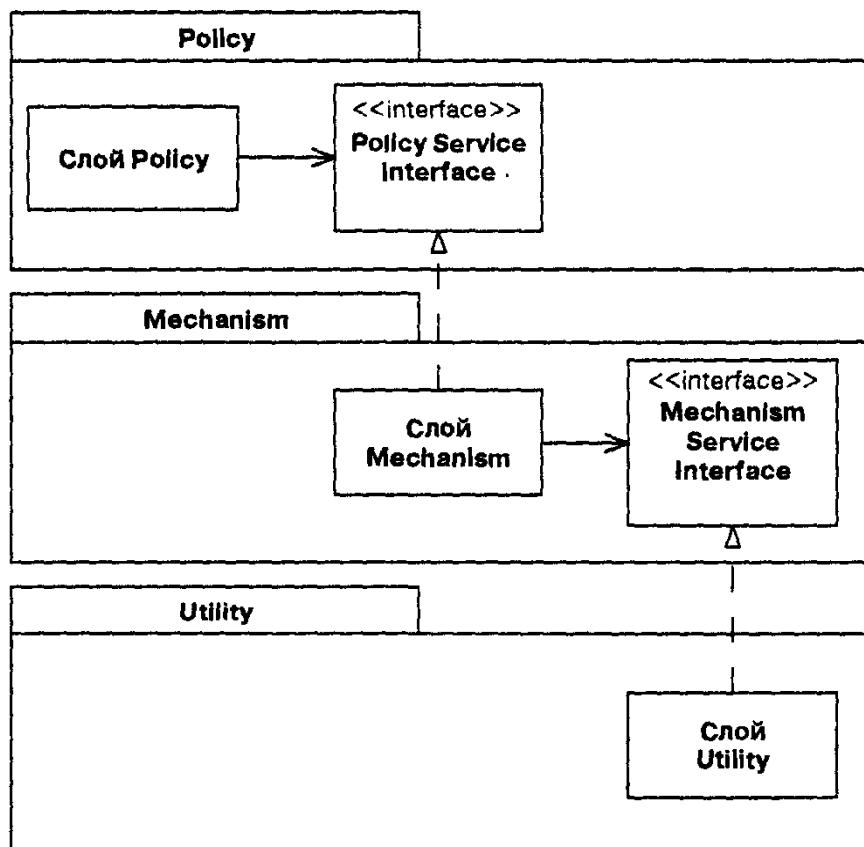


Рис. 11.2. Инвертированные слои

более высокого уровня с помощью абстрактного интерфейса использует следующий нижайший уровень. Таким образом, слои верхнего уровня не зависят от слоев низкого уровня. Вместо этого слои, расположенные ниже, зависят от абстрактных служебных интерфейсов, *объявленных* в верхних слоях. Нарушается не только транзитивная зависимость `PolicyLayer` от `UtilityLayer`, но также и непосредственная зависимость `PolicyLayer` от `MechanismLayer`.

## Инверсия отношений собственности

Обратите внимание, что инверсия затрагивает не только отношения зависимости, но также и отношения собственности для интерфейсов. Можно представлять, что библиотеки утилит содержат собственные интерфейсы. Применяя принцип DIP, заметим, что клиенты располагают абстрактными интерфейсами, а серверы просто их используют.

Здесь напрашивается аналогия с известным принципом Голливуда: “Не надо обращаться к нам, мы сами вас найдем”<sup>2</sup>. Модули нижнего уровня поддерживают реализацию для объявленных интерфейсов и обращаются к модулям более высоких уровней.

Используя инверсию собственности, `PolicyLayer` не затрагивается изменениями на слоях `MechanismLayer` или `UtilityLayer`. Более того, `PolicyLayer` может повторно применяться в любом контексте, определяющем модули нижних уровней в соответствии с `PolicyServiceInterface`. Таким образом, при инвертировании зависимостей получаем более гибкую, надежную и переносимую структуру.

## Зависимость от абстракций

В упрощенной форме суть принципа DIP можно интерпретировать с помощью эвристического утверждения о “зависимости от абстракций”. Проще говоря, рекомендуется не использовать зависимость от статичного класса — все взаимоотношения в программе поддерживаются с помощью абстрактного класса или интерфейса. В соответствии с этим утверждением получим ряд выводов:

- ни одна переменная не должна содержать указатель или ссылку на статичный класс;
- ни один класс не должен порождаться статичным классом;
- ни один метод не должен отвергать метода реализации любого из базовых классов.

Конечно, это эвристическое утверждение хотя бы раз нарушается в любой программе. Необходимо создавать экземпляры статичных классов, и какие-ли-

<sup>2</sup>[Swee' 5].

бо модули оказываются связанными с ними<sup>3</sup>. Кроме того, нет причин, чтобы не следовать этому эвристическому утверждению при работе со статичными (неизменяемыми) классами. Если в статичный класс редко вносятся изменения, и он не порождает аналогичных образований, вполне можно доверять приведенному утверждению.

Например, во многих системах именно статичный класс описывает строки. Для Java, например, идет речь о статичном классе `String`. В этот класс редко вносятся изменения. Поэтому можно устанавливать зависимость непосредственно от этого класса.

Но большинство статичных классов, *создаваемых* как часть прикладной программы, являются все же изменяемыми. И непосредственная зависимость от этих статичных классов весьма нежелательна. Непостоянство подобных классов можно изолировать, отделив их с помощью абстрактного интерфейса.

Данное решение не является исчерпывающим. Случается, что в интерфейс непостоянного класса следует вносить изменения, и эти изменения должны распространяться на абстрактный интерфейс, представляющий класс. Подобные изменения нарушают изоляцию абстрактного интерфейса.

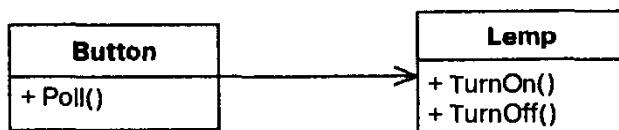
Именно в этом и проявляется наивность приведенного выше эвристического утверждения. С другой стороны, если обратить внимание, что клиентские классы объявляют необходимые им сервисные интерфейсы, можно заметить, что интерфейс изменяется только в случае изменения *клиента*. Изменения же в классах, реализующих абстрактный интерфейс, не оказывают влияния на клиента.

## Простой пример

Инверсия зависимостей проявляется всякий раз, когда один класс направляет сообщение другому. Например, рассмотрим взаимоотношения объектов `Button` и `Lamp`.

Объект `Button` воспринимает воздействие со стороны внешней среды. После получения сообщения `Poll` этот объект определяет, “нажал” ли соответствующую кнопку пользователь. Не имеет значения механизм связи с внешней средой. Пиктограмма кнопки может относиться к графическому интерфейсу пользователя, воздействие на реальную кнопку осуществляется путем нажима пользователем, а движение объекта, связанного с кнопкой, — отслеживаться домашней системой безопасности. Объект `Button` уточняет, активизировал ли пользователь в данный момент кнопку.

<sup>3</sup>Фактически, при использовании строк в процессе создания классов можно обойти это ограничение. Методы Java и других языков программирования позволяют поступать именно так. В таких языках названия статичных классов можно передавать в программу в виде конфигурационных данных.



**Рис. 11.3.** Примитивная модель связи между объектами Button и Lamp

На объект Lamp оказывает влияние внешняя среда. После получения сообщения `TurnOn` зажигается свет. При получении сообщения `TurnOff` — свет гаснет. При этом не важен физический механизм произведенных действий. В качестве источника света может использоваться светодиод на консоли компьютера, ртутная лампа на автостоянке или лазер в лазерном принтере.

Каким образом сконструировать систему, в которой объект `Button` управляет объектом `Lamp`? На рис. 11.3 приводится соответствующая примитивная схема. Объект `Button` получает сообщения `Poll`, уточняет, не нажата ли кнопка, а затем просто пересыпает объекту `Lamp` сообщение `TurnOn` или `TurnOff`.

В чем же состоит примитивизм подобной схемы? Обратимся к коду Java, использующему эту модель (листинг 11.1). Обратите внимание, что класс `Button` непосредственно зависит от класса `Lamp`. Вследствие этой зависимости на класс `Button` оказывают влияние изменения в классе `Lamp`. Более того, отсутствует возможность повторного использования `Button` для контроля над объектом `Motor`. При данной схеме разработки объекты `Button` контролируют объекты `Lamp` и только объекты `Lamp`.

---

#### Листинг 11.1. Button.java

---

```

public class Button
{
 private Lamp itsLamp;
 public void poll()
 {
 if /*условие*/
 itsLamp.turnOn();
 }
}

```

---

Приведенное решение нарушает основополагающие условия, накладываемые принципом DIP. Высокоуровневая политика приложения не отделена от реализации низкого уровня. Абстракции не отделены от деталей. Без подобного отделения политика высокого уровня автоматически попадает в зависимость от модулей низких уровней, а абстракции — от деталей.

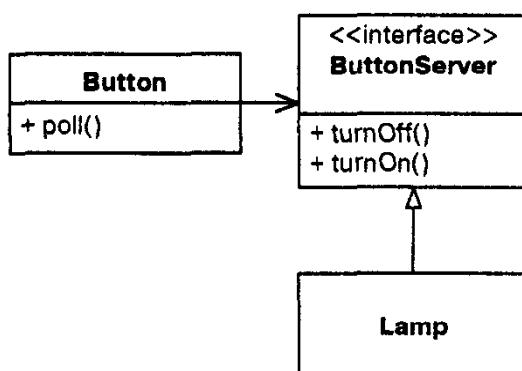
## Поиск основополагающих абстракций

Что же представляет собой политика высокого уровня? Это — абстракция, находящаяся в основе приложения, истина, неизменная при переменах в деталях. Прибегая к метафоре, можно заключить, что речь идет о системе *внутри* системы. В случае с Button/Lamp основополагающая абстракция состоит в проверке пользовательского жеста, ведущего к подключению/отключению, и трансляции этого жеста на целевой объект. Какой механизм проверяет жест пользователя? Не имеет значения! Что собой представляет целевой объект? Не имеет значения! Все это — детали, не оказывающие влияния на абстракцию.

Схема, показанная на рис. 11.3, может быть усовершенствована путем инвертирования зависимости от объекта Lamp. На рис. 11.4 показано, что теперь Button содержит ассоциацию с объектом ButtonServer. ButtonServer поддерживает абстрактные методы, которые может использовать Button для подключения или отключения чего бы то ни было. Lamp реализует интерфейс ButtonServer. Поэтому Lamp теперь формирует зависимость, а не является зависимым.

Схема, изложенная на рис. 11.4, позволяет объекту Button контролировать работу любого устройства, реализующего интерфейс ButtonServer. Этот подход значительно более гибкий, чем рассмотренный ранее. Также объекты Button получают возможность контроля над объектами, которые появятся только в будущем.

Но это решение также накладывает ограничение на любой объект, контролируемый объектом Button. Подобный объект *должен* реализовывать интерфейс ButtonServer. Это ограничение существенно, поскольку данные объекты также могут контролироваться объектом Switch либо некоторым объектом, отличным от Button.



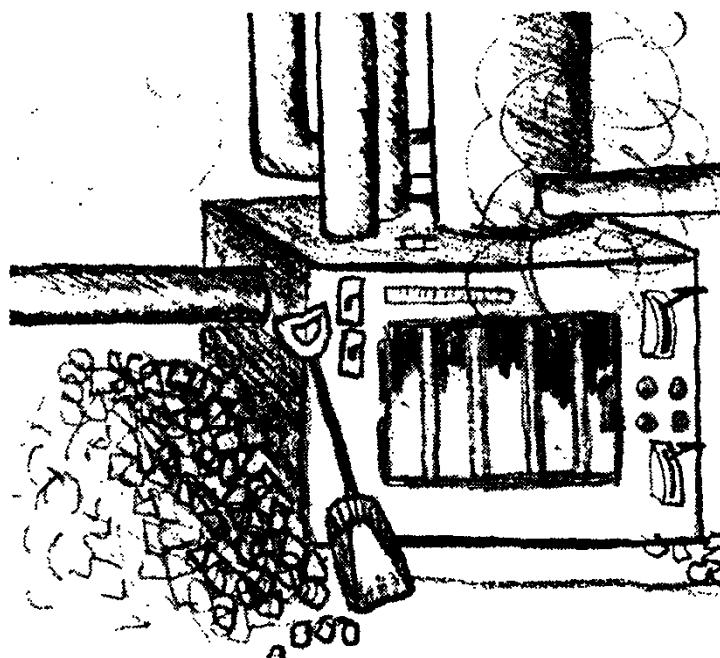
**Рис. 11.4.** Принцип инверсии зависимостей, примененный по отношению к объекту Lamp

Инвертируя направление зависимости и, вовлекая объект Lamp в активную зависимость вместо пассивного зависимого состояния, мы получим тот результат, что Lamp зависит от другой детали — объекта Button. Не так ли?

Несомненно, объект Lamp зависит от ButtonServer, но ButtonServer не зависит от объекта Button. Любой объект, манипулирующий интерфейсом ButtonServer, получает возможность контроля над объектом Lamp. Таким образом, указанная зависимость осталась только в названии. Изменяя название ButtonServer на название более общего характера, типа SwitchableDevice, можно устраниить эту видимую зависимость. Также можно позаботиться о том, чтобы Button и SwitchableDevice хранились в отдельных библиотеках, тогда при использовании SwitchableDevice необязательно обращаться к объекту Button.

В этом случае никто не является собственником интерфейса. Имеем интересную ситуацию, когда интерфейсом может пользоваться большое количество различных клиентов, и данный интерфейс может реализоваться различными серверами. Поэтому интерфейс следует располагать отдельно и не соотносить с какой-либо группой. При работе с C++ подобный интерфейс помещается в отдельное пространство имен (namespace) и библиотеку. В Java он размещается в отдельном пакете<sup>4</sup>.

## Пример с электроплиткой



Обратимся к более интересному примеру. Предположим, что ПО контролирует регулятор степени нагрева электроплитки. Программным образом контролирует-

<sup>4</sup> В динамических языках программирования (Smalltalk, Python или Ruby) этот интерфейс просто не существует в виде четко очерченной сущности источника-кода.

ся текущая температура в каналах ввода/вывода, и инструкция о подключении или отключении регулятора передается другому каналу ввода/вывода. Структура данного алгоритма представлена в листинге 11.2.

---

### Листинг 11.2. Простой алгоритм работы терmostата

---

```
#define THERMOMETER 0x86
#define FURNACE 0x87
#define ENGAGE 1
#define DISENGAGE 0

void Regulate(double minTemp, double maxTemp)
{
 for (;;)
 {
 while (in(THERMOMETER) > minTemp)
 wait(1);
 out (FURNACE, ENGAGE);

 while (in (THERMOMETER) < maxTemp)
 wait(1);
 out(FURNACE, DISENGAGE);
 }
}
```

---

Ясно, что данный алгоритм имеет высокий уровень, но код представлен в виде набора разрозненных деталей низкого уровня. Этот код невозможно повторно использовать при обращении к другой программе.

Если объем кода невелик, подобной ситуацией можно пренебречь. Но даже в этом случае хотелось бы использовать данный алгоритм повторно. Попытаемся инвертировать зависимости, в результате чего получим схему, изображенную на рис. 11.5.

Регуляторная функция имеет два аргумента, которые являются интерфейсами. Показания интерфейса Thermometer можно считывать, а интерфейс Heater можно применять либо не применять в работе. Так полностью записывается алгоритм Regulate. Теперь этот алгоритм имеет вид, показанный в листинге 11.3.

---

### Листинг 11.3. Обобщенный регулятор температуры

---

```
void Regulate(Thermometers t, Heater& h,
 double minTemp, double maxTemp)
{
 for (;;)
 {
 while (t.Read() > minTemp)
 wait (1);
 h.Engage();
```

```

 while (t.Read() < maxTemp)
 wait(1);
 h.Disengage();
 }
 }
}

```

---

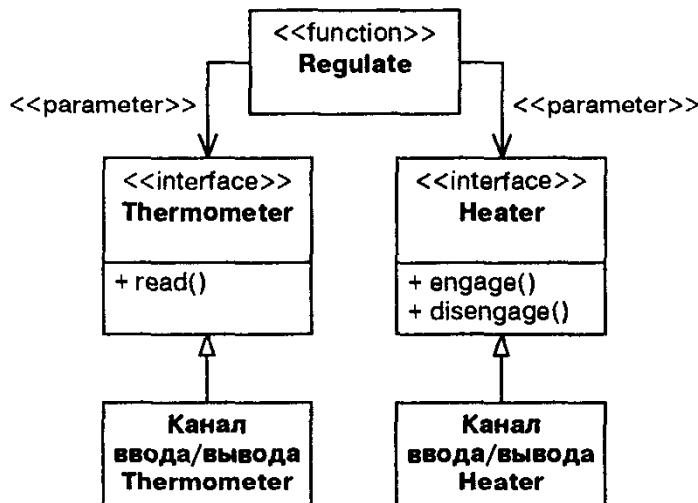


Рис. 11.5. Обобщенный регулятор температуры

Мы получили результат инвертирования зависимостей, теперь высокоуровневая регуляторная политика не зависит от каких-либо определенных деталей, связанных с термометром или электроплиткой. Данный алгоритм легко применяется повторно.

## Динамический и статический полиморфизм

Мы получили инверсию зависимостей, создали обобщенную функцию `Regulate` с помощью динамического полиморфизма (т.е. абстрактных классов и интерфейсов). Но существует и другой путь. Благодаря шаблонам C++ можно воспользоваться статической формой полиморфизма. Обратите внимание на листинг 11.4.

---

### Листинг 11.4. Демонстрация статичной формы полиморфизма

```

template <typename THERMOMETER, typename HEATER>
class Regulate(THERMOMETER& t, HEATER& h,
 double minTemp, double maxTemp)
{
 for (;;)
 {
 while (t.Read() > minTemp)
 wait(1);
 h.Engage();
 }
}

```

---

```

 while (t.Read() < maxTemp)
 wait (1);
 h.Disengage();
 }
}

```

---

Благодаря этому подходу получим аналогичную инверсию зависимостей без использования возможностей динамического полиморфизма. При работе с C++ методы `Read`, `Engage` и `Disengage` могут быть неприменимы. Более того, любой класс, объявляющий эти методы, может реализоваться с помощью шаблона. Эти методы не наследуются из общей базы.

Аналогично шаблону, класс `Regulate` не зависит от определенной реализации этих функций. Требуется только, чтобы класс, заменяемый классом `HEATER`, располагал методами `Engage` и `Disengage`, а класс, заменяемый классом `THERMOMETER`, обладал функцией `Read`. Тогда эти классы реализуют интерфейс, определенный шаблоном. Другими словами, как `Regulate`, так и классы, применяющие `Regulate`, должны использовать один и тот же интерфейс, и обе стороны связаны этим соглашением.

Статический полиморфизм устраняет зависимость “источник-кода”, но не решает массу других проблем, что под силу динамическому полиморфизму. Среди недостатков использования подхода, связанного с шаблонами, следует обязательно отметить следующие:

- типы `HEATER` и `THERMOMETER` нельзя изменить в процессе выполнения программы;
- обращение к новому виду `HEATER` или `THERMOMETER` потребует повторной компиляции и повторного развертывания.

Поэтому, несмотря на убедительное превосходство в скорости при использовании статического полиморфизма, динамический полиморфизм обладает рядом других преимуществ.

## Резюме

Традиционное процедурное программирование формирует структуру зависимостей, при которой политика зависит от деталей. Это весьма неудобно, поскольку политики чутко реагируют на изменения в деталях. Объектно-ориентированное программирование инвертирует эту структуру зависимости, тогда и детали, и политики зависят от абстракции, а сервисные интерфейсы часто оказываются в собственности у клиентов.

Действительно, инверсия зависимостей является признаком хорошо выполненной объектно-ориентированной разработки. И не имеет значения, какой язык программирования при этом используется. Если зависимости инвертированы,

речь идет об объектно-ориентированной разработке. Если зависимости не инвертированы, имеет место процедурная разработка.

Принцип инверсной зависимости является фундаментальным механизмом низкого уровня, находящимся в основе многих выгодных аспектов, отличающих объектно-ориентированную технологию. При создании повторно применимых схем (frameworks) необходимо соответствующим образом применять эту технологию. Также этот подход имеет критическое значение при конструировании гибко трансформируемых кодов. Если абстракции и детали изолированы друг от друга, код значительно легче поддерживать в будущем.

## Литература

1. Booch G. *Object Solutions*. Menlo Park, CA: Addison-Wesley, 1996.
2. Gamma. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
3. Sweet R. E. The Mesa Programming Environment. *SIGPLAN Notices*, 1985.

# 12

## ISP: принцип отделения интерфейса

Этот принцип позволяет преодолевать недостатки, связанные с чрезмерной “толщиной” интерфейсов. Классы, имеющие “тучные” интерфейсы, недостаточно компактны. Эти интерфейсы можно разбить на группы методов, и каждая группа обслуживает иной набор клиентов. Поэтому каждая группа клиентов использует определенную группу членов-функций.

Принцип ISP ориентируется на наличие объектов, нуждающихся в несвязанных интерфейсах; также учитывается, что клиенты не должны представлять их как отдельный класс. Вместо этого клиентам следует иметь представление об абстрактных базовых классах, имеющих связанные интерфейсы.

### Проблемы, возникающие при формировании интерфейса

Рассмотрим систему обеспечения безопасности. В этой системе имеются объекты `Door`, которые могут быть как блокированы, так и открыты (листинг 12.1.)

---

#### Листинг 12.1.

---

```
Security Door
class Door
{
public:
 virtual void Lock() = 0;
 virtual void Unlock() = 0;
 virtual bool IsDoorOpen() = 0;
};
```

---

Данный класс абстрактен, поэтому клиенты могут применять объекты, относящиеся к интерфейсу `Door`, вне зависимости от определенных реализаций `Door`.

Рассмотрим одну из таких реализаций, `TimedDoor`, которая подает звуковой сигнал, если дверь открыта слишком долго. Чтобы выполнить эти действия, объект `TimedDoor` связывается с другим объектом под названием `Timer`. (листинг 12.2.)

---

### Листинг 12.2.

---

```
class Timer
{
public:
 void Register(int timeout, TimerClient* clients);
};

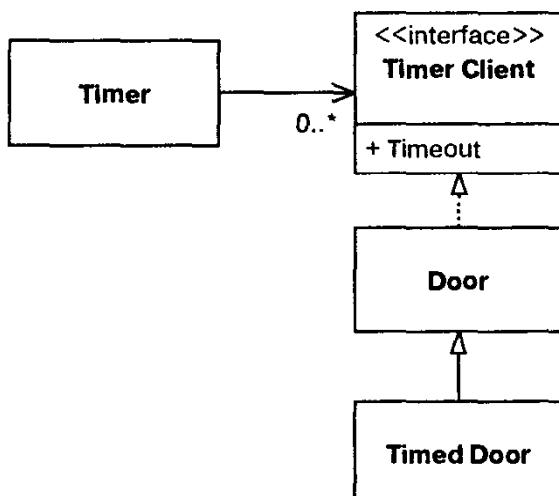
class TimerClient
{
public:
 virtual void TimeOut() = 0;
};
```

---

Если объект желает проинформировать о перерыве, вызывается функция `Register` для `Timer`. Аргументами этой функции являются время перерыва и указатель объекта `TimerClient`, чья функция `TimeOut` и вызывается по завершении перерыва.

Каким образом связать класс `TimerClient` с классом `TimedDoor`, чтобы код в `TimedDoor` уведомлял о наступившем перерыве? Существуют несколько альтернатив. На рис. 12.1 проиллюстрировано примитивное решение. Структура сформирована таким образом, что классы `Door` и `TimedDoor` наследуются из `TimerClient`. Получаем гарантию, что `TimerClient` сможет зарегистрироваться с помощью `Timer` и получить сообщение `TimeOut`.

Несмотря на популярность этого решения, оно проблематично. Основным недостатком этого подхода является зависимость класса `Door` от `TimerClient`.



**Рис. 12.1.** Объект `Timer Client`, находящийся в верхней части иерархии

Далеко не всегда `Door` нуждается в использовании таймера. В действительности, исходная абстракция `Door` не имеет отношения к таймеру. При формировании для `Door` различных нововведений, они по-прежнему будут поддерживать устаревшие реализации метода `TimeOut`, что является потенциальным нарушением принципа LSP. Более того, приложения, применяющие эти нововведения должны импортировать определение класса `TimerClient`, даже в том случае, если он не используется. А это уже чревато проявлениями неоправданной сложности и излишнего повторения.

Рассмотренный пример иллюстрирует неаккуратность при формировании интерфейса, которая является распространенным синдромом в языках статического типа, таких как C++ и Java. Эти особенности интерфейса `Door` проявляются из-за применения неадекватных методов при его создании. Выгодно использовать эту методику для создания одного из подклассов, поэтому она положена в основу формирования всего интерфейса. Если следовать подобной практике, всякий раз, когда нововведение потребует применения нового метода, этот метод будет добавляться в базовый класс. А это приведет к еще большей неаккуратности интерфейса базового класса, усиливая степень его “тучности”.

Более того, всякий раз при добавлении нового метода к базовому классу, этот метод должен реализовываться (или разрешаться по умолчанию) в производных классах. В действительности, обычно эти интерфейсы приобретают вырожденные реализации и добавляются к базовому классу. Тогда производные классы не имеют дополнительной нагрузки по их реализации. Как указывалось ранее, эта практика противоречит принципу LSP, приводит к многочисленным проблемам, связанным с поддержкой и повторным использованием.

## **Отделение клиентов равнозначно изоляции интерфейсов**

Объекты `Door` и `TimerClient` представляют интерфейсы, применяемые совершенно разными клиентами. Объект `Timer` использует интерфейс `TimerClient`, а классы, манипулирующие дверьми, применяют объект `Door`. Поскольку клиенты отделены, интерфейсы также должны быть выделены. Зачем? Клиенты оказывают существенное влияние на используемые интерфейсы.

## **Обратное влияние клиентов на интерфейсы**

Когда речь идет о воздействиях, вызывающих изменения в программах, обычно это относится к тому, каким образом изменения в интерфейсах влияют на пользователей. Например, при внесении изменений в интерфейс `TimerClient` они отразятся на всех пользователях `TimerClient`. Но существует и обратное влияние. Иногда пользователь вносит изменения в интерфейс.

Например, некоторые пользователи `Timer` регистрируют более одного запроса за единицу времени. Рассмотрим объект `TimedDoor`. Если проверка показала, что объект `Door` открыт, объекту `Timer` направляется сообщение `Register`, содержащее запрос о перерыве. Но еще до окончания перерыва дверь закрывается, остается некоторое время закрытой, потом открывается снова. Вследствие этого регистрируется *новый* запрос об интервале еще до завершения первого интервала. Наконец, завершается первый интервал, вызывается функция `Timeout` из `TimedDoor`. Звуковой сигнал `Door` подается ошибочно.

Можно исправить эту ситуацию с помощью соглашения, проиллюстрированного кодом из листинга 12.3. В каждый интервал регистрации включается уникальный код `timeOutId`, и этот код повторяется при вызове `Timeout` для `TimerClient`. Тогда каждое нововведение для `TimerClient` будет поставлено в известность об интервале регистрации, на который следует откликаться.

---

### Листинг 12.3.

---

```
Timer with ID
class Timer
{
public:
 void Register(int timeout,
 int timeOutId,
 TimerClient* client);
};

class TimerClient
{
public:
 virtual void TimeOut(int timeOutId) = 0;
};
```

---

Ясно, что это изменение окажет влияние на всех пользователей `TimerClient`. Это происходит вследствие недостатков, присущих `timeOutId`. Но в схеме, приведенной на рис. 12.1 также существует `Door`, и на всех клиентов `Door` оказывают влияние эти особенности! А это является прямым указанием на наличие признаков вязкости и закрепощенности. Почему же ошибка в `TimerClient` влияет на клиентов тех нововведений `Door`, которые не нуждаются в использовании таймера? Когда изменение в одной части программы оказывает влияние на другие полностью независимые части программы, последствия этого, а также возможные потери, трудно прогнозировать, что резко повышает риск сбоев в работе всей программы.

## ISP: принцип отделения интерфейса

*Клиенты не должны попадать в зависимость от методов, которыми они не пользуются.*

Когда клиенты вынужденно зависят от методов, которыми не пользуются, они становятся субъектами тех изменений, которым подвержены эти методы. В результате между всеми клиентами могут возникать непредсказуемые сопряжения. Другими словами, если клиент находится в зависимости от класса, содержащего неиспользуемые данным клиентом методы, которые *применяются* другими клиентами, то клиент находится под влиянием изменений, вносимых в класс этими клиентами. Желательно избегать подобных сопряжений, поэтому необходимо отделить интерфейсы.

### Интерфейсы классов и объектов

Рассмотрим снова `TimedDoor`. Этот объект располагает двумя отдельными интерфейсами, которые используются двумя клиентами — `Timer` и пользователями `Door`. Оба интерфейса *должны* реализоваться в одном объекте, поскольку манипулируют одними и теми же данными. Каким образом можно согласовать это обстоятельство с ISP? Как можно отделить интерфейсы, если они должны оставаться вместе?

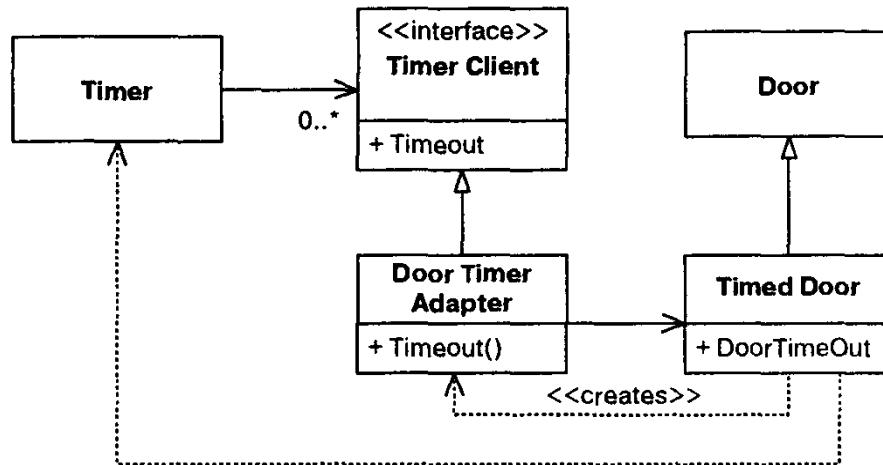
Ответы на эти вопросы можно получить, если учесть, что клиенты не получают к объекту доступ с помощью интерфейса объекта. Вместо этого доступ реализуется путем делегирования или посредством базового класса объекта.

### Отделение путем делегирования

Одно из решений состоит в создании объекта, производного от `TimerClient`, который передает полномочия `TimedDoor`. На рис. 12.2 проиллюстрировано это решение.

Если `TimedDoor` желает зарегистрировать запрос о перерыве с помощью `Timer`, формируется `DoorTimerAdapter`, и запрос регистрируется с помощью `Timer`. Когда `Timer` пересыпает сообщение `TimeOut` к `DoorTimerAdapter`, `DoorTimerAdapter` делегирует сообщение назад, `TimedDoor`.

Это решение согласовано с принципом ISP и препятствует сопряжению клиентов `Door` с `Timer`. Если производятся изменения в `Timer`, демонстрируемые кодом из листинга 12.3, это не оказывает влияния на пользователей `Door`. Более того, `TimedDoor` не должен иметь точно такой же интерфейс, что и `TimerClient`. `DoorTimerAdapter` может *транслировать* интерфейс `TimerClient` в интерфейс `TimedDoor`. Поэтому данное решение носит кардинальный характер. (листинг 12.4.)



**Рис. 12.2.** Класс Door Timer Adapter

#### Листинг 12.4. TimedDoor.cpp

```
class TimeedDoor : public Door
{
public:
 virtual void DoorTimeOut (int timeOutId);
};

class DoorTimerAdapter : public TimerClient
{
public:
 DoorTimerAdapter (TimedDoor& theDoor)
 : itsTimedDoor (theDoor)
 {}

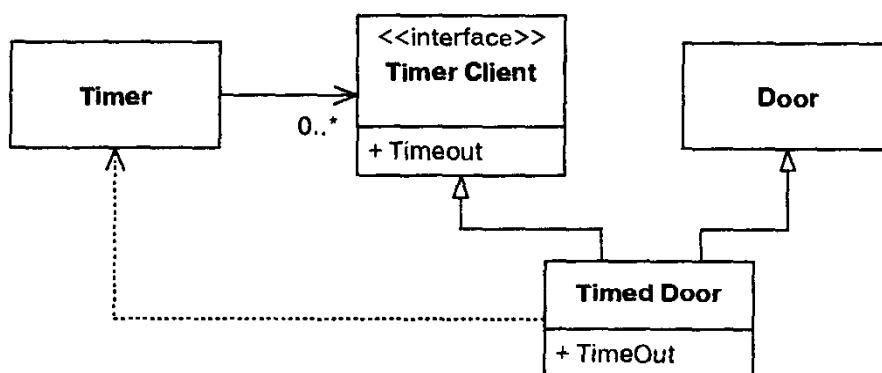
 virtual void TimeOut (int timeOutId)
 {itsTimedDoor.DoorTimeOut (timeOutId);}

private:
 timedDoor& itsTimedDoor;
};
```

Но это решение не является элегантным. Всякий раз, когда необходимо зафиксировать факт перерыва, формируется новый объект. Более того, процесс делегирования требует хоть и очень малого, но не нулевого объема памяти и отнимает время при выполнении программы. Существуют такие приложения, как, например, встроенные системы управления в реальном времени, для которых дефицит времени выполнения и объема памяти имеет определяющее значение.

## **Отделение с помощью множественного наследования**

Рисунок 12.3 и листинг 12.5 демонстрируют, каким образом можно воспользоваться множественным наследованием для соблюдения принципа ISP. В этой модели `TimedDoor` наследует как из `Door`, так и из `TimerClient`. Несмотря



**Рис. 12.3. Множественное наследование для Timed Door**

на то, что клиенты обоих базовых классов могут использовать `TimedDoor`, ни один практически не зависит от класса `TimedDoor`. Поэтому они пользуют один и тот же объект с помощью отдельных интерфейсов.

---

#### Листинг 12.5. `TimedDoor.cpp`

---

```

class TimedDoor : public Door, public TimerClient
{
public:
 virtual void TimeOut(int timeOutId);
};

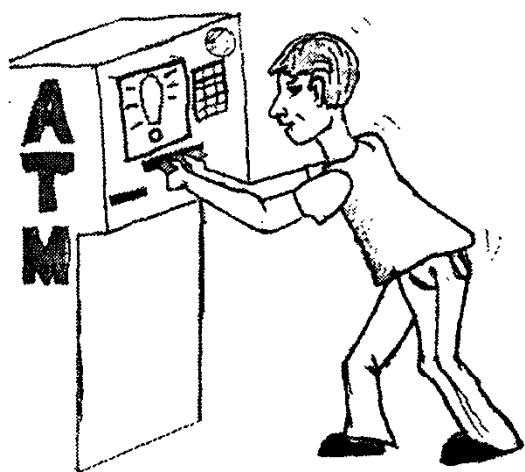
```

---

Именно этому решению отдается предпочтение. Обычно решению, представленному на рис. 12.2 отдают предпочтение перед решением, показанным на рис. 12.3, только в том случае, если с помощью объекта `DoorTimerAdapter` необходимо выполнить трансляцию либо в разное время требуются различные трансляции.

## Пример пользовательского интерфейса: ATM

Теперь рассмотрим более интересный пример. Проблема связана с использованием традиционного банковского автомата (ATM, Automated teller machine). Пользовательский интерфейс для ATM должен быть довольно гибким. Выходные значения транслируются на многих языках. Они могут представляться на экране либо с помощью системы чтения Брайля, либо озвучиваться синтезатором. Ясно, что для этого необходим абстрактный базовый класс, располагающий абстрактными же методами для всех видов сообщений, которые могут быть представлены данным интерфейсом.



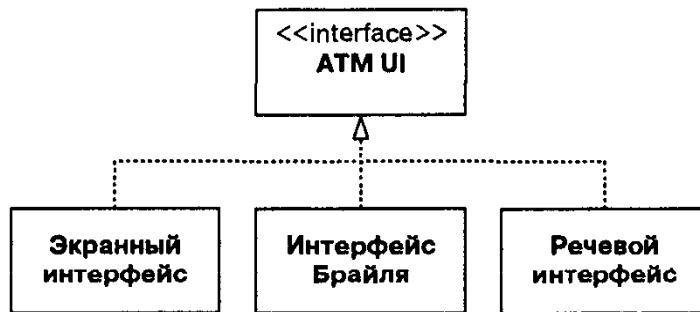


Рис. 12.4. Схема интерфейса: ATM

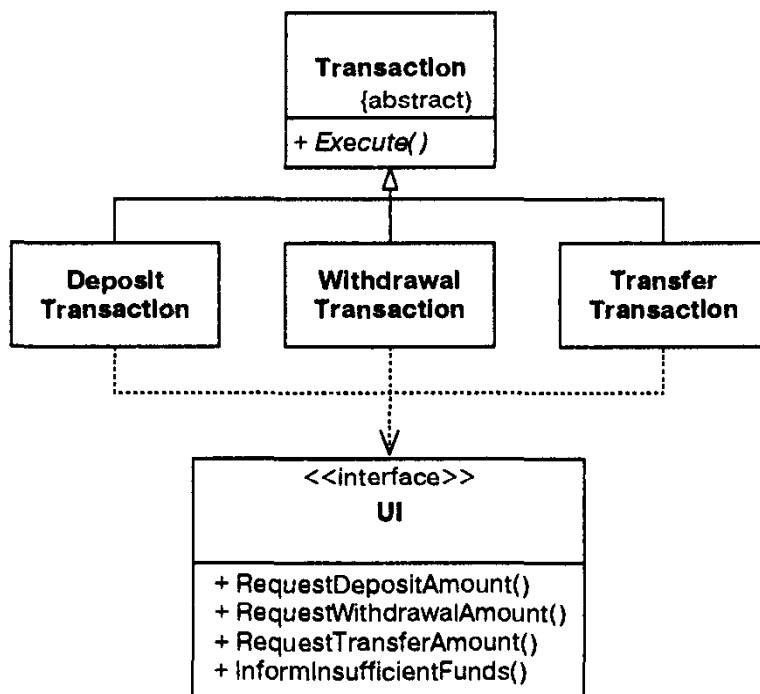


Рис. 12.5. Иерархия ATM-транзакций

Также предположим, что каждая транзакция, выполняемая ATM, инкапсулируется в качестве производного объекта для класса Transaction. Поэтому можно получить классы типа DepositTransaction, WithdrawalTransaction и TransferTransaction. Каждый класс вызывает методы, относящиеся к классу UI. Например, при запросе пользователя о величине средств, помещаемых на депозит, объект DepositTransaction вызывает метод RequestDepositAmount класса UI. Более того, при запросе пользователя об объемах средств, перемещаемых со счета на счет, объект TransferTransaction вызывает метод RequestTransferAmount из класса UI. Этот процесс отражен на диаграмме рис. 12.5.

Обратите внимание, что именно подобные ситуации, с точки зрения ISP, следует избегать. Каждая транзакция применяет методы из UI, не используемые другими классами. Возникает вероятность, что изменения в одной из производных Transaction приведут к изменениям в UI. Тогда в зоне воздействия окажутся все остальные производные классы Transaction, а также все классы, зави-

сящие от UI интерфейса. Подобное положение дел можно охарактеризовать как закрепощенность и неустойчивость.

Например, если добавить `PayGasBillTransaction`, необходимо внести в UI новые методы, обрабатывающие уникальные сообщения, которые может отображать эта транзакция. К сожалению, поскольку `DepositTransaction`, `WithdrawalTransaction` и `TransferTransaction` зависят от интерфейса, придется выполнить повторную компиляцию. Хуже того, если транзакции развернуты как компоненты отдельных DLL или общедоступных библиотек, тогда и эти компоненты должны разворачиваться повторно, даже при стабильности логического построения. Нет ли тут влияния вязкости?

Подобной нежелательной соптыковки можно избежать при разделении UI интерфейса на отдельные интерфейсы, такие как `DepositUI`, `WithdrawUI` и `TransferUI`. Затем из этих выделенных интерфейсов формируется заключительный интерфейс, UI путем применения множественного наследования. На рис. 12.6 и в листинге 12.6 продемонстрирована эта модель.

При создании нового производного класса от `Transaction` необходимо обращаться к соответствующему базовому классу для абстрактного интерфейса UI. Поэтому должен измениться как UI интерфейс, так и все его производные. Но эти классы не применяются в широком масштабе. В действительности, они вероятнее

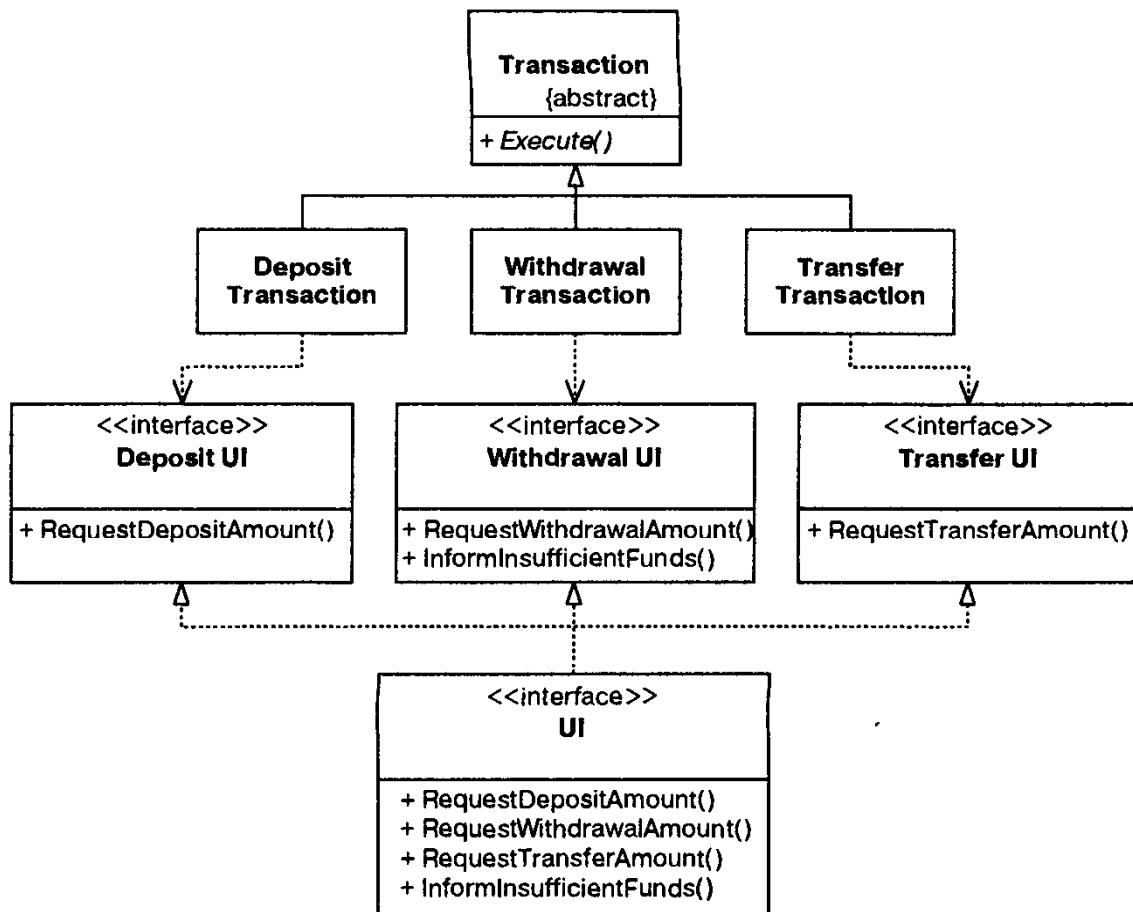


Рис. 12.6. Отделенный интерфейс ATM UI

всего применяются только в main или каком-либо ином процессе, выполняющем загрузку системы и создающем конкретный UI экземпляр. Поэтому минимизировано влияние, возникающее при добавлении новых базовых классов UI.

---

**Листинг 12.6. Отделенный ATM UI интерфейс**

---

```
class DepositUI
{
public:
 virtual void RequestDepositAmount() = 0;
};

class DepositTransaction : public Transaction
{
public:
 DepositTransaction(DepositUI& ui)
 : itsDepositUI(ui)
 {}

 virtual void Execute()
 {
 ...
 itsDepositUI.RequestDepositAmount();
 ...
 }
private:
 DepositUI& itsDepositUI;
};

class WithdrawalUI
{
public:
 virtual void RequestWithdrawalAmount() = 0;
};

class WithdrawalTransaction : public Transaction
{
public:
 WithdrawalTransaction(WithdrawalUI& ui)
 : itsWithdrawalUI(ui)
 {}

 virtual void Execute()
 {
 ...
 itsWithdrawalUI.RequestWithdrawalAmount();
 ...
 }
private:
 WithdrawalUI& itsWithdrawalUI;
```

```

class TransferUI
{
public:
 virtual void RequestTransferAmount() = 0;
};

class TransferTransaction : public Transaction
{
public:
 TransferTransaction(TransferUI& ui)
 : itsTransferUI(ui)
 {}

 virtual void Execute()
 {
 ...
 itsTransferUI.RequestTransferAmount();
 ...
 }
private:
 TransferUI& itsTransferUI;
};

class UI : public DepositUI
 , public WithdrawalUI
 , public TransferUI
{
public:
 virtual void RequestDepositAmount ();
 virtual void RequestWithdrawalAmount();
 virtual void RequestTransferAmount();
};

```

При внимательном изучении кода из листинга 12.6 можно заметить один из моментов использования принципа ISP, который не столь очевиден в примере *TimedDoor*. Заметим, что каждая транзакция должна каким-либо образом составить представление об определенной версии UI. *DepositTransaction* должна представить содержание *DepositUI*, *WithdrawTransaction* – *WithdrawUI* и т.д. В листинге 12.6, этот момент получил отражение путем размещения в каждой транзакции ссылки на определенный UI. Заметим, что благодаря этому в листинге 12.7 можно воспользоваться идиомой.

---

#### Листинг 12.7. Идиома инициализации интерфейса

---

```

UI Gui; // глобальный объект;

void f()
{
 DepositTransaction dt(Gui);

```

Этот подход удобен лишь отчасти, поскольку каждая транзакция должна содержать ссылку на соответствующий интерфейс пользователя. В целях адресации также можно создать набор глобальных констант, как показано в листинге 12.8. Использование глобальных переменных не всегда свидетельствует о “бедности” разработки. В данном случае именно глобальные переменные позволяют облегчить доступ. Поскольку они являются ссылками, изменить их невозможно. Поэтому, ими нельзя манипулировать, что и должно радовать других пользователей.

#### Листинг 12.8. Отделенные глобальные указатели

```
// в определенном модуле, связанном с
// оставшейся частью app.

static UI Lui; // объект не является глобальным;
DepositUI& GdepositUI = Lui;
WithdrawalUI& GwithdrawalUI = Lui;
TransferUI& GtransferUI = Lui;

// в модуле depositTransaction.h

class WithdrawalTransaction : public Transaction
{
public:

 virtual void Execute()
 {
 ...
 GwithdrawalUI.RequestWithdrawalAmount();
 ...
 };
}
```

При программировании на C++ можно допустить оплошность, размещая все глобальные переменные из листинга 12.8 в отдельный класс, чтобы воспрепятствовать неаккуратному использованию глобального пространства имен. В листинге 12.9 демонстрируется подобный подход. Но следует обратить внимание и на нежелательный эффект. Чтобы применить `UIGlobals`, следует воспользоваться конструкцией `#include ui_globals.h`. Но тогда, в свою очередь, необходимо сделать и такие записи: `#includes depositUI.h`, `withdrawUI.h` и `transferUI.h`. Значит, любой модуль, обращающийся к использованию какого-либо UI-интерфейса, транзитивно зависит и от всех остальных — складывается ситуация, которую следует избегать, согласно указаниям ISP. Если во всех UI-интерфейсах произведены изменения, модули, содержащие `#include "ui_globals .h"`, должны быть компилированы заново. Класс

`UIGlobals` содержит комбинации интерфейсов, которые с таким большим трудом были отделены!

---

#### Листинг 12.9. Оформление `Globals` в виде класса

---

```
// в ui_globals.h

#include "depositUI.h"
#include "withdrawalUI.h"
#include "transferUI.h"

class UIGlobals
{
public:
 static WithdrawalUI& withdrawal;
 static DepositUI& deposit;
 static TransferUI& transfer
};

// в ui_globals.cc

static UI Lui; // объект не является глобальным;
DepositUI& UIGlobals::deposit = Lui;
WithdrawalUI& UIGlobals::withdrawal = Lui;
TransferUI& UIGlobals::transfer = Lui;
```

---

## Поливалентность и моновалентность

Рассмотрим функцию, получающую доступ как к `DepositUI`, так и к `TransferUI`. Также предположим, что этой функции передаются `UI`. Следует ли записывать прототип функции в следующем виде?

```
void g(DepositUI&, TransferUI&);
```

Или же необходимо записать этот прототип таким образом?

```
void g(UI&);
```

Довольно велико стремление записать этот прототип функции в последней (моновалентной) форме. В конце концов, известно, что в прежней (многовалентной) форме оба аргумента ссылаются на *один и тот же объект*. Кроме того, при использовании многовалентной формы обращение примет следующий вид:

```
g(ui, ui);
```

Однако, кажется, что тут что-то не так.

Несмотря на нашу недоверчивость, поливалентная форма часто является предпочтительнее моновалентной. Моновалентная форма свидетельствует о том, что `g` зависит от каждого интерфейса, содержащегося в `UI`. Таким образом, при изменении `WithdrawalUI` как `g`, так и все клиенты `g` будут подвергаться воздействию. А это обстоятельство вызывает еще больше подозрений, чем `g(ui, ui)`! Более

того, нельзя гарантировать, что оба аргумента *с всегда* ссылаются на один и тот же объект! В будущем может так случиться, что объекты по какой-либо причине отделятся. И тот факт, что все интерфейсы объединены в одном объекте, не входит в обязательный объем информации, которым располагает *с*. Поэтому для подобных функций предпочтительнее пользоваться поливалентной формой.

## Группировка клиентов

Часто клиенты группируются вместе в соответствии со служебными методами, к которым они обращаются. Подобная группировка позволяет создавать выделенные интерфейсы для каждой группы, а не для каждого клиента. При этом подходе значительно сокращается количество интерфейсов, реализуемых данной службой, причем в данном случае служба не должна зависеть от каждого типа клиента.

Иногда методы, вызываемые различными группами клиентов, “накладываются” друг на друга. Если “наложение” невелико, тогда интерфейсы этих групп остаются отделенными. Общие функции объявляются во всех частично совпадающих интерфейсах. Серверный класс наследует общие функции из каждого из этих интерфейсов, но реализует их только один раз.

## Внесение изменений в интерфейсы

При поддержке объектно-ориентированных приложений часто изменяются интерфейсы существующих классов и компонентов. Иногда эти изменения имеют большое влияние и приводят к повторному компилированию и развертыванию очень больших частей системы. Эти процессы можно упростить, добавляя к существующим объектам новые интерфейсы без изменения существующего интерфейса. Клиенты “старых” интерфейсов, которым необходим доступ к новому интерфейсу, могут запрашивать объект для этого интерфейса, как показано в листинге 12.10.

---

### Листинг 12.10.

```
void Client(Service* s)
{
 if (NewService* ns = dynamic_cast<NewService*>(s))
 {
 // применение нового служебного интерфейса
 }
}
```

---

В соответствии с изложенными принципами, заботу следует проявлять, не слишком усердствуя при этом. Спектр класса с сотнями различных интерфейсов, некоторые из которых отделены от клиента, а другие — отделены от версии, может приобрести просто устрашающий вид.

## Резюме

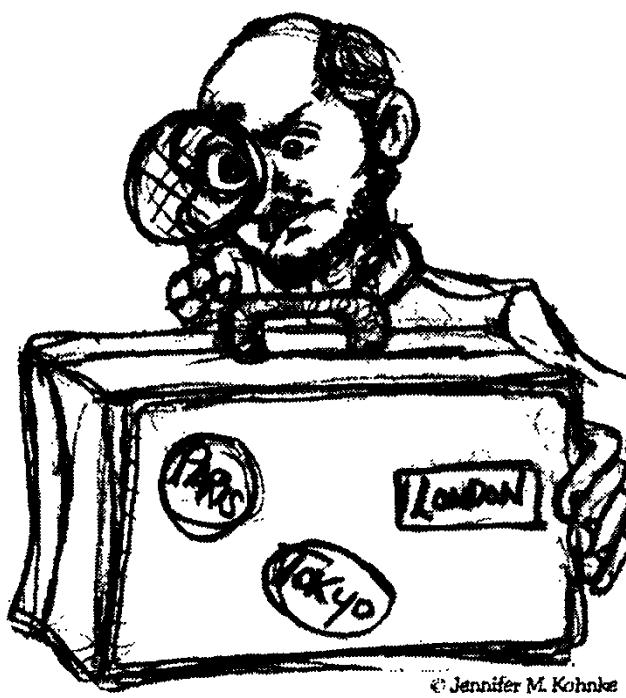
“Тучные” классы могут приводить к причудливым и нежелательным стыковкам между клиентами. Когда один клиент вносит изменение в подобный класс, это оказывает влияние на всех остальных клиентов. Поэтому клиенты должны зависеть от методов, которые реально ими вызываются. Чтобы достичь этого, интерфейс подобных “тучных” классов разбивается на большое количество интерфейсов, специфичных для каждого клиента. Каждый интерфейс, связанный с определенным клиентом, объявляет только функции, которые вызываются данным клиентом или клиентской группой. Затем “тучный” класс может наследовать все интерфейсы, связанные с определенными клиентами, и реализовывать их. Таким образом прерывается зависимость между клиентами и методами, к которым они не обращаются, что позволяет поддерживать независимость клиентов друг от друга.

## Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

# ЧАСТЬ III

## Практическое занятие: программа расчета зарплаты



© Jennifer M. Kohnke

В предыдущих частях книги рассмотрены вопросы, связанные с рабочими практиками и принципами экстремального программирования. Были обговорены основные вопросы, связанные с разработкой программных проектов. Также обсуждались методики тестирования и планирования. Итак, пришло время первого практического занятия.

В следующих нескольких главах речь пойдет о проектировании и практической реализации системы пакетов, выполняющей расчет зарплаты. Элементарная спецификация этой системы будет приведена немного позже. В процессе проектирования и реализации будут применяться несколько проектных шаблонов. К этим шаблонам относятся следующие: *Command*, *Template method*, *Strategy*, *Singleton*, *Null object*, *Factory* и *Facade*. Именно эти шаблоны составляют предмет рассмотрения следующих нескольких глав. В главе 18

рассматривается проектирование и реализация приложения, реализующего расчет зарплаты.

Материал этого практического задания может изучаться несколькими способами.

- Читайте все подряд, начиная с изучения проектных шаблонов, а затем — методов их практического применения в программе, выполняющей расчет зарплаты.
- Если вы уже знакомы с шаблонами и не нуждаетесь в пополнении своих знаний, переходите сразу к главе 18.
- Сначала ознакомьтесь с главой 18, затем вернитесь к чтению глав, в которых описываются используемые вами шаблоны.
- При чтении главы 18 используйте “фрагментарный” подход. Если встречаетесь с незнакомым шаблоном, вернитесь к главе, описывающей этот шаблон, затем снова перейдите к главе 18.
- Конечно, на этом перечень всех правил не исчерпывается. Выберите (или изобретите) стратегию, которая подходит вам наилучшим образом.

## Элементарная спецификация системы расчета зарплаты

Ниже приводятся некоторые соображения, которые следует учитывать при беседе с заказчиком.

Разрабатываемая нами система состоит из базы данных, в которой хранятся записи о работниках компании. В этой базе данных накапливаются вспомогательная информация, например, табели учета отработанного времени. В результате автоматизируется оплата труда каждого сотрудника. При этом требуется начислять корректные суммы заработанных денег, основываясь на повременной системе оплаты или любом другом методе. Также допускается возможность формирования различных отчетов на базе начисленной зарплаты.

- Некоторые сотрудники работают на почасовой основе. Для них используется почасовая ставка, которая вводится в одно из полей записи, соответствующей этому работнику. В качестве источника входных данных применяются ежедневные карточки табельного учета, в которые вносятся записи, включающие сведения о датах и количестве отработанных часов. Ставка за сверхурочные часы в полтора раза превышает обычную ставку. Выплата денег производится по пятницам.
- Некоторые работники получают фиксированную ставку. В этом случае выплата денег производится в последний рабочий день месяца. Величина месячной зарплаты хранится в одном из полей записи для данного работника.

- Некоторые из сотрудников получают комиссионные, вычисленные на основе их окладов. Им приходится заполнять квитанции, в которых они указывают даты и величину оклада. Величина комиссионных указывается в одном из полей записи, соответствующей сотруднику. Все комиссионные выплачиваются по пятницам.
- Сотрудники могут сами указывать метод платежа. Они могут получать платежные чеки, отсылаемые по избранному почтовому адресу, или передавать свои платежные чеки кассиру; также можно сделать так, что платежные чеки будут переводиться на банковский депозит на указанный вами счет.
- Некоторые сотрудники входят в состав определенных объединений. В этом случае соответствующая запись включает поле, в котором определяется величина еженедельного взноса. Причем величина взноса вычисляется на основании размера зарплаты. Помимо этого, от самого объединения может исходить инициатива относительно изменения размера взносов для отдельных членов объединения. Размеры этих взносов определяются еженедельно, а также они должны вычисляться на основе суммы следующей выплаты.
- Приложение, выполняющее расчет зарплаты, запускается по рабочим дням, в результате чего определяется величина зарплаты сотрудников, работавших в этот день. Система сообщает дату, которой закрываются выплаты сотрудникам. Благодаря этому можно вычислить суммы платежей, начиная от даты последней оплаты и завершая указанной датой.

## Упражнение

Перед тем как продолжить работу, следует продумать (прежде всего, для себя) методы проектирования программы по расчету зарплаты. Может потребоваться выполнить наброски некоторых начальных UML-диаграмм. А еще лучше, если выполнить на практике тестирование нескольких практических занятий. Воспользуйтесь методиками и принципами, изученными вами ранее, и попытайтесь создать сбалансированный и “здоровый” проект.

А теперь обратите внимание на варианты использования, краткий обзор которых приводится ниже. Вы можете спокойно пропустить этот материал, поскольку он излагается повторно в следующих главах.

## Вариант использования 1: добавление сведений о новом сотруднике

Факт добавления информации о новом сотруднике подтверждается выполнением транзакции AddEmp. Эта транзакция включает имя сотрудника, адрес, а также присвоенный ему табельный номер. Сама транзакция выступает в трех формах:

```
AddEmp <EmpID> "<name>" "<address>" H <hourly-rate>
AddEmp <EmplD> "<name>" "<address>" S <monthly-salary>
AddEmp <EmplD> "<name>" "<address>" C <monthly-salary> <commission-rate>
```

При этом создается запись о сотруднике, поля которой получают соответствующие значения.

#### Сообщение об ошибке

**An error in the transaction structure (Ошибка в структуре транзакции)**

Если структура транзакции некорректна, выводится сообщение об ошибке, причем какие-либо действия не предпринимаются.

### **Вариант использования 2: удаление записи, соответствующей сотруднику**

Записи о сотрудниках удаляются в процессе выполнения транзакции DelEmp. Эта транзакция имеет следующий вид:

```
DelEmp <EmplD>
```

При выполнении этой транзакции удаляется запись соответствующего работника.

#### Сообщение об ошибке

**Invalid or unknown EmpID (Неизвестный или некорректно указанный табельный номер работника)**

Если поле <EmplD> структурировано неправильно, или указана некорректная запись работника, транзакция выводит сообщение об ошибке, причем не предпринимаются какие-либо другие действия.

### **Вариант использования 3: отсылка карточки табельного учета**

В результате выполнения транзакции TimeCard в системе создается запись, соответствующая карточке табельного учета, которая связывается с записью соответствующего работника.

```
TimeCard <EmplD> <date> <hours>
```

#### Сообщение об ошибке 1

**The selected employee is not hourly (Для выбранного работника не указан табель)**

В этом случае система выводит соответствующее сообщение об ошибке и не предпринимает каких-либо действий в дальнейшем.

**Сообщение об ошибке 2****An error in the transaction structure (Ошибка в структуре транзакции)**

Система выводит соответствующее сообщение об ошибке, не предпринимая при этом каких-либо дальнейших действий.

**Вариант использования 4: отсылка торговой квитанции**

В результате выполнения транзакции SalesReceipt система создает новую запись, соответствующую торговой квитанции, и связывает ее с работником, которому выплачиваются комиссионные.

`SalesReceipt <EmpID> <date> <amount>`

**Сообщение об ошибке 1****The selected employee is not hourly (Выбранному сотруднику не начислены комиссионные)**

В этом случае система выводит на печать соответствующее сообщение об ошибке, не предпринимая при этом каких-либо действий.

**Сообщение об ошибке 2****An error in the transaction structure (Ошибка в структуре транзакции)**

Система выводит соответствующее сообщение об ошибке, не предпринимая при этом каких-либо действий.

**Вариант использования 5: отсылка сообщения о плате за членство в организации**

В результате выполнения этой транзакции система создает запись, связанную с оплатой членства, и связывает ее с кодом определенной организации-получателя платежа.

`ServiceCharge <memberID> <amount>`

**Сообщение об ошибке****Poorly formed transaction (Плохо сформированная транзакция)**

Если транзакция недостаточно хорошо сформирована или `<memberID>` не определяет номер существующей организации, транзакция выводит на печать соответствующее сообщение об ошибке.

**Вариант использования 6: изменение сведений о работнике**

В результате выполнения этой транзакции система изменяет сведения, указанные в записи для соответствующего работника. Существует несколько возможных вариантов этой транзакции.

|                                              |                                                 |
|----------------------------------------------|-------------------------------------------------|
| ChgEmp <EmpID> Name <name>                   | изменение имени работника                       |
| ChgEmp <EmpID> Address <address>             | изменение адреса работника                      |
| ChgEmp <EmpID> Hourly <hourlyRate>           | изменение почасовой выработки                   |
| ChgEmp <EmpID> Salaried <salary>             | изменение величины оклада                       |
| ChgEmp <EmpID> Commissioned <salary> <rate>  | изменение величины комиссии-онных               |
| ChgEmp <EmpID> Hold                          | хранение платежного чека                        |
| ChgEmp <EmpID> Direct <bank> <account>       | направление на депозит                          |
| ChgEmp <EmpID> Mail <address>                | отсылка платежного чека                         |
| ChgEmp <EmpID> Member <memberID> Dues <rate> | назначение членства в организаций для работника |
| ChgEmp <EmpID> NoMember                      | исключение работника из организации             |

### Сообщение об ошибке

#### Transaction Errors (Ошибки транзакции)

Если структура транзакции некорректна или идентификатор <Emp1D> не указывает на реального работника или <member1D> всегда указывает на члена организации, выводится соответствующее сообщение об ошибке, причем не предпринимаются какие-либо действия.

### Вариант использования 7: выполнение программы расчета зарплаты для одного платежного дня

В результате получения транзакции Payday система осуществляет поиск всех работников, оплата труда которых должна быть произведена к указанной дате. Затем определяется величина оплаты, после чего производится оплата в соответствии с выбранным методом платежа.

Payday <date>

# 13

## Шаблоны Command и Active Object



Ни один человек не обладает изначально правом командовать своими ближними.

Дидро

Из всех описанных в последние годы шаблонов проектирования меня больше всего впечатляет структура Command — одна из самых простых и элегантных. Но как мы увидим далее, видимая простота часто бывает обманчивой. Поэтому диапазон применения шаблона Command практически не имеет ограничений.

Простота применения шаблона Command, как показано на рис. 13.1, является почти курьезной. Листинг 13.1 также не является слишком “серьезным”. Может показаться абсурдным, что наш шаблон состоит из интерфейса с одним методом.

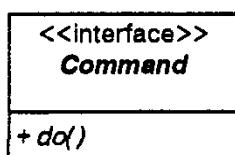


Рис. 13.1. Шаблон Command

### Листинг 13.1. Command.java

```

public interface Command
{
 public void do();
}

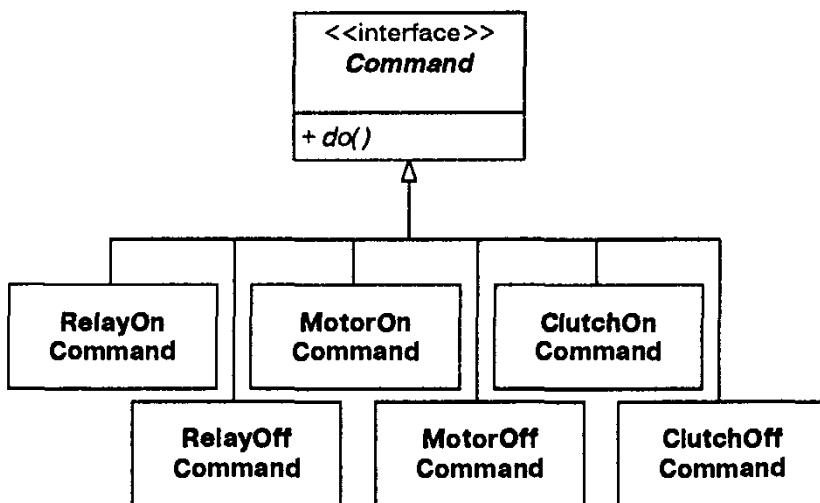
```

Однако на самом деле данный шаблон обладает очень интересной особенностью. Именно в ней заключается его “скрытая” сложность. Большинство классов ассоциируются с набором методов, а также с соответствующим им множеством переменных. Шаблон Command устроен по-другому. Можно сказать, что он включает в себя функции без переменных.

Если строго придерживаться основ объектно-ориентированного программирования, то получается, что такая система приводит к функциональной декомпозиции. Она поднимает значимость и свойства функции до уровня класса. Неслыянно! Однако при столкновении двух этих принципов происходят вещи, заслуживающие пристального внимания и изучения.

## Простые команды

Несколько лет назад я консультировал представителей фирмы, производящей копировальную технику (ксероксы). Я помогал одной из команд разработчиков в проектировании и внедрении встроенной программы, работающей в режиме реального времени, которая была предназначена для нового поколения ксероксов.



**Рис. 13.2.** Некоторые простые команды программы управления ксероксом

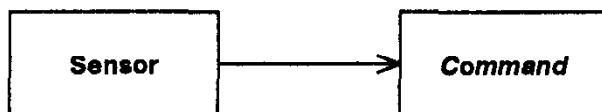
Мы буквально натолкнулись на идею использования шаблона *Command* для контроля аппаратных средств. Была создана иерархия, похожая на ту, что изображена на рис. 13.2.

Роль данных классов очевидна. Когда вы вызываете функцию *do()* из класса *RelayOnCommand*, она включает некоторые реле. При вызове *do()* из *MotorOffCommand* происходит отключение двигателя. Адрес двигателя или реле включается в объект как аргумент.

Имея в своем распоряжении такую структуру, мы можем оперировать объектами *Command* в пределах всей системы и применять к ним функцию *do()*, не задаваясь при этом вопросом какой именно тип объекта *Command* будет использован. Это ведет к дальнейшим любопытным упрощениям.

Данная система управляетя событиями. Реле срабатывали и отключались, двигатели включались и выключались, муфты сцеплялись и расцеплялись в зависимости от определенных событий, происходящих в системе. Многие из этих событий обнаруживаются сенсорами. Например, когда оптический сенсор “видит”, что лист бумаги достиг определенной точки, необходимо включить соответствующий захват. Мы обеспечили это путем простой привязки *ClutchOnCommand* к объекту, который контролировал данный оптический сенсор (рис. 13.3).

Такая простая структура обладает массой преимуществ. Объект *Sensor* “не знает”, что он делает. При обнаружении события он просто вызывает функцию



**Рис. 13.3.** Программное обеспечение сенсорного контроля

`do()` из объекта `Command`, к которому она относится. То есть, все объекты `Sensors` не должны быть связаны со всеми захватами или реле. Им также не требуется информация о структуре и пути листа бумаги в аппарате. Функция сенсора в высшей степени проста.

Все сложности, связанные с идентификацией нужных реле и их связи с сенсорами, делегируются функции инициализации. В определенный момент при инициализации всей системы каждый объект `Sensor` связывается с конкретным объектом `Command`. Это позволяет поместить всю схему *проводки*<sup>1</sup> (*логические связи между сенсорами и "командами"*) в одном месте и выделить ее из общего кода системы. При этом представится возможность создать простой текстовый файл, содержащий описание всех связей между объектами `Sensors` и `Commands`. Программа инициализации будет считывать этот файл, и конфигурировать систему соответствующим образом. Т.е. вся *проводка* системы будет описываться вне общего тела программы и при ее изменении не придется перекомпилировать весь код.

После включения *понятия* команды описанная схема позволяет нам “отделить” логические связи от используемых при этом устройств. Благодаря этому достигаются серьезные преимущества.

## Транзакции

Еще одно применение шаблона `Command`, удобное при работе с программой расчета зарплаты, — создание и выполнение транзакций. Предположим, например, что мы пишем программу, поддерживающую базу данных всех сотрудников компании (рис. 13.4). Существует множество операций, которые пользователи могут выполнять при работе с базой. Они могут добавлять в нее информацию о новых работниках, удалять информацию о старых или изменять атрибуты существующих сотрудников.

Как только пользователь решает добавить запись о новом работнике, он должен указать всю информацию, необходимую для успешного создания новой записи. Прежде чем оперировать полученной информацией, система должна проверить ее на предмет синтаксической и семантической корректности. В этой работе может помочь шаблон `Command`. Объект `Command` в этом случае служит в качестве хранилища непроверенных данных, использует методы их проверки, а затем выполняет транзакцию.

К примеру, обратите внимание на рис. 13.5. Объект `AddEmployeeTransaction` содержит такие же поля данных, как и `Employee`. В нем также присутствует указатель на объект `payClassification`. Эти поля и объект создаются на основе данных, введенных пользователем при добавлении записи о новом сотруднике.

<sup>1</sup>Логические взаимосвязи между `Sensors` и `Commands`.

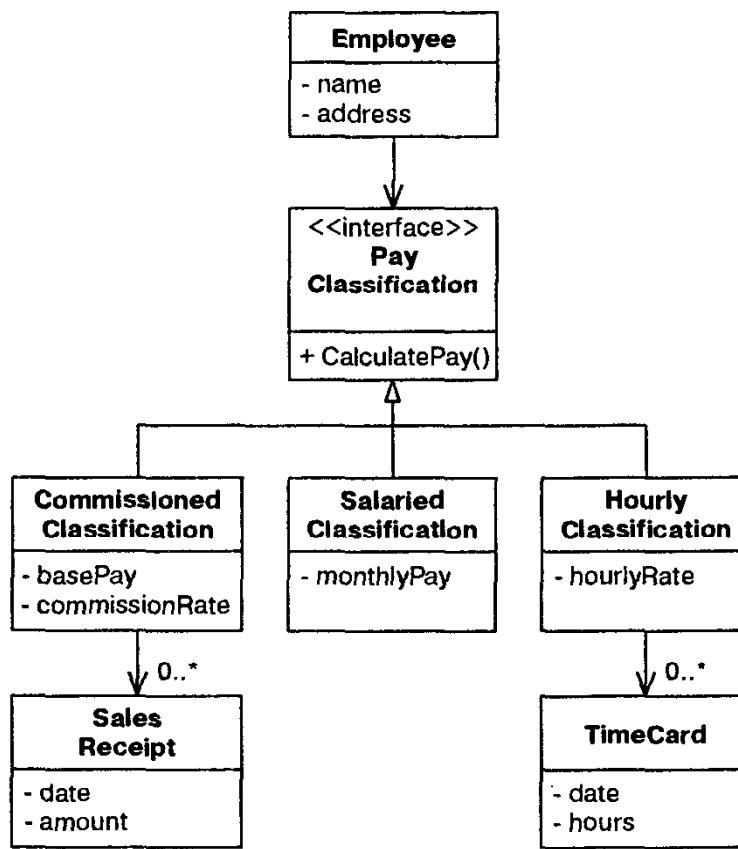


Рис. 13.4. База данных сотрудников

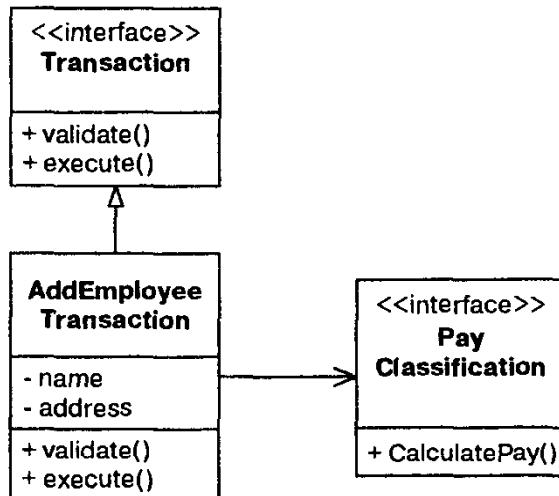


Рис. 13.5. Транзакция AddEmployee

Метод проверки корректности данных обрабатывает все данные без исключения. Он проверяет их синтаксическую и семантическую корректность. Можно даже проверить, не противоречат ли новые данные уже существующим. К примеру, есть возможность удостовериться, что записи о данном работнике в базе еще не существует (исключить дублирование данных).

После завершения проверки происходит внесение информации в базу данных. В нашем простом примере новый объект **Employee** будет создан и загружен

с использованием полей данных объекта `AddEmployeeTransaction`. Объект `PayClassification` будет перемещен или скопирован в `Employee`.

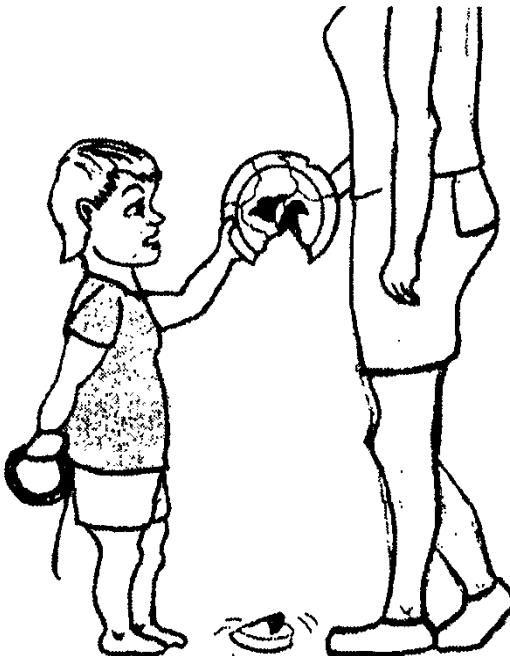
## Физическое и временное разделение связей

Основное преимущество данной технологии заключается в разделении кода, который получает данные от пользователя, кода, который проверяет их (и оперирует ими) и всеми остальными объектами программы. К примеру, для ввода данных о новых сотрудниках можно использовать диалоговые окна графического интерфейса. При этом нет смысла включать в код интерфейса алгоритмы проверки и загрузки этих данных — в случае подобного объединения нельзя будет использовать данные алгоритмы при работе с другими интерфейсами. Путем выделения алгоритмов в отдельный класс `AddEmployeeTransaction`, мы физически отделяем их код от интерфейса. Более того, мы отделяем код, управляющий работой базы, от остальных объектов.

## Временное разделение

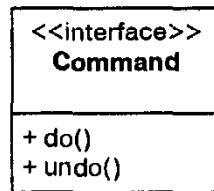
Можно также разделить код проверки и загрузки (обработки) другим способом. При вводе данных необязательно сразу же вызывать процедуры проверки и загрузки их в базу. Объекты могут быть внесены в отдельный список, а операции над ними можно произвести позже.

Допустим, что мы работаем с базой данных, которая должна оставаться неизменной в течение дня. Изменения могут вноситься только в промежуток времени между 24.00 и 01.00. Дожидаться полуночи и пытаться успеть внести все данные и команды за час — не самое лучшее решение. Гораздо удобнее вносить данные по мере их поступления, чтобы затем они хранились в определенном месте (после проверки), а после полуночи автоматически загружались в базу. Схема `command` предоставляет нам такую возможность.



## Отмена действия (UNDO)

На рис. 13.6 в шаблон `Command` добавлен метод `undo()`. Он основан на следующем принципе: если метод `do()` производного класса `Command` можно использовать для запоминания деталей операций, им же и выполняемым, то ме-



**Рис. 13.6.** Отмена действия  
в шаблоне Command

тод `undo()` можно использовать для “отката” (отмены предыдущих действий) и возврата системы к исходному состоянию.

Представим себе, к примеру, приложение, которое позволяет пользователю рисовать на экране геометрические фигуры. На панели инструментов находятся кнопки, дающие пользователю возможность рисовать окружности, квадраты, треугольники и так далее. Предположим, что пользователь нажимает на кнопку “нарисовать окружность”. Система создает объект `DrawCircleCommand`, а затем вызывает для него функцию `do()`. `DrawCircleCommand` отслеживает перемещения мыши пользователем и ожидает нажатия в рабочей области окна (“окне рисования”). Когда пользователь щелкает мышью, `DrawCircleCommand` регистрирует его, устанавливая точку щелчка в качестве центра окружности, и рисует анимированный круг, радиус которого изменяется вместе с перемещением мыши. Когда пользователь снова щелкает мышью, объект `DrawCircleCommand` перестает выводить на экран анимированную окружность и добавляет новый объект к списку форм, отображаемых на экране. Он также сохраняет идентификатор новой окружности в виде переменной, которая затем возвращается методом `do()`. После этого система помещает `DrawCircleCommand` в стек завершенных (выполненных) команд.

Через какое-то время пользователь щелкает на кнопке отмены действия на панели инструментов. Система обращается к данным из стека и вызывает функцию `undo()` результирующего объекта `Command`, при получении сообщения от `undo()` объект `DrawCircleCommand` удаляет окружность, сверяясь со списком идентификаторов объектов, отображаемых на экране.

Используя такую технику, вы можете легко встроить функцию отмены предыдущих действий практически в любое приложение. Код, отменяющий команду, всегда сходен с кодом, запускающим ее на выполнение.

## Шаблон Active Object

Одним из моих любимых применений шаблона `Command` является реализация шаблона `Active Object`<sup>2</sup>. Это очень старый способ обеспечить управление

<sup>2</sup>[Lavender96].

сразу несколькими потоками. В той или иной форме он использовался при создании многозадачных ядер для множества систем, применяемых в промышленности.

В основе данной технологии лежит очень простой принцип. Обратим внимание на листинги 13.2 и 13.3. Объект `ActiveObjectEngine` поддерживает связанный список объектов `Command`. Пользователи могут добавлять новые команды в основную программу или вызывать функцию `run()`. Функция `run()` просто обрабатывает связанный список, выполняя и затем удаляя каждую команду.

---

#### Листинг 13.2. ActiveObjectEngine.java

---

```
import java.util.LinkedList;
import java.util.Iterator;

public class ActiveObjectEngine
{
 LinkedList itsCommands = new LinkedList();

 public void addCommand(Command c)
 {
 itsCommands.add(c);
 }

 public void run()
 {
 while (!itsCommands.isEmpty())
 {
 Command c = (Command) itsCommands.getFirst();
 itsCommands.removeFirst();
 c.execute();
 }
 }
}
```

---



---

#### Листинг 13.3. Command.java

---

```
public interface Command
{
 public void execute() throws Exception;
}
```

---

На первый взгляд, этот код не производит сильного впечатления. Но представим, что произойдет, если один из объектов `Command` в связном списке скопирует сам себя и поместит копию обратно в список. Тогда список никогда не опустеет, а функция `run()` никогда не завершит работу.

Обратим внимание на тестовый случай, код которого приведен в листинге 13.4. Программа создает нечто под названием `sleepCommand`. Кроме того,

перед созданием объекта SleepCommand имеет место задержка в 1000 миллисекунд. Затем SleepCommand помещается в ActiveObjectEngine. При вызове функции run() подразумевается, что прошло нужное количество времени.

---

**Листинг 13.4. TestSleepCommand.java**

---

```
import junit.framework.*;
import junit.swingui.TestRunner;

public class TestSleepCommand extends TestCase
{
 public static void main(String[] args)
 {
 TestRunner.main(new String[]{"TestSleepCommand"});
 }

 public TestSleepCommand(String name)
 {
 super(name);
 }

 private boolean commandExecuted = false;

 public void testSleep() throws Exception
 {
 Command wakeup = new Command()
 {
 public void execute() {commandExecuted = true;}
 };
 ActiveObjectEngine e = new ActiveObjectEngine();
 SleepCommand c = new SleepCommand(1000,e,wakeup);
 e.addCommand(c);
 long start = System.currentTimeMillis();
 e.run();
 long stop = System.currentTimeMillis();
 long sleepTime = (stop-start);
 assert("SleepTime " + sleepTime + " expected > 1000",
 sleepTime > 1000);
 assert("SleepTime " + sleepTime + " expected < 1100",
 sleepTime < 1100);
 assert("Command Executed", commandExecuted);
 }
}
```

---

Давайте рассмотрим этот тестовый случай более внимательно. Конструктор (функция-член класса с тем же именем, что и сам класс, создающая и инициализирующая объект данного класса) объекта sleepCommand содержит три аргумента. Первый аргумент — время задержки в миллисекундах. Второй — ActiveObjectEngine, внутри которого команда будет запущена на исполнение.

ние. И наконец, существует третий командный объект-“будильник”. Смысл его заключается в том, что `SleepCommand` будет ждать определенное время (указанное в миллисекундах), а затем даст команду пробуждения системы.

Листинг 13.5 представляет реализацию объекта `SleepCommand`. При выполнении программы `SleepCommand` проверяет, не был ли он запущен ранее. Если нет, то объект записывает время старта. Если время задержки еще не прошло, он помещает себя обратно в `ActiveObjectEngine`. По прошествии указанного времени в `ActiveObjectEngine` помещается `wakeup`.

---

#### Листинг 13.5. `SleepCommand.java`

---

```
public class SleepCommand implements Command
{
 private Command wakeupCommand = null;
 private ActiveObjectEngine engine = null;
 private long sleepTime = 0;
 private long startTime = 0;
 private boolean started = false;

 public SleepCommand(long milliseconds, ActiveObjectEngine e,
 Command wakeupCommand)
 {
 sleepTime = milliseconds;
 engine = e;
 this.wakeupCommand = wakeupCommand;
 }

 public void execute() throws Exception
 {
 long currentTime = System.currentTimeMillis();
 if (!started)
 {
 started = true;
 startTime = currentTime;
 engine.addCommand(this);
 }
 else if ((currentTime - startTime) < sleepTime)
 {
 engine.addCommand(this);
 }
 else
 {
 engine.addCommand(wakeupCommand);
 }
 }
}
```

---

Можно провести аналогию между данной программой и многопоточным приложением, ожидающим определенного события. Когда поток в многопоточной

программе ожидает конкретного события, он часто запускает процедуру системного вызова, блокирующую данный поток до тех пор, пока ожидаемое событие не произойдет. Программа, код которой представлен в листинге 13.5, не обладает функцией блокирования. Вместо этого, если ожидаемое событие (`(currentTime - startTime) < sleepTime`) не произошло, она просто помещает себя обратно в `ActiveObjectEngine`.

При создании многопоточных систем использовалось множество вариантов подобной технологии, в настоящее время она не менее популярна. Потоки такого типа называются “исполняемыми до завершения” (*run-to-completion*) заданиями (RTC), поскольку каждый экземпляр `Command` исполняется, а затем завершает свою работу, и только после этого запускается следующий такой объект. Само название указывает на то, что объекты `Command` не блокируют друг друга.

Тот факт, что отдельные экземпляры `Command` исполняются по очереди, позволяет RTC-потокам использовать стек исполняемой программы. В отличие от потоков данных в обычной многопоточной системе, совсем не обязательно определять или выделять отдельный стек для каждого потока. Это может оказаться большим преимуществом, если память системы ограничена, а потоков данных должно быть очень много.

В продолжение нашего примера рассмотрим листинг 13.6, где приводится код простой программы, использующей объект `sleepCommand` и отображающей поведение многопоточной системы. Программа называется `DelayedTyper`.

#### Листинг 13.6. `DelayedTyper.java`

```
public class DelayedTyper implements Command
{
 private long itsDelay;
 private char itsChar;
 private static ActiveObjectEngine engine =
 new ActiveObjectEngine();
 private static boolean stop = false;

 public static void main(String args[])
 {
 engine.addCommand(new DelayedTyper(100,'1'));
 engine.addCommand(new DelayedTyper(300,'3'));
 engine.addCommand(new DelayedTyper(500,'5'));
 engine.addCommand(new DelayedTyper(700,'7'));

 Command stopCommand = new Command()
 {
 public void execute() {stop=true;}
 };

 engine.addCommand(
 new SleepCommand(20000,engine,stopCommand));
 }
}
```

```

 engine.run();
 }

public DelayedTyper(long delay, char c)
{
 itsDelay = delay;
 itsChar = c;
}

public void execute() throws Exception
{
 System.out.print(itsChar);
 if(!stop)
 delayAndRepeat();
}

private void delayAndRepeat() throws Exception
{
 engine.addCommand(new SleepCommand(itsDelay,engine,this));
}
}

```

---

Обратите внимание, что `DelayedTyper` включает в себя `Command`. Метод `Execute` просто выводит на печать символ, введенный в структуру, проверяет наличие флага `stop` и, если он не установлен, активизирует `delayAndRepeat`. Метод `delayAndRepeat` создает объект `SleepCommand`, используя задержку, величина которой передается инструкции. Затем `SleepCommand` внедряется в `ActiveObjectEngine`.

Поведение объекта `Command` легко предсказать. Он просто запускает цикл, периодически выводящий на печать определенный символ и выжидаящий затем заданное время. Цикл прекращается в случае установки флага `stop`.

Основная программа запускает сразу несколько экземпляров `DelayedTyper` в `ActiveObjectEngine`, каждый из которых выводит на печать свой собственный символ и обладает своим временем задержки. Затем она активизирует модуль `SleepCommand`, который устанавливает флаг `stop` через некоторое время. Результатом работы программы является простая строка, состоящая из символов '1', '2', '3', '5' и '7'. Повторный запуск выводит на экран похожую строку, символы в которой будут расположены в другом порядке. Ниже показаны результаты двух запусков программы:

```
1357113115113711131511317151311131517311135111371153111357...
135711131513171131511311713511131151731113151131711351113117...
```

Эти строки отличаются друг от друга потому, что генератор тактовой частоты процессора и генератор импульсов времени не могут быть идеально синхронизированы. Этот вид недетерминированного поведения является "визитной карточкой" многопоточных систем.

Недетерминированное поведение системы зачастую является настоящим “проклятием для разработчика”. Любой, кто работал со встроенными системами реального времени, знает, насколько трудно бывает отлаживать программу с недетерминированным поведением.

## Резюме

Простота шаблона *Command* объясняется его гибкостью и универсальностью. Этот шаблон имеет массу применений — от управления базами данных до контроля за аппаратными средствами, от работы с ядром многопоточной системы до создания инструментов для графического интерфейса.

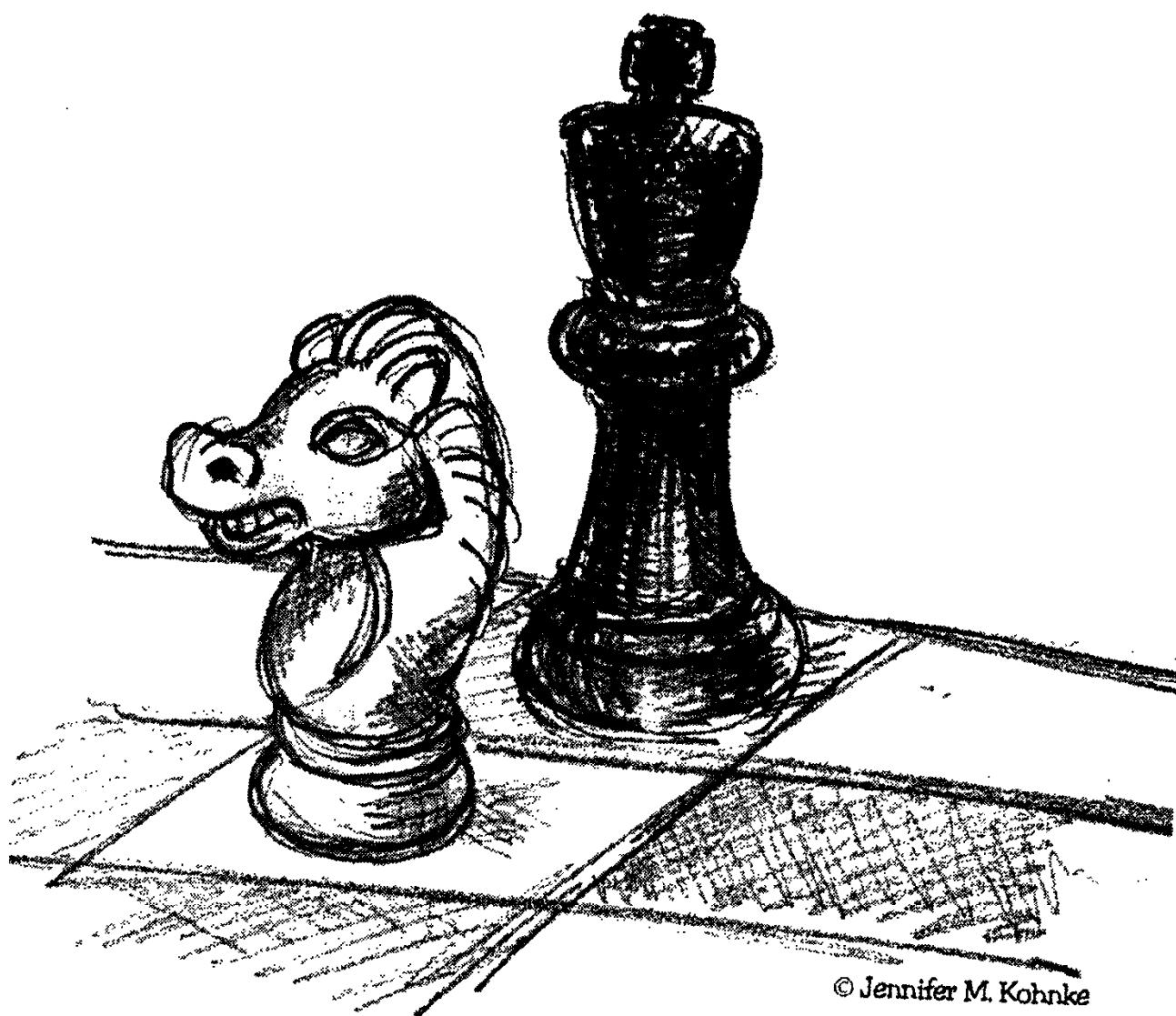
Многие полагают, что шаблон *Command* плохо вписывается в парадигму объектно-ориентированного программирования, поскольку в нем функции ставятся впереди классов. Иногда это действительно так, но в реальности такая технология вполне может оказаться полезной для любого разработчика программ.

## Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
2. Lavender R. G., Schmidt D. C. *Active Object: An Object Behavioral Pattern for Concurrent Programming*, in “Pattern Languages of Program Design” Reading, MA: Addison-Wesley, 1996.

# 14

## Шаблоны Template Method и Strategy: наследование и делегирование



© Jennifer M. Kohnke

Лучшая стратегия в жизни — усердие.

Китайская пословица

Вспомните начало 90-х годов прошлого века (“детство” объектно-ориентированного программирования), когда все были увлечены идеей наследования. Следствия этого увлечения были достаточно серьезными. Благодаря наследованию можно легко создавать *различные программы!* Например, предположим, что имеется некоторый класс, который содержит набор полезных свойств. В этом случае можно создать подкласс, исключив свойства, которые не представляют особого интереса. Появляется возможность повторного использования кода путем простого наследования! При этом можно выполнять всеобъемлющие таксономии программных структур, отделяя каждый уровень повторно используемого кода от верхних уровней. В результате перед нами открывается новый мир.

Как и с большинством “других новых миров”, порой кажется, что нам предстаются неограниченные возможности. К 1995 году стало понятно, что методом наследования очень часто злоупотребляют, причем последствия подобного злоупотребления могут быть очень серьезными. Согласно высказыванию Гамма (Gamma), Хелма (Helm), Джонсона (Johnson) и Виссида (Vlissides), “*хорошо выполненная композиция объектов лучше наследования классов*”<sup>1</sup>. Поэтому следует сократить использование наследования, заменяя ее композицией или делегированием.

В настоящей главе рассматриваются два шаблона, которые позволяют лучше понять различия между наследованием и делегированием. Достаточно часто шаблоны **Template Method** и **Strategy** позволяют решать подобные проблемы, а также являются взаимозаменяемыми. Следует отметить, что шаблон **Template Method** применяет технологию наследования, а шаблон **Strategy** – делегирование.

Шаблоны **Template Method** и **Strategy** позволяют решить проблемы, связанные с отделением обобщенного алгоритма от детализированного содержимого. Весьма часто именно эти шаблоны оказываются полезными в процессе разработки программного проекта. В нашем распоряжении окажется обобщенный алгоритм, который может применяться во многих случаях. В целях соответствия принципу инверсии зависимостей (DIP, Dependency-Inversion Principle) следует убедиться в том, что обобщенный алгоритм не зависит от детализированной реализации. Точнее, обобщенный алгоритм и его детализированная реализация должны зависеть от абстракций.

<sup>1</sup>[GOF95], с. 20.

## Шаблон Template Method

Вспомните все программы, написанные вами в течение жизни. Не правда ли, что многие из них содержат следующую фундаментальную циклическую структуру (`main`).

```
Initialize();
while (!done()) // цикл main
{
 Idle(); // выполнение некоторых действий.
}
Cleanup();
```

Сначала инициализируется приложение. Затем вводится основной цикл. В теле этого цикла определяются все действия, выполняемые программой. Например, можно обрабатывать события, связанные с графическим интерфейсом пользователя, или обрабатывать записи в базе данных. После завершения перечисленных действий осуществляется выход из основного цикла, но перед выходом осуществляется очистка.

Эта структура носит настолько обобщенный характер, что имеет смысл поместить ее в отдельный класс, именованный `Application`. Затем этот класс может повторно использоваться при создании каждой новой программы. Прекрасная идея! Ведь нет ничего противнее, чем каждый раз переписывать цикл<sup>2</sup>!

Например, обратим пристальное внимание на листинг 14.1. Здесь присутствуют все элементы стандартной программы. Инициализируются переменные `inputStreamReader` и `BufferedReader`. Здесь мы имеем дело с основным циклом, в процессе выполнения которогочитываются значения температуры, присвоенные переменной `BufferedReader` (по шкале Фаренгейта), которые затем преобразуются с использованием шкалы Цельсия. После завершения выполнения программы печатается соответствующее сообщение.

---

### Листинг 14.1. ftoc raw

---

```
import java.io.*;
public class ftocraw
{
 public static void main(String[] args) throws Exception
 {
 InputStreamReader isr = new InputStreamReader(System.in);
 BufferedReader br = new BufferedReader(isr);
 boolean done = false;
 while (!done)
 {
 String fahrString = br.readLine();
 if (fahrString == null || fahrString.length() == 0)
 done = true;
 }
 }
}
```

<sup>2</sup>Я с удовольствием поделюсь с вами идеями, дабы облегчить ваше существование.

```

 }
 else
 {
 double fahr = Double.parseDouble(fahrString);
 double celcius = 5.0/9.0*(fahr-32);
 System.out.println("F=" + fahr + ", C=" + celcius);
 }
}
System.out.println("ftoc exit");
}
}

```

---

Описанная программа включает все элементы циклической структуры main. Сначала выполняется инициализация, затем реализуется цикл main, после чего выполняется очистка и выход из программы.

Можно отделить эту фундаментальную структуру от программы ftoc с помощью шаблона Template Method. Благодаря применению этого метода весь обобщенный код помещается в реализованный метод для абстрактного базового класса. Этот метод перехватывает обобщенный алгоритм, но обработку всех деталей передает абстрактным методам базового класса.

Так, например, можно перехватить циклическую структуру main в абстрактном базовом классе Application (листинг 14.2).

---

#### Листинг 14.2. Application.java

---

```

public abstract class Application
{
 private boolean isDone = false;

 protected abstract void init();
 protected abstract void idle();
 protected abstract void cleanup();

 protected void setDone()
 {isDone = true; }

 protected boolean done()
 {return isDone; }

 public void run()
 {
 init();
 while(!done())
 idle();
 cleanup();
 }
}

```

Этот класс реализует описание обобщенного приложения, включающего цикл main. Этот цикл можно видеть в реализованной функции run. Нетрудно заметить, что вся работа передается абстрактным методам init, idle и cleanup. Метод init выполняет необходимую инициализацию. Метод idle выполняет основную работу в программе и может вызываться несколько раз до тех пор, пока не будет вызван setDone. Метод cleanup выполняет все, что требуется перед выходом из программы.

Можно также переписать класс ftoc, выполнив наследование из Application и просто указав параметры для абстрактных методов. Пример реализации на практике подобной технологии приводится в листинге 14.3.

---

#### Листинг 14.3. ftocTemplateMethod.java

---

```
import java.io.*;
public class ftocTemplateMethod extends Application
{
 private InputStreamReader isr;
 private BufferedReader br;

 public static void main(String[] args) throws Exception
 {
 (new ftocTemplateMethod()).run();
 }

 protected void init()
 {
 isr = new InputStreamReader(System.in);
 br = new BufferedReader(isr);
 }

 protected void idle()
 {
 String fahrString = readLineAndReturnNullIfError();
 if (fahrString == null || fahrString.length() == 0)
 setDone();
 else
 {
 double fahr = Double.parseDouble(fahrString);
 double celcius = 5.0/9.0*(fahr-32);
 System.out.println("F=" + fahr + ", C=" + celcius);
 }
 }

 protected void cleanup()
 {
 System.out.println("ftoc exit");
 }

 private String readLineAndReturnNullIfError()
```

```

{
 String s;
 try
 {
 s = br.readLine();
 }
 catch(IOException e)
 {
 s = null;
 }
 return s;
}
}

```

---

Обработка исключений занимает некоторое время, хотя при этом упрощается процесс рассмотрения того, как прежнее приложение `ftoc` преобразуется с помощью шаблона `Template Method`.

## Не злоупотребляйте шаблонами

Наверное, вы уже подумали следующее: *“И это он серьезно? Неужели он думает, что я буду использовать класс Application при разработке новых приложений? В любом случае это меня не сильно касается и, как мне кажется, вся проблема просто “высосана из пальца”.*

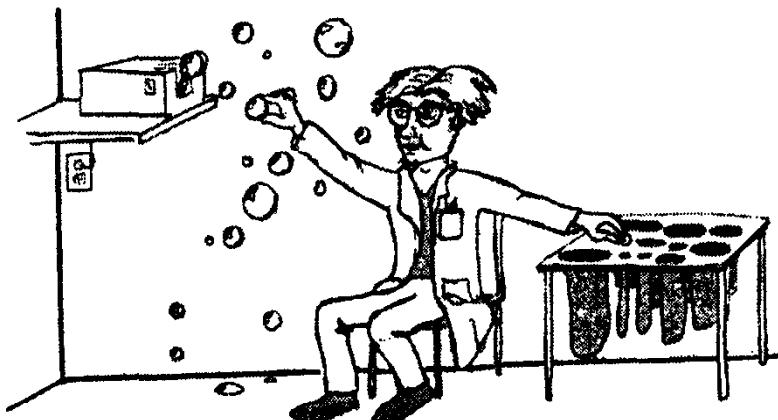
Выбор рассматриваемого примера мотивируется его простотой, а также тем, что в этом случае поддерживается хорошая платформа, которая позволяет увидеть механику метода `Template Method` в действии. Хотя на самом деле разрабатывать приложения, подобные `ftoc`, не рекомендуется.

Рассматриваемый пример демонстрирует пагубные последствия, к которым может привести злоупотребление шаблонами. Использование в данном случае шаблона `Template Method` является нецелесообразным, поскольку это ведет к усложнению программы и к увеличению объема кода. Инкапсуляция цикла `main` при разработке любого приложения на первый взгляд кажется неплохой идеей, хотя на практике эта идея может оказаться “бесплодной”.

Проектные шаблоны обеспечивают разработчикам целый ряд возможностей, а также обеспечивают разрешение многих проблем, возникающих в процессе проектирования. Но отсюда вовсе не следует, что эти шаблоны должны применяться повсеместно. В данном случае, несмотря на то, что шаблон `Template Method` может применяться в целях устранения тех или иных проблем, его использование все же не рекомендуется. Накладные расходы, связанные с применением шаблона, превышают возможную прибыль.

Ну а теперь обратите внимание на более сложный пример. (листинг 14.4.)

## Сортировка методом “пузырька”<sup>3</sup>



**Листинг 14.4. BubbleSorter.java**

```
public class BubbleSorter
{
 static int operations = 0;
 public static int sort(int[] array)
 {
 operations = 0;
 if (array.length <= 1)
 return operations;

 for (int nextToLast = array.length-2;
 nextToLast >= 0; nextToLast--)
 for (int index = 0; index <= nextToLast; index++)
 compareAndSwap(array, index);

 return operations;

 private static void swap(int[] array, int index)
 {
 int temp = array[index];
 array[index] = array[index+1];
 array[index+1] = temp;
 }

 private static void compareAndSwap(int[] array, int index)
 {
 if (array[index] > array[index+1]) swap(array, index);
 operations++;
 }
 }
}
```

<sup>3</sup>Как и в случае с Application, приложение Bubble Sort можно считать идеальным для учебных целей. Не следует применять его для сортировки больших объемов данных, поскольку в этом случае применяются более совершенные алгоритмы.

Класс `BubbleSorter` “знает” о том, каким образом следует сортировать целочисленный массив с помощью алгоритма сортировки “пузырьком”. Метод сортировки `BubbleSorter` использует алгоритм, включающий метод сортировки “пузырьком”. Два вспомогательных метода, `swap` и `compareAndSwap`, реализуют детали, связанные с обработкой целых чисел и массивов, а также механику, требуемую алгоритмом сортировки.

С помощью шаблона `Template Method` можно выделять алгоритм сортировки “пузырьком” в виде абстрактного базового класса, именуемого `BubbleSorter`. Этот класс включает реализацию функции сортировки, которая вызывает абстрактный метод `outOfOrder` и другой метод — `swap`. Метод `outOfOrder` выполняет сравнение двух смежных элементов массива. При этом возвращается значение `true`, если элементы неупорядочены. Метод `swap` выполняет перестановку двух смежных элементов массива.

Метод `sort` ничего не “знает” ни о самом массиве, ни о том, как следует обрабатывать объекты, находящиеся в этом массиве. Он просто вызывает функцию `outOfOrder` с различными индексами массива, а также определяет, была ли выполнена перестановка индексов (листинг 14.5).

---

#### Листинг 14.5. `BubbleSorter.java`

---

```
public abstract class BubbleSorter
{
 private int operations = 0;
 protected int length = 0;

 protected int doSort()
 {
 operations = 0;
 if (length <= 1)
 return operations;

 for (int nextToLast = length-2;
 nextToLast >= 0; nextToLast--)
 for (int index = 0; index <= nextToLast; index++)
 {
 if (outOfOrder(index))
 swap(index);
 operations++;
 }

 return operations;
 }

 protected abstract void swap(int index);
 protected abstract boolean outOfOrder(int index)
```

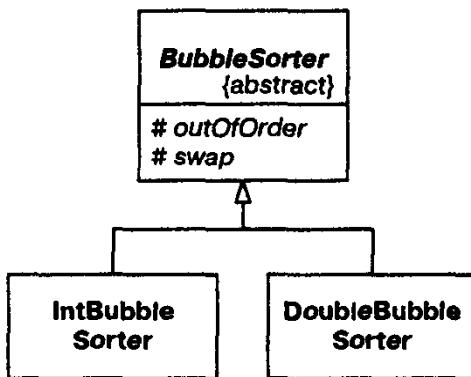


Рис. 14.1. Структура класса Bubble-Sorter

Располагая `BubbleSorter`, можно создавать простые производные методы, которые могут применяться для сортировки любых объектов. Например, можно сформировать класс `intBubbleSorter`, обеспечивающий сортировку целочисленных массивов, а также класс `DoubleBubbleSorter`, применяемый для сортировки массивов, содержащих вещественные числа удвоенной точности. (Рис. 14.1, листинг 14.6 и листинг 14.7.)

---

#### Листинг 14.6. `IntBubbleSorter.java`

---

```

public class IntBubbleSorter extends BubbleSorter
{
 private int[] array = null;
 public int sort(int[] theArray)
 {
 array = theArray;
 length = array.length;
 return doSort();
 }

 protected void swap(int index)
 {
 int temp = array[index];
 array[index] = array[index+1];
 array[index+1] = temp;
 }

 protected boolean outOfOrder(int index)
 {
 return (array[index] > array[index+1]);
 }
}

```

---

**Листинг 14.7. DoubleBubbleSorter.java**

```

public class DoubleBubbleSorter extends BubbleSorter
{
 private double[] array = null;
 public int sort(double[] theArray)
 {
 array = theArray;
 length = array.length;
 return doSort();
 }

 protected void swap(int index)
 {
 double temp = array[index];
 array[index] = array[index+1];
 array[index+1] = temp;
 }

 protected boolean outOfOrder(int index)
 {
 return (array[index] > array[index+1]);
 }
}

```

Шаблон *Template Method* демонстрирует одну из классических форм метода повторного использования, применяемого в объектно-ориентированном программировании. Обобщенные алгоритмы размещаются в базовом классе, а затем наследуются в различных детализированных контекстах. Но использование этой техники предполагает определенные затраты. Это связано с тем, что между производными и базовыми классами устанавливается очень сложная взаимосвязь.

Например, функции *outOfOrder* и *swap* из класса *IntBubbleSorter* представляют необходимый “минимальный набор”, применяемый другими видами алгоритмов сортировки. И все же не существует способа повторного применения *outOfOrder* и *swap* другими алгоритмами сортировки. Реализовав наследование с помощью *BubbleSorter*, мы “обречем” *IntBubbleSorter* на вечное ограничение рамками *BubbleSorter*. В этом случае дополнительные возможности предлагаются шаблоном *Strategy*.

## Шаблон *Strategy*

Благодаря использованию шаблона *Strategy* разрешается проблема инвертирования зависимостей между обобщенным алгоритмом и его детализированной реализацией. Причем в этом случае могут применяться самые различные методы.

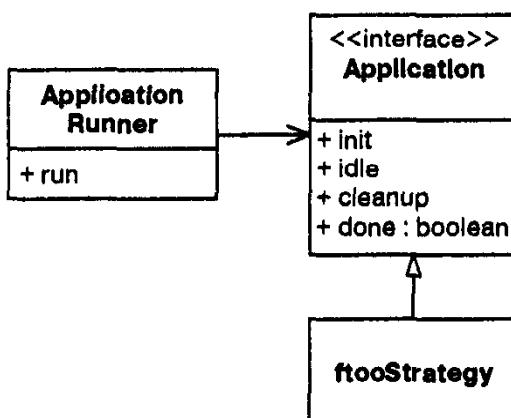


Рис. 14.2. Шаблон Strategy в составе алгоритма Application

Снова вернемся к проблеме, связанной с неоправданным применением шаблонов в классе `Application`.

Вместо того чтобы обобщенный алгоритм приложения включать в абстрактный базовый класс, в данном случае он будет размещен в *конкретном* классе, именуемом `ApplicationRunner`. В интерфейсе `Application` будут определены абстрактные методы, которые должен вызывать обобщенный алгоритм. На базе этого интерфейса будет наследоваться `ftocStrategy`, а затем передаваться классу `ApplicationRunner`. Затем именно `ApplicationRunner` будет делегирован для данного интерфейса. (Рис. 14.2, а также листинги с 14.8 по 14.10.)

---

#### Листинг 14.8. ApplicationRunner.java

---

```

public class ApplicationRunner
{
 private Application itsApplication = null;

 public ApplicationRunner(Application app)
 {
 itsApplication = app;
 }
 public void run()
 {
 itsApplication.init();
 while (!itsApplication.done())
 itsApplication.idle();
 itsApplication.cleanup();
 }
}

```

---



---

#### Листинг 14.9. Application.java

---

```

public interface Application

```

```
public void init();
public void idle();
public void cleanup();
public boolean done();
}
```

---

**Листинг 14.10. ftocStrategy.java**

---

```
import java.io.*;
public class ftocStrategy implements Application
{
 private InputStreamReader isr;
 private BufferedReader br;
 private boolean isDone = false;

 public static void main(String[] args) throws Exception
 {
 (new ApplicationRunner(new ftocStrategy())).run();
 }

 public void init()
 {
 isr = new InputStreamReader(System.in);
 br = new BufferedReader(isr);
 }

 public void idle()
 {
 String fahrString = readLineAndReturnNullIfError();
 if (fahrString == null || fahrString.length() == 0)
 isDone = true;
 else
 {
 double fahr = Double.parseDouble(fahrString);
 double celcius = 5.0/9.0*(fahr-32);
 System.out.println("F=" + fahr + ", C=" + celcius);
 }
 }

 public void cleanup()
 {
 System.out.println("ftoc exit");
 }

 public boolean done()
 {
 return isDone;
 }

 private String readLineAndReturnNullIfError()
```

```

 {
 String s;
 try
 {
 s = br.readLine();
 }
 catch(IOException e)
 {
 s = null;
 }
 return s;
 }
}

```

---

Очевидно, что применяемая в данном случае структура обеспечивает преимущества, а также снижает уровень накладных расходов (относительно структуры *Template Method*). Шаблон *Strategy* вовлекает более полные классы, а также носит косвенный характер (по сравнению с *Template Method*). Представление указателя при выполнении *ApplicationRunner* связано с несколько большими накладными расходами (если за шкалу отсчета брать время выполнения и объем, занимаемый хранимыми данными), чем в случае наследования. С другой стороны, если требуется запустить на выполнение большое количество приложений, можно использовать повторно экземпляр *ApplicationRunner*, осуществляя передачу параметров в нескольких экземплярах *Application*. В результате уменьшается степень связывания между обобщенным алгоритмом и контролируемыми им деталями.

Ни один из видов упомянутых преимуществ и накладных расходов не перекрывает “соседа”. Как правило, наибольшее беспокойство приносит дополнительный класс, требуемый шаблону *Strategy*. Ну а теперь перейдем к рассмотрению других вопросов.

## И снова сортировка

Обратимся к рассмотрению сортировки методом “пузырька”, реализуемой с помощью шаблона *Strategy*. (Листинги с 14.11 по 14.13.)

---

### Листинг 14.11. BubbleSorter.java

---

```

public class BubbleSorter
{
 private int operations = 0;
 private int length = 0;
 private SortHandle itsSortHandle = null;

 public BubbleSorter(SortHandle handle)
 {
 itsSortHandle = handle;
 }
}

```

```

}

public int sort(Object array)
{
 itsSortHandle.setArray(array);
 length = itsSortHandle.length();
 operations = 0;
 if (length <= 1)
 return operations;

 for (int nextToLast = length-2;
 nextToLast >= 0; nextToLast--)
 for (int index = 0; index <= nextToLast; index++)
 {
 if (itsSortHandle.outOfOrder(index))
 itsSortHandle.swap(index);
 operations++;
 }

 return operations;
}
}

```

---

**Листинг 14.12. SortHandle.java**

```

public interface SortHandle
{
 public void swap(int index);
 public boolean outOfOrder(int index);
 public int length();
 public void setArray(Object array);
}

```

---

**Листинг 14.13. Листинг 14.13. IntSortHandle.java**

```

public class IntSortHandle implements SortHandle
{
 private int[] array = null;

 public void swap(int index)
 {
 int temp = array[index];
 array[index] = array[index+1];
 array[index+1] = temp;
 }

 public void setArray(Object array)
 {
 this.array = (int[])array;
 }
}

```

```

public int length()
{
 return array.length;
}

public boolean outOfOrder(int index)
{
 return (array[index] > array[index+1]);
}
}

```

---

Обратите внимание на то, что класс `IntSortHandle` “ничего не знает” о классе `BubbleSorter`. Не существует зависимости независимо от реализации метода сортировки “пузырьком”. В этом случае шаблон `Template Method` неприемлем. Снова обратите внимание на листинг 14.6, после чего вы наверняка заметите, что класс `intBubbleSorter` непосредственно зависит от класса `BubbleSorter`, включающего алгоритм сортировки методом “пузырька”.

На самом деле подход с применением шаблона `Template Method` приводит к частичному нарушению принципа DIP. Это связано с тем, что реализованы методы `swap` и `outOfOrder`, которые непосредственно зависят от сортировки методом “пузырька”. Подобная зависимость не характерна в случае применения подхода с шаблоном `Strategy`. Благодаря этому можно применять класс `IntSortHandle` вместе с реализациями `Sorter` вместо того, что применять `BubbleSorter`.

Например, можно разработать вариант алгоритма сортировки методом “пузырька”, который завершает свою работу досрочно в случае, если массив будет упорядочен (листинг 14.14). Алгоритм `QuickBubbleSorter` также может использовать класс `IntSortHandle` или любой другой класс, производный от класса `SortHandle`.

---

#### Листинг 14.14. QuickBubbleSorter.java

---

```

public class QuickBubbleSorter
{
 private int operations = 0;
 private int length = 0;
 private SortHandle itsSortHandle = null;

 public QuickBubbleSorter(SortHandle handle)
 {
 itsSortHandle = handle;
 }

 public int sort(Object array)
 {
 itsSortHandle.setArray(array);
 length = itsSortHandle.length();

```

```

operations = 0;
if (length <= 1)
 return operations;

boolean thisPassInOrder = false;
for (int nextToLast = length-2; nextToLast >= 0 &&
 !thisPassInOrder; nextToLast--)
{
 thisPassInOrder = true; // потенциально.
 for (int index = 0; index <= nextToLast; index++)
 {
 if (itsSortHandle.outOfOrder(index))
 {
 itsSortHandle.swap(index);
 thisPassInOrder = false;
 }
 operations++;
 }
}
return operations;
}
}

```

Как видите, шаблон **Strategy** обеспечивает одно дополнительное преимущество по сравнению с шаблоном **Template Method**. В то время, как шаблон **Template Method** позволяет сформировать обобщенный алгоритм, который позволяет манипулировать несколькими возможными детализированными реализациями, шаблон **Strategy** полностью совместим с принципом DIP. Благодаря этому каждая детализированная реализация может обрабатываться каждым из различных обобщенных алгоритмов.

## Резюме

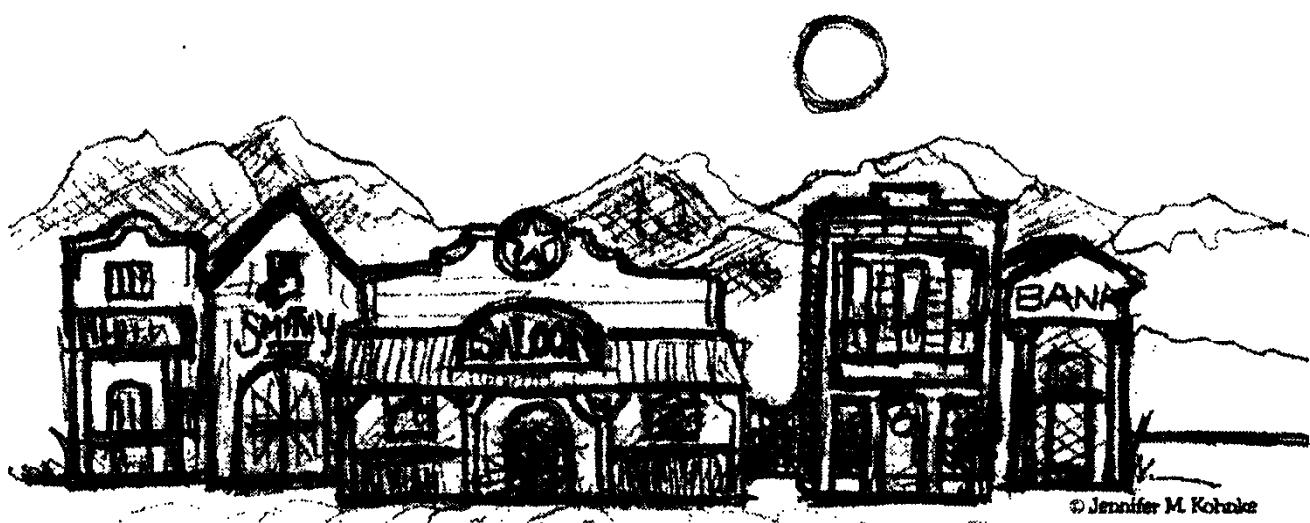
Оба шаблона, **Template Method** и **Strategy**, позволяют отделять алгоритмы высокого уровня от низкоуровневых подробностей. Также обеспечивается повторное использование алгоритмов высокого уровня, причем исключается зависимость от деталей. Шаблон **Strategy** также позволяет организовать использование деталей независимо от алгоритма высокого уровня. Это достигается за счет незначительного усложнения, затрат несколько большего количества памяти, а также времени выполнения.

## Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley. 1995.
2. Martin. Robert C. и др. *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley, 1998.

# 15

## Шаблоны Facade и Mediator



© Jennifer M. Kohlke

А сейчас мы приступим к рассмотрению шаблонов общего назначения. Они применяются для установки некоторого рода политики по отношению к другой группе объектов. Шаблон *Facade* устанавливает политику в направлении “сверху вниз”, а шаблон *Mediator* – в направлении “снизу вверх”. Эффект применения *Facade* очевиден, хотя весьма ограничен. В случае с шаблоном *Mediator* эффект будет невидимым, а политика его применения носит рекомендательный характер.

### Шаблон *Facade*

Шаблон *Facade* применяется в том случае, если требуется установить простой, но в то же время специфичный интерфейс для группы объектов, для которых выбран сложный родовой интерфейс. Например, обратите внимание на класс

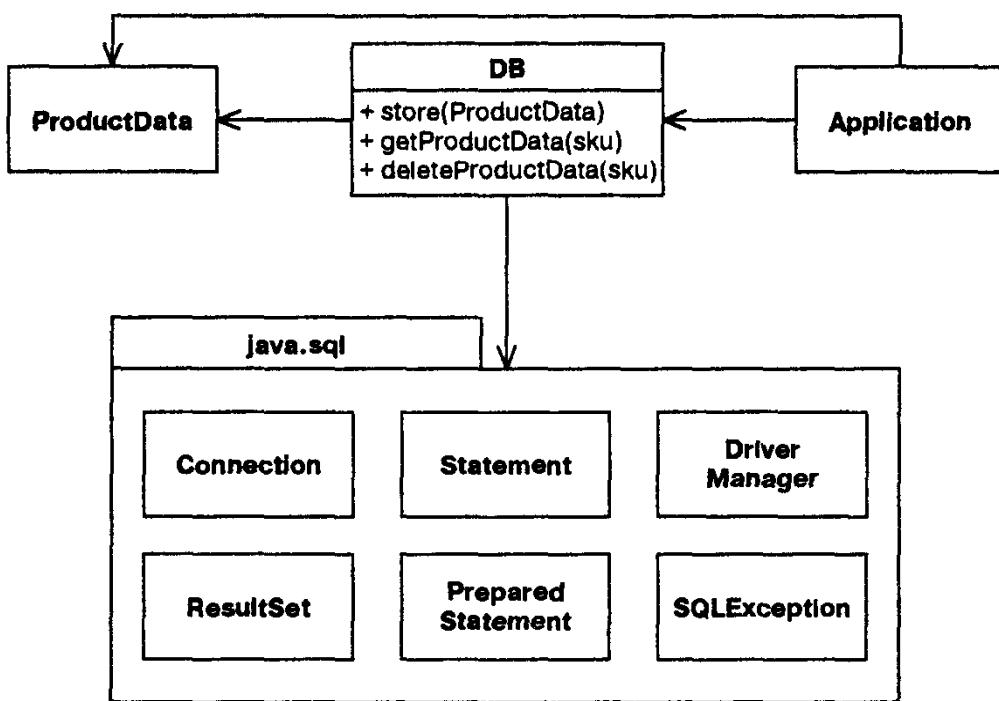


Рис. 15.1. Шаблон DB Facade

**DB.java**, код которого включен в листинг 26.9 (глава 26). Этот класс применяет очень простой интерфейс, который является специфичным для **productData** относительно сложных родовых интерфейсов классов в пакете **java.sql**. На рис. 15.1 приводится соответствующая структура.

Обратите внимание, что класс **DB** предотвращает потребность **Application** в том, чтобы знать детали реализации пакета **Java.sql**. Он скрывает сложность и обобщенный характер **java.sql**, маскируя их очень простым, но в то же время специфичным интерфейсом.

Шаблон **Facade**, как и **DB**, устанавливает применение политики в пакете **java.sql**. Он “знает”, каким образом следует инициализировать и закрывать подключение к базе данных. Также известно, каким образом следует транслировать элементы **ProductData** в поля базы данных (и наоборот). Также известно, каким образом следует формировать подходящие запросы и команды, позволяющие выполнять обработку информации в базе данных. В этом случае излишняя сложность скрывается от пользователей. С “точки зрения” **Application**, **java.sql** не существует; поскольку он скрыт за **Facade**.

В процессе использования шаблона **Facade** разработчикам следует адаптировать соглашение таким образом, чтобы все вызовы базы данных направлялись через **DB**. Если же какая-то часть кода **Application** направляется непосредственно **java.sql** вместо того, чтобы использовать **Facade**, в этом случае соглашение будет нарушено. Следовательно, **Facade** устанавливает политики для приложения. Согласно соглашению, **DB** выступает в качестве единственного брокера возможностей **java.sql**.

## Шаблон Mediator

Шаблон **Mediator** также применяется для установки политики. В то время как **Facade** устанавливает политику, используя видимый и ограничивающий метод, **Mediator** устанавливает свои политики скрытым и не ограничивающим способом. Например, класс **QuickEntryMediator**, код которого приводится в листинге 15.1, находится “за кулисами”, а также связан с полями ввода текста в списке. Если вы вводите данные в поле ввода текста, выделяется первый элемент в списке, который соответствует введенному тексту. Благодаря этому облегчается ввод аббревиатур, а также обеспечивается быстрый вывод элементов из списка.

---

### Листинг 15.1. QuickEntryMediator.java

---

```
package utility;

import javax.swing.*;
import javax.swing.event.*;

/**
 * QuickEntryMediator. Этот класс использует JTextField и
 * JList. Пользователь вводит символы в JTextField,
 * предваряемые записями в JList. Автоматически
 * выбирается первый элемент в Jlist, который
 * соответствует текущему префиксу JTextField.
```

Если JTextField – нуль или его префикс не совпадает с элементами в JList, выбор JList устраняется.

Отсутствуют методы, применяемые для вызова этого объекта. Можно просто создать объект и забыть о нем. (Но не удаляйте его вместе с ‘‘мусором’’...)

Пример:

```
JTextField t = new JTextField();
JList l = new JList();

QuickEntryMediator qem = new QuickEntryMediator(t, l);
// а вот и сама программа.
```

автор Роберт К. Мартин, Роберт С. Косс  
©дата 30 июня, 1999 г, 2113 (SLAC)  
\*/

```
public class QuickEntryMediator {
 public QuickEntryMediator(JTextField t, JList l) {
 itsTextField = t;
 itsList = l;
```

```
itsTextField.getDocument().addDocumentListener(
 new DocumentListener() {
 public void changedUpdate(DocumentEvent e){
 textFieldChanged();
 }

 public void insertUpdate(DocumentEvent e) {
 textFieldChanged();
 }

 public void removeUpdate(DocumentEvent e) {
 textFieldChanged();
 }
 } // новый DocumentListener);
} // addDocumentListener
}//QuickEntryMediator()

private void textFieldChanged() {
 String prefix = itsTextField.getText();

 if (prefix.length() == 0) {
 itsList.clearSelection();
 return;
 }

 ListModel m = itsList.getModel();
 boolean found = false;
 for (int i = 0; found == false && i < m.getSize(); i++) {
 Object o = m.getElementAt(i);
 String s = o.toString();
 if (s.startsWith(prefix)){
 itsList.setSelectedValue(o, true);
 found = true;
 }
 }

 if (!found) {
 itsList.clearSelection();
 }
} // textFieldChanged

private JTextField itsTextField;
private JList itsList;
} // класс QuickEntryMediator
```

Структура QuickEntryMediator продемонстрирована на рис. 15.2. Экземпляр QuickEntryMediator конструируется на основе JList и JTextField. Объект QuickEntryMediator регистрирует анонимный DocumentListener вместе с JTextField. Этот слушатель вызывает метод textFieldChanged

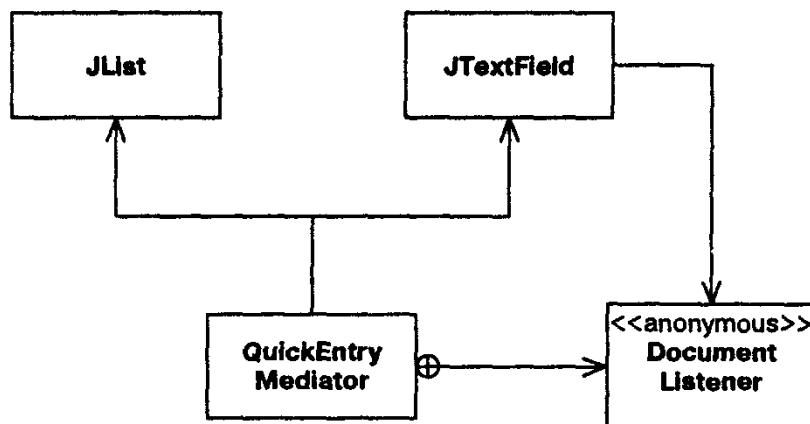


Рис. 15.2. Объект QuickEntryMediator

при наличии каких-либо изменений в тексте. При вызове этого метода производится поиск элементов в `JList`, которые предваряются текстом, с последующим их выделением.

Пользователи, работающие с `JList` и `JTextField`, не подозревают о существовании `Mediator`. Этот шаблон “ведет себя тихо”, формируя политику по отношению к этим объектам. При этом не остаются какие-либо видимые “следы”.

## Резюме

Если политика должна быть яркой и заметной, следует воспользоваться услугами шаблона `Facade`. Если же во главу угла ставится скрытность и аккуратность, более подходящим является шаблон `Mediator`. Как правило, фасады (`Facade`) являются наиболее заметными. Любой из нас согласится работать с фасадами вместо объектов, скрываемых под ним. Медиатор (`Mediator`) скрыт от пользователей. Предлагаемая политика обычно является негласной и не регулируется какими-либо соглашениями.

## Литература

1. Gamma и др. *Design Patterns*. Reading. MA: Addison-Wesley, 1995.

# 16

## Шаблоны Singleton и Monostate



Бесконечно блаженство существования! Именно оно и ничего кроме него.

---

Эдвин А. Эбботт

Между классами и их экземплярами обычно устанавливаются взаимосвязи “один ко многим”. Для большинства классов можно создавать несколько экземпляров. Экземпляры создаются по мере необходимости, а их применение реализуется в том случае, если это приносит какую-либо пользу. Они появляются в потоке действий, реализующих распределение и очистку памяти.

Некоторые классы обладают лишь одним экземпляром. Этот экземпляр активизируется в том случае, если программа запускается на выполнение. После завершения выполняемой программы экземпляр класса разрушается. Подобные

объекты иногда выступают в качестве своего рода “корней” приложения. Начиная с “корней”, можно отслеживать многие другие объекты в системе. Иногда в качестве этих объектов применяются так называемые фабрики, с помощью которых можно создавать другие объекты в системе. Порой эти объекты являются диспетчерами, несущими ответственность за отслеживание некоторых других объектов, а также реализующими управление ими.

Какими бы ни были рассматриваемые объекты, возможны серьезные логические ошибки в случае, если создается более одного объекта. Если было создано более одного “корня”, доступ к объектам приложения будет зависеть от выбранного “корня”. Программисты, не подозревающие о существовании более чем одного “корня”, могут столкнуться с ситуацией, когда приходится иметь дело с поднабором объектов приложения, о которых ничего не известно. Если существует несколько фабрик, “опека” созданных объектов может быть скомпрометирована. Если существует несколько диспетчеров, последовательные действия могут быть преобразованы в параллельную форму.

Возможно, все дело в том, что механизмы, придающие особенность этим объектам, являются избыточными. Ведь в процессе инициализации приложения создается один из таких механизмов, который используется в дальнейшей работе. На самом деле подобный алгоритм действий является наилучшим. Следует избегать применения механизма, если в этом нет насущной потребности. Также ожидается, что наш код будет взаимодействовать с содержимым. Если механизм, порождающий особенности, устроен просто, преимущества, обеспечиваемые коммуникациями, превышают затраты, связанные с эксплуатацией механизма.

Материал этой главы описывает два шаблона, призванных придавать некие особенности объектам. Этим шаблонам присущи различные соотношения “выгода-цена”. Во многих случаях затраты достаточно малы, чтобы негативно влиять на положительный эффект выразительности, связанный с их применением.

## Шаблон **Singleton**<sup>1</sup>

Шаблон **Singleton** устроен очень просто. Его функционирование демонстрируется с помощью тестового случая, код которого приводится в листинге 16.1. Первая тестовая функция демонстрирует технологию доступа к экземпляру **Singleton** с помощью общедоступного статического метода **Instance**. Здесь также показано, что, если метод **Instance** вызывается несколько раз, возвращается ссылка на один и тот же интерфейс. Во втором тестовом случае показано, что если класс **Singleton** не включает общедоступные конструкторы, невозможно создать экземпляр, не воспользовавшись методом **Instance**.

<sup>1</sup>[GOF95], с. 127.

---

**Листинг 16.1. Тестовый случай для шаблона Singleton**

---

```
import junit.framework.*;
import Java.lang.reflect.Constructor;

public class TestSimpleSingleton extends TestCase
{
 public TestSimpleSingleton(String name)
 {
 super(name);
 }

 public void testCreateSingleton()
 {
 Singleton s = Singleton.Instance();
 Singleton s2 = Singleton.Instance();
 assertEquals(s, s2);
 }

 public void testNoPublicConstructors() throws Exception
 {
 Class singleton = Class.forName("Singleton");
 Constructor[] constructors = singleton.getConstructors();
 assertEquals("public constructors.", 0, constructors.length);
 }
}
```

---

Этот тестовый случай представляет собой спецификацию шаблона Singleton. Логическим следствием этого тестового случая является код, приведенный в листинге 16.2. Этот код очень прост для понимания, поскольку он является ничем иным, как экземпляром класса Singleton, который находится в области действия статической переменной Singleton.theInstance.

---

**Листинг 16.2. Реализация класса Singleton**

---

```
public class Singleton
{
 private static Singleton theInstance = null;
 private Singleton() {}

 public static Singleton Instance()
 {
 if (theInstance == null)
 theInstance = new Singleton();
 return theInstance;
 }
}
```

---

## Преимущества шаблона Singleton

- **Перекрестные платформы.** Подходящее промежуточное ПО (например, RMI) позволяет расширять возможности класса `Singleton` таким образом, что обеспечивается его использование на многих JVM, а также на различных типах компьютеров.
- **Применимость к любому классу.** Можно преобразовать любой класс таким образом, что будет создан класс `Singleton`. Для этого достаточно преобразовать конструкторы класса так, чтобы они стали частными, а также добавить соответствующие статические функции и переменные.
- **Возможность создания производного класса.** При наличии какого-либо класса можно создать его подкласс, который и будет соответствовать шаблону `Singleton`.
- **“Ленивое” развитие.** Если класс `Singleton` никогда не использовался, он и не будет создан.

## Недостатки шаблона Singleton

- **Неопределенное разрушение.** Не существует однозначного способа, позволяющего разрушить или зарезервировать класс `Singleton`. Если был добавлен метод `decommission`, обнуляющий `outTheInstance`, другие модули системы могут сохранять установленную ссылку на экземпляр `Singleton`. Последовательные вызовы `Instance` приводят к созданию других экземпляров, причем в этом случае могут одновременно существовать два экземпляра. Эта проблема особенно остро стоит в C++, где экземпляр объекта *может уничтожаться*, приводя к возможному “разыменованию” устраниемоего объекта.
- **Отсутствие наследования.** Класс, наследуемый от `Singleton`, не будет обладать свойствами исходного класса. Если требуется, чтобы он имел тип `Singleton`, необходимо добавить статическую функцию и переменную.
- **Эффективность.** Каждый вызов `Instance` сопровождается вызовом конструкции `if`. При большинстве вызовов конструкция `if` бесполезна.
- **Непрозрачность.** Пользователи `Singleton` знают о том, что работают с этим классом, поскольку им приходится вызывать метод `Instance`.

## Пример использования шаблона Singleton

Предположим, что мы имеем дело с Web-системой, которая открывает доступ пользователям к защищенным областям Web-сервера. Подобные системы снабжены базами данных, включающими имена пользователей, пароли, а также описания других пользовательских атрибутов. Более того, предположим, что доступ к базе

данных осуществляется с помощью API от независимых производителей. Можно получить доступ к базе данных непосредственно из каждого модуля, который должен быть написан (и прочтен) пользователем. В результате API от независимых производителей будут “рассеяны” по всему коду, в результате чего не останется места для соглашений о структуре или методе доступа.

Один из лучших способов, используемых в данном случае, представляет шаблон *Facade*, а также класс *UserDatabase*, поддерживающий методы, которые применяются для чтения и записи объектов *User*. Эти методы обеспечивают доступ API от независимых производителей к базе данных, выполняя операцию трансляции между объектами *User* и таблицами (строками) базы данных. Находясь в *UserDatabase*, можно устанавливать соглашения о структуре и доступе. Например, можно сделать так, что ни одна из записей *User* не будет фиксироваться до тех пор, пока не указано имя пользователя. Либо можно реализовать последовательную форму доступа к записи *User*, в результате чего исключается возможность одновременного чтения и записи для двух модулей.

Код, приведенный в листингах 16.3 и 16.4, демонстрирует шаблон *Singleton* в действии. Класс *Singleton* в данном случае называется *UserDatabaseSource*. Он реализует интерфейс *UserDatabase*. Обратите внимание, что статический метод *instance()* не включает традиционную конструкцию *if*, предохраняющую от создания нескольких экземпляров. Вместо этого используются возможности инициализации, предоставляемые Java.

---

#### Листинг 16.3. Интерфейс *UserDatabase*

---

```
public interface UserDatabase
{
 User readUser(String userName);
 void writeUser(User user);
}
```

---

---

#### Листинг 16.4. *UserDatabaseSource* *Singleton*

---

```
public class UserDatabaseSource implements UserDatabase
{
 private static UserDatabase theInstance =
 new UserDatabaseSource();

 public static UserDatabase instance()
 {
 return theInstance;
 }

 private UserDatabaseSource()
```

---

```

public User readUser(String userName)
{
 // реализация
 return null; // осталось скомпилировать
}

public void writeUser(User user)
{
 // реализация
}

```

---

Итак, вы ознакомились с примером максимально обобщенного использования шаблона Singleton. Применение этого примера на практике гарантирует, что доступ к базе данных осуществляется через единственный экземпляр `UserDatabaseSource`. В результате облегчается установка проверок, счетчиков и “замков” в `UserDatabaseSource`, который формирует упомянутые ранее соглашения по доступу и структуре.

## Шаблон Monostate<sup>2</sup>

Другой метод установки особенностей объектов обеспечивает шаблон Monostate. Но он использует совершенно другой рабочий механизм. Наглядная демонстрация работы этого механизма производится в тестовом случае Monostate, код которого приводится в листинге 16.5.

Первая тестовая функция просто описывает объект, для переменной `x` которого может устанавливаться и считываться значение. Во втором тестовом случае показаны два экземпляра одного и того же класса, которые ведут себя как *единое целое*. Если переменной `x`, соответствующей одному экземпляру, было присвоено то или иное значение, можно считывать это значение, получив доступ к переменной `x` из другого экземпляра. Получается, что два экземпляра являются носителями различных имен одного и того же объекта.

---

### Листинг 16.5. Тестовый случай Monostate

---

```

import junit.framework.*;

public class TestMonostate extends TestCase
{
 public TestMonostate(String name)
 {
 super(name);
 }

```

---

<sup>2</sup>[BALL2000].

```
public void testInstance()
{
 Monostate m = new Monostate();
 for (int x = 0; x<10; x++)
 {
 m.setX(x);
 assertEquals(x, m2.getX());
 }
}

public void testInstancesBehaveAsOne()
{
 Monostate m1 = new Monostate();
 Monostate m2 = new Monostate();

 for (int x = 0; x<10; x++)
 {
 m1.setX(x);
 assertEquals(x, m2.getX());
 }
}
```

Если вы включили класс `Singleton` в этот тестовый случай и заменили конструкциями `new Monostate` вызовы `Singleton.Instance`, тестовый случай по-прежнему будет выполняться. Это связано с тем, что данный тестовый случай описывает *поведение Singleton*, не учитывая ограничения единственного экземпляра!

Каким же образом два экземпляра могут вести себя так, как будто речь идет об одном объекте? В данном случае это означает то, что два объекта должны совместно использовать одни и те же переменные. Это легко достигается в том случае, если все переменные будут статическими. Листинг 16.6 демонстрирует реализацию `Monostate`, который проходит указанный выше тестовый случай. Обратите внимание, что переменная `itsX` является статической. Также обратите внимание на то, что *ни один из методов не является статическим*. Это обстоятельство играет важную роль в дальнейшем.

---

#### Листинг 16.6. Реализация шаблона Monostate

---

```
public class Monostate
{
 private static int itsX = 0;
 public Monostate() {}

 public void setX(int x)
 {
 itsX = x;
```

```
public int getX()
{
 return itsX;
}
```

---

Я рассматриваю приведенный код как превосходный пример “переплетенного” шаблона. Независимо от количества создаваемых экземпляров `Monostate`, все они ведут себя подобно *одному объекту*. Можно даже уничтожать или резервировать все текущие экземпляры, причем это не приводит к потере данных.

Также обратите внимание на различия между шаблонами, сказывающиеся на их поведении и структуре. Шаблон `Singleton` направлен на формирование структуры особенностей. Он предотвращает возможность создания более одного экземпляра. Ну а шаблон `Monostate` влияет на *особое поведение*, не учитывая структурные ограничения. В целях выделения этого различия отметим, что тестовый случай `Monostate` приемлем для класса `Singleton`, но тестовый случай `Singleton` завершается сбоем в случае с классом `Monostate`.

## Преимущества шаблона `Monostate`

- **Прозрачность.** Пользователи шаблона `Monostate` ведут себя подобно пользователям обычного объекта. Им вовсе не требуется знать, что данный объект называется `Monostate`.
- **Способность к наследованию.** Производные классы `Monostate` также имеют тип `Monostate`. Более того, все производные объекты `monostate` являются частью *одного и того же* объекта `Monostate`. Они используют совместно одни и те же статические переменные.
- **Полиморфизм.** Поскольку методы `Monostate` не являются статическими, они могут перекрываться производными методами. Различные производные методы характеризуются разными типами поведения на базе одного и того же набора статических переменных.
- **Хорошо определенные процессы создания и разрушения.** Переменные из класса `Monostate`, будучи статическими, включают хорошо определенные процессы создания и разрушения.

## Недостатки шаблона `Monostate`

- **Невозможность преобразования.** Обычный класс не может быть преобразован в класс `Monostate` с помощью наследования.
- **Эффективность.** Класс `Monostate` может “пережить” множество созданий и разрушений, поскольку он является реальным объектом. Эти операции часто сопряжены со значительными накладными расходами.

- **Присутствие.** Переменные класса `Monostate` требуют наличия свободного пространства, даже если `Monostate` не используется.
- **Локальная платформа.** Невозможно использовать `Monostate` на нескольких экземплярах JVM или нескольких платформах.

## Пример использования шаблона Monostate

Рассмотрим реализацию простой машины, имеющей конечное количество состояний. Эта машина моделирует турникет в метро, а ее схема приведена на рис. 16.1. Начальное состояние турникета — `Locked`. После того как в приемник бросается жетон, он переходит в состояние `Unlocked`, разблокирует пропускное устройство, отменяет режим охраны, а также пересыпает жетон в ящик приемника. После того как пассажир проходит через турникет, он переходит обратно в состояние `Locked`, а также блокирует пропускное устройство.

При этом могут возникать два аномальных состояния. Если пассажир бросает два или большее количество жетонов, но не проходит через турникет, лишние жетоны возвращаются обратно, а пропускное устройство не блокируется. Если же пассажир хочет пройти, не оплатив свой проезд, звучит сигнал, а пропускное устройство остается заблокированным.

Тестовая программа, моделирующая описанную ситуацию, приведена в листинге 16.7. Обратите внимание, что тестовые методы предполагают, что `Turnstile` относится к классу `Monostate`. Предполагается возможность отсылать события, а также получать запросы из различных экземпляров. Это имеет смысл, если существует не более одного экземпляра `Turnstile`.

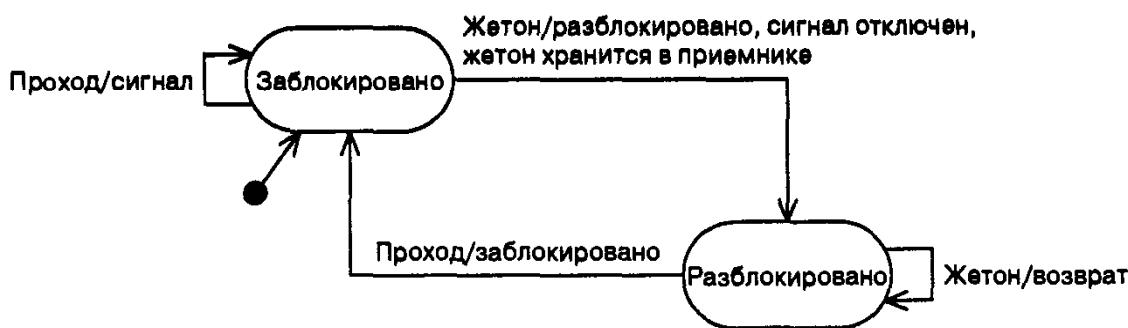


Рис. 16.1. Машина с конечными состояниями, моделирующая турникет в метро

---

### Листинг 16.7. TestTurnstile

```
import junit.framework.*;

public class TestTurnstile extends TestCase
{
 public TestTurnstile(String name)
```

```
super(name);
}

public void setUp()
{
 Turnstile t = new Turnstile();
 t.reset();
}

public void testInit()
{
 Turnstile t = new Turnstile();
 assert(t.locked());
 assert(!t.alarm());
}

public void testCoin()
{
 Turnstile t = new Turnstile();
 t.coin();
 Turnstile tl = new Turnstile();
 assert(!tl.locked());
 assert(!tl.alarm());
 assertEquals(1, tl.coins());
}

public void testCoinAndPass()
{
 Turnstile t = new Turnstile();
 t.coin();
 t.pass();

 Turnstile tl = new Turnstile();
 assert(tl.locked());
 assert(!tl.alarm());
 assertEquals("coins", 1, tl.coins());
}

public void testTwoCoins()
{
 Turnstile t = new Turnstile();
 t.coin();
 t.coin();

 Turnstile tl = new Turnstile();
 assertEquals("unlocked", !tl.locked());
 assertEquals("coins", 1, tl.coins());
 assertEquals("refunds", 1, tl.refunds());
 assert(!tl.alarm());
}
```

```
public void testPass()
{
 Turnstile t = new Turnstile();
 t.pass();
 Turnstile tl = new Turnstile();
 assert("alarm", tl.alarm());
 assert("locked", tl.locked());
}

public void testCancelAlarm()
{
 Turnstile t = new Turnstile();
 t.pass();
 t.coin();
 Turnstile tl = new Turnstile();
 assert("alarm", !tl.alarm());
 assert("locked", !tl.locked());
 assertEquals("coin", 1, tl.coins());
 assertEquals("refund", 0, tl.refunds());
}

public void testTwoOperations()
{
 Turnstile t = new Turnstile();
 t.coin();
 t.pass();
 t.coin();
 assert("unlocked", !t.locked());
 assertEquals("coins", 2, t.coins());
 t.pass();
 assert("locked", t.locked());
}
```

Реализация `Turnstile`, применяющая моносостояния (monostate), приводится в листинге 16.8. Базовый класс `Turnstile` делегирует две событийные функции (`coin` и `pass`) двум производным классам `Turnstile` (`Locked` и `Unlocked`), которые представляют машину с конечными состояниями.

---

#### Листинг 16.8. Класс `Turnstile`

---

```
public class Turnstile
{
 private static boolean isLocked = true;
 private static boolean isAlarming = false;
 private static int itsCoins = 0;
 private static int itsRefunds = 0;
 protected final static Turnstile LOCKED = new Locked();
 protected final static Turnstile UNLOCKED = new Unlocked();
 protected static Turnstile itsState = LOCKED;
```

```
public void reset()
{
 lock(true);
 alarm(false);
 itsCoins = 0;
 itsRefunds = 0;
 itsState = LOCKED;
}

public boolean locked()
{
 return isLocked;
}

public boolean alarm()
{
 return isAlarming;
}

public void coin()
{
 itsState.coin();
}

public void pass()
{
 itsState.pass();
}

protected void lock(boolean shouldLock)
{
 isLocked = shouldLock;
}

protected void alarm(boolean shouldAlarm)
{
 isAlarming = shouldAlarm;
}

public int coins()
{
 return itsCoins;
}

public int refunds()
{
 return itsRefunds;
}

public void deposit()
```

```
{
 itsCoins++;
}

public void refund()
{
 itsRefunds++;
}
}

class Locked extends Turnstile
{
 public void coin()
 {
 itsState = UNLOCKED;
 lock(false);
 alarm(false);
 deposit();
 }

 public void pass()
 {
 alarm(true);
 }
}

class Unlocked extends Turnstile
{
 public void coin()
 {
 refund();
 }

 public void pass()
 {
 lock(true);
 itsState = LOCKED;
 }
}
```

В представленном примере демонстрируются некоторые полезные свойства шаблона Monostate. Здесь также используется преимущество, вытекающее из способности производных классов Monostate быть полиморфными, а также тот факт, что производные классы Monostate сами имеют отношение к Monostates. Этот пример также демонстрирует трудности, с которыми порой сопряжено включение класса Monostate в обычный класс. Структура применяемого в этом случае решения очень зависит от природы Monostate объекта Turnstile. И если потребуется контролировать более одного турникета, использу-

зая описанную машину конечных состояний, потребуется выполнить значительный объем работ по рефакторингу для данного кода.

Возможно, у вас вызвало удивление необычный характер использования наследования в рассматриваемом примере. Поскольку `Unlocked` и `Locked` наследуются из `Turnstile`, это выглядит как нарушение стандартных принципов объектно-ориентированного программирования. Но поскольку `Turnstile` относится к `Monostate`, отсутствуют отдельные экземпляры этого объекта. Поэтому `Unlocked` и `Locked` не являются фактически отдельными объектами. В то же время они представляют собой часть абстракции `Turnstile`, поэтому `Unlocked` и `Locked` получают доступ к тем же самым переменным и методам, что и `Turnstile`.

## Резюме

Часто бывает необходимо устанавливать ограничение, которое позволяет для того или иного объекта создавать единственную копию. В этой главе были рассмотрены две весьма различные методики. Шаблон `Singleton` обеспечивает использование частных конструкторов, статических переменных и функций в целях контроля и ограничения создания экземпляров. Шаблон `Monostate` просто придает всем переменным объекта статический характер.

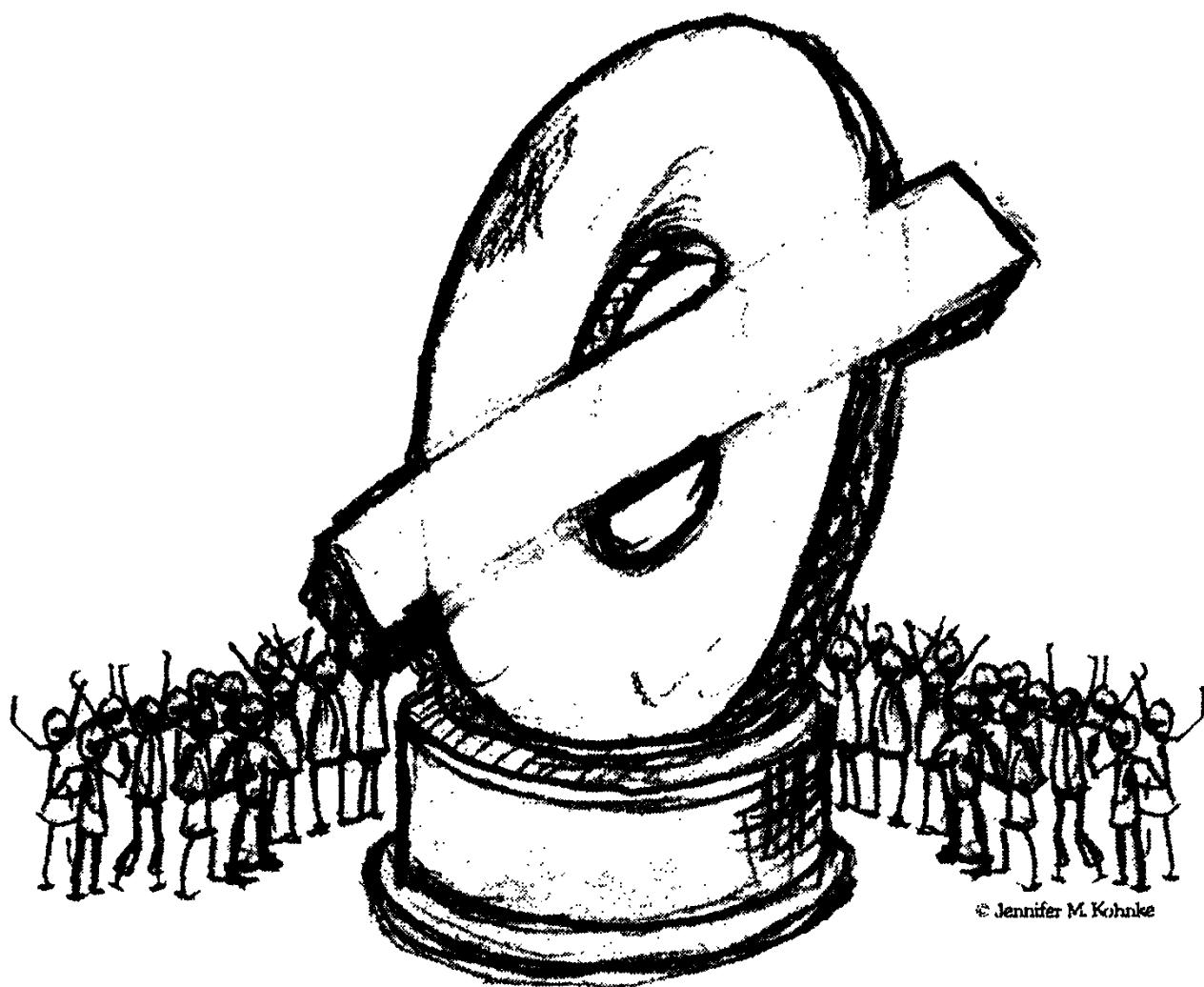
Шаблон `Singleton` лучше всего использовать в том случае, если требуется ограничить существующий класс через наследование. В этом случае также ставится задача предотвратить доступ пользователя, вызывающего метод `Instance()`. Шаблон `Monostate` идеален в том случае, если требуется “особую” природу класса сделать “прозрачной” для пользователей, или когда требуются полиморфные производные классы для одного объекта.

## Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley. 1995.
2. Martin Robert C. и др. *Pattern Languages of Program Design 3*. Reading. MA: Addison-Wesley, 1998.
3. Ball S., and Crawtbrd J. *Monostate Classes: The Power of One*. Published in *More C++ Gems*. compiled by Robert C. Martin. Cambridge. UK: Cambridge University Press, 2000.

# 17

## Объект Null



© Jennifer M. Kohnke

Безошибочная некорректность, холод простоты, превосходный нуль.  
Мертвое совершенство и ничего более.

---

Альфред Теннисон

Обратите внимание на следующий код.

```
Employee e = DB.getEmployee("Bob");
if (e != null && e.isTimeToPay(today))
 e.pay();
```

В данном случае базе данных отсылается запрос на получение объекта `Employee`, именуемого "Bob". Объект `DB` возвращает `null` в случае, если подобный объект не существует. Если же объект существует, возвращается запрошенный экземпляр `Employee`. Если в базу данных внесена информация об этом работнике и пришло время выплачивать ему зарплату, в этом случае вызывается метод `pay`.

В этом случае напрашивается идея написать фрагмент кода, подобный указанному выше. Суть общего закона заключается в том, что, поскольку первая часть выражения `&&` оценивается в C-подобном языке, оценивание второй части выполняется только в том случае, если значение первой части равно `true`. Большинство из нас также забывает выполнить тестирование с возвратом значения `null`. Результатом подобного подхода является появление ряда ошибок в дальнейшем.

Можно устранить тенденцию, грозящую возникновением ошибок, если возвращать исключение потока `DB.getEmployee`, а не `null`. Однако в этом случае работа с блоками `try/catch` может быть даже труднее, чем проверка на наличие `null`. И что еще хуже, так это то, что применение исключений приводит к необходимости их объявления в конструкциях `throws`. А это, в свою очередь, затрудняет изменение исключений в существующих приложениях.

Перечисленные проблемы могут быть успешно устраниены, если воспользоваться шаблоном `Null Object`<sup>1</sup>. Благодаря применению этого шаблона часто исключается потребность в реализации проверки на наличие `null`, в результате чего достигается упрощение программного кода.

На рис. 17.1 демонстрируется пример соответствующей структуры. Здесь `Employee` выступает в качестве интерфейса, который включает две реализации. Обычной является реализация `Employee-Implementation`. Она включает все методы и переменные, которые только могут быть у объекта `Employee`. Как только `DB.getEmployee` находит запись о работнике в базе данных, возвращается экземпляр `EmployeeImplementation`. Объект `NullEmployee` возвращается только в том случае, если `DB.getEmployee` не может найти запись о работнике.

Объект `NullEmployee` реализует все методы `Employee`, заключающиеся в "ничегонеделании". Каким же образом это зависит от метода? Например, пусть ожидается, что `isTimeToPay` реализован таким образом, что возвращается `false`, если не производятся платежи `NullEmployee`.

---

<sup>1</sup>[PLOPD3], с. 5. Эта восхитительная статья, написанная Бобби Вульфом (Bobby Woolf), переволнена шутками, иронией, а также включает конкретные практические советы.

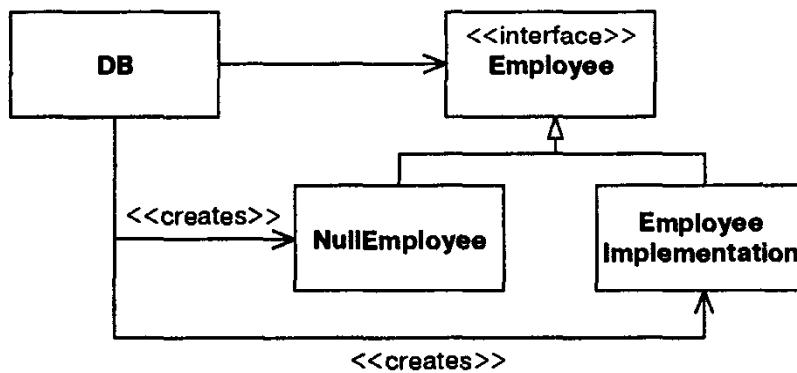


Рис. 17.1. Шаблон Null Object

Используя шаблон, можно изменить исходный программный код следующим образом:

```
Employee e = DB.getEmployee("Bob");
if (e.isTimeToPay(today))
 e.pay();
```

В этом случае “ужасные ошибки” не наблюдаются. Более того, имеет место радующая глаз совместимость. Объект `DB.getEmployee` всегда возвращает экземпляр `Employee`. Этот экземпляр ведет себя предсказуемым образом, независимо от того, была ли найдена запись о работнике.

Конечно, возникают ситуации, когда мы по-прежнему хотим знать о том, был ли сбой при выполнении `DB.getEmployee` в процессе поиска записи о работнике. Эта проблема решается путем создания переменной `static final` в `Employee`, которая хранит один и только один экземпляр `NullEmployee`.

В листинге 17.1 демонстрируется тестовый случай для `NullEmployee`. В базе данных отсутствует запись о работнике по имени “Bob”. Обратите внимание на то, что в данном тестовом случае ожидается, что `isTimeToPay` будет возвращать значение `false`. Также предполагается, что запись о работнике возвращается `DB.getEmployee` в `Employee.NULL`.

---

#### Листинг 17.1. TestEmployee.java (частичная реализация)

---

```
public void testNull() throws Exception
{
 Employee e = DB.getEmployee("Bob");
 if (e.isTimeToPay(new Date()))
 fail();
 assertEquals(Employee.NULL, e);
}
```

---

Код класса `DB` приводится в листинге 17.2. Обратите внимание, что в условиях нашего теста метод `getEmployee` просто возвращает `Employee.NULL`.

**Листинг 17.2. DB.java**

```
public class DB
{
 public static Employee getEmployee(String name)
 {
 return Employee.NULL;
 }
}
```

---

Код интерфейса Employee показан в листинге 17.3. Обратите внимание на то, что он включает статическую переменную Null, которая содержит анонимную реализацию Employee. Эта анонимная реализация представляет единственный экземпляр “нулевого” работника. В результате его выполнения isTimeToPay возвращает false, следовательно платеж не производится.

---

**Листинг 17.3. Employee.java**

```
import java.util.Date;
public interface Employee
{
 public boolean isTimeToPay(Date payDate);

 public void pay();

 public static final Employee NULL = new Employee()
 {
 public boolean isTimeToPay(Date payDate)
 {
 return false;
 }

 public void pay()
 {
 }
 };
}
```

---

Создание для “нулевого” работника анонимного внутреннего класса — единственный способ убедиться в существовании единственного экземпляра класса для этого работника. В данном случае не идет речь о классе NullEmployee самом по себе. Никто другой не может создавать другие экземпляры “нулевого” работника. И это хорошо, поскольку допускаются следующие синтаксические конструкции:

```
if (e == Employee.NULL)
```

Эта конструкция может быть ненадежной в случае, если возможно создание нескольких экземпляров “нулевого работника”.

## Резюме

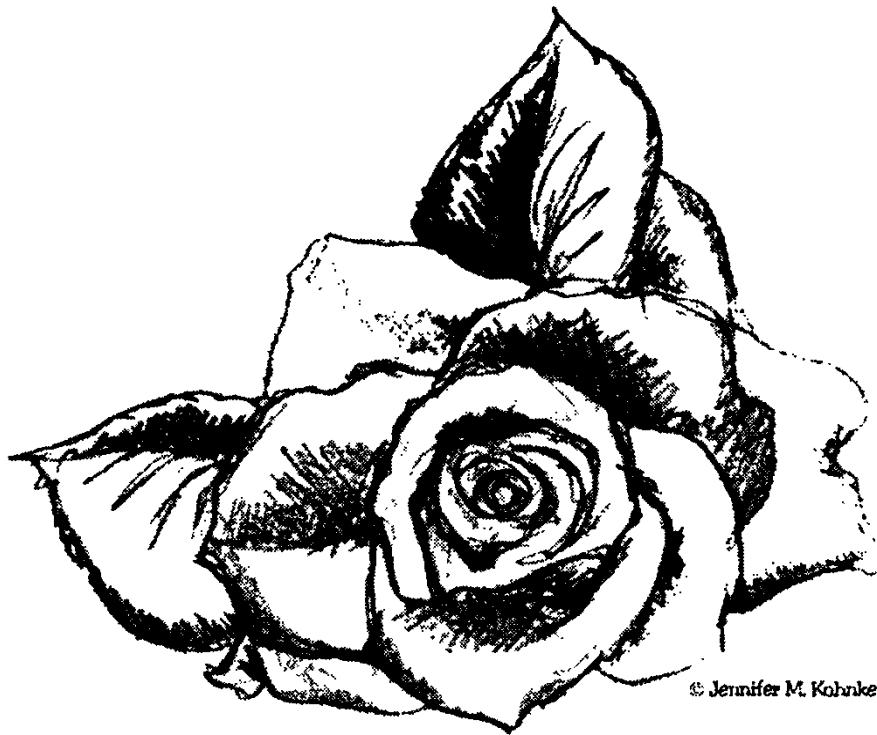
Те из нас, кто длительное время программировал на С-подобных языках, привыкли к функциям, которые возвращают `null` или 0 в случае каких-либо сбоев. Мы полагаем, что возвращаемые значения для подобных функций требуется тестировать. Шаблон `null object` приводит к изменению сложившейся ситуации. Благодаря его применению можно гарантировать возврат функциями корректных объектов, даже если возникает какой-либо сбой. Эти объекты представляют сбой или “ничегонеделание”.

## Литература

1. Martin R., Dirk R., Buschmann F. *Pattern Languages of Program Design 3*. Reading. MA: Addison-Wesley, 1998.

# 18

## Практическое занятие: программа по расчету зарплаты (первая итерация)



Все, что делается красивым, является красивым самим по себе, не требуя восхвалений со стороны восторженных поклонников.

---

Марк Аврелий

## Введение

В настоящем практическом занятии описывается первая итерация процесса разработки простой пакетной программы, выполняющей расчет зарплаты. Пользовательские истории, включенные в это практическое занятие, предельно упрощены. Например, упрощенным является алгоритм расчета налогов. Это связано с тем, что сейчас мы имеем дело с ранней версией программы. И ее коммерческая ценность составляет лишь весьма малую долю того, что на самом деле требуют заказчики.

В этой главе буде выполнен некоторый быстрый анализ, а также сеанс проектирования, которые обычно имеют место в начальный момент стандартной итерации. Заказчик просто выбрал истории для этой итерации, а теперь нам потребуется перевести их в форму, пригодную для практической реализации. Проводимые при этом сеансы проектирования будут беглыми и краткими, как и вся эта глава. Приведенные в главе UML-диаграммы являются не более чем поверхностными эскизами, набросанными на чертежной доске. Реальный рабочий проект будет рассмотрен в следующей главе (наравне с модульными тестами и примерами реализации).

## Спецификация

Следующие заметки появились в результате бесед с заказчиком. Предметом этих бесед были пользовательские истории, выбранные для этапа первой итерации.

- Некоторые сотрудники работают на почасовой основе. Для них используется почасовая ставка, которая вводится в одно из полей записи, соответствующей этому работнику. В качестве источника входных данных применяются ежедневные карточки табельного учета, в которые вносятся записи, включающие сведения о датах и количестве отработанных часов. Ставка за сверхурочные часы в полтора раза превышает обычную ставку. Выплата денег производится по пятницам.
- Некоторые работники получают фиксированную ставку. В этом случае выплата денег производится в последний рабочий день месяца. Величина месячной зарплаты хранится в одном из полей записи для данного работника.
- Некоторые из сотрудников получают комиссионные, начисленные на основе их окладов. Им приходится заполнять торговые квитанции, в которых они указывают даты и суммы заключенных сделок. Величина комиссионных указывается в одном из полей записи, соответствующей сотруднику. Все комиссионные выплачиваются по пятницам.
- Сотрудники могут сами выбирать метод платежа. Они могут получать платежные чеки, отсылаемые по избранному почтовому адресу; они могут пе-

редавать свои платежные чеки кассиру; также можно сделать так, чтобы платежные чеки переводились на банковский депозит на указанный вами счет.

- Некоторые сотрудники входят в определенные объединения (союзы). В этом случае соответствующая запись включает поле, в котором определяется величина еженедельного взноса. Причем величина взноса вычисляется на основании размера зарплаты. Помимо этого, от самого объединения может исходить инициатива относительно изменения размера взносов для отдельных членов объединения. Размеры этих взносов определяются еженедельно, а также они должны вычисляться на основе суммы следующей выплаты.
- Приложение, выполняющее расчет зарплаты, запускается по рабочим дням, в результате чего определяется величина зарплаты сотрудников, работавших в этот день. Система сообщает дату, которой закрываются выплаты сотрудникам. Благодаря этому можно вычислить суммы платежей, начиная от даты последней оплаты и завершая указанной датой.

Начало работы сопровождается генерированием схемы базы данных. В процессе решения этой проблемы используется некий вид реляционной базы данных, а сформулированные пользователем требования являются источником идей относительно типов применяемых полей и таблиц. Проектирование рабочей схемы и переход к разработке запросов не составит особого труда. В результате применения подобного подхода генерируется приложение, центральное место в котором занимает база данных.

На самом деле базы данных представляют *детали реализации*. Поэтому переход к рассмотрению самих баз данных должен осуществляться как можно позже. Достаточно много примеров приложений, которые сложным образом переплетены с базами данных только потому, что изначально повышенное внимание уделялось именно базам данных. Обратите внимание на определение абстракции: *расширение является существенным, а исключение — неуместно*. Рассмотрение баз данных неуместно на данной стадии проекта, поскольку их функции ограничиваются хранением и обеспечением доступа к базам данных.

## Анализ по методу вариантов использования

Давайте вместо организации хранения данных в системе начнем с рассмотрения поведения системы. Именно этот фактор является ключевым в деле определения ценности самой системы.

Один из методов получения сведений о поведении системы и их дальнейший анализ заключается в создании *вариантов использования*. Согласно определению Джекобсона (Jacobson), формат записи вариантов использования напоминает формат пользовательских историй, распространенных в практике экстремального программирования. Отличие варианта использования от пользовательской истории

заключается в более тщательной “проработке” деталей (в первом случае). Подобные усовершенствования имеют смысл только в том случае, если была выбрана реализация пользовательской истории в текущей итерации.

В процессе проведения анализа с помощью метода вариантов использования рассматриваются пользовательские истории и тесты приемлемости в целях поиска сигналов, генерируемых пользователями системы. Затем предпринимаются попытки формулирования ответных реакций системы на эти сигналы.

В качестве примера обратите внимание на перечень пользовательских историй, выбранных заказчиком для следующей итерации.

1. Добавление записи о новом работнике.
2. Удаление записи в случае увольнения работника.
3. Проводка табеля.
4. Проводка торговых квитанций.
5. Проводка членских взносов.
6. Изменение сведений о работнике (почасовая ставка, величина пошлины).
7. Ежедневное выполнение программы расчета зарплаты.

Ну а теперь попытаемся преобразовать каждую пользовательскую историю в проработанный вариант использования. При этом могут понадобиться дополнительные подробности, позволяющие спроектировать код таким образом, чтобы соответствовать требованиям каждой истории.

## Добавление записи о работнике

### Вариант использования 1: добавление записи о новом работнике

Факт добавления нового сотрудника подтверждается выполнением транзакции AddEmp. Эта транзакция включает имя сотрудника, адрес, а также назначенный ему табельный номер. Сама транзакция выступает в трех формах:

```
AddEmp <EmpID> "<name><address>" H <hourly-rate>
AddEmp <EmpID> "<name><address>" S <monthly-salary>
AddEmp <EmpID> "<name><address>" C <monthly-salary> <commission-rate>
```

При этом создается запись о сотруднике, поля которой получают соответствующие значения.

#### Сообщение об ошибке

**An error in the transaction structure (Ошибка в структуре транзакции)**

Если структура транзакции некорректна, выводится сообщение об ошибке, причем не предпринимаются какие-либо действия.

Существует три формы транзакции AddEmp, причем все они используют поля <EmpID>, <name> и <address>. Можно воспользоваться шаблоном Command

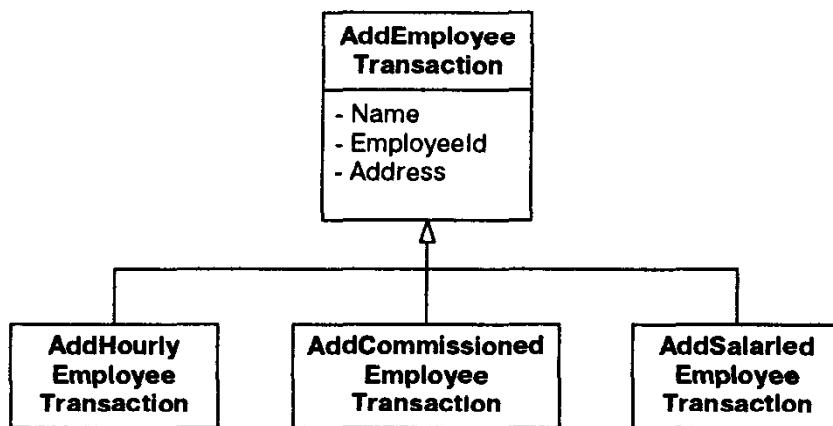


Рис. 18.1. Иерархия классов AddEmployeeTransaction

в целях создания абстрактного базового класса `AddEmployeeTransaction` с тремя производными классами: `AddHourlyEmployeeTransaction`, `AddSalariedEmployeeTransaction` и `AddCommissionedEmployeeTransaction`. (рис. 18.1.)

Описанная структура соответствует принципу персональной ответственности (SRP, Single-Responsibility Principle), поскольку каждое задание выполняется в собственном отдельном классе. Альтернативный вариант — помещение всех выполняемых заданий в единственный модуль. Хотя при этом сокращается количество классов в системе, что приводит к ее упрощению, но, в то же время весь код, выполняющий обработку транзакций, сосредотачивается в одном месте. В результате, создается большой модуль, чреватый появлением ошибок.

В первом варианте использования рассматривается запись о работнике. Ее использование автоматически ведет к появлению некоторого вида базы данных. И снова рассуждения о структуре записей или полей приводят нас к мысли о необходимости перехода к базе данных, хотя не следует поддаваться этому искушению. На самом деле, в рассматриваемом практическом случае требуется создание записи о работнике. Какова применяемая в этом случае объектная модель? Каковы функции трех различных транзакций в этом случае? Как мне кажется, эти транзакции создают три различных вида объектов, моделирующих записи о работниках, причем при этом имитируются три различных вида транзакций `AddEmp`. На рис. 18.2 демонстрируется применяемая в этом случае структура.

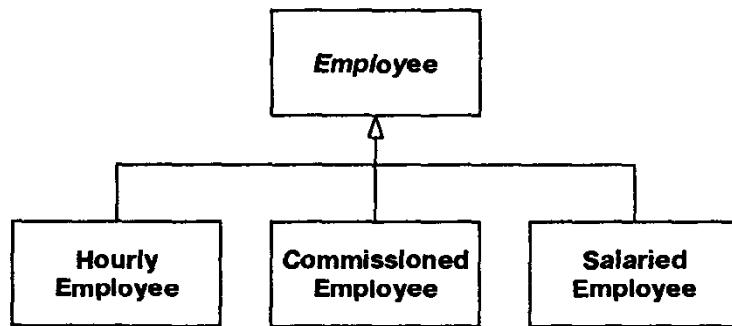


Рис. 18.2. Возможная иерархия классов Employee

## Удаление записи о работнике

### Вариант использования 2: удаление записи о работнике

Записи о сотрудниках удаляются в случае, если выполняется транзакция `DelEmp`. Эта транзакция имеет следующий вид.

`DelEmp <EmpID>`

При выполнении этой транзакции удаляется запись, соответствующего работника.

#### Сообщение об ошибке

`Invalid or unknown EmpID` (Неизвестный или некорректно указанный табельный номер работника)

Если поле `<EmpID>` структурировано некорректно или указана некорректная запись работника, транзакция выводит сообщение об ошибке, причем не предпринимаются какие-либо другие действия.

Этот вариант использования не является особо оригинальным, поэтому отложим его подробное рассмотрение на следующий раз.

## Проводка карточки табельного учета

### Вариант использования 3: проводка карточки табельного учета

В результате выполнения транзакции `TimeCard` в системе создается запись, соответствующая карточке табельного учета, которая связывается с записью соответствующего работника.

`TimeCard <EmplId> <date> <hours>`

#### Сообщение об ошибке 1

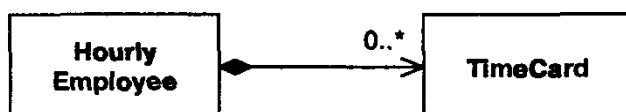
`The selected employee is not hourly` (Для выбранного работника не указан табель)

В этом случае система выводит соответствующее сообщение об ошибке и не предпринимает каких-либо действий в дальнейшем.

**Сообщение об ошибке 2****Ошибка в структуре транзакции**

Система выводит соответствующее сообщение об ошибке, не предпринимая при этом какие-либо дальнейшие действия.

При выполнении этого варианта использования некоторые транзакции применяются к определенным категориям работников, подчеркивая идею о том, что различные категории представлены разными классами. В этом случае также проявляется взаимосвязь между карточками табельного учета и работниками на почасовой оплате. На рис. 18.3 продемонстрирована возможная статическая модель, демонстрирующая подобный род взаимосвязи.



**Рис. 18.3.** Связь между торговыми квитанциями HourlyEmployee и TimeCard

**Проводка торговых квитанций****Вариант использования 4: проводка торговой квитанции**

В результате выполнения транзакции SalesReceipt система создает новую запись, соответствующую торговой квитанции, и связывает ее с работником, которому выплачиваются комиссионные.

`SalesReceipt <EmpID> <date> <amount>`

**Сообщение об ошибке 1**

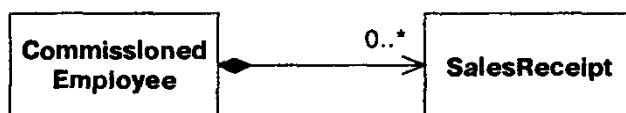
**The selected employee is not commissioned** (Выбранному сотруднику не начислены комиссионные)

В этом случае система выводит на печать соответствующее сообщение об ошибке, не предпринимая при этом какие-либо действия.

**Сообщение об ошибке 2****An error in transaction structure (Ошибка в структуре транзакции)**

Система выводит соответствующее сообщение об ошибке, не предпринимая при этом каких-либо действий.

Этот вариант использования напоминает вариант 3. Здесь появляется структура, показанная на рис. 18.4.



**Рис. 18.4.** Работники, получающие комиссионные с продаж, и торговые квитанции

## Проведение оплаты за членство в организации

### Вариант использования 5: проведение оплаты за членство в организации

В результате выполнения этой транзакции система создает запись, связанную с оплатой членства в организации, и связывает ее с кодом определенной организации-получателя платежа.

`ServiceCharge <memberID> <amount>`

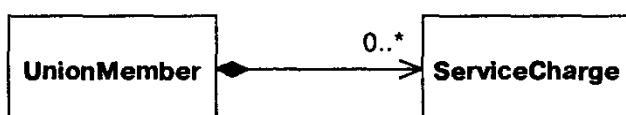
#### Сообщение об ошибке

#### Poorly formed transaction (Плохо сформированная транзакция)

Если транзакция недостаточно хорошо сформирована или `<memberID>` указывает на номер несуществующей организации, транзакция выводит на печать соответствующее сообщение об ошибке.

Этот вариант использования демонстрирует, что для членов какой-либо организации (объединения) не всегда достаточно указания табельного номера работника. Каждая организация поддерживает собственную схему нумерации для своих членов. Поэтому системой должна предоставляться возможность устанавливать связь между членами какой-то организации и прочими работниками. Существует множество способов установки такого рода взаимосвязи, поэтому во избежание возможной ошибки отложим принятие окончательного решения. Возможно, что накладываемые другими компонентами системы ограничения позволят выбрать оптимальное решение.

Лишь одно можно утверждать вполне определенно. Существует непосредственная взаимосвязь между членами организации и их издержками. На рис. 18.5 демонстрируется возможная статическая модель, иллюстрирующая эту взаимосвязь.



**Рис. 18.5.** Проводка платежей за членство в организации

## Изменение сведений о работнике

### Вариант использования 6: изменение сведений о работнике

В результате выполнения этой транзакции система изменяет сведения, указанные в записи для соответствующего работника. Существует несколько возможных вариантов этой транзакции.

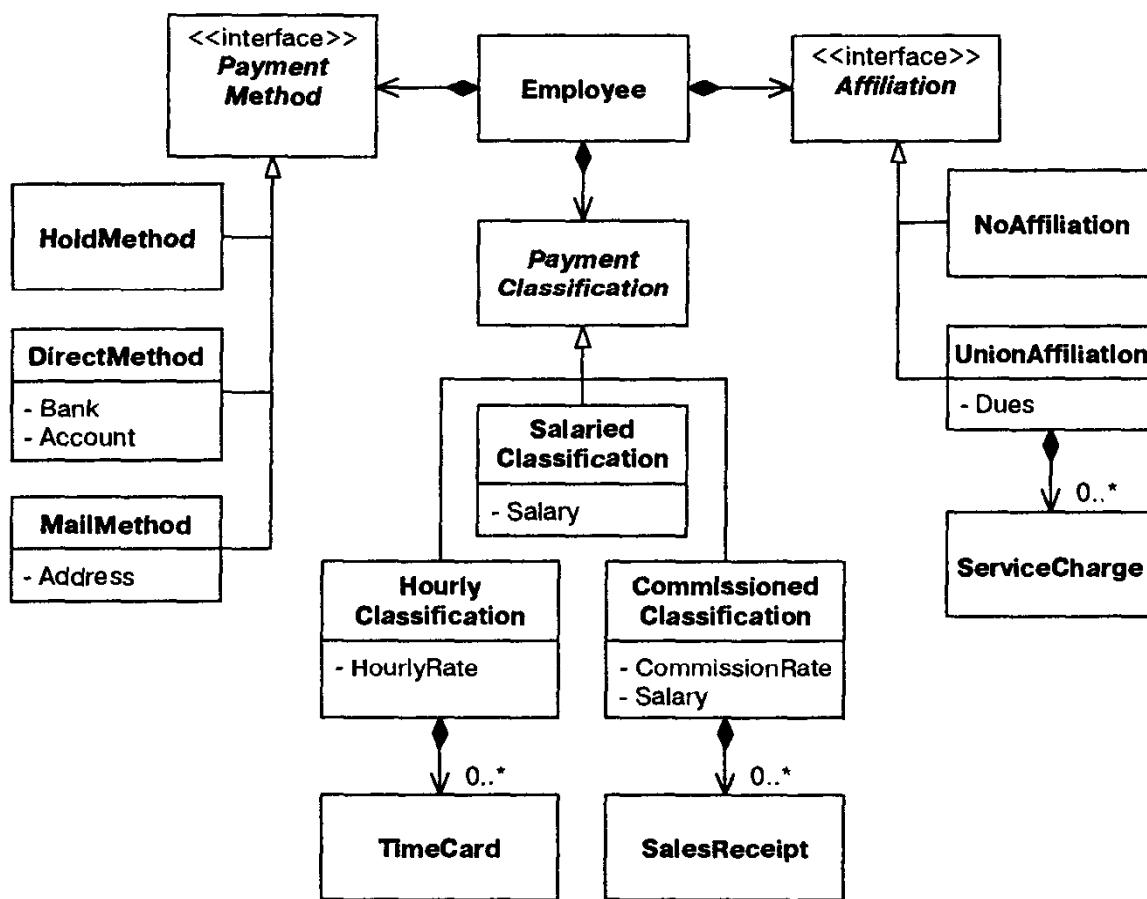
|                                              |                                                 |
|----------------------------------------------|-------------------------------------------------|
| ChgEmp <EmpID> Name <name>                   | изменение имени работника                       |
| ChgEmp <EmpID> Address <address>             | изменение адреса работника                      |
| ChgEmp <EmpID> Hourly <hourlyRate>           | изменение почасовой выработки                   |
| ChgEmp <EmpID> Salaried <salary>             | изменение величины оклада                       |
| ChgEmp <EmpID> Commissioned <salary> <rate>  | изменение величины комиссионных                 |
| ChgEmp <EmpID> Hold                          | хранение платежного чека                        |
| ChgEmp <EmpID> Direct <bank> <account>       | направление на депозит                          |
| ChgEmp <EmpID> Mail <address>                | проводка платежного чека                        |
| ChgEmp <EmpID> Member <memberID> Dues <rate> | назначение членства в организации для работника |
| ChgEmp <EmpID> NoMember                      | исключение работника из организации             |

### Сообщение об ошибке

#### Transaction Errors (Ошибки транзакции)

Если структура транзакции некорректна или идентификатор <Emp1D> не указывает на реального работника, или <member1D> всегда указывает на члена организации, выводится соответствующее сообщение об ошибке, причем не предпринимаются какие-либо действия.

Этот вариант использования весьма нагляден. Изучая его, можно легко прийти к выводу о том, что все связанные с работником аспекты могут изменяться. Тот факт, что можно переводить работника с почасовой ставки на оклад свидетельствует о том, что изображенная на рис. 18.2 диаграмма не совсем верна. Поэтому при вычислении величины зарплаты более подходящим может оказаться шаблон *Strategy*. Класс *Employee* может включать “стратегический” класс *PaymentClassification*, как показано на рис. 18.6. При этом налицо преимущество, которое заключается в том, что можно изменять объект *PaymentClassification*, не затрагивая какую-либо часть объекта *Employee*. Как только производится перевод работника с почасовой ставки на оклад, объект



**Рис. 18.6.** Пересмотренная диаграмма классов для программы расчета зарплаты: базовая модель

*HourlyClassification* соответствующего объекта *Employee* заменяется соответствующим объектом *SalariedClassification*.

Существует три версии объекта *PaymentClassification*. Объекты *HourlyClassification* поддерживают почасовую ставку, а также список объектов *TimeCard*. Объекты *SalariedClassification* поддерживают месячную тарифную сетку. Объекты *CommissionedClassification* поддерживают величины месячной зарплаты, комиссионных, а также перечень объектов *SalesReceipt*. Здесь автор воспользовался композицией взаимосвязей для рассматриваемых случаев, поскольку справедливо полагал, что объекты *TimeCards* и *SalesReceipts* должны уничтожаться после удаления записи работника.

Также следует обеспечить возможность изменения метода платежа. На рис. 18.6 представлена реализация этой идеи путем применения шаблона *Strategy*, а также с помощью генерирования трех различных типов классов *PaymentMethod*. Если метод *Employee* включает объект *MailMethod*, соответствующему работнику отсылаются платежный чек.

Адрес, по которому отсылаются чеки, фиксируется объектом *MailMethod*. Если объект *Employee* включает объект *DirectMethod*, платеж переводится непосредственно на банковский счет, который хранится объектом *DirectMethod*.

Если объект `Employee` включает объект `HoldMethod`, платежные чеки отсылаются непосредственно кассиру.

На рис. 18.6 также демонстрируется применение шаблона `Null Object` относительно членства в организации. Каждому объекту `Employee` соответствует объект `Affiliation`, выступающий в двух формах. Если объект `Employee` включает объект `NoAffiliation`, платеж не может адаптироваться в соответствии с организацией, отличной от организации-работодателя. Если же объект `Employee` включает объект `UnionAffiliation`, работник платит членские взносы, а также другие платежи, направленные на содержание организаций. При этом соответствующие записи фиксируются объектом `UnionAffiliation`.

Благодаря использованию перечисленных шаблонов достигается совместимость системы с принципом открытия-закрытия (OCP, Open-Closed Principle). Класс `Employee` закрыт для изменений метода платежа, классификации платежей, а также членских взносов. Новые методы, приемы классификации, а также членские взносы могут добавляться в систему, не затрагивая класс `Employee`.

На рис. 18.6 демонстрируется наша *основная модель* (архитектура). Эта модель представляет основу всего, что может выполнять система по расчету зарплаты. Существуют многие другие проекты и классы, относящиеся к приложению расчета зарплаты, которые играют вторичную роль в описанной нами фундаментальной структуре. Конечно, эта структура не является застывшей, поскольку может развиваться наравне с другими структурными элементами.

## Платежный день

### Вариант использования 7: выполнение программы расчета зарплаты для одного платежного дня

В результате получения транзакции `Payday` система осуществляет поиск всех работников, оплата труда которых должна быть произведена к указанной дате. Затем определяется размер оплаты, после чего производится оплата в соответствии с выбранным методом платежа.

`Payday <date>`

Несмотря на то, что понять назначение этого варианта использования не составляет особого труда, определить степень его воздействия на статическую структуру, изображенную на рис. 18.6, уже сложнее. В этом случае нужно ответить на некоторые вопросы.

Во-первых, каким образом объект `Employee` получает сведения о методах вычисления платежей? Если оплата труда работника производится на почасовой основе, система подсчитывает количество карточек табельного учета, умножая его на величину почасовой ставки. Если работник получает комиссионные, система подсчитывает торговые квитанции, выполняя умножение на величину комисси-

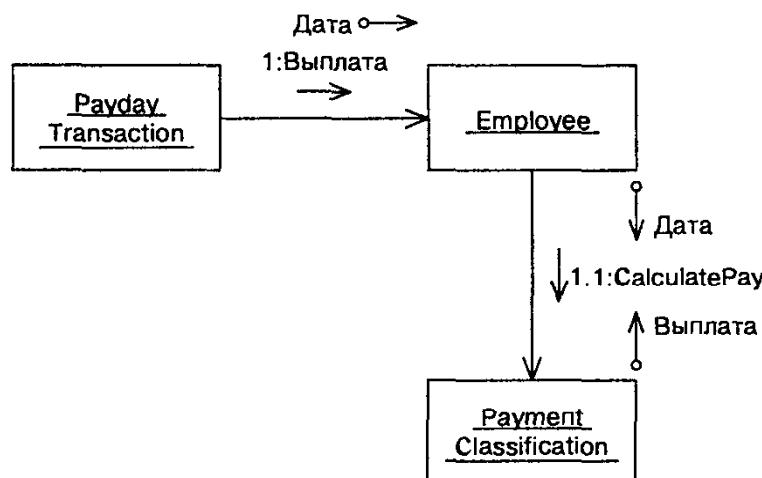


Рис. 18.7. Вычисление платежей

онной ставки, а затем добавляя полученный результат к сумме базовой зарплаты. И где же производятся все эти вычисления? Идеальным местом являются производные классы `PaymentClassification`. Эти объекты включают записи, применяемые при вычислении величины оплаты (причем могут даже включаться методы, позволяющие рассчитать сумму платежа). На рис. 18.7 показана диаграмма сотрудничества, описывающая все, о чем рассказывалось выше.

Как только объект `Employee` получает запрос на вычисление величины платежа, запрос перенаправляется объекту `PaymentClassification`. Фактически используемый в этом случае алгоритм зависит от типа объекта `PaymentClassification`, который включает объект `Employee`. На рис. 18.8–18.10 демонстрируются три возможных в этом случае сценария.

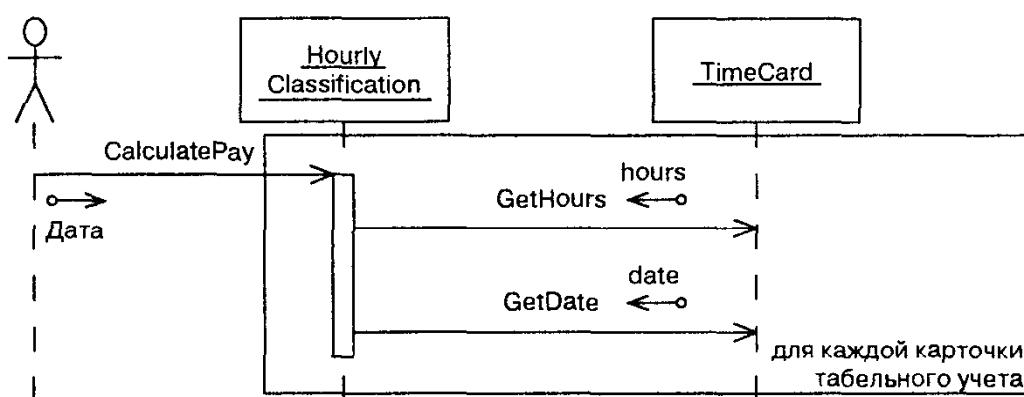
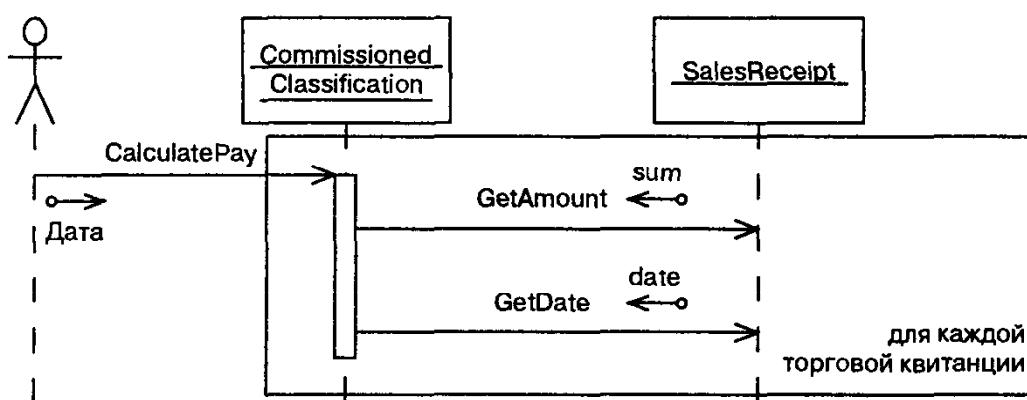
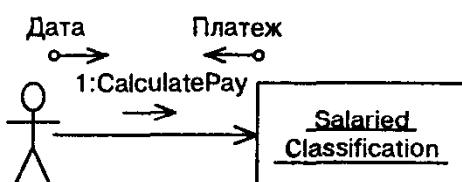


Рис. 18.8. Расчет зарплаты для работников, получающих почасовую ставку



**Рис. 18.9.** Расчет зарплаты для работников, получающих комиссионные



**Рис. 18.10.** Расчет зарплаты для работников, получающих фиксированный оклад

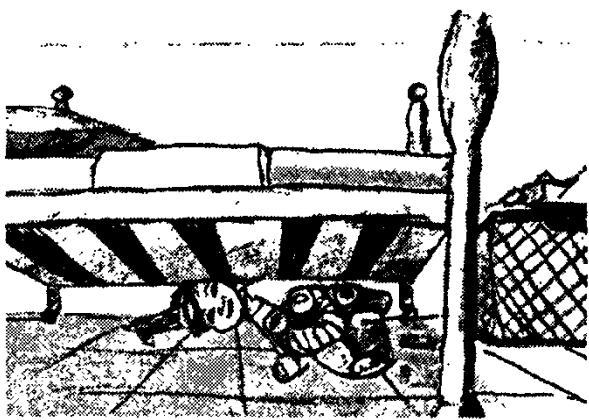
## Первые итоги

К настоящему моменту времени мы осознали, каким образом простой анализ практических случаев обеспечивает поддержку богатого набора данных, а также позволяет “заглянуть внутрь” системного проекта.

## В поисках базовых абстракций

В целях эффективного применения принципа ОСР следует “устроить охоту” на базовые абстракции, определив те из них, которые образуют основу приложения. Достаточно часто эти абстракции не формулируются и даже не упоминаются в требованиях к данному приложению или к практическим случаям. Это связано с тем, что в случае описания базовых абстракций происходит “распускание” требований и практических случаев.

Что же можно сказать относительно базовых абстракций приложения по расчету зарплаты? Давайте снова обратимся к описанию требований. В их качестве выступают такие утверждения, как “Некоторые работники трудятся почасово”, “Некоторые работники получают обычную зарплату” и “Некоторые [...] работни-



ки получают комиссионные”. Все эти высказывания можно обобщить следующим образом: “Все работники получают зарплату в соответствии с различными схемами”. Соответствующая абстракция формулируется следующим образом: “Все работники получают зарплату”. Модель класса `PaymentClassification`, приведенная на рис. 18.7–18.10, хорошо выражает основную идею рассматриваемой абстракции. Поэтому эту абстракцию можно считать определенной (наравне с нашими пользовательскими историями).

## Абстракция, связанная с календарным планированием

В процессе рассмотрения других абстракций мы обнаруживаем следующее: “выплаты осуществляются по пятницам”, “выплаты производятся в последний рабочий день месяца” и “выплаты производятся каждую вторую пятницу”. В этом случае будет уместным сделать следующее обобщение: “Все работники получают зарплату в соответствии с неким календарным графиком”. Описанная здесь абстракция выражает *календарный график*. Его наличие позволяет сформировать запрос объекту `Employee` о том, является ли текущая дата платежным днем. Именно этот факт упоминается в практических случаях. С помощью требований осуществляется связывание календарного плана работника с соответствующей классификацией платежей. Например, работники с почасовой ставкой получают деньги еженедельно, работники с фиксированным окладом получают деньги ежемесячно, а работники, схема платежей для которых предусматривает комиссионные, получают зарплату раз в две недели. Может ли описанная политика измениться таким образом, что работники получат возможность выбрать понравившийся им календарный график? Или работники, относящиеся к различным отделам и подразделениям, будут располагать различным календарным графиком? Может ли политика, определяющая календарный график, изменяться независимо от политики платежей? По крайней мере, весьма вероятно, что это именно так.

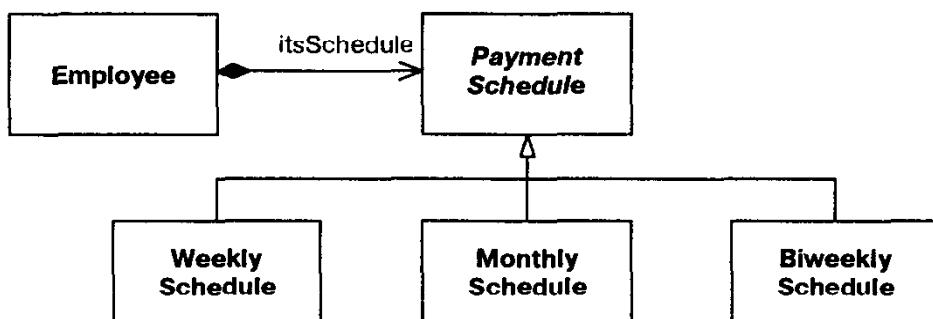
Если (как и предполагается требованиями) решение вопросов, связанных с календарным графиком, будет передано классу `PaymentClassification`, это не означает защиту класса от изменений в календарном графике. Поэтому в случае изменения политики платежей потребуется протестировать календарный график. После изменения календарных графиков следует протестировать политику платежей. В результате можно предотвратить возможное нарушение принципов ОСР и SRP.

Наличие связи между политикой определения календарного графика и платежей приводит к появлению ошибок, когда изменение в конкретной политике платежей приводит к составлению некорректного календарного графика для некоторых работников. Подобные ошибки могут в определенной степени “напрягать” программистов, а также сеять ужас в сердцах менеджеров и пользователей. Они опасаются (и совершенно справедливо), что если будут нарушены календарные графики (в случае изменения политики платежей), *любые* изменения, выполненные

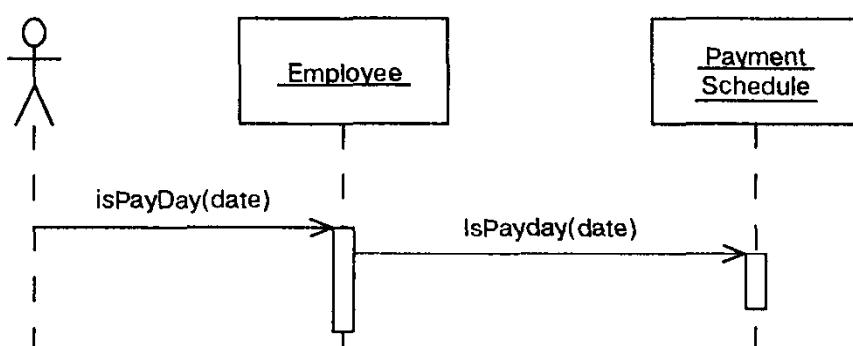
ные в любом месте, могут привести к возникновению проблем в любом другом независимом компоненте системы. Также они побаиваются того, что не смогут предвидеть все возможные эффекты, связанные с выполненными изменениями. В этом случае о конфиденциальности не может быть и речи, а сама программа получает статус “опасной и нестабильной” (по крайней мере, так о ней отзываются ее пользователи и менеджеры).

Несмотря на столь выдающуюся роль абстракции календарного планирования, применение анализа с помощью практических случаев не позволяет нам обнаружить ее существование. В этом случае может помочь тщательное изучение требований, а также консультации с пользователями будущей программы. К неприятностям может привести излишнее доверие к инструментам и процедурам, а также недоверие к знаниям и опыту специалистов.

На рис. 18.11 и 18.12 показаны статическая и динамическая модели абстракции календарного планирования. И снова используется шаблон *Strategy*. Класс *Employee* включает абстрактный класс *PaymentSchedule*. Существует три вариации класса *PaymentSchedule*, соответствующие трем известным календарным графикам, согласно которым работники получают зарплату.



**Рис. 18.11.** Статическая модель абстракции календарного планирования



**Рис. 18.12.** Динамическая модель абстракции календарного планирования

## Методы платежей

Существует другое обобщение, вытекающее на основе сформулированных требований: “Все работники получают зарплату в результате применения определенного метода платежа”. В этом случае абстракцией является класс `PaymentMethod`. Именно эта абстракция отражена на рис. 18.6.

## Взносы

Сформулированные требования приводят к логичному выводу о том, что работники платят взносы за членство в определенных организациях. В эту категорию включаются благотворительные взносы. Работники также могут автоматически отчислять взносы в профессиональные ассоциации. Обобщение в данном случае имеет следующий вид: “Работники могут отчислять взносы различным организациям, которые автоматически оплачивают представленные им платежные документы”.

Соответствующую абстракцию в этом случае представляет класс `Affiliation`, который показан на рис. 18.6. На этом рисунке не показан класс `Employee`, содержащий более одного класса `Affiliation`, а просто представлен класс `NoAffiliation`. Рассмотренный нами проект недостаточно хорошо соответствует именно той абстракции, в которой мы сейчас нуждаемся. На рис. 18.13 и 18.14 показаны статическая и динамическая модели, представляющие абстракцию `Affiliation`.

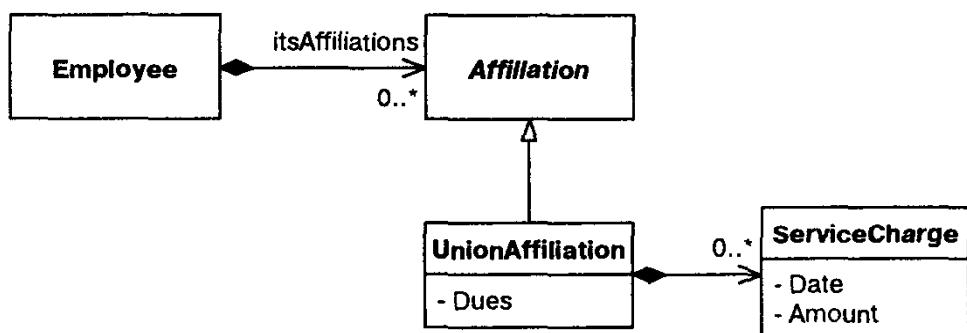


Рис. 18.13. Статическая структура абстракции `Affiliation`

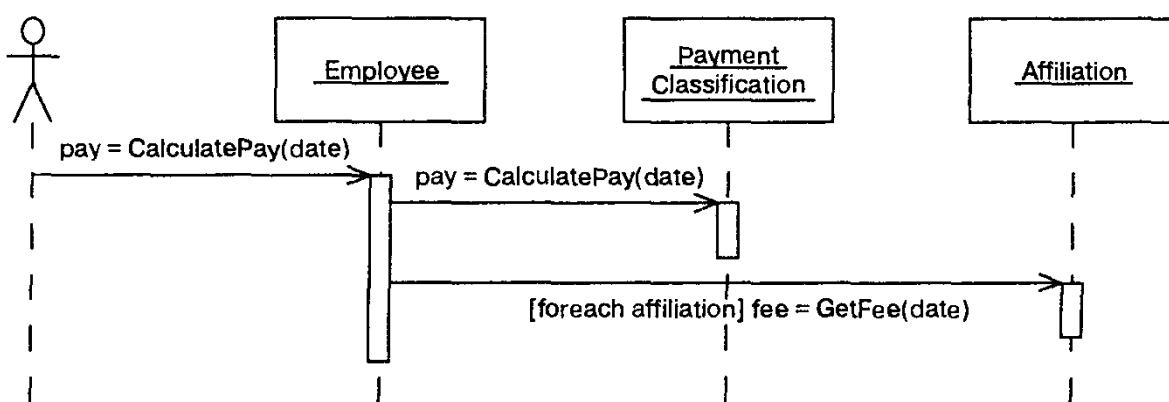


Рис. 18.14. Динамическая структура абстракции `Affiliation`

При работе с перечнем объектов `Affiliation` не используется шаблон `Null Object` для указания работников, не отчисляющих членские взносы. Если есть такие работники, соответствующий им список членских взносов будет пустым.

## Резюме

В самом начале разработки проекта участники команды разработчиков собираются вместе на совещание. Они высказывают мнения о будущем проекте, а также обговаривают пользовательские истории, выбранные на этой итерации. Подобные *сессии быстрого проектирования* обычно делятся меньше одного часа. При этом на доске остаются UML-диаграммы. Как правило, подобные диаграммы не нужно переносить на бумагу. Такой сеанс дает толчок к началу процесса обдумывания, способствует выработке общей ментальной модели, столь необходимой для начала работы. При этом не ставится цель *жесткого закрепления* проекта. Эта глава представляет собой текстуальный эквивалент сеанса быстрого проектирования.

## Литература

1. Jacobson I. *Object-Oriented Software Engineering, A Use-Case-Driven Approach*. Wokingham, England: Addison-Wesley, 1992.

# Практическое занятие: реализация программы по расчету зарплаты



Уже достаточно давно автор приступил к написанию кода, который поддерживает и проверяет выполняемые проекты. Код создавался по частям и служил иллюстрацией к соответствующим разделам текста. Рассматривая полностью сформированные части кода, не следует думать, что именно такой вид они имели изначально. Фактически, каждая часть кода десятки раз редактировалась, подвергалась многочисленным компиляциям и неоднократно тестировалась. Причем каждый раз в код вносились изменения.

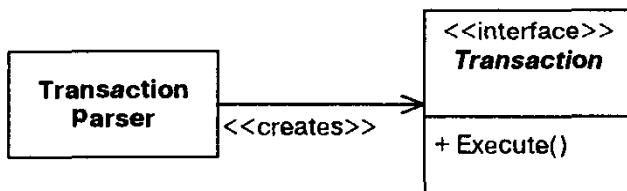


Рис. 19.1. Интерфейс Transaction

Также вам немного пришлось ознакомиться и с языком UML. Данный вариант UML можно рассматривать как метод формирования набросков диаграммы, поскольку UML является удобной средой взаимодействия между разработчиками.

На рис. 19.1 показано, что транзакции представляют абстрактный базовый класс под названием `Transaction`, включающий метод экземпляра `Execute()`. Конечно же, речь идет о шаблоне `Command`. Листинг 19.1 включает реализацию класса `Transaction`.

---

#### Листинг 19.1. Transaction.h

---

```

#ifndef TRANSACTION_H
#define TRANSACTION_H

class Transaction
{
public:
 virtual ~Transaction();
 virtual void Execute() = 0;
};

#endif

```

---

## Добавление записей о новых работниках

На рис. 19.2 демонстрируется потенциальная структура для транзакций, выполняющих добавление записей о работниках. Заметим, что при выполнении этих транзакций штатное расписание работников ассоциируется со штатной классификацией. Такой подход довольно удобен, поскольку транзакции представляют часть основной модели. Поэтому основная модель находится вне ассоциативных связей; ассоциация является лишь частью постоянно изменяемой вариации. Например, легко можно добавить транзакцию, разрешающую изменение календарных планов сотрудников.

Также заметим, что заданный по умолчанию метод начисления зарплаты предполагает проверку производимых кассиром выплат. Если при начислении жалования работнику используется другая методика, соответствующее изменение можно внести с помощью транзакции `ChgEmp`.

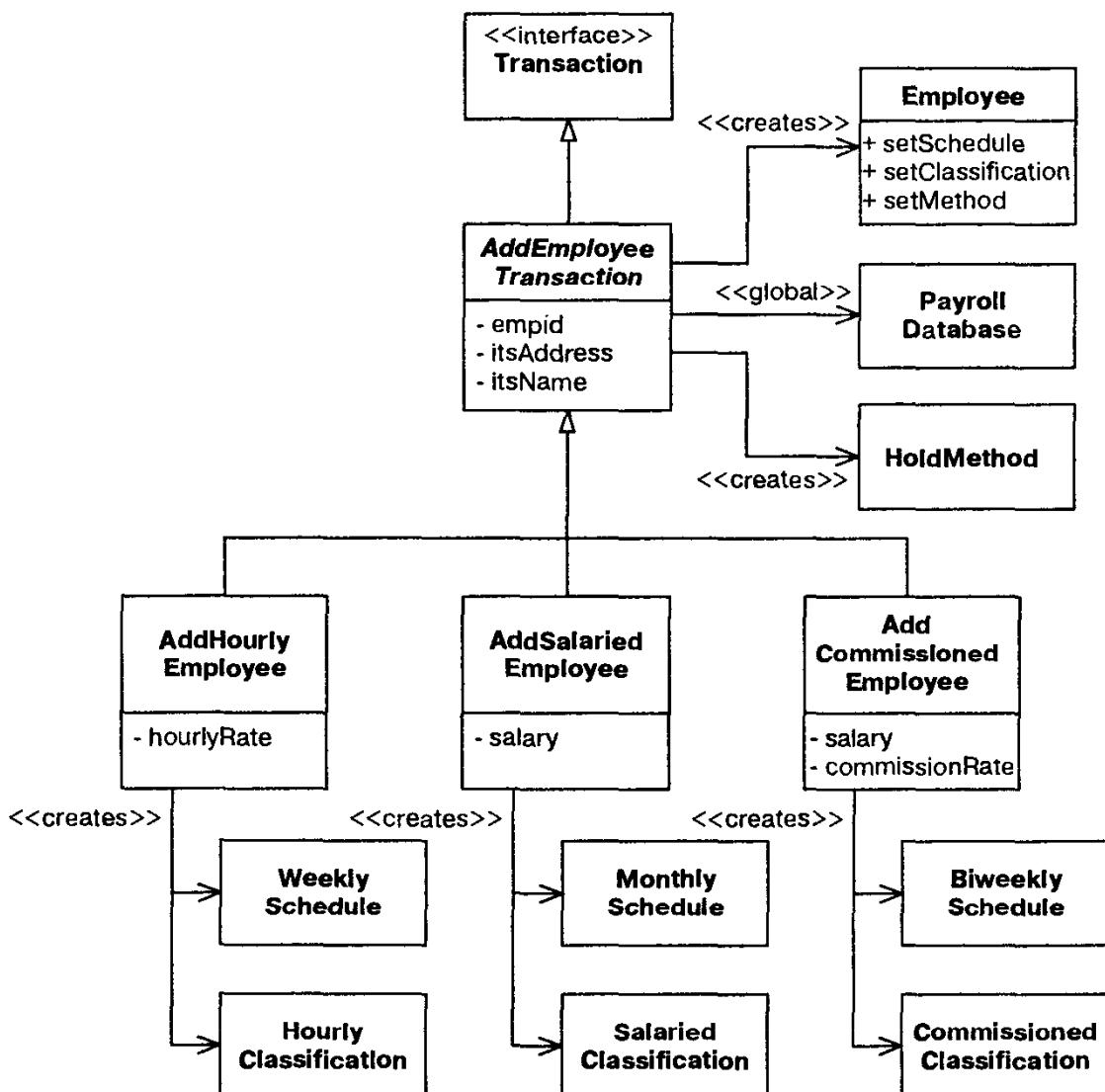


Рис. 19.2. Статическая модель AddEmployeeTransaction

Как обычно, написание кода начинается с создания тестов. Листинг 19.2 представляет тестовый случай, демонстрирующий корректную работу AddSalariedTransaction. Приведенный код успешно прошел через тестовый случай.

#### Листинг 19.2. PayrollTest::TestAddSalariedEmployee

```

void PayrollTest::TestAddSalariedEmployee()
{
 int empId = 1;
 AddSalariedEmployee t(empId, "Bob", "Home", 1000.00);
 t.Execute();

 Employee* e == GpayrollDatabase.GetEmployee(empId);
 assert("Bob" == e->GetName());

 PaymentClassification* pc = e->GetClassification();
 SalariedClassification* sc = dynamic_cast<SalariedClassification*>(pc
 assert(sc);
}

```

```

assertEquals(1000.00, sc->GetSalary(), .001);
PaymentSchedule* ps = e->GetSchedule();
MonthlySchedule* ms = dynamic_cast<MonthlySchedule*>(ps);
assert(ms);
PaymentMethod* pm = e->GetMethod();
HoldMethod* hm = dynamic_cast<HoldMethod*>(pm);
assert(hm);
}

```

---

## База данных Payroll

Класс `AddEmployeeTransaction` применяет класс под названием `PayrollDatabase`. Этот класс поддерживает все имеющиеся в наличии объекты `Employee` в `Dictionary`, контролируемые с помощью `empID`. Также поддерживается и `Dictionary`, отражающий связь `memberIDs` и `empIDs`. На рис. 19.3 представлена структура этого класса. Класс `PayrollDatabase` является примером использования шаблона `Facade`.

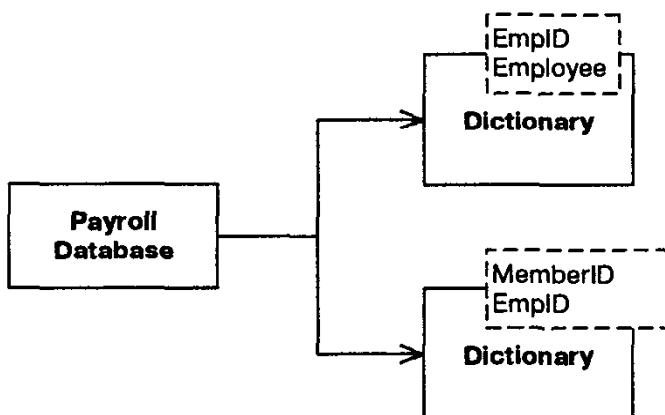


Рис. 19.3. Статическая структура PayrollDatabase

Листинги 19.3 и 19.4 иллюстрируют элементарную реализацию для `PayrollDatabase`. Эта реализация поможет в процессе прогона начальных тестовых случаев. В нее даже не входит словарь, определяющий соответствие между идентификаторами членов и экземплярами `Employee`.

---

### Листинг 19.3. PayrollDatabase.h

```

#ifndef PAYROLLDATABASE_H
#define PAYROLLDATABASE_H

#include <map>

class Employee;

class PayrollDatabase

```

```
public:
 virtual ~PayrollDatabase();
 Employee* GetEmployee(int empId);
 void AddEmployee(int empid, Employee*);
 void clear() {itsEmployees.clear();}
private:
 map<int, Employee*> itsEmployees;
};
#endif
```

---

#### Листинг 19.4. PayrollDatabase.cpp

---

```
#include "PayrollDatabase.h"
#include "Employee.h"

PayrollDatabase Gpayroll Database;

PayrollDatabase::~PayrollDatabase ()
{
}

Employee* PayrollDatabase::GetEmployee(int empid)
{
 return itsEmployees[empid];
}

void PayrollDatabase::AddEmployee(int empid,Employee* e)
{
 itsEmployees[empid] = e;
}
```

---

В общем случае реализации для базы данных рассматриваются достаточно подробно. Изучение нюансов приведенных решений откладывается на более поздний срок. Не имеет значения, используются ли они для реализации конкретной реляционной базы данных, табличных файлов или объектно-ориентированной реляционной базы данных. Теперь перейдем к созданию API, поддерживающих службы баз данных для оставшейся части приложения. Соответствующие реализации базы данных рассматриваются в дальнейшем.

Изучение деталей откладывается не столь часто, но именно этот подход заслуживает одобрения. Решения, имеющие отношение к базам данных, следует принимать в том случае, когда накопится определенный багаж знаний о разрабатываемом ПО и требованиях к нему. Также можно избежать перегрузки базы данных сведениями об инфраструктуре.

## Использование шаблона Template Method при добавлении записей о работниках

На рис. 19.4 показана динамическая модель, применяемая при добавлении записей о новых работниках. Заметим, что объект AddEmployeeTransaction для получения доступа к соответствующим объектам PaymentClassification и Payment Schedule пересыпает сообщения *самому себе*. Эти сообщения реализуются в производных модулях класса AddEmployeeTransaction. Таким образом реализуется применение шаблона Template Method.

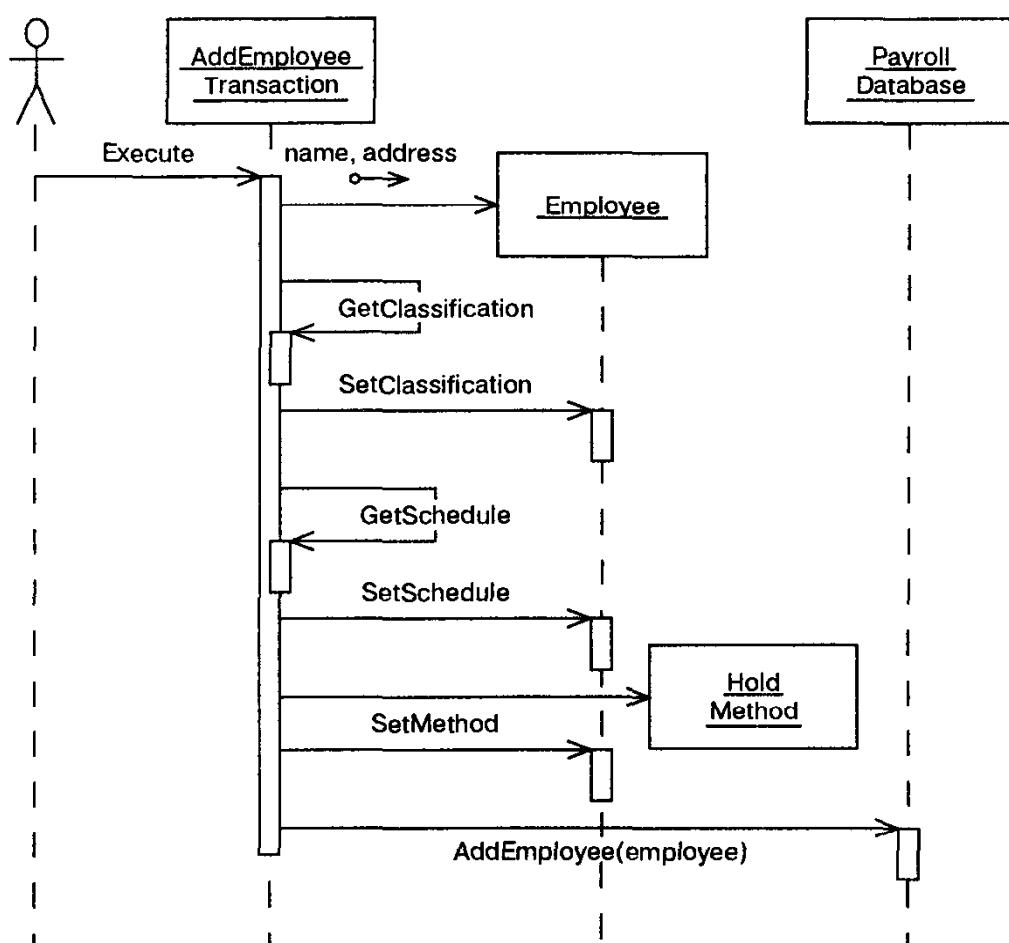


Рис. 19.4. Динамическая модель добавления записей о новых работниках

Листинги 19.5 и 19.6 демонстрируют реализацию шаблона Template Method в классе AddEmployeeTransaction. Этот класс применяет метод `Execute()` для вызова двух виртуальных функций, реализуемых с помощью производных модулей. Рассматриваемые функции, `GetSchedule()` и `GetClassification()`, возвращают объекты `PaymentSchedule` и `PaymentClassification`, которые требуются вновь создаваемому `Employee`. Затем метод `Execute()` привязывает эти объекты к `Employee` и сохраняет `Employee` в `PayrollDatabase`.

---

**Листинг 19.5. AddEmployeeTransaction.h**

---

```
#ifndef ADDEMPLOYEETRANSACTION_H
#define ADDEMPLOYEETRANSACTION_H

#include "Transaction.h"
#include <string>

class PaymentClassification;
class PaymentSchedule;
class AddEmployeeTransaction : public Transaction
{
public:
 virtual ~AddEmployeeTransaction();
 AddEmployeeTransaction(int empid, string name, string address);
 virtual PaymentClassification* GetClassification() const = 0;
 virtual PaymentSchedule* GetSchedule() const = 0;
 virtual void Execute();

private:
 int itsEmpid;
 string itsName;
 string itsAddress;
};

#endif
```

---

---

**Листинг 19.6. AddEmployeeTransaction.cpp**

---

```
#include "AddEmployeeTransaction.h"
#include "HoldMethod.h"
#include "Employee.h"
#include "PayrollDatabase.h"

class PaymentMethod;
class PaymentSchedule;
class PaymentClassification;

extern PayrollDatabase GpayrollDatabase;

AddEmployeeTransaction::~AddEmployeeTransaction()
{
}

AddEmployeeTransaction::
AddEmployeeTransaction(int empid, string name, string address)
: itsEmpid(empid)
, itsName(name)
, itsAddress(address)
```

```

}

void AddEmployeeTransaction::Execute()
{
 PaymentClassification* pc = GetClassification();
 PaymentSchedule* ps = GetSchedule();
 PaymentMethod* pm = new HoldMethod();
 Employee* e = new Employee(itsEmpid, itsName, itsAddress);
 e->SetClassification(pc);
 e->SetSchedule(ps);
 e->SetMethod(pm);
 GpayrollDatabase.AddEmployee(itsEmpid, e);
}

```

---

Листинги 19.7 и 19.8 иллюстрируют реализацию класса AddSalariedEmployee. Данный класс является производным от класса AddEmployeeTransaction и реализует методы GetSchedule() и GetClassification(), возвращающие соответствующие объекты методу AddEmployeeTransaction::Execute().

---

#### Листинг 19.7. AddSalariedEmployee.h

---

```

#ifndef ADDSALARIEDEMPLOYEE_H
#define ADDSALARIEDEMPLOYEE_H

#include "AddEmployeeTransaction.h"

class AddSalariedEmployee : public AddEmployeeTransaction
{
public:
 virtual ~AddSalariedEmployee();
 AddSalariedEmployee(int empid, string name,
 string address, double salary);
 PaymentClassification* GetClassification() const;
 PaymentSchedule* GetSchedule() const;
private:
 double itsSalary;
};

#endif

```

---



---

#### Листинг 19.8. AddSalariedEmployee.cpp

---

```

#include "AddSalariedEmployee.h"
#include "SalariedClassification.h"
#include "MonthlySchedule.h"

AddSalariedEmployee::~AddSalariedEmployee()
{

```

```

AddSalariedEmployee:::
AddSalariedEmployee(int empid, string name,
 string address, double salary)
: AddEmployeeTransaction(empid, name, address)
, itsSalary(salary)
{
}

PaymentClassification*
AddSalariedEmployee::GetClassification() const
{
 return new SalariedClassification(itsSalary);
}

PaymentSchedule* AddSalariedEmployee::GetSchedule() const
{
 return new MonthlySchedule();
}

```

В качестве упражнений для читателя осталось выполнить реализацию классов AddHourlyEmployee и AddCommissionedEmployee. Не забудьте сначала сформировать тестовые случаи.

## Удаление записей о работниках

На рис. 19.5 и 19.6 представлены статистическая и динамическая модели для транзакций, выполняющих удаление записей о работниках.

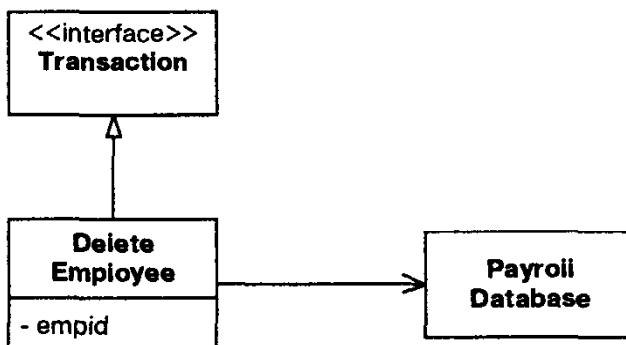


Рис. 19.5. Статическая модель транзакции DeleteEmployee

Листинг 19.9 иллюстрирует тестовый случай, реализующий удаление записей, соответствующих работникам. Листинги 19.10 и 19.11 демонстрируют реализацию класса DeleteEmployeeTransaction. Данная реализация является типичной для шаблона Command. Конструктор хранит данные, которые постепенно обрабатываются с помощью метода Execute().

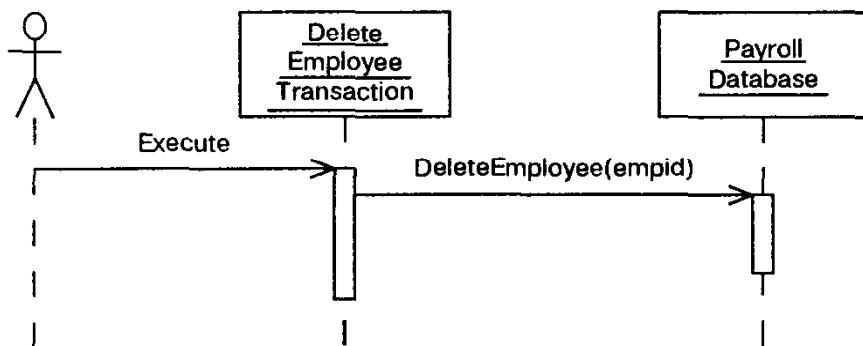


Рис. 19.6. Динамическая модель транзакции DeleteEmployee

**Листинг 19.9. PayrollTest::TestDeleteEmployee()**

```

void PayrollTest::TestDeleteEmployee()
{
 cerr << "TestDeleteEmployee" << endl;
 int empId = 3;
 AddCommissionedEmployee t(empId, "Lance", "Home", 2500, 3.2);
 t.Execute();
 {
 Employee* e = GpayrollDatabase.GetEmployee(empId);
 assert(e);
 }
 DeleteEmployeeTransaction dt(empId);
 dt.Execute();
 {
 Employee* e = GpayrollDatabase.GetEmployee(empId);
 assert(e == 0);
 }
}

```

**Листинг 19.10. DeleteEmployeeTransaction.h**

```

#ifndef DELETEEMPLOYEETRANSACTION_H
#define DELETEEMPLOYEETRANSACTION_H

#include "Transaction.h"

class DeleteEmployeeTransaction : public Transaction
{
public:
 virtual ~DeleteEmployeeTransaction();
 DeleteEmployeeTransaction(int empid);
 virtual void Execute();
private:
 int itsEmpid;
};

#endif

```

---

**Листинг 19.11. DeleteEmployeeTransaction.cpp**

```
#include "DeleteEmployeeTransaction.h"
#include "PayrollDatabase.h"

extern PayrollDatabase GpayrollDatabase;
DeleteEmployeeTransaction::~DeleteEmployeeTransaction()
{
}

DeleteEmployeeTransaction::DeleteEmployeeTransaction (int empid)
: itsEmpid(empid)
{
}

void DeleteEmployeeTransaction::Execute()
{
 GpayrollDatabase.DeleteEmployee(itsEmpid);
}
```

---

## Глобальные переменные

Теперь понятно, что база данных `GpayrollDatabase` носит глобальный характер. Иногда глобальные переменные используются программистами крайне неохотно. На самом деле применение этих переменных не таит в себе ничего дурного. Именно рассматриваемая ситуация представляет собой идеальный случай, когда использование глобальных переменных существенно упрощает составление кода. Пожалуй, так происходит только при работе с одним экземпляром класса `PayrollDatabase`, что следует сделать достоянием широкой аудитории.

Можно предположить, что в данном случае удобнее воспользоваться шаблонами `Singleton` или `Monostate`. Действительно, эти шаблоны вполне подходят в данной ситуации. Но в них также применяются глобальные переменные. Шаблонам `Singleton` и `Monostate` по определению присущ глобальный характер. Можно заключить, что в данном случае использование `Singleton` или `Monostate` приводит к избыточной сложности. Проще сохранить экземпляр базы данных в глобальном виде.

## Ведение карточек табельного учета, торговых квитанций и поддержка сведений об оплате услуг

На рис. 19.7 показана статическая структура для транзакции по ведению карточек табельного учета работников. Рисунок 19.8 иллюстрирует динамическую

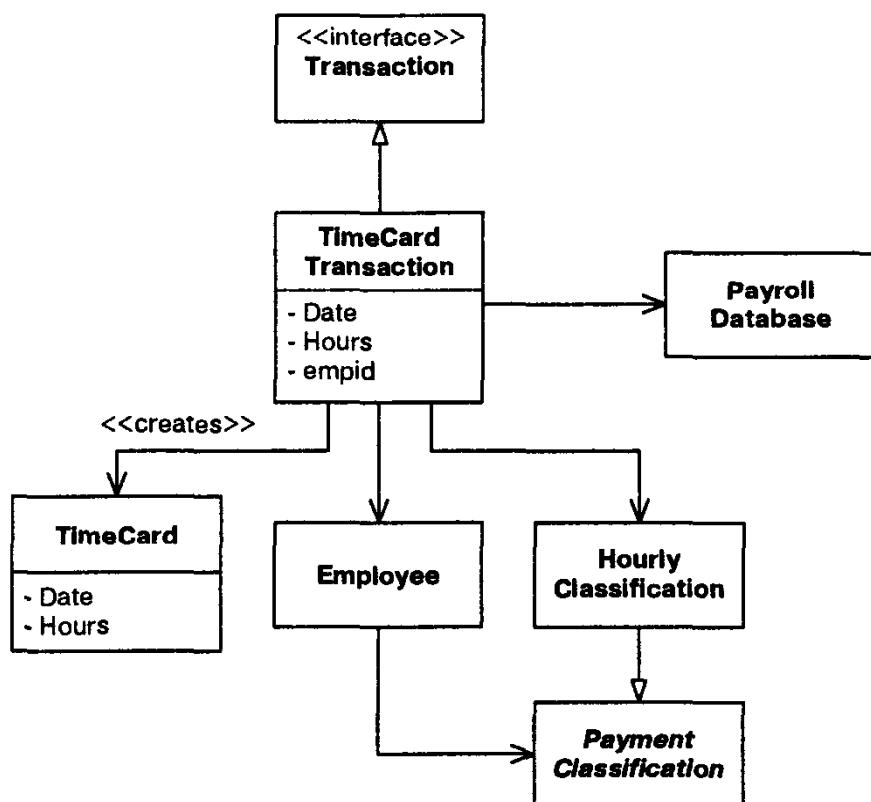


Рис. 19.7. Статическая структура TimeCardTransaction

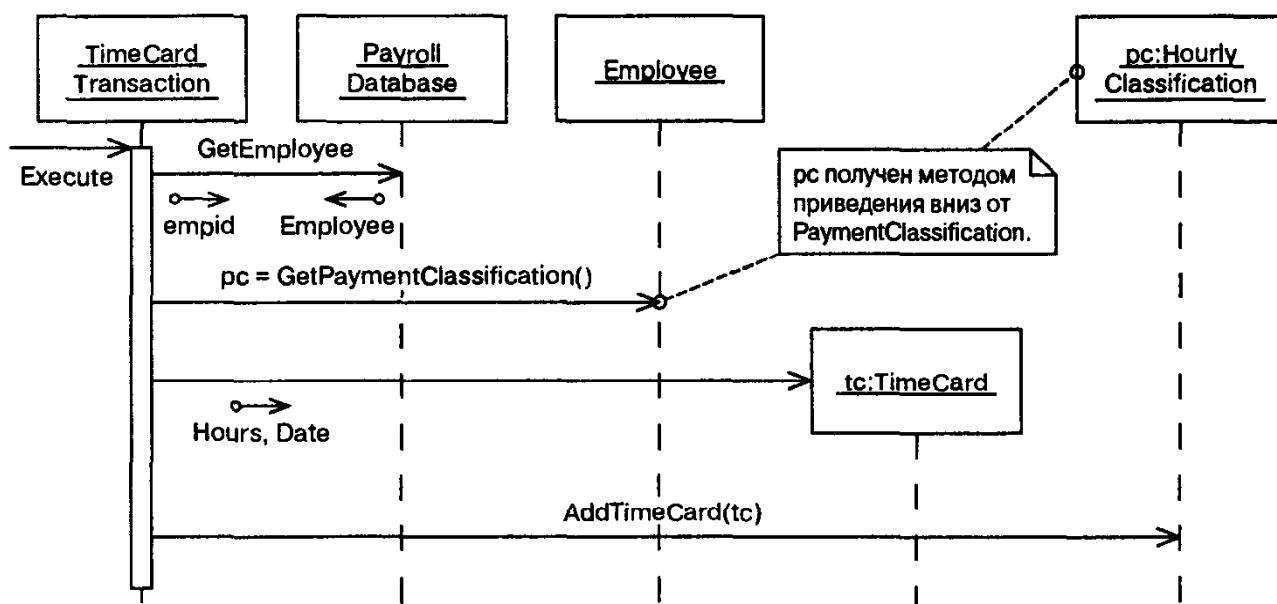


Рис. 19.8. Динамическая модель, имитирующая отсылку TimeCard

модель. Основная идея состоит в том, что транзакция получает объект Employee из PayrollDatabase, запрашивает Employee по поводу соответствия объекту PaymentClassification, затем создает объект TimeCard и вносит его в необходимый реестр PaymentClassification.

Обратите внимание, что объекты TimeCard нельзя добавлять в обобщенные объекты PaymentClassification; рассматриваемые объекты можно вно-

сить только в объекты HourlyClassification. Отсюда следует, что объект PaymentClassification, полученный из объекта Employee, следует привести вниз к объекту HourlyClassification. В данном случае удобно воспользоваться оператором `dynamic_cast` из C++, как показано в листинге 19.15.

Код из листинга 19.12 демонстрирует один из тестовых случаев, при выполнении которого учитываются карточки табельного учета работников, имеющих почасовую оплату. Эти карточки вносятся в базу данных. Затем создается TimeCardTransaction и вызывается метод Execute(). И наконец, проверяется, включает ли HourlyClassification соответствующий TimeCard.

---

#### Листинг 19.12. PayrollTest::TestTimeCardTransaction()

---

```
void PayrollTest::TestTimeCardTransaction ()
{
 cerr << "TestTimeCardTransaction" << endl;
 int empId = 2;
 AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
 t.Execute();
 TimeCardTransaction tct(20011031, 8.0, empId);
 tct.Execute();
 Employee* e = GpayrollDatabase.GetEmployee(empId);
 assert(e);
 PaymentClassification* pc = e->GetClassification();
 HourlyClassification* hc =
 dynamic_cast<HourlyClassification*>(pc);
 assert(hc);
 TimeCard* tc = hc->GetTimeCard(20011031);
 assert(tc);
 assertEquals(8.0, tc->GetHours());
}
```

---

Код из листинга 19.13 реализует класс TimeCard. При данных условиях это — оптимальный вариант. Этот класс является классом данных. Обратите внимание, что для представления данных используется тип long integer, поскольку отсутствует пригодный для этого класс Date. Вероятно, со временем это число заменится на более привлекательное, но в настоящее время и этот вариант получит распространение. Сейчас не хотелось бы отвлекаться на решение этой задачи, поскольку необходимо завершить тестовые случаи. Со временем можно записать тестовый случай с помощью реального класса Date. Когда это произойдет, вернемся назад и воспользуемся классом TimeCard.

---

#### Листинг 19.13. TimeCard.h

---

```
#ifndef TIMECARD_H
#define TIMECARD_H

class TimeCard
```

```

{
public:
 virtual ~TimeCard();
 TimeCard(long date, double hours);
 long GetDate {return itsDate;}
 double GetHours() {return itsHours;}
private:
 long itsDate;
 double itsHours;
};

#endif

```

Листинги 19.14 и 19.15 демонстрируют реализацию класса TimeCardTransaction. Обратите внимание, что используются простые строчные исключения. Эта практика не всегда себя оправдывает, но вполне достаточна для дальнейшего совершенствования программы. После выяснения, какие исключения можно реально выполнить, можно вернуться и создать значимые классы исключений. Также заметим, что экземпляр TimeCard создается только в том случае, если не отказываются от использования исключения. Исключение не может привести к утечке памяти. Довольно просто создать код, нерационально использующий память или ресурсы и не применяющий исключений, поэтому в данном случае следует проявлять осторожность<sup>1</sup>.

---

#### Листинг 19.14. TimeCardTransaction.h

---

```

#ifndef TIMECARDTRANSACTION_H
#define TIMECARDTRANSACTION_H

#include "Transaction.h"

class TimeCardTransaction : public Transaction
{
public:
 virtual ~TimeCardTransaction();
 TimeCardTransaction(long date, double hours, int empid);
 virtual void Execute();

private:
 int itsEmpid;
 long itsDate;
 double itsHours;
};

#endif

```

---

<sup>1</sup> Желательно ознакомиться с книгами Херба Саттера (Herb Sutter) *Exceptional C++* и *More Exceptional C++*. Эти книги значительно облегчат вашу работу с исключениями в C++.

**Листинг 19.15. TimeCardTransaction.cpp**

```
#include "TimeCardTransaction.h"
#include "Employee.h"
#include "PayrollDatabase.h"
#include "HourlyClassification.h"
#include "TimeCard.h"

extern PayrollDatabase GpayrollDatabase;

TimeCardTransaction::~TimeCardTransaction()
{
}

TimeCardTransaction::TimeCardTransaction(long date,
 double hours,
 int empid)
: itsDate(date)
, itsHours(hours)
, itsEmpid(empid)
{
}

void TimeCardTransaction::Execute()
{
 Employee* e = GpayrollDatabase.GetEmployee(itsEmpid);
 if (e){
 PaymentClassification* pc = e->GetClassification();
 if (HourlyClassification* hc =
 dynamic_cast<HourlyClassification*>(pc)){
 hc->AddTimeCard(new TimeCard(itsDate, itsHours));
 } else
 throw("Tried to add timecard to non-hourly employee");
 } else
 throw("No such employee.");
}
```

На рис. 19.9 и 19.10 показана аналогичная разработка для транзакции, которая направляет торговые квитанции работникам, получающим комиссионные. Реализация этих классов послужит хорошим упражнением для читателя.

На рис. 19.11 и 19.12 демонстрируется структура транзакции, которая реализует оплату услуг членов объединения.

Описанные структуры позволяют установить неполное соответствие между моделью транзакции и создаваемой моделью. Основной объект Employee может реализовывать отчисления взносов различным организациям, но модель транзакции ограничивает кооперацию формой объединения. Поэтому модель транзакции не позволяет идентифицировать форму кооперации. В данном случае просто пред-

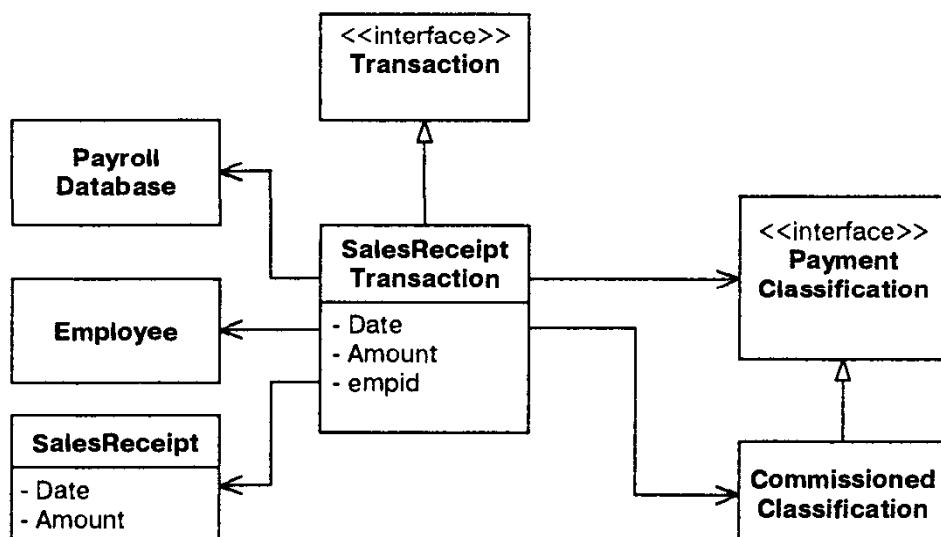


Рис. 19.9. Статическая модель для SalesReceiptTransaction

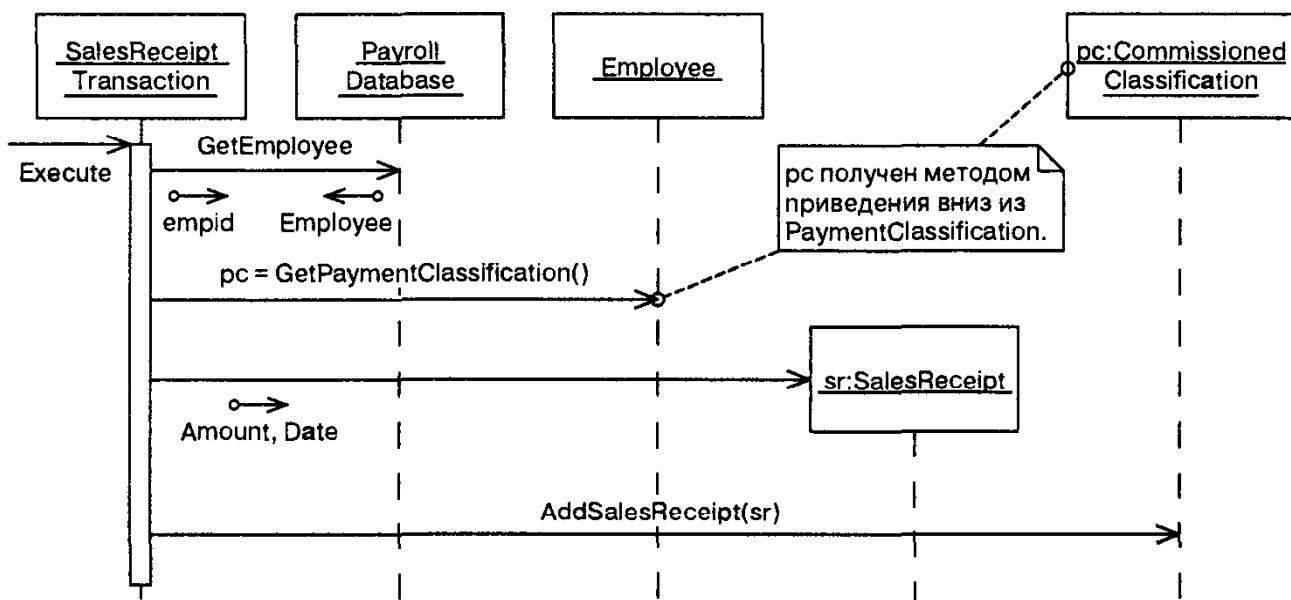
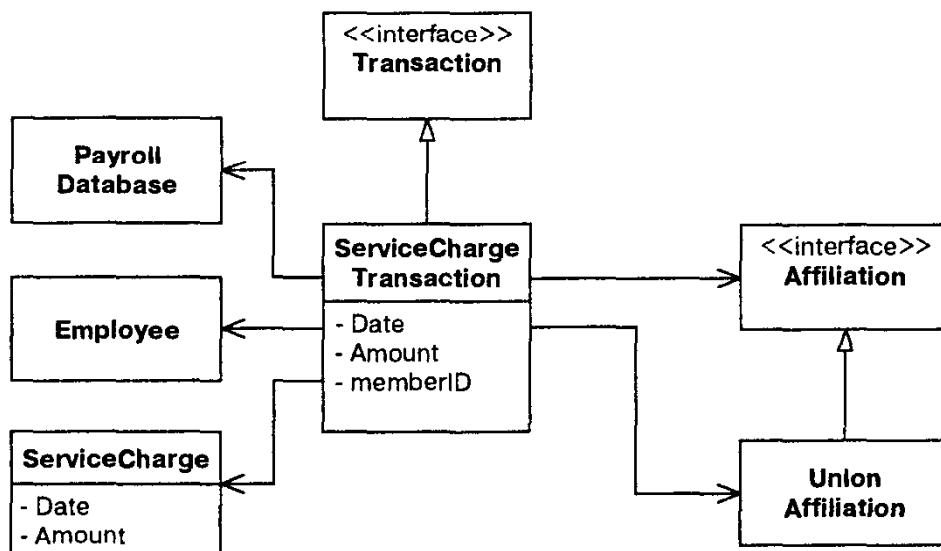


Рис. 19.10. Динамическая модель для SalesReceiptTransaction

полагается, что работник, который платит членские взносы, был принят в объединение.

В динамической модели этого затруднения удается избежать следующим образом. Выполняется поиск набора объектов *Affiliation*, включенных в объект *Employee* для объекта *UnionAffiliation*. Затем объект *ServiceCharge* добавляется в *UnionAffiliation*.

Листинг 19.16 демонстрирует тестовый случай для класса *ServiceChargeTransaction*. Формируется список работников с почасовой оплатой труда, и *UnionAffiliation* добавляется туда же. Также предполагается, что с помощью *PayrollDatabase* регистрируется соответствующий идентификатор члена объединения. Затем создается и выполняется

Рис. 19.11. Статическая модель для `ServiceChargeTransaction`Рис. 19.12. Динамическая модель для `ServiceChargeTransaction`

`ServiceChargeTransaction`. Наконец, точно известно, что соответствующий `ServiceCharge` вносится в `UnionAffiliation` объекта `Employee`.

#### Листинг 19.16. `PayrollTest::TestAddServiceCharge()`

```

void PayrollTest::TestAddServiceCharge()
{
 cerr << "TestAddServiceCharge" << endl;
 int empId = 2;
 AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
 t.Execute();
 Employee* e = GpayrollDatabase.GetEmployee(empId);
 assert(e);
 UnionAffiliation* af = new UnionAffiliation(12.5);
 e->SetAffiliation(af);

```

```
int memberId =86; // Maxwell Smart
GpayrollDatabase.AddUnionMember(memberId, e);
ServiceChargeTransaction set(memberId, 20011101, 12.95);
set.Execute();
ServiceCharge* sc = af->GetServiceCharge(20011101);
assert(sc);
assertEquals(12.95, sc->GetAmount(), .001);
}
```

---

## Код и UML-диаграммы

При создании UML-диаграммы, показанной на рис. 19.12, предполагалось, что удобно заменить `NoAffiliation` списком объединений. Казалось, что этот вариант является несложным и достаточно гибким. В конце концов, новые объединения можно вносить по мере необходимости, и нет нужды в создании класса `NoAffiliation`. Но при написании тестового случая, код которого приводится в листинге 19.16, автор пришел к выводу, что предпочтительнее вызывать метод `SetAffiliation` для `Employee`, чем `AddAffiliation`. В конце концов, не требуется, чтобы работник относился более, чем к одному классу `Affiliation`, поэтому нет необходимости в использовании `dynamic_cast` для выбора между потенциально различными образованиями. Возможность выбора чрезмерно усложняет задание.

Этот пример показывает, почему излишнее использование UML-диаграммы без проведения дополнительной проверки в коде столь негативно сказывается на функционировании программы. Код может дать дополнительную информацию о проекте, которая не может быть получена с помощью диаграммы. В данном случае нет необходимости размещать структуры в UML-диаграмме. Вероятно, размещение структур в UML-диаграмме представляет удобство на определенном этапе работы, но не следует постоянно пользоваться этим приемом. Результат не оправдывает прилагаемых усилий.

В данном случае, несмотря на относительно несложную поддержку `dynamic_cast`, не предполагается ее реализация. Значительно проще исключить список объектов `Affiliation`. Итак, вместо класса `NoAffiliation` сохраняется шаблон `Null Object`.

Код в листингах 19.17 и 19.18 демонстрирует реализацию класса `ServiceChargeTransaction`. Значительно проще разыскивать объекты `UnionAffiliation` без применения цикла. Проще получить сведения о работнике (`Employee`) из базы данных, направить соответствующее `Affiliation` в `UnionAffiliation` и добавить к нему класс `ServiceCharge`.

---

### Листинг 19.17. ServiceChargeTransaction.h

---

```
#ifndef SERVICECHARGE TRANSACTION_H
#define SERVICECHARGE TRANSACTION_H
```

```
#include "Transaction.h"

class ServiceChargeTransaction : public Transaction
{
public:
 virtual ~ServiceChargeTransaction();
 ServiceChargeTransaction(int memberId, long date, double charge);
 virtual void Execute();
private:
 int itsMemberId;
 long itsDate;
 double itsCharge;
};

#endif
```

---

#### Листинг 19.18. ServiceChargeTransaction.cpp

---

```
#include "ServiceChargeTransaction.h"
#include "Employee.h"
#include "ServiceCharge.h"
#include "PayrollDatabase.h"
#include "UnionAffiliation.h"

extern PayrollDatabase GpayrollDatabase;

ServiceChargeTransaction::~ServiceChargeTransaction();
{
}

ServiceChargeTransaction::ServiceChargeTransaction(int memberId, long date, double charge)
:itsMemberId(memberId)
, itsDate(date)
, itsCharge(charge)
{
}

void ServiceChargeTransaction::Execute()
{
 Employee* e = GpayrollDatabase.GetUnionMember(itsMemberId);
 Affiliation* af = e->GetAffiliation();
 if (UnionAffiliation* uaf = dynamic_cast<UnionAffiliation*>(af)) {
 uaf->AddServiceCharge(itsDate, itsCharge);
 }
}
```

## Изменение записей о работниках

На рисунках 19.13 и 19.14 показана статическая структура для транзакций, изменяющих атрибуты работника. Эта структура легко выводится из варианта использования под номером 6. Во всех транзакциях используется аргумент EmpID, поэтому можно создать базовый класс высокого уровня под названием `ChangeEmployeeTransaction`. Ниже этого базового класса находятся классы, изменяющие отдельные атрибуты, такие как `ChangeNameTransaction` и `ChangeAddressTransaction`. Транзакции, вносящие изменения в классификации, используются в общих целях — обновляют одно и то

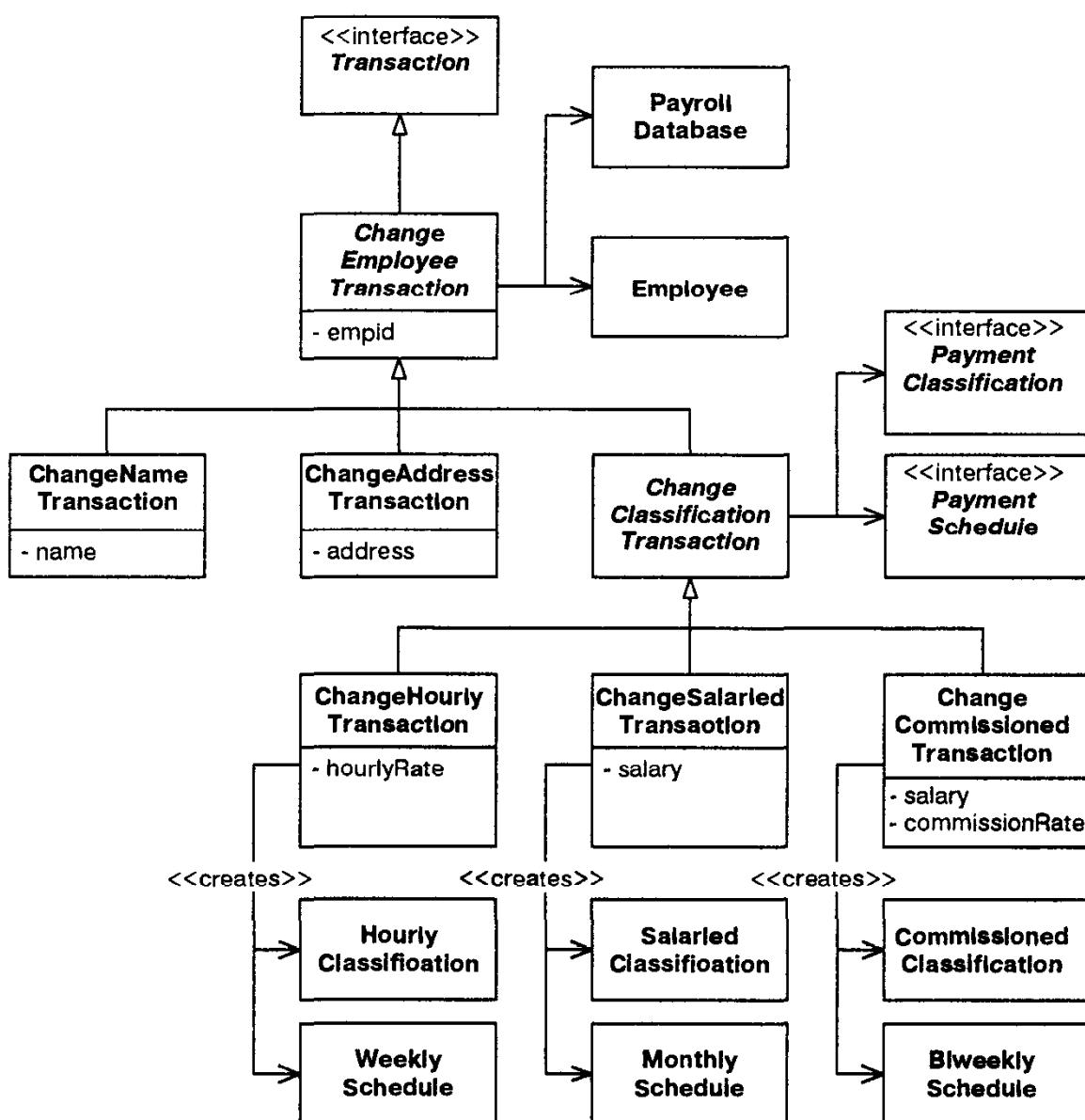
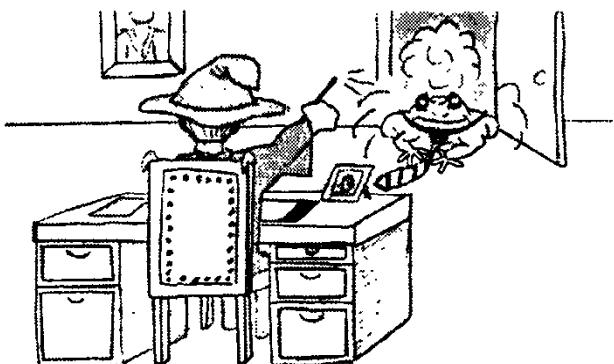


Рис. 19.13. Статическая модель для `ChangeEmployeeTransaction`

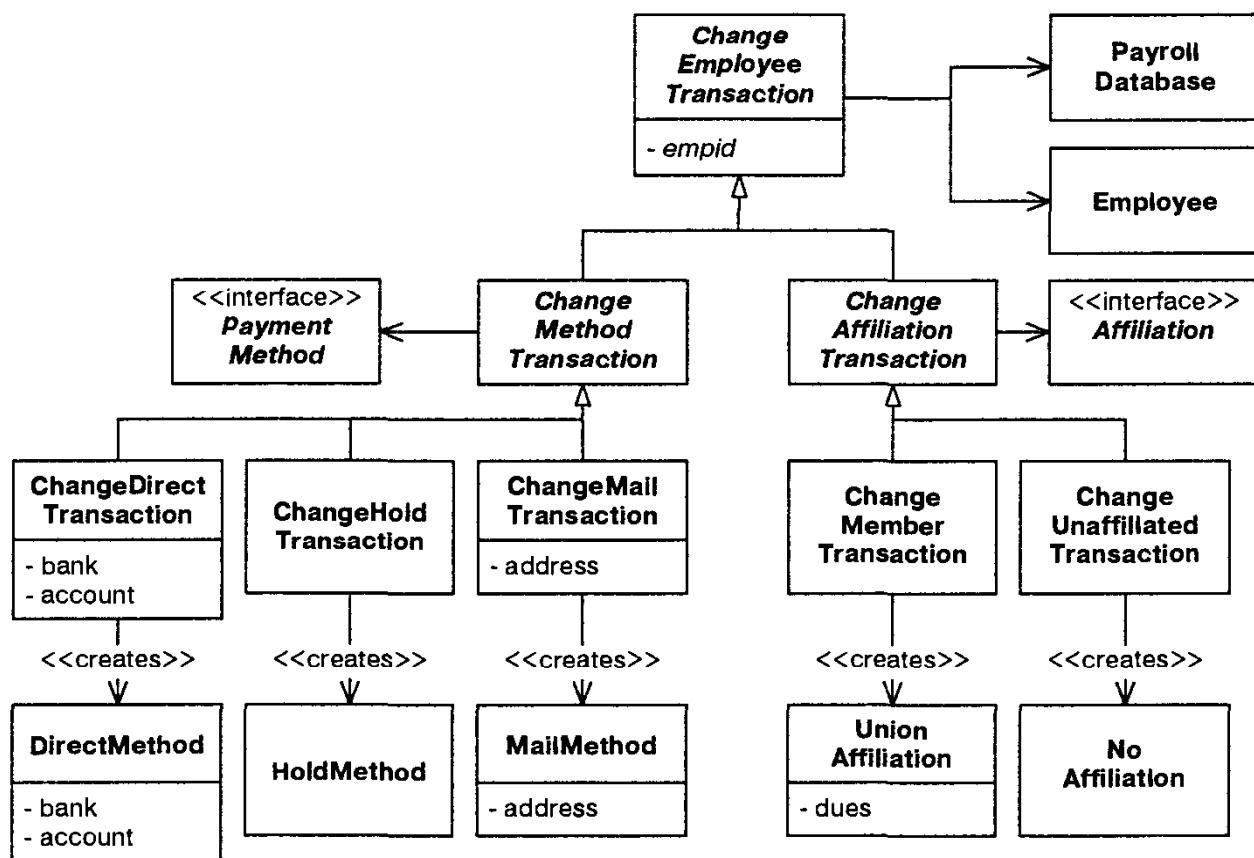


Рис. 19.14. Статическая модель для ChangeEmployeeTransaction (продолжение)

же поле объекта **Employee**. Поэтому они сгруппированы вместе с применением абстрактной базы, **ChangeClassificationTransaction**. Аналогичный подход реален и для транзакций, изменяющих размеры оплаты и отношение к объединениям. Для этого используются структуры **ChangeMethodTransaction** и **ChangeAffiliationTransaction**.

На рис. 19.15 показана динамическая модель для всех транзакций, выполняющих изменения. Снова применяется шаблон **Template Method**. В любом случае объект **Employee**, соответствующий EmpID, должен выбираться из **Payroll Database**.

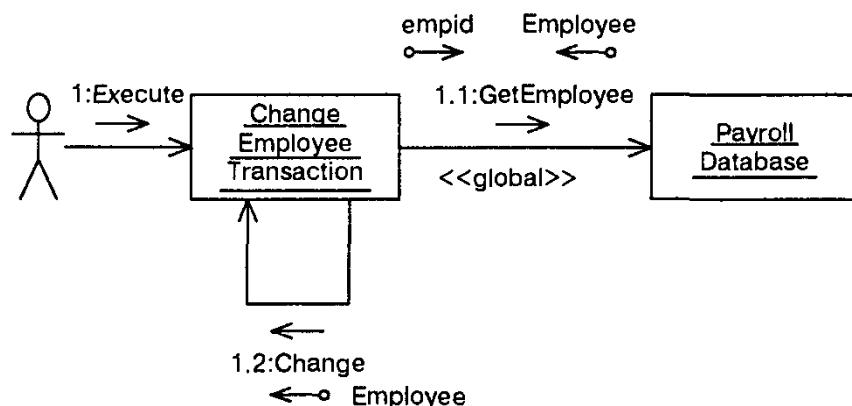


Рис. 19.15. Динамическая модель для ChangeEmployee-Transaction

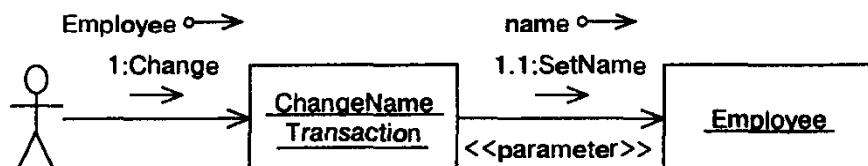


Рис. 19.16. Динамическая модель для ChangeNameTransaction

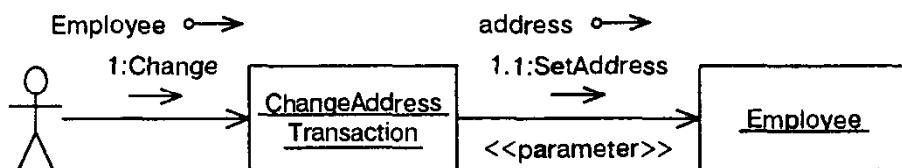


Рис. 19.17. Динамическая модель для ChangeAddressTransaction

`rollDatabase`. Поэтому, функция `Execute` из `ChangeEmployeeTransaction` реализует это поведение и затем направляет себе сообщение `Change`. Этот метод объявляется как виртуальный и реализуется с помощью производных модулей, как показано на рис. 19.16 и 19.17.

Листинг 19.19 демонстрирует тестовый случай для `ChangeNameTransaction`. Данный тестовый случай реализован довольно просто. Транзакция `AddHourlyEmployee` применяется для формирования записи о работнике-почасовике по имени Билл (Bill). Затем создается и выполняется транзакция `ChangeNameTransaction`, изменяющая имя работника на Боб (Bob). Наконец, производится обращение к экземпляру `Employee` из `PayrollDatabase`, после чего подтверждается, что имя работника было изменено.

#### Листинг 19.19. PayrollTest::TestChangeNameTransaction()

```

void PayrollTest::TestChangeNameTransaction()
{
 cerr << "TestChangeNameTransaction" << endl;
 int empId = 2;
 AddHourlyEmployee t (empId, "Bill", "Home", 15.25);
 t.Execute();
 ChangeNameTransaction cnt(empId, "Bob");
 cnt.Execute();
 Employee* e = GpayrollDatabase.GetEmployee(empId);
 assert(e);
 assert("Bob" == e->GetName());
}

```

Листинги 19.20 и 19.21 представляют реализацию абстрактного базового класса `ChangeEmployeeTransaction`. Структура шаблона `Template Method` довольно проста. Метод `Execute()` просматривает соответствующий экземпляр

Employee из PayrollDatabase и в случае удачи вызывает чисто виртуальную функцию Change().

---

#### Листинг 19.20. ChangeEmployeeTransaction.h

---

```
#ifndef CHANGEEMPLOYEETRANSACTION_H
#define CHANGEEMPLOYEETRANSACTION_H

#include "Transaction.h"
#include "Employee.h"

class ChangeEmployeeTransaction : public Transaction
{
public:
 ChangeEmployeeTransaction(int empid);
 virtual ~ChangeEmployeeTransaction();
 virtual void Execute();
 virtual void Change(Employee&) = 0;
private:
 int itsEmpId;
};

#endif
```

---

---

#### Листинг 19.21. ChangeEmployeeTransaction.cpp

---

```
#include "ChangeEmployeeTransaction.h"
#include "Employee.h"
#include "PayrollDatabase.h"

extern PayrollDatabase GpayrollDatabase;

ChangeEmployeeTransaction::~ChangeEmployeeTransaction()
{
}

ChangeEmployeeTransaction::ChangeEmployeeTransaction(int empid)
: itsEmpId(empid)
{
}

void ChangeEmployeeTransaction::Execute()
{
 Employee* e = GpayrollDatabase.GetEmployee(itsEmpId);
 if (e != 0)
 Change(* e);
}
```

---

Листинги 19.22 и 19.23 демонстрируют реализацию для ChangeNameTransaction. Здесь можно легко заметить вторую часть шаблона Template Method.

Для изменения названия аргумента Employee реализуется метод Change(). Структура ChangeAddressTransaction имеет подобный вид, поэтому ее реализация оставлена читателям в качестве упражнения.

---

#### Листинг 19.22. ChangeEmployeeTransaction.h

---

```
#ifndef CHANGENAMETRANSACTION_H
#define CHANGENAMETRANSACTION_H

#include "ChangeEmployeeTransaction.h"
#include <string>

class ChangeNameTransaction : public ChangeEmployeeTransaction
{
public:
 virtual ~ChangeNameTransaction();
 ChangeNameTransaction(int empid, string name);
 virtual void Change (Employee&:);
private:
 string itsName;
};

#endif
```

---



---

#### Листинг 19.23. ChangeNameTransaction.cpp

---

```
#include "ChangeNameTransaction.h"

ChangeNameTransaction::~ChangeNameTransaction()
{
}

ChangeNameTransaction::ChangeNameTransaction(int empid, string name)
: ChangeEmployeeTransaction(empid)
, itsName(name)
{
}

void ChangeNameTransaction::Change(Employee& e)
{
 e.SetName(itsName);
}
```

---

## Внесение изменений в классификацию

На рис. 19.18 представлен прогноз поведения ChangeClassification-Transaction. Снова используется шаблон Template Method. Эти транзакции призваны создать новый объект PaymentClassification и затем передать его объекту Employee. Для достижения этой цели класс GetClassification отсылает сам себе сообщения.

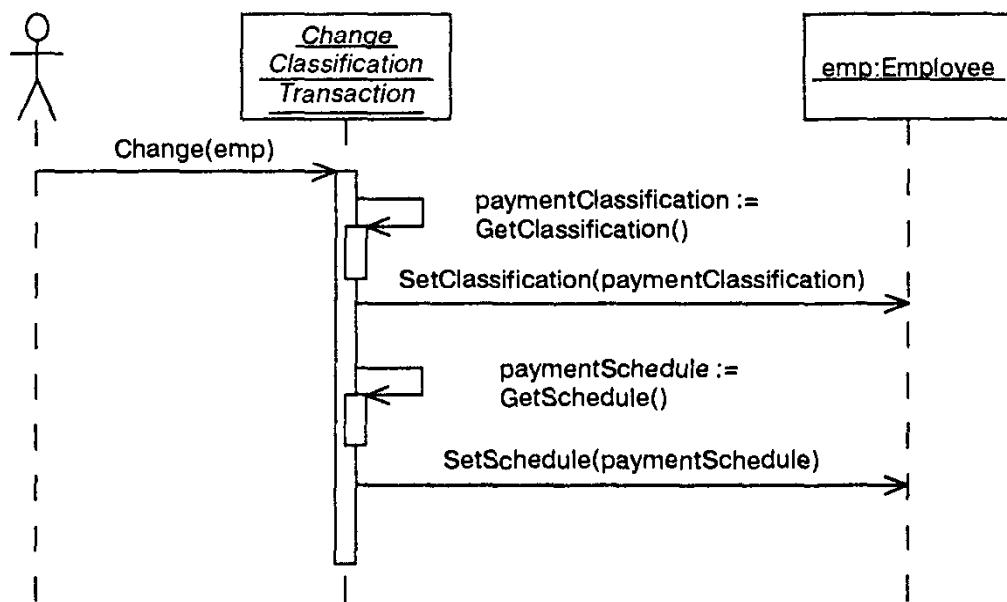


Рис. 19.18. Динамическая модель ChangeClassificationTransaction

Этот абстрактный метод реализуется в каждом классе, производном от ChangeClassificationTransaction, что и проиллюстрировано на рис. 19.19-19.21.

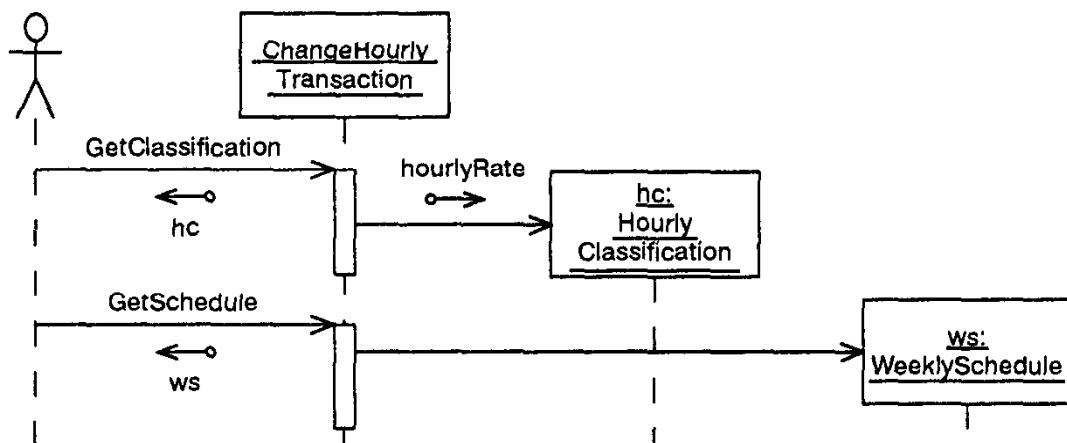


Рис. 19.19. Динамическая модель для ChangeHourlyTransaction

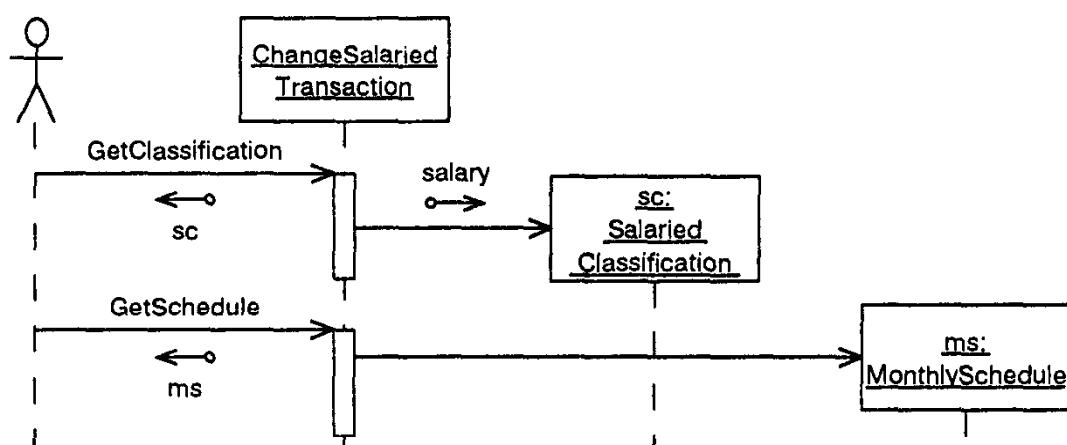


Рис. 19.20. Динамическая модель для ChangeSalariedTransaction

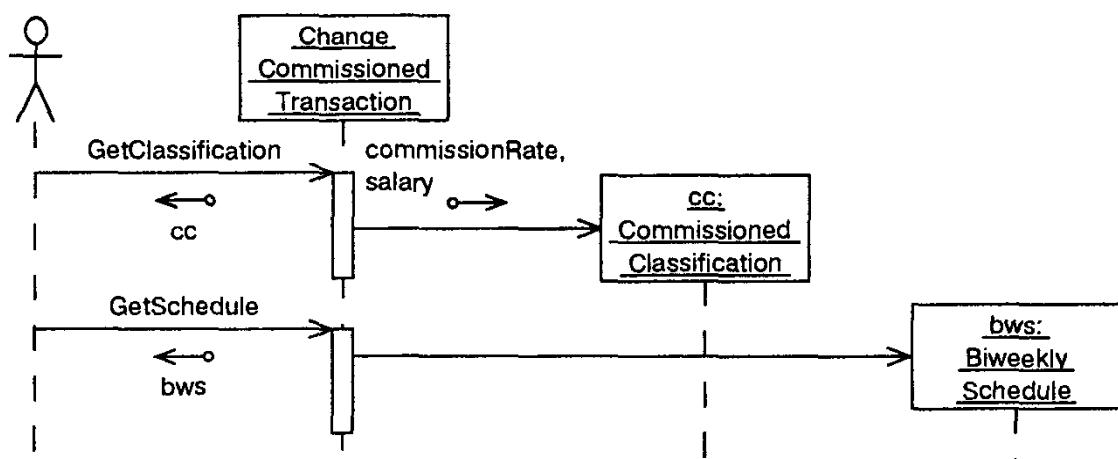


Рис. 19.21. Динамическая модель для `ChangeCommissionedTransaction`

Листинг 19.24 демонстрирует тестовый случай для `ChangeHourlyTransaction`. Здесь транзакция `AddCommissionedEmployee` применяется для определения работника, получающего комиссионные. Затем создается и выполняется `ChangeHourlyTransaction`. Изменяется положение работника и подтверждается, что его `PaymentClassification` является `HourlyClassification` с соответствующей почасовой ставкой, а `PaymentSchedule` данного работника является `WeeklySchedule`.

#### Листинг 19.24. PayrollTest::TestChangeHourlyTransaction()

```

void PayrollTest::TestChangeHourlyTransaction()
{
 cerr << "TestChangeHourlyTransaction" << endl;
 int empId = 3;
 AddCommissionedEmployee t(empId, "Lance", "Home", 2500, 3.2);
 t.Execute();
 ChangeHourlyTransaction cht(empId, 27.52);
 cht.Execute();
 Employee* e = GpayrollDatabase.GetEmployee(empId);
 assert(e);
 PaymentClassification* pc = e->GetClassification();
 assert(pc);
 HourlyClassification* hc =
 dynamic_cast<HourlyClassification*>(pc);
 assert(hc);
 assertEquals(27.52, hc->GetRate(), .001);
 PaymentSchedule* ps = e->GetSchedule();
 WeeklySchedule* ws = dynamic_cast<WeeklySchedule*>(ps);
 assert(ws);
}

```

Листинги 19.25 и 19.26 демонстрируют реализацию абстрактного базового класса `ChangeClassificationTransaction`. Снова легко просматривается использование шаблона `Template Method`. Метод `Change()` вызывает две

виртуальные функции, `GetClassification()` и `GetSchedule()`. Возвращаемые значения этих функций применяются для классификации `Employee` и составления календарного графика.

---

**Листинг 19.25. ChangeClassificationTransaction.h**

---

```
#ifndef CHANGECLASSIFICATIONTRANSACTION_H
#define CHANGECLASSIFICATIONTRANSACTION_H

#include "ChangeEmployeeTransaction.h"

class PaymentClassification;
class PaymentSchedule;
class ChangeClassificationTransaction : public
 ChangeEmployeeTransaction
{
public:
 virtual ~ChangeClassificationTransaction();
 ChangeClassificationTransaction(int empid);
 virtual void Change(Employee& e);
 virtual PaymentClassification* GetClassification() const = 0;
 virtual PaymentSchedule* GetSchedule() const = 0;
};

#endif
```

---

---

**Листинг 19.26. ChangeClassificationTransaction.cpp**

---

```
#include "ChangeClassificationTransaction.h"

ChangeClassificationTransaction::~ChangeClassificationTransaction()
{
}

ChangeClassificationTransaction::ChangeClassificationTransaction(int
 empid)
: ChangeEmployeeTransaction(empid)
{
}

void ChangeClassificationTransaction::Change(Employee& e)
{
 e.SetClassification(GetClassification());
 e.SetSchedule(GetSchedule());
}
```

---

Листинги 19.27 и 19.28 включают реализацию класса транзакции `ChangeHourlyTransaction`. Этот класс обращается к шаблону `Template Method` путем реализации методов `GetClassification()` и `GetSchedule()`, наследуемых из `ChangeClassificationTransaction`. Вследствие этого `Get-`

`Classification()` возвращает заново созданный `HourlyClassification`.  
 Также `GetSchedule()` возвращает заново созданный `WeeklySchedule`.

---

**Листинг 19.27. ChangeHourlyTransaction.h**

---

```
#ifndef CHANGEHOURLYTRANSACTION_H
#define CHANGEHOURLYTRANSACTION_H

#include "ChangeClassificationTransaction.h"

class ChangeHourlyTransaction : public ChangeClassificationTransaction
{
public:
 virtual ~ChangeHourlyTransaction();
 ChangeHourlyTransaction(int empid, double hourlyRate);
 virtual PaymentSchedule* GetSchedule() const;
 virtual PaymentClassification* GetClassification() const;
private:
 double itsHourlyRate;
};

#endif
```

---



---

**Листинг 19.28. ChangeHourlyTransaction.cpp**

---

```
#include "ChangeHourlyTransaction.h"
#include "WeeklySchedule.h"
#include "HourlyClassification.h"

ChangeHourlyTransaction::~ChangeHourlyTransaction()
{
}

ChangeHourlyTransaction::ChangeHourlyTransaction(int empid, double
 hourlyRate)
: ChangeClassificationTransaction(empid)
, itsHourlyRate(hourlyRate)
{
}

PaymentSchedule* ChangeHourlyTransaction::GetSchedule() const
{
 return new WeeklySchedule();
}

PaymentClassification* ChangeHourlyTransaction::GetClassification()
 const
{
 return new HourlyClassification(itsHourlyRate);
```

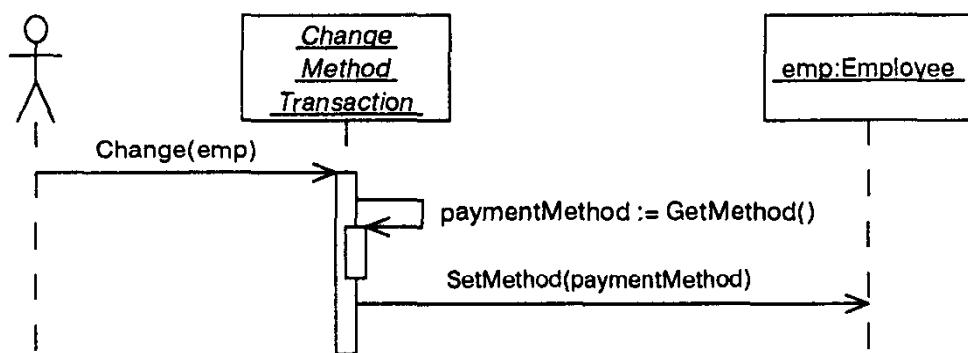


Рис. 19.22. Динамическая модель для `ChangeMethodTransaction`

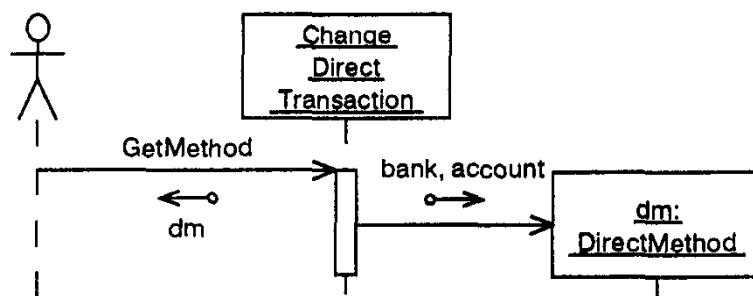


Рис. 19.23. Динамическая модель для `ChangeDirectTransaction`

Как всегда, реализация `ChangeSalariedTransaction` и `ChangeCommissionedTransaction` остаются в качестве упражнений для читателя.

Аналогичный механизм применяется для реализации `ChangeMethodTransaction`. Абстрактный метод `GetMethod` используется для выбора подходящего производного модуля для `PaymentMethod`, который затем направляется объекту `Employee`. (рис. 19.22–19.25.)

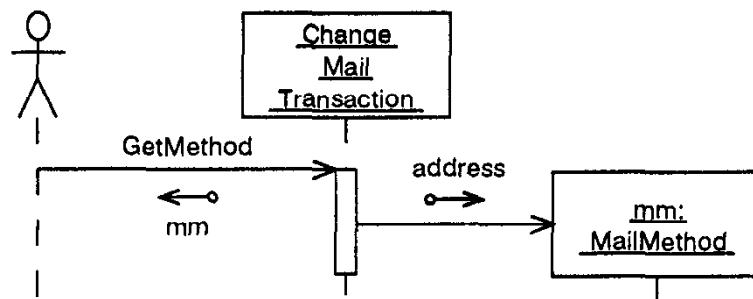
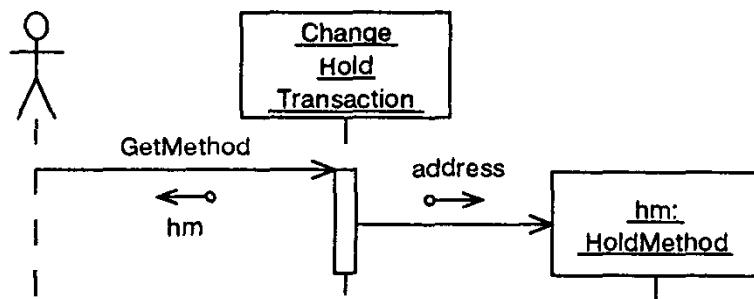


Рис. 19.24. Динамическая модель для `ChangeMailTransaction`

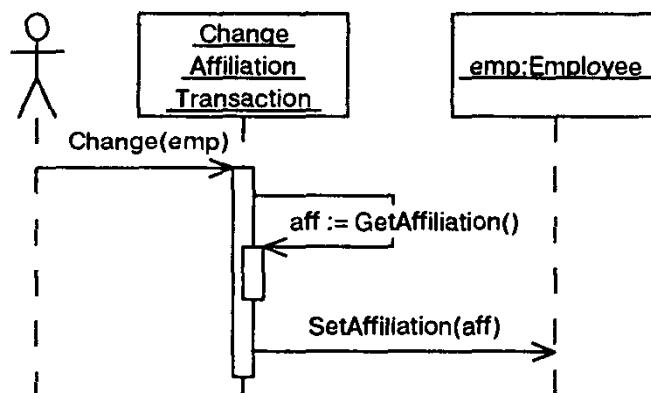
Реализация этих классов выглядит прямолинейной и неэкономичной. Они также оставлены в качестве упражнения для читателя.

На рис. 19.26 показана реализация класса транзакции `ChangeAffiliationTransaction`. Снова для выделения производного модуля `Affiliation` ис-

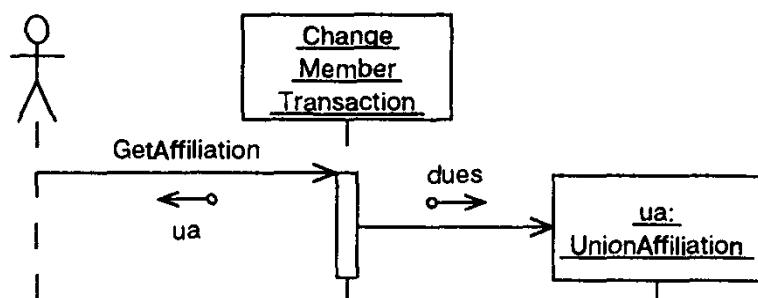
пользуется шаблон **Template Method**, затем этот модуль передается объекту **Employee**. (рис. 19.27–19.29).



**Рис. 19.25.** Динамическая модель для Change-HoldTransaction



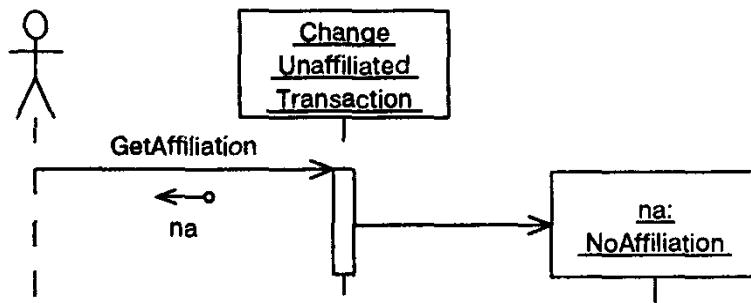
**Рис. 19.26.** Динамическая модель Change-AffiliationTransaction



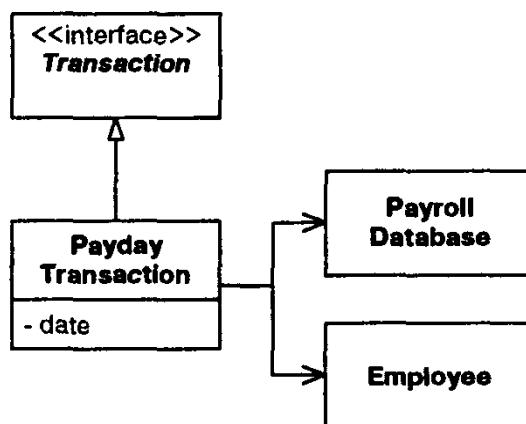
**Рис. 19.27.** Динамическая модель для Change-MemberTransaction

## Некоторые итоги

При реализации рассматриваемого проекта возникли определенные трудности. Просмотрите внимательно динамические диаграммы для транзакций, относящихся к объединениям. Замечаете ли вы особенности, проводящие к затруднениям?



**Рис. 19.28.** Динамическая модель ChangeUnaffiliatedTransaction



**Рис. 19.29.** Статическая модель PaydayTransaction

Как всегда, реализация началась с записи тестового случая для ChangeMemberTransaction. Код этого тестового случая представлен в листинге 19.29. Тестовое испытание проходит несколько прямолинейно. Формируется запись для работника-помощника по имени Билл (Bill), затем создается и выполняется ChangeMemberTransaction для привязки Билла к объединению (союзу). Проводится проверка, что Билл входит в UnionAffiliation, а UnionAffiliation характеризуется соответствующей величиной членских взносов.

---

#### Листинг 19.29. PayrollTest::TestChangeMemberTransaction()

---

```

void PayrollTest::TestChangeMemberTransaction ()
{
 cerr << "TestChangeMemberTransaction" << endl;
 int empId = 2;
 int memberId = 7734;
 AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
 t.Execute();
 ChangeMemberTransaction cmt(empId, memberId, 99.42);
 cmt.Execute();
 Employee* e = GpayrollDatabase.GetEmployee(empId);
 assert(e);
 Affiliation* af = e->GetAffiliation();
 assert(af);
}

```

```

UnionAffiliation* uf = dynamic_cast<UnionAffiliation*>(af);
assert(uf);
assertEquals(99.42, uf->GetDues(), .001);
Employee* member = PayrollDatabase.GetUnionMember(memberId);
assert(member);
assert(e == member);
}

```

---

“Собака зарыта” в нескольких последних строках кода. Эти строки позволяют заключить, что PayrollDatabase фиксирует членство Билла в объединении. Но в приведенных UML-диаграммах ничего подобного не предполагается. Диаграмма имеет только некоторое отношение к соответствующему производному модулю *Affiliation*, ограниченному *Employee*. Эту особенность трудно заметить, не так ли?

После несложного кодирования транзакций для каждой из диаграмм приходим к сбоям при выполнении всего теста. После сбоя в работе становится понятно, где допущена неточность. Но устранить неточность не так просто. Каким образом можно установить членство для *ChangeMemberTransaction*, но устраниТЬ его при обращении к *ChangeUnaffiliatedTransaction*?

Для устранения этого затруднения к *ChangeAffiliationTransaction* добавляется другая исключительно виртуальная функция под названием *RecordMembership* (*Employee\**). Эта функция реализуется в *ChangeMemberTransaction*, позволяя связать *memberId* с экземпляром *Employee*. В *ChangeUnaffiliatedTransaction* этот подход используется для устранения записи о членстве в объединении.

Листинги 19.30 и 19.31 демонстрируют результирующую реализацию абстрактного базового класса *ChangeAffiliationTransaction*. Снова очевидным образом применяется шаблон *Template Method*.

---

#### Листинг 19.30. ChangeAffiliationTransaction.h

---

```

#ifndef CHANGEAFFILIATIONTRANSACTION_H
#define CHANGEAFFILIATIONTRANSACTION_H

#include "ChangeEmployeeTransaction.h"

class ChangeAffiliationTransaction: public ChangeEmployeeTransaction
{
public:
 virtual ~ChangeAffiliationTransaction();
 ChangeAffiliationTransaction(int empid);
 virtual Affiliation* GetAffiliation() const = 0;
 virtual void RecordMembership(Employee*) = 0;
 virtual void Change (Employee&::);
};

#endif

```

---

**Листинг 19.31. ChangeAffiliationTransaction.cpp**

```
#include "ChangeAffiliationTransaction.h"

ChangeAffiliationTransaction::~ChangeAffiliationTransaction()
{
}

ChangeAffiliationTransaction::ChangeAffiliationTransaction(int empid)
: ChangeEmployeeTransaction(empid)
{
}

void ChangeAffiliationTransaction::Change(Employee& e)
{
 RecordMembership(&e);
 e.SetAffiliation(GetAffiliation());
}
```

---

Листинги 19.32 и 19.33 демонстрируют реализацию ChangeMemberTransaction. Это несложно и не столь интересно. С другой стороны, реализация ChangeUnaffiliatedTransaction в листингах 19.34 и 19.35 изложена более основательно. Функция RecordMembership определяет, является ли данный работник членом объединения. Если это так, из UnionAffiliation направляется memberId, и запись о членстве в объединении удаляется.

---

**Листинг 19.32. ChangeMemberTransaction.h**

```
#ifndef CHANGEMEMBERTRANSACTION_H
#define CHANGEMEMBERTRANSACTION_H

#include "ChangeAffiliationTransaction.h"

class ChangeMemberTransaction : public ChangeAffiliationTransaction
{
public:
 virtual ~ChangeMemberTransaction();
 ChangeMemberTransaction(int empid, int memberid, double dues);
 virtual Affiliation* GetAffiliation() const;
 virtual void RecordMembership(Employee*);
private:
 int itsMemberId;
 double itsDues;
};

#endif
```

---

**Листинг 19.33. ChangeMemberTransaction.cpp**

```
#include "ChangeMemberTransaction.h"
#include "UnionAffiliation.h"
#include "PayrollDatabase.h"

extern PayrollDatabase GpayrollDatabase;

ChangeMemberTransaction::~ChangeMemberTransaction()
{
}

ChangeMemberTransaction::ChangeMemberTransaction(int empid, int memberid, double dues)
: ChangeAffiliationTransaction(empid)
, itsMemberId(memberid)
, itsDues(dues)
{
}

Affiliation* ChangeMemberTransaction::GetAffiliation() const
{
 return new UnionAffiliation(itsMemberId, itsDues);
}

void ChangeMemberTransaction::RecordMembership(Employee* e)
{
 GpayrollDatabase.AddUnionMember(itsMemberId, e);
}
```

---

**Листинг 19.34. ChangeUnaffiliatedTransaction.h**

```
#ifndef CHANGEUNAFFILIATEDTRANSACTION_H
#define CHANGEUNAFFILIATEDTRANSACTION_H

#include "ChangeAffiliationTransaction.h"

class ChangeUnaffiliatedTransaction : public
 ChangeAffiliationTransaction
{
public:
 virtual ~ChangeUnaffiliatedTransaction();
 ChangeUnaffiliatedTransaction(int empId);
 virtual Affiliation* GetAffiliation() const;
 virtual void RecordMembership(Employee*);
};
```

---

#endif

---

**Листинг 19.35. ChangeUnaffiliatedTransaction.cpp**

```
#include "ChangeUnaffiliatedTransaction.h"
#include "NoAffiliation.h"
#include "UnionAffiliation.h"
#include "PayrollDatabase.h"

extern PayrollDatabase GpayrollDatabase;

ChangeUnaffiliatedTransaction::~ChangeUnaffiliatedTransaction()
{
}

ChangeUnaffiliatedTransaction::ChangeUnaffiliatedTransaction(int empId)
: ChangeAffiliationTransaction(empId)
{
}

Affiliation* ChangeUnaffiliatedTransaction::GetAffiliation() const
{
 return new NoAffiliation();
}

void ChangeUnaffiliatedTransaction::RecordMembership(Employee* e)
{
 Affiliation* af = e->GetAffiliation();
 if (UnionAffiliation* uf = dynamic_cast<UnionAffiliation*>(af))
 {
 int memberId = uf->GetMemberId();
 GpayrollDatabase.RemoveUnionMember(memberId);
 }
}
```

---

Вряд ли можно сказать, что данный проект безупречен. Нежелательно, чтобы `ChangeUnaffiliatedTransaction` получал информацию об `UnionAffiliation`. Для устранения этой связи абстрактные методы `RecordMembership` и `EraseMembership` помещаются в класс `Affiliation`. Но тогда `UnionAffiliation` и `NoAffiliation` получают сведения о `PayrollDatabase`, что нежелательно<sup>2</sup>.

Обратите внимание, что реализация довольно проста и лишь в незначительной степени нарушит принцип ОСР. Прекрасно, что только несколько модулей системы поддерживают связи с `ChangeUnaffiliatedTransaction`, поэтому в данном случае дополнительные зависимости не будут нарастать подобно снежному кому.

---

<sup>2</sup>Для решения этой проблемы можно воспользоваться шаблоном `Visitor`, но это довольно сложный путь.

## Оплата труда работников

Наконец, пришло время рассмотреть основную транзакцию данного приложения: эта транзакция позволяет реализовать оплату труда определенных работников. На рис. 19.29 показана статическая структура класса `PaydayTransaction`. На рис. 19.30–19.33 описана динамическая модель.

Динамические модели отражают широкий спектр полиморфного поведения.

Алгоритм, поддерживаемый сообщением `CalculatePay`, зависит от типа `PaymentClassification`, который содержится в объекте `Employeee`. Этот алгоритм применяется для уточнения того, зависит ли день выдачи жалованья от типа `PaymentSchedule`, содержащегося в `Employeee`. Алгоритм используется для пересылки суммы оплаты классу `Employeee` в зависимости от типа объекта `PaymentMethod`. Эта высокая степень абстракции позволяет закрывать алгоритмы при добавлении новых разновидностей классификационных оплат, расписаний, объединений или методов платежа.

Алгоритмы, проиллюстрированные на рис. 19.32 и 19.33, вводят понятие *проводки*. После выполнения соответствующих начислений и направления их `Employeee`, оплата “проводится”; то есть, обновляются записи, используемые при оплате. Поэтому метод `CalculatePay` можно определить как функцию подсчета выплат, начиная с последней “проводки” до указанной даты.

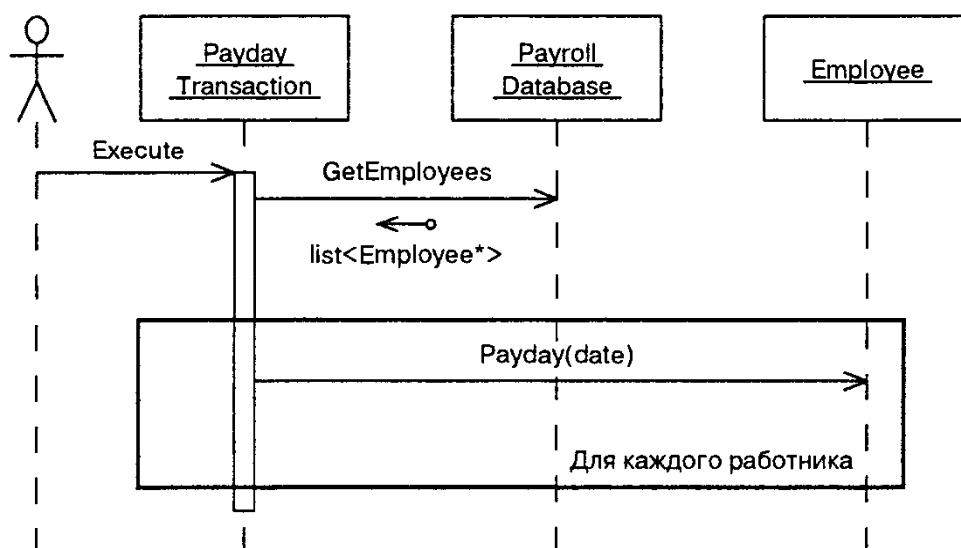
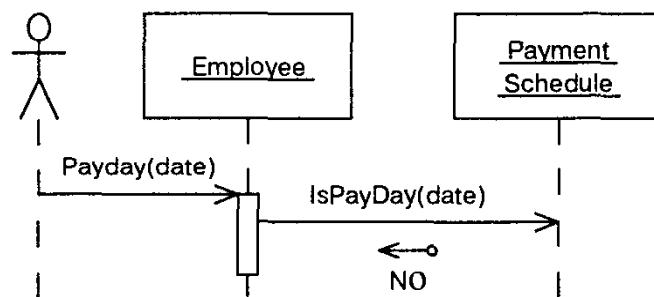
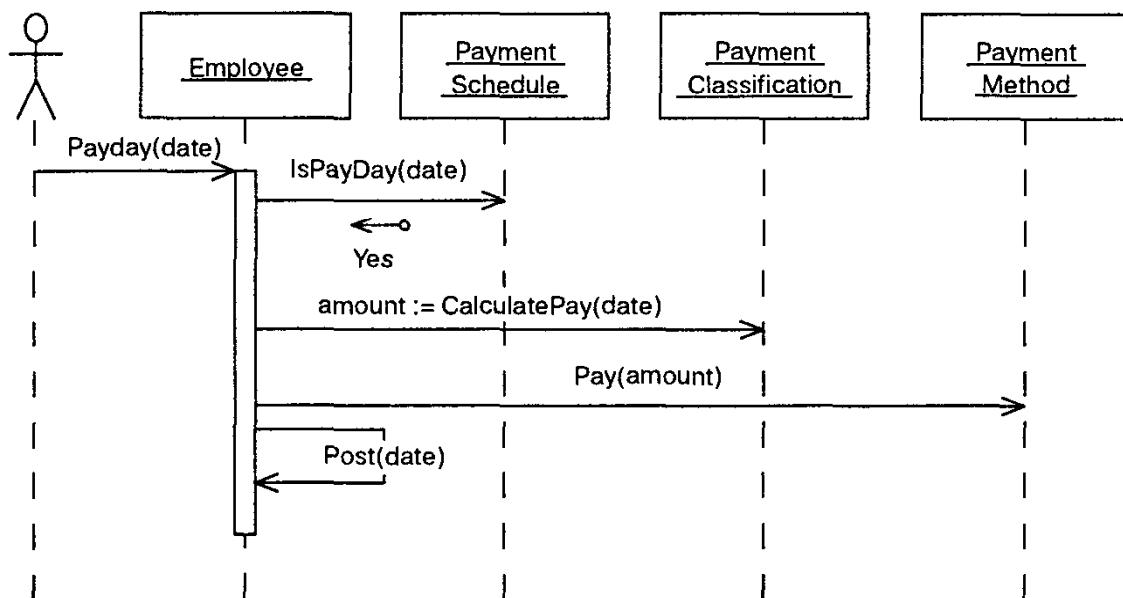


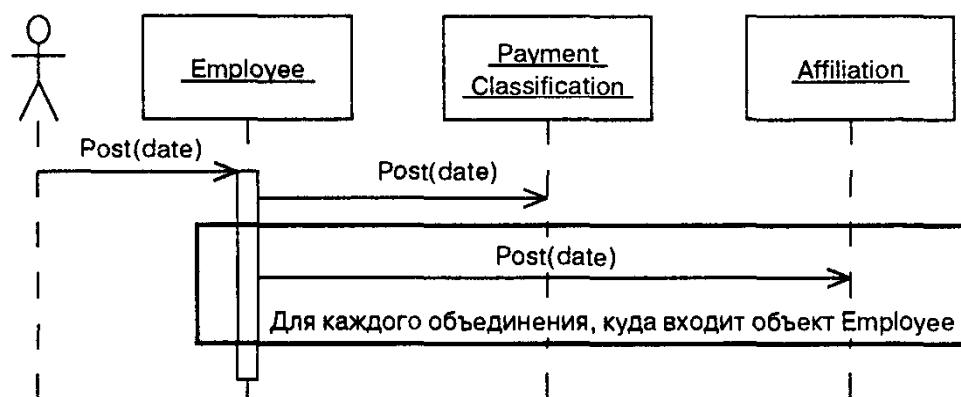
Рис. 19.30. Динамическая модель `PaydayTransaction`



**Рис. 19.31. Динамическая модель сценария “Выплата зарплаты — не сегодня”**



**Рис. 19.32. Динамическая модель сценария “Выплата зарплаты — сегодня”**



**Рис. 19.33. Динамическая модель сценария “Проводка зарплаты”**

## Следует ли разработчикам принимать бизнес-решения?

Откуда пришло понятие “проводки”? Обычно его не упоминают пользователи, оно не используется на практических занятиях по программированию. Оно необходимо для решения возникшей проблемы. К методу `Payday` можно обращаться

несколько раз, используя одну и ту же дату либо дату, относящуюся к одному периоду оплаты. Желательно удостовериться, что работник не получил зарплату за один период несколько раз. Эти ограничения естественны, они не подсказаны заказчиком.

Фактически, речь идет о принятии бизнес-решения. При неоднократном выполнении программы по поддержке платежной ведомости получаем различные результаты. Необходимо переговорить с заказчиком или менеджером проекта по поводу этого обстоятельства.

Из беседы с заказчиком можно заключить, что идея об использовании “проводок” не соответствует его замыслам.<sup>3</sup> Заказчик желает работать с системой по расчету зарплаты, а затем просматривать результирующие записи об оплате. Если же что-то в записях насторожит заказчика, он скорректирует сведения об оплате и снова выполнить программу по расчету зарплаты. Отдельно отмечено, что не следует рассматривать карточки табельного учета или объемы выручки с продаж, если эти сведения поступили после начисления текущих платежей.

Итак, остановимся на схеме “проводок”. Предположительно, идея неплоха, но заказчик ее обычно не поддерживает.

## Оплата труда штатных сотрудников

Листинг 19.36 включает описание двух тестовых случаев. Тестирование позволяет узнать, правильно ли проведено начисление зарплаты штатному работнику. В первом тесте предполагается, что работник получает зарплату в последний день месяца. Второй тест гарантирует, что работник не получит ее до наступления последнего дня месяца.

---

### Листинг 19.36. PayrollTest::TestPaySingleSalariedEmployee & co.

---

```
void PayrollTest::TestPaySingleSalariedEmployee()
{
 cerr << "TestPaySingleSalariedEmployee" << endl;
 int empId = 1;
 AddSalariedEmployee t(empId, "Bob", "Home", 1000.00);
 t.Execute();
 Date payDate(11,30,2001);
 PaydayTransaction pt(payDate);
 pt.Execute();
 Paycheck* pc = pt.GetPaycheck(empId);
 assert(pc);
 assert(pc->GetPayDate() == payDate);
 assertEquals(1000.00, pc->GetGrossPay(), .001);
 assert("Hold" == pc->GetField("Disposition"));
 assertEquals(0.0, pc->GetDeductions(), .001);
 assertEquals(1000.00, pc->GetNetPay(), .001);
```

---

<sup>3</sup>Прекрасно, заказчик – это я.

```

}

void PayrollTest::TestPaySingleSalariedEmployeeOnWrongDate()
{
 cerr << "TestPaySingleSalariedEmployeeWrongDate" << endl;
 int empId = 1;
 AddSalariedEmployee t(empId, "Bob", "Home", 1000.00);
 t.Execute();
 Date payDate(11, 29, 2001);
 PaydayTransaction pt(payDate);
 pt.Execute();
 Paycheck* pc = pt.GetPaycheck(empId);
 assert(pc == 0);
}

```

---

Вернемся назад, к листингу 19.13, где при реализации класса `TimeCard` для представления данных используется тип данных `long integer`. Теперь же речь идет о реальном классе `Date`. Эти два тестовых испытания не выполняются, если не указать, что день выплаты зарплаты совпадает с последним днем месяца.

К классу `Date` автор обращался около 10 лет назад, когда преподавал основы языка C++. Порывшись в архивах, необходимые материалы удалось найти в хранилище данных старого мэйнфрейма.<sup>4</sup> Автор переместил материалы в среду разработки и выполнил компиляцию за считанные минуты. Эти материалы предназначались для работы в Linux и не использовались при работе с Windows 2000.

Листинг 19.37 демонстрирует код функции `Execute()` для `PaydayTransaction`. Она выполняет итерацию по всем объектам `Employee` в базе данных. Каждый работник опрашивается на предмет того, совпадает ли дата этой транзакции с днем выплаты жалованья. Если это так, для работника создается новый платежный документ, и он должен его заполнить.

---

#### Листинг 19.37. `PaydayTransaction::Execute()`

---

```

void PaydayTransaction::Execute()
{
 list<int> empIds;
 GpayrollDatabase.GetAllEmployeeIds(empIds);
 list<int>::iterator i = empIds.begin();
 for (; i != empIds.end(); i++) {
 int empId = *i;
 if (Employee* e = GpayrollDatabase.GetEmployee(empId)) {
 if (<e->IsPayDate(itsPayDate)) {

```

---

<sup>4</sup> В оригиналe отa.com. Речь идет о мэйнфрейме, за который пришлось уплатить 6000 долларов компании, выполнившей проект, который затем отменили. Это была хорошая сделка в 1994 году. Поскольку компьютер до сих пор применяется в сети Object Mentor, можно сделать вывод о том, насколько доброкачественно делали технику раньше.

```
 Paycheck* pc = new Paycheck(itsPayDate);
 itsPaychecks[empId] = pc;
 e->Payday(*pc);
}
}
}
}
```

В листинг 19.38 включен фрагмент кода `MonthlySchedule.cpp`. Обратите внимание, что `IsPayDate` реализуется для возвращения `true` только в том случае, если аргументом является дата, совпадающая с последним днем месяца. Благодаря этому алгоритму можно понять предназначение класса `Date`. Выполнение подобных несложных вычислений трудно реализовать без применения хорошего класса `Date`.

**Листинг 19.38.** MonthlySchedule.cpp (фрагмент)

```
namespace
{
 bool IsLastDayOfMonth(const Date& date)
 {
 int m1 = date.GetMonth();
 int m2 = (date+1).GetMonth();
 return (m1 != m2);
 }
}

bool MonthlySchedule::IsPayDate(const Date& payDate) const
{
 return IsLastDayOfMonth(payDate);
}
```

Листинг 19.39 демонстрирует реализацию Employee::PayDay(). Эта функция представляет обобщенный алгоритм для начисления платежей, и направления их всем работникам. Заметим, что в данном случае широко используется шаблон Strategy. Все подробные вычисления содержатся во включенных стратегических классах: itsClassification, itsAffiliation и itsPaymentMethod.

### Листинг 19.39. Employee::PayDay()

```
void Employee::Payday(Paycheck& pc)
{
 double grossPay = itsClassification->CalculatePay(pc);
 double deductions = itsAffiliation->CalculateDeductions(pc);
 double netPay = grossPay - deductions;
 pc.SetGrossPay(grossPay);
 pc.SetDeductions(deductions);
 pc.SetNetPay(netPay);
 itsPaymentMethod->Pay(pc);
```

}

## Оплата труда работников с почасовой ставкой

Введение оплаты труда работников с почасовой ставкой является хорошим примером доработки тестируемой ранее программы. Начнем с тривиальных тестовых испытаний, а затем усложним их. Ниже показаны тестовые испытания, а потом приводится результирующий код, созданный на их основе.

Листинг 19.40 демонстрирует самый простой вариант. Имя работника с почасовой оплатой вносится в ведомость, и для него начисляется зарплата. Поскольку карточки табельного учета не ведутся, в платежной ведомости может оказаться нулевое значение. Функция `ValidateHourlyPaycheck` выполняет рефакторинг, который реализуется позже. Во-первых, этот код просто утерян внутри тестовой функции. Данный тестовый случай выполняется без внесения каких-либо изменений в оставшуюся часть кода.

---

### Листинг 19.40. `TestPaySingleHourlyEmployeeNoTimeCards`

---

```
void PayrollTest::TestPaySingleHourlyEmployeeNoTimeCards()
{
 cerr << "TestPaySingleHourlyEmployeeNoTimeCards" << endl;
 int empId = 2;
 AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
 t.Execute();
 Date payDate(11, 9, 2001); // пятница
 PaydayTransaction pt(payDate);
 pt.Execute();
 ValidateHourlyPaycheck(pt, empId, payDate, 0.0);
}

void PayrollTest::ValidateHourlyPaycheck(PaydayTransaction& pt,
 int empid,
 const Date& payDate,
 double pay)
{
 Paycheck* pc = pt.GetPaycheck(empid);
 assert(pc);
 assert(pc->GetPayDate() == payDate);
 assertEquals(pay, pc->GetGrossPay(), .001);
 assertEquals("Hold" == pc->GetField("Disposition"));
 assertEquals(0.0, pc->GetDeductions(), .001);
 assertEquals(pay, pc->GetNetPay(), .001);
}
```

---

Листинг 19.41 демонстрирует два тестовых случая. В первом случае проверяется, можно ли оплачивать труд работника после внесения единственной карточки

табельного учета. В другом случае уточняется, можно ли оплачивать сверхурочный труд по карточке табельного учета, где зафиксировано это обстоятельство. Оба эти тестовых случая не создавались одновременно. В действительности, они создавались последовательно.

---

#### Листинг 19.41. Тест... OneTimeCard

---

```
void PayrollTest::TestPaySingleHourlyEmployeeOneTimeCard()
{
 cerr << "TestPaySingleHourlyEmployeeOneTimeCard" << endl;
 int empId = 2;
 AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
 t.Execute();
 Date payDate(11, 9, 2001); // пятница

 TimeCardTransaction tc(payDate, 2.0, empId);
 tc.Execute();
 PaydayTransaction pt(payDate);
 pt.Execute();
 ValidateHourlyPaycheck(pt, empId, payDate, 30.5);
}

void PayrollTest::TestPaySingleHourlyEmployeeOvertimeOneTimeCard()
{
 cerr << "TestPaySingleHourlyEmployeeOvertimeOneTimeCard" << endl;
 int empId = 2;
 AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
 t.Execute();
 Date payDate(11, 9, 2001); // пятница

 TimeCardTransaction tc(payDate, 9.0, empId);
 tc.Execute();
 PaydayTransaction pt(payDate);
 pt.Execute();
 ValidateHourlyPaycheck(pt, empId, payDate, (8 + 1.5) * 15.25);
}
```

---

При рассмотрении первого тестового случая внесены изменения в `HourlyClassification::CalculatePay`. Здесь используется цикл по карточкам табельного учета работников, добавляются часы и умножаются на коэффициент из тарифной сетки. Во втором teste выполнен рефакторинг функции, вычисляющей объем в часах для плановых и сверхурочных работ.

Тестовый случай, код которого приведен в листинге 19.42, гарантирует, что работники-почасовики получат оплату в определенный день месяца (в `PaydayTransaction` используется пятница).

---

#### Листинг 19.42. TestPaySingleHourlyEmployeeOnWrongDate

```
void PayrollTest::TestPaySingleHourlyEmployeeOnWrongDate()
{
 cerr << "TestPaySingleHourlyEmployeeOnWrongDate" << endl;
 int empId = 2;
 AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
 t.Execute();
 Date payDate(11,8,2001); // четверг

 TimeCardTransaction tc(payDate, 9.0, empId);
 tc.Execute();
 PaydayTransaction pt(payDate);
 pt.Execute();
 Paycheck* pc = pt.GetPaycheck(empId);
 assert(pc == 0);
}
```

Листинг 19.43 представляет тестовый случай, гарантирующий начисление платежей для работников, имеющих более одной карточки табельного учета.

#### Листинг 19.43. TestPaySingleHourlyEmployeeTwoTimeCards

```
void PayrollTest::TestPaySingleHourlyEmployeeTwoTimeCards()
{
 cerr << "TestPaySingleHourlyEmployeeTwoTimeCards" << endl;
 int empId = 2;
 AddHourlyEmployee t(empId, "Bill", "Home", 15.25);
 t.Execute();
 Date payDatef11,9,2001); // пятница

 TimeCardTransaction tc(payDate, 2.0, empId);
 tc.Execute();
 TimeCardTransaction tc2(Date(11,8,2001), 5.0, empId);
 tc2.Execute();
 PaydayTransaction pt(payDate);
 pt.Execute();
 ValidateHourlyPaycheck(pt, empId, payDate, 7*15.25);
}
```

Наконец, тестовый случай в листинге 19.44 показывает, что работники получают оплату только по тем карточкам табельного учета, которые сданы в текущий период времени. Карточки табельного учета, относящиеся к другим периодам, не оплачиваются.

#### Листинг 19.44.

#### TestPaySingleHourlyEmployeeWithTimeCardsSpanningTwoPayPeriods

```
void PayrollTest::TestPaySingleHourlyEmployeeWithTimeCardsSpanningTwoPayPeriods()
{
 cerr << "TestPaySingleHourlyEmployeeWithTimeCards" <<
```

```

 "SpanningTwoPayPeriods" << endl;
int empId = 2;
AddHourlyEmployee (empId, "Bill", "Home", 15.25);
t.Execute();
Date payDate(11,9,2001); // пятница
Date dateInPreviousPayPeriod(11,2,2001);
TimeCardTransaction tc(payDate, 2.0, empId);
tc.Execute();
TimeCardTransaction tc2(dateInPreviousPayPeriod, 5.0, empId);
tc2.Execute();
PaydayTransaction pt(payDate);
pt.Execute();
ValidateHourlyPaycheck(pt, empId, payDate, 2*15.25);
}

```

Этот код позволяет постепенно наращивать диапазон рассмотрения. Структура приведенного ниже кода состоит из последовательности тестовых испытаний. Листинг 19.45 отображает соответствующие фрагменты для HourlyClassification.cpp. Используется цикл по обработке карточек табельного учета. Для каждой карточки табельного учета проводится проверка совпадения данных с периодом оплаты. Если ответ положительный, проводится начисление платежей.

#### Листинг 19.45. HourlyClassification.cpp (фрагмент)

```

double HourlyClassification::CalculatePay(Paycheck& pc) const
{
 double totalPay = 0;
 Date payPeriod = pc.GetPayDate();
 map<Date, TimeCard*>::const_iterator i;
 for (i=itsTimeCards.begin(); i != itsTimeCards.end(); i++) {
 TimeCard * tc = (*i).second;
 if (IsInPayPeriod(tc, payPeriod))
 totalPay += CalculatePayForTimeCard(tc);
 }
 return totalPay;
}

bool HourlyClassification::IsInPayPeriod(TimeCard* tc,
 const Date& payPeriod) const
{
 Date payPeriodEndDate = payPeriod;
 Date payPeriodStartDate = payPeriod - 5;
 Date timeCardDate = tc->GetDate();
 return (timeCardDate >= payPeriodStartDate) && (timeCardDate <=
 payPeriodEndDate);
}

double HourlyClassification::
CalculatePayForTimeCard(TimeCard* tc) const

```

```
{
 double hours = tc->GetHours();
 double overtime = max(0.0, hours - 8.0);
 double straightTime = hours - overtime;
 return straightTime * itsRate + overtime * itsRate * 1.5;
}
```

Листинг 19.46 показывает, что `WeeklySchedule` составлено с учетом выплат по пятницам.

---

#### Листинг 19.46. `WeeklySchedule::IsPayDate`

---

```
bool WeeklySchedule::IsPayDate(const Date& theDate) const
{
 return theDate.GetDayOfWeek() == Date::friday;
}
```

---

Читателю в качестве упражнения предлагается выполнить начисления работникам, получающим комиссионные. В этом случае не должно возникнуть трудностей. Более интересный вариант упражнения возникает, если разрешить “проводку” карточек табельного учета по выходным дням, в конце недели, а также корректно высчитать сверхурочные часы работы.

## Периоды выплат: трудности проектирования

Теперь можно рассмотреть реализацию оплаты взносов членов объединения, а также оплаты услуг. После пристального изучения тестового испытания, когда в платежную ведомость вносились штатные работники, оно было взято за основу. После конвертирования записей штатных работников в записи членов объединения, можно начислить им оплату, а затем вычесть из нее взносы. Этот вариант и представлен в листинге 19.47.

---

#### Листинг 19.47. `PayrollTest::TestSalariedUnionMemberDues`

---

```
void PayrollTest::TestSalariedUnionMemberDues()
{
 cerr << "TestSalariedUnionMemberDues" << endl;
 int empId = 1;
 AddSalariedEmployee t(empId, "Bob", "Home", 1000.00);
 t.Execute();
 int memberId = 7734;
 ChangeMemberTransaction cmt(empId, memberId, 9.42);
 cmt.Execute();
 Date payDate(11,30,2001);
 PaydayTransaction pt(payDate);
 pt.Execute();
 ValidatePaycheck(pt, empId, payDate, 1000.0 - ???);
}
```

Обратите внимание на наличие ??? в последней строке тестового случая. Что же там должно находиться? Дело в том, что взносы для объединения собираются еженедельно, а зарплату штатным работникам выдают ежемесячно. Сколько недель в каждом месяце? Обязательно ли умножать еженедельный взнос на четырех? Здесь имеется неточность, которую поможет устранить заказчик<sup>5</sup>.

Заказчик пояснил, что членские взносы в объединение поступают каждую пятницу. Поэтому следует подсчитать количество пятниц, попадающих в оплачиваемый период, и умножить на объем еженедельного взноса. В ноябре 2001 года, а именно этот месяц отражен в тестовом случае, содержится пять пятниц. Поэтому преобразуем тестовое испытание соответствующим образом.

Для подсчета количества пятниц оплачиваемого периода необходимо иметь представление относительно его размеров в целом. Эти вычисления производит функция `IsInPayPeriod` из листинга 19.45 (возможно, вы написали подобную функцию для `CommissionedClassification`). Эта функция применяется функцией `CalculatePay` объекта `HourlyClassification`, что позволяет гарантировать рассмотрение только тех карточек табельного учета, которые соответствуют оплачиваемому периоду. Кажется, что объект `UnionAffiliation` также должен вызывать эту функцию.

Но подождите! Какие задачи выполняет эта функция в классе `HourlyClassification`? Известно, что ассоциативная связь между графиком выплат и классификацией платежей не является устойчивой. Функция, определяющая период выплат должна находиться в классе `PaymentSchedule`, но не в классе `PaymentClassification`!

Интересно, что UML-диаграммы не отражают этого затруднения. Проблема возникла при обсуждении тестовых случаев для `UnionAffiliation`. Это уже второй пример того, как важно при работе над любым проектом кодировать получаемую информацию. Диаграммы, несомненно, полезны, но без дополнительного анализа кодов не обойтись.

Итак, период платежей извлекается из иерархии `PaymentSchedule` и вносится в иерархии `PaymentClassification` и `Affiliation`? Эти иерархии не имеют сведений друг о друге. Определение периода платежей можно поместить в объект `Paycheck`. Сейчас `Paycheck` указывает конечную дату периода платежей. Необходимо также внести туда начальную дату этого периода.

Листинг 19.48 демонстрирует изменения, произведенные в `PaydayTransaction::Execute()`. Обратите внимание, что при создании `Paycheck` туда передаются как начальная, так и конечная даты периода платежей. Если “перепрыгнуть” листинг 19.55, можно заметить, что в `PaymentSchedule` содержатся вычисления обеих дат. Изменения, внесенные в `Paycheck` очевидны.

<sup>5</sup>И снова надо спрашивать у самого себя (автор). Обратитесь на Web-узел [www.google.com/groups](http://www.google.com/groups) и просмотрите статью Роберта Мартина “Schizophrenic”.

**Листинг 19.48. PaydayTransaction::Execute()**

```

void PaydayTransaction::Execute()
{
 list<int> empIds;
 GpayrollDatabase.GetAllEmployeeIds(empIds);

 list<int>::iterator i = empIds.begin();
 for (; i != empIds.end(); i++) {
 int empId = *i;
 if (Employee* e = GpayrollDatabase.GetEmployee(empId)) {
 if (e->IsPayDate(itsPayDate)) {
 Paycheck* pc = new Paycheck(e->
 GetPayPeriodStartDate(itsPayDate), itsPayDate);
 itsPaychecks[empId] = pc;
 e->Payday(*pc);
 }
 }
 }
}

```

Эти две функции в классах HourlyClassification и CommissionedClassification определяют, содержатся ли TimeCards и SalesReceipts в оплачиваемом периоде, а также перемещаются в базовый класс PaymentClassification (листинг 19.49).

**Листинг 19.49. PaymentClassification::IsInPayPeriod(...)**

```

bool PaymentClassification::
IsInPayPeriod(const Date& theDate, const Paycheck& pc) const
{
 Date payPeriodEndDate = pc.GetPayPeriodEndDate();
 Date payPeriodStartDate = pc.GetPayPeriodStartDate();
 return (theDate >= payPeriodStartDate)
 && (theDate <= payPeriodEndDate);
}

```

Теперь можно подсчитать взносы работников в объединение с помощью UnionAffiliation::CalculateDeductions. Код в листинге 19.50 показывает этот процесс. Две даты, определяющие период платежей, извлекаются из платежной ведомости и передаются функции утилиты, подсчитывающей количество пятниц для оплачиваемого периода. Затем это число умножается на величину еженедельного отчисления, и подсчитывается размер взноса за указанный период.

**Листинг 19.50. UnionAffiliation::CalculateDeductions()**

```

namespace
{
 int NumberOfFridaysInPayPeriod(const Date& payPeriodStart,

```

```

 const Date& payPeriodEnd)
{
 int fridays = 0;
 for (Date day = payPeriodStart; day <= payPeriodEnd; day++)
 {
 if (day.GetDayOfWeek() == Date::friday)
 fridays++;
 }
 return fridays;
}

double UnionAffiliation::
CalculateDeductions(Paycheck& pc) const
{
 double totalDues = 0;
 int fridays =
 NumberOfFridaysInPayPeriod(pc.GetPayPeriodStartDate(),
 pc.GetPayPeriodEndDate());
 totalDues = itsDues * fridays;
 return totalDues;
}

```

---

Два последних тестовых случая посвящены оплате членских взносов. Первый тестовый случай показан в листинге 19.51. Использование приведенного программного кода гарантирует правильное удержание означенных сумм.

---

#### Листинг 19.51. PayrollTest::TestHourlyUnionMemberServiceCharge

---

```

void PayrollTest::TestHourlyUnionMemberServiceCharge()
{
 cerr << "TestHourlyUnionMemberServiceCharge" << endl;
 int empId = 1;
 AddHourlyEmployee t(empId, "Bill", "Home", 15.24);
 t.Execute();
 int memberId = 7734;
 ChangeMemberTransaction cmt(empId, memberId, 9.42);
 cmt.Execute();
 Date payDate(11,9,2001);
 ServiceChargeTransaction set(memberId, payDate, 19.42);
 set.Execute();
 TimeCardTransaction tct(payDate, 8.0, empId);
 tct.Execute();
 PaydayTransaction pt(payDate);
 pt.Execute();
 Paycheck* pc = pt.GetPaycheck(empId);
 assert(pc);
 assert(pc->GetPayPeriodEndDate() == payDate);
 assertEquals(8*15.24, pc->GetGrossPay(), .001);
 assertEquals("Hold", pc->GetField("Disposition"));
}
```

```
assertEquals(9.42 + 19.42, pc->GetDeductions(), .001);
assertEquals((8*15.24)-(9.42 + 19.42), pc->GetNetPay(), .001);
}
```

Второй тестовый случай связан с определенными затруднениями. Эти особенности получили отражение в листинге 19.52. Данное тестовое испытание гарантирует, что оплата услуг, даты оказания которых не охватываются периодом платежей, производиться не будет.

#### Листинг 19.52.

#### PayrollTest::TestServiceChargesSpanningMultiplePayPeriods

```
void PayrollTest::
TestServiceChargesSpanningMultiplePayPeriods()
{
 cerr << "TestServiceChargesSpanningMultiplePayPeriods" << endl;
 int empId = 1;
 AddHourlyEmployee t(empId, "Bill", "Home", 15.24);
 t.Execute();
 int memberId = 7734;
 ChangeMemberTransaction cmt(empId, memberId, 9.42);
 cmt.Execute();
 Date earlyDate(11,2,2001); // предыдущая пятница
 Date payDate(11,9,2001);
 Date lateDate(11,16,2001); // следующая пятница
 ServiceChargeTransaction sct(memberId, payDate, 19.42);
 set.Execute();
 ServiceChargeTransaction sctEarly(memberId, earlyDate, 100.00);
 sctEarly.Execute();
 ServiceChargeTransaction sctLate(memberId, lateDate, 200.00);
 sctLate.Execute();
 TimeCardTransaction tct(payDate, 8.0, empId);
 tct.Execute();
 PaydayTransaction pt(payDate);
 pt.Execute();
 Paycheck* pc = pt.GetPaycheck(empId);
 assert(pc);
 assert<pc->GetPayPeriodEndDate() == payDate>;
 assertEquals(8*15.24, pc->GetGrossPay(), .001);
 assert("Hold" == pc->GetField("Disposition"));
 assertEquals(9.42 + 19.42, pc->GetDeductions(), .001);
 assertEquals((8*15.24)-(9.42 + 19.42), pc->GetNetPay(), .001);
}
```

В целях реализации этого подхода желательно, чтобы `UnionAffiliation::CalculateDeductions` вызывал `IsInPayPeriod`.

К сожалению, `IsInPayPeriod` размещается в классе `PaymentClassification` (листинг 19.49). Именно там удобно его поместить, поскольку он является производным модулем `PaymentClassification`, который к нему обращается.

ется. Но теперь к `IsInPayPeriod` обращаются и другие классы. Поэтому функция перемещена в класс `Date`. В конце концов, эта функция просто определяет, находится ли рассматриваемая дата между двумя другими датами (листинг 19.53).

---

**Листинг 19.53. `Date::IsBetween`**

---

```
static bool IsBetween(const Date& theDate,
 const Date& startDate,
 const Date& endDate)
{
 return (theDate >= startDate) && (theDate <= endDate);
}
```

---

Теперь можно завершить формирование функции `UnionAffiliation::CalculateDeductions`. Этот процесс оставлен в качестве упражнения для читателей.

Листинги 19.54 и 19.55 демонстрируют реализацию класса `Employee`.

---

**Листинг 19.54. `Employee.h`**

---

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>

class PaymentSchedule;
class PaymentClassification;
class PaymentMethod;
class Affiliation;
class Paycheck;
class Date;

class Employee
{
public:
 virtual ~Employee();
 Employee(int empid, string name, string address);
 void SetName(string name);
 void SetAddress(string address);
 void SetClassification(PaymentClassification*);
 void SetMethod(PaymentMethod*);
 void SetSchedule(PaymentSchedule*);
 void SetAffiliation(Affiliation*);
 int GetEmpid() const {return itsEmpid;}
 string GetName() const {return itsName;}
 string GetAddress() const {return itsAddress;}
 PaymentMethod* GetMethod() {return itsPaymentMethod;}
 PaymentClassification* GetClassification() {return
 itsClassification;}
 PaymentSchedule* GetSchedule() {return itsSchedule;}
```

```
Affiliation* GetAffiliation () {return itsAffiliation; }

void Payday(Paycheck&);
bool IsPayDate(const Date& payDate) const;
Date GetPayPeriodStartDate(const Date& payPeriodEndDate) const;

private:
 int itsEmpid;
 string itsName;
 string itsAddress;
 PaymentClassification* itsClassification;
 PaymentSchedule* itsSchedule;
 PaymentMethod* itsPaymentMethod;
 Affiliation* itsAffiliation;
};

#endif
```

---

#### Листинг 19.55. Employee.cpp

---

```
#include "Employee.h"
#include "NoAffiliation.h"
#include "PaymentClassification.h"
#include "PaymentSchedule.h"
#include "PaymentMethod.h"
#include "Paycheck.h"

Employee::~Employee()
{
 delete itsClassification;
 delete itsSchedule;
 delete itsPaymentMethod;
}

Employee::Employee(int empid, string name, string address)
: itsEmpid(empid)
, itsName(name)
, itsAddress(address)
, itsAffiliation(new NoAffiliation())
, itsClassification(0)
, itsSchedule(0)
, itsPaymentMethod(0)
{
}

void Employee::SetName(string name)
{
 itsName = name;
}

void Employee::SetAddress(string address)
```

```
{
 itsAddress = address;
}

void Employee::SetClassification(PaymentClassification* pc)
{
 delete itsClassification;
 itsClassification = pc;
}

void Employee::SetSchedule(PaymentSchedule* ps)
{
 delete itsSchedule;
 itsSchedule = ps;
}

void Employee::SetMethod(PaymentMethod* pm)
{
 delete itsPaymentMethod;
 itsPaymentMethod = pm;
}

void Employee::SetAffiliation(Affiliation* af)
{
 delete itsAffiliation;
 itsAffiliation = af;
}

bool Employee::IsPayDate(const Date& payDate) const
{
 return itsSchedule->IsPayDate(payDate);
}

Date Employee::GetPayPeriodStartDate(const Date& payPeriodEndDate) const
{
 return itsSchedule->GetPayPeriodStartDate(payPeriodEndDate);
}

void Employee::Payday(Paycheck& pc)
{
 Date payDate = pc.GetPayPeriodEndDate();
 double grossPay = itsClassification->CalculatePay(pc);
 double deductions = itsAffiliation->CalculateDeductions(pc);
 double netPay = grossPay - deductions;
 pc.SetGrossPay(grossPay);
 pc.SetDeductions(deductions);
 pc.SetNetPay(netPay);
 itsPaymentMethod->Pay(pc);
}
```

## Основная программа

Основная программа по расчету зарплаты может быть описана как цикл, анализирующий транзакции, выполняющиеся на основе введенных данных. На рис. 19.34 и 19.35 отображены статика и динамика основной программы. Программный код не является сложным: `PayrollApplication` содержит в цикле, альтернативно запрашивая транзакции из `TransactionSource` и направляя эти объекты `Transaction` к `Execute`. Заметим, что данная схема отличается от диаграммы на рис. 19.1. Можно наблюдать смещение представления в область большей абстракции.

Класс `TransactionSource` является абстрактным, и его можно реализовать несколькими способами. Статическая диаграмма демонстрирует производный модуль под названием `TextParserTransactionSource`, который просматривает вводный поток текста и анализирует транзакции описанными выше способами. Затем этот объект создает соответствующие объекты `Transaction` и пересыпает их к `PayrollApplication`.

Отделение интерфейса от реализации в `TransactionSource` позволяет источнику транзакций иметь абстрактную природу; например, можно легко устано-

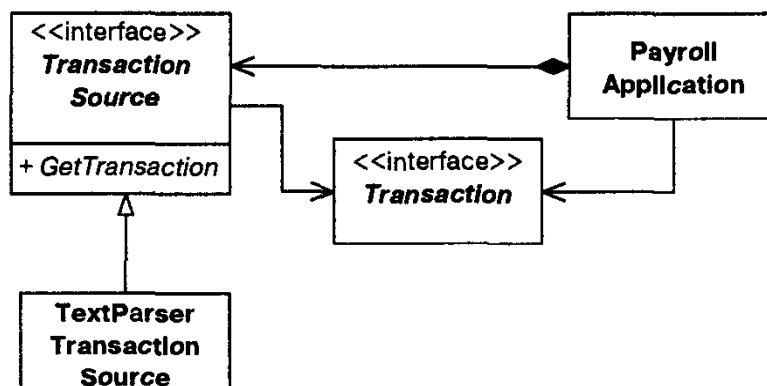


Рис. 19.34. Статическая модель основной программы

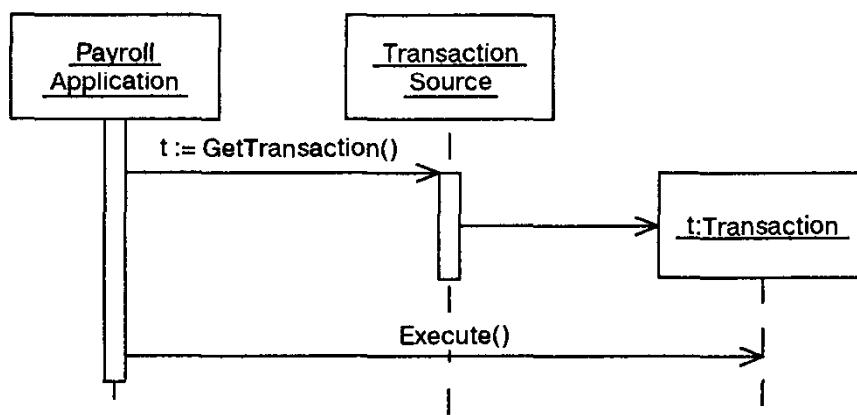


Рис. 19.35. Динамическая модель основной программы

вить контакт между `PayrollApplication` и `GUITransactionSource` либо `RemoteTransactionSource`.

## База данных

После анализа итерации, разработки и реализации рассмотрим роль, отводимую базе данных. Класс `PayrollDatabase` включает неизменные данные. Объекты, содержащиеся в `PayrollDatabase`, должны иметь более “долгую жизнь”, чем любая другая часть приложения. Каким образом это можно реализовать? Ясно, что временный механизм, применяемый в тестовых испытаниях, не может использоваться в реальных системах. У нас есть несколько возможностей.

Можно реализовать `PayrollDatabase` с помощью системы управления объектно-ориентированными базами данных (OODBMS, object-oriented database management system). Тогда реальные объекты будут располагаться в пределах постоянного хранилища базы данных. С точки зрения разработчиков, работы прибавится, поскольку ОО-СУБД не внесет новых вариаций в проект. Одним из преимуществ ОО-СУБД-программ является тот факт, что они почти не влияют на объектную модель приложений. С точки зрения структуры проекта, существование базы данных едва заметно.<sup>6</sup>

Другая возможность состоит в использовании для записи данных простого текстового файла. Начиная с инициализации, объект `PayrollDatabase` просматривает этот файл и формирует в памяти необходимые объекты. В конце программы объект `PayrollDatabase` записывает новую версию этого текстового файла. Конечно, эта возможность не является достаточной для компании, где трудятся сотни и тысячи работников, либо в том случае, когда необходимо в реальном времени получать параллельный доступ к базе данных программы расчета зарплаты. Но описанный подход вполне приемлем для небольшой компании и может применяться в качестве механизма по тестированию оставшихся классов приложения без вторжения в крупную систему базы данных.

Также можно рассмотреть возможность использования системы управления реляционными базами данных (RDBMS, relational database management system) и включения ее в объект `PayrollDatabase`. Затем реализация объекта `PayrollDatabase` сформирует соответствующие запросы к СУБД, что позволит своевременно создавать в памяти необходимые объекты.

Следует отметить, что, в зависимости от приложения, базы данных представляют собой простые механизмы по управлению хранилищами. Они не должны рассматриваться в качестве главного фактора разработки и реализации. Вслед-

---

<sup>6</sup>Это замечание носит оптимистический характер. В простом приложении типа `Payroll` применение ОО-СУБД не оказывает заметного влияния на разработку программы. Но по мере усложнения приложения возрастает и влияние ОО-СУБД на него. Но это воздействие значительно меньше, чем при использовании СУБД.

ствие этого базы данных можно рассматривать в последнюю очередь и довольно подробно<sup>7</sup>. В данном случае осталось вне рассмотрения большое количество интересных возможностей, для реализации которых нужна настойчивость, а также определенные механизмы по проведению тестирования оставшейся части приложения. Также проект не привязан к определенной технологии баз данных или программного продукта. Можно выбирать базу данных по своему усмотрению, опираясь на оставшуюся часть проекта. Авторы только приветствуют возможную замену в будущем базы данных программного продукта.

## Резюме по проекту расчета зарплаты

Около 50 диаграмм и 3300 строк кода демонстрируют процесс разработки и реализации одной итерации приложения по расчету зарплаты. Данный проект зиждется на абстракциях и полиморфизме. В результате большие части проекта довольно устойчивы к различным изменениям политики по проведению платежей. Например, приложение можно изменять с учетом поквартальной выплаты жалованья, используя разработку по ведению обычных выплат и расписание бонусных платежей. Подобное изменение может потребовать *дополнения* к проекту, но мало что изменится в существующем проекте и коде.

В процессе работы над проектом трудно выделить, когда необходимо заниматься анализом, разработкой либо реализацией. Вместо этого все внимание концентрируется на ясности и точности изложения. Там, где это было возможно, предпринимались попытки переходить к абстракциям. В результате получился неплохой начальный проект приложения по расчету зарплаты, также ядро классов, вполне соответствующее рассматриваемым заданиям.

## Историческая справка

Диаграммы, представленные в главе, созданы на основе диаграмм Буча (Booch), описанных в соответствующей главе книги автора *Designing Object-Oriented C++ Applications using the Booch Method*, 1995. Эти диаграммы созданы в 1994 году. При формировании диаграмм автор написал часть кодов по их реализации, что позволило уточнить диаграммы. Но весь приведенный здесь код не создавался в реальности. Поэтому диаграммы не лишены недостатков, иллюстрирующих работу автора.

В каждом случае тестовые случаи записывались перед продуктивным кодом. Во многих случаях эти тесты создавались по частям, тогда использовался и про-

<sup>7</sup>Иногда природа базы данных оказывает существенное влияние на приложение. СУБД поддерживает универсальную систему по обработке запросов и систему отчетности, которая может включаться в перечень требований к приложению. Но даже при наличии четких требований разработчики должны отделять работу над приложением от разработки баз данных. Создаваемое приложение не должно зависеть от типа базы данных.

дуктивный код. Код программы записывался в соответствии с диаграммами там, где это имело смысл. В некоторых случаях от этого правила приходилось отступать, тогда в проект кода вносились изменения.

Первый раз это случилось тогда, когда в нескольких экземплярах `Affiliation` объекта `Employee` обнаружились неточности. В следующий раз так произошло тогда, когда обнаружилось, что в `ChangeMemberTransaction` не учтено членство работника в объединении.

Такой подход является вполне приемлемым. Если разработка ведется без контроля, неизбежны ошибки. Необходимо на практике проверять тестовые испытания и выполнять код, тогда ошибки выявляются значительно легче.

## Источники

Заключительную версию описанного в главе кода можно найти на Web-узле Prentice Hall или по адресу [www.objectmentor.com/PPP](http://www.objectmentor.com/PPP).

## Литература

1. Jacobson I. *Object-Oriented Software Engineering, A Use-Case-Driven Approach*. Wokingham, UK: Addison-Wesley, 1992.

# ЧАСТЬ IV

## Упаковка программы расчета зарплаты

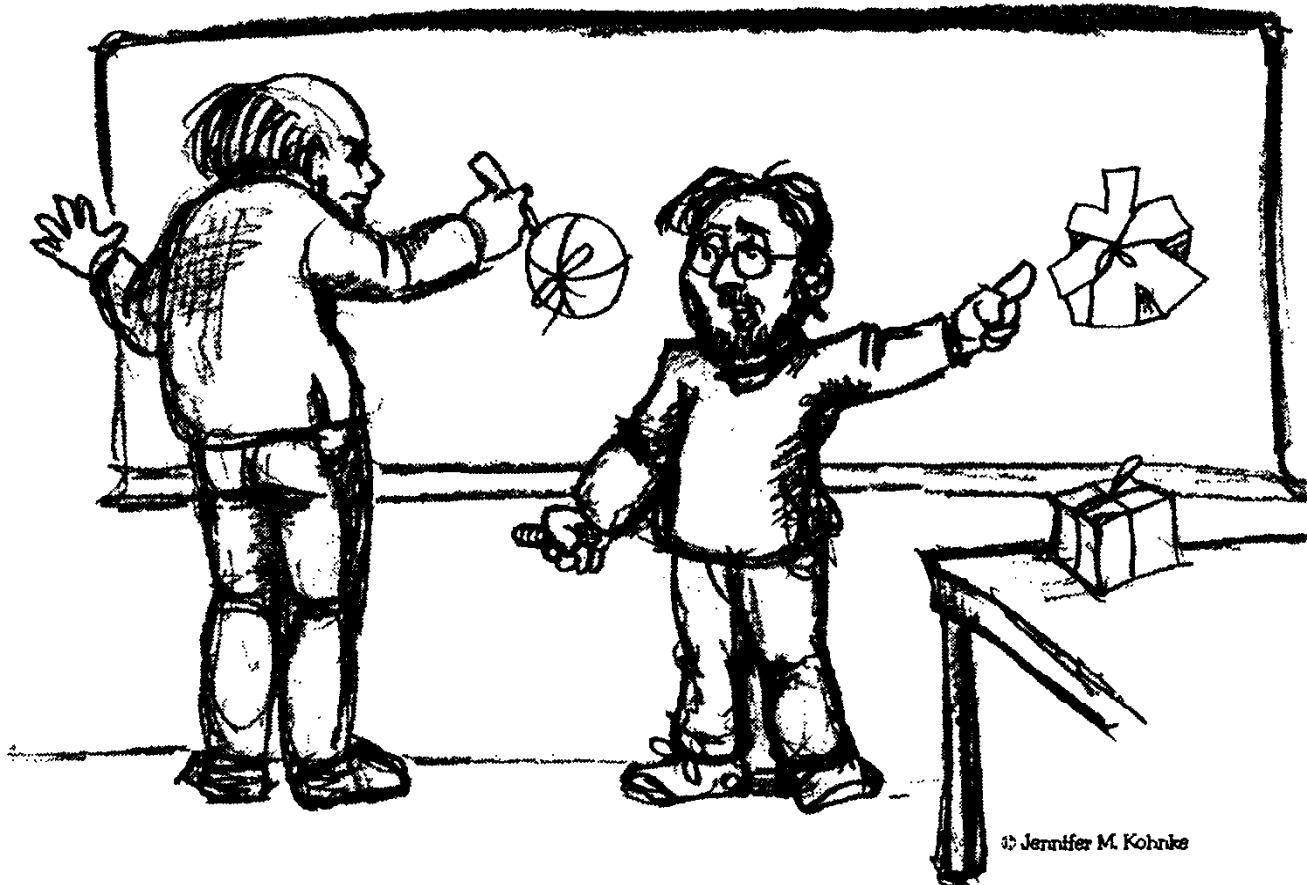


© Jennifer M. Kohnke

В этой части будут рассмотрены принципы проектирования программ, применение которых позволит “разбивать” большие программные системы, формируя на их основе пакеты. В первой главе части производится обзор этих принципов, вторая глава посвящена описанию шаблона, с помощью которого совершенствуется структура, применяемая в процессе формирования пакетов. В третьей главе рассматривается шаблон и основные принципы, которые применяются в отношении программы расчета зарплаты.

# 20

## Принципы упаковки программных проектов



По мере того, как возрастают размеры и сложность программных приложений, требуется их некая высокоуровневая организация. Классы, которые являются весьма удобными в процессе разработки небольших по размеру приложений, ведут к излишней детализации в случае больших по размеру приложений, предназначенных для уровня всей организации. Объект, “превышающий” по размеру класс, облегчает структурирование объемных приложений. Подобный объект называется *пакетом*.

В главе кратко рассматриваются шесть принципов, имеющих отношение к работе с пакетами. Первые три принципа имеют отношение к *цеплению пакетов*. Их применение облегчает выделение классов в пакеты. Остальные три принципа управляют *связыванием пакетов*. Они позволяют определять степень связности пакетов. Последние два принципа также описывают набор *метрик управления зависимостями* (*DM, Dependency Management*). Эти характеристики позволяют разработчикам измерять и характеризовать структуру зависимостей, которая наблюдается в их проектах.

## Пакеты и проектирование

В языке UML пакеты применяются в качестве контейнеров для групп, включающих классы. Группировка классов в пакеты является одной из причин, приводящей к проектированию на более высоком уровне абстракции. Можно также воспользоваться пакетами в целях управления разработкой и распространением ПО. Формулируемая в этом случае цель заключается в распределении классов, имеющих отношение к приложению. Также предусматривается включение классов в состав пакетов, помещенных в сформированных заранее разделах.

Следует учитывать, что классы часто включают зависимости от других классов, причем эти зависимости весьма часто могут выходить за границы пакетов. В результате, пакеты включают взаимосвязи между зависимостями. Взаимосвязи между пакетами выражают высокоуровневую организацию приложения, поэтому нуждаются в управлении.

Ниже представлены основные вопросы, имеющие отношение к пакетам.

1. Каковы принципы распределения классов в пакеты?
2. Каким образом принципы проектирования отражаются на взаимосвязях между пакетами?
3. Должны ли пакеты проектироваться раньше классов (“сверху вниз”)? Или классы следует проектировать раньше пакетов (“снизу вверх”)?
4. В какой форме производится физическое представление пакетов? Средствами языка C++? Средствами языка Java? В среде разработки?
5. Каким образом могут применяться пакеты после завершения их разработки?

В данной главе рассмотрены шесть принципов проектирования, которые управляют созданием, установкой взаимных связей, а также применением пакетов. Первые три принципа ответственны за распределение классов по пакетам. Последние три принципа ответственны за взаимосвязи между пакетами.

## Степень детализации: принципы сцепления пакетов

Три принципа, управляющих сцеплением пакетов, облегчают разработчикам принятие решения относительно распределения классов в пакеты. Точная формулировка принципов зависит от того, были ли исследованы, как минимум, некоторые классы и связи между ними. Поэтому в отношении этих принципов применяется методика распределения на разделы “снизу вверх”.

### Принцип эквивалентности повторно применяемых выпусков (REP, Reuse-Release Equivalence Principle)

*Уровень детализации повторного применения соответствует уровню детализации выпуска.*

Что следует ожидать от разработчика библиотеки классов, которую он планирует повторно использовать? Вполне естественно ожидать хорошо написанную документацию, рабочий программный код, корректно определенные интерфейсы и так далее. Но существуют и другие моменты, которые также следует учитывать.

Во-первых, особую ценность приобретает возможность повторного применения кода, написанного этим разработчиком. В этом случае вам следует убедиться в том, что разработчик гарантирует поддержку разработанного им программного кода. После всего, если вы собираетесь самостоятельно поддерживать в дальнейшем эту программу, это потребует колоссальных затрат времени. Это время лучше потратить на проектирование меньших по размеру (и лучших по сути) пакетов “для внутреннего потребления”.

Во-вторых, было бы неплохо, если бы автор информировал вас заранее о любых планируемых им изменениях интерфейса и функциональных возможностях разработанного им программного кода. Но одного извещения в этом случае будет недостаточно. Разработчик код должен предоставить вам возможность отказа от любых новых версий программного кода. Может случиться так, что на внедрение новой версии программного кода просто не найдется свободного времени, либо изменения кода могут оказаться несовместимыми с вашей системой.

В любом случае, если вы откажетесь от использования новой версии программы, разработчик должен гарантировать поддержку используемой вами старой версии программы. Возможно, что поддержка ограничится тремя месяцами или одним годом, но она должна быть. Разработчик не должен ограничивать вашу “свободу” и отказывать в поддержке. Если же разработчик не соглашается поддерживать устаревшие версии ПО, следует серьезно подумать о том, следует ли пользоваться его программой и подвергаться опасности непредсказуемых изменений кода.

Решаемый в данном случае вопрос носит "политический" характер. Ответ зависит от величины моральных и материальных издержек, которые должны нести другие пользователи в случае, если примут решение о повторном применении кода. Эти издержки могут оказаться серьезное влияние на упакованную структуру ПО. В целях обеспечения гарантий, требуемых приверженцами повторного использования кода, разработчикам следует организовать свои программы в виде повторно применяемых пакетов, а затем отслеживать эти пакеты в различных выпусках.

Принцип REP утверждает, что уровень детализации повторного использования (пакет) не может быть меньшим, чем уровень детализации выпуска. Все повторно используемые программы также должны выпускаться и отслеживаться. Вряд ли реально, чтобы разработчик создал класс, а затем требовал, чтобы этот класс повторно использовался. Реальное повторное использование возможно в том случае, если включено отслеживание системы, в результате чего гарантируется своевременно извещение, безопасность, а также необходимая поддержка.

Принцип REP дает нам первую подсказку относительно разбиения проекта на пакеты. Поскольку повторное использование основывается на пакетах, повторно применяемые пакеты должны включать повторно используемые классы. Поэтому, по крайней мере, некоторые пакеты обязаны включать повторно используемые наборы классов.

Конечно, если "политические" соображения оказывают влияние на разбиение ПО, это вызывает некоторое беспокойство. Причина тревоги заключается в том, что любая программа весьма отличается от исключительно математической сущности, которая может быть структурирована в соответствии со строгими математическими правилами. На самом деле любая программа — это продукт человеческого труда. Программы создаются и применяются людьми. И если программное обеспечение будет использоваться повторно, оно должно разбиваться на пакеты таким образом, чтобы не создавать неудобства пользователям.

А теперь обратимся к внутренней структуре пакета. Любое внутреннее содержимое должно рассматриваться с точки зрения потенциального повторного использования. Если пакет включает программу, которая будет повторно использована, он не должен одновременно содержать программу, которая не предназначена для повторного использования. *Либо все классы, входящие в состав пакета, являются повторно используемыми, либо ни один из них не входит в эту категорию.*

К понятию повторного использования нельзя подходить как к некоему критерию; здесь следует учитывать личность того, кто намеревается повторно использовать программу. Пусть библиотека контейнера классов повторно используется, но в то же время и формирует схему для финансовых расчетов. Не всегда следует включать эти возможности в состав одного и того же пакета. Некоторые пользователи захотят воспользоваться библиотекой контейнеров классов, но вряд ли их привлекут возможности схемы для финансовых расчетов. Поэтому весьма

желательно, чтобы все классы, входящие в состав пакета, повторно использовались одной и той же аудиторией. Вряд ли будет целесообразно, если пакет будет включать классы, некоторые из которых подходят пользователям, а другие — совершенно не подходят.

## **Принцип общего повторного использования (CRP, Common-Reuse Principle)**

*Классы, входящие в состав пакета, могут повторно использоваться совместно. Если вы можете повторно использовать один из классов, входящих в состав пакета, вам доступны все остальные классы.*

Именно этот принцип позволяет принять решение о том, следует ли рассматриваемые классы включать в пакет. Его смысл заключается в том, что совместно используемые классы принадлежат одному и тому же пакету.

Достаточно редко реализуется совместное использование классов само по себе. Как правило, совместно используемые классы взаимодействуют с другими классами, которые являются частью абстракции повторного использования. Принцип CRP утверждает, что эти классы входят в состав одного и того же пакета. В подобный пакет могут включаться классы, между которыми устанавливается множество зависимостей.

В качестве простого примера можно рассматривать контейнерный класс, а также связанные с ним итераторы. Эти классы повторно используются (одновременно), поскольку они тесно связаны друг с другом. Поэтому они могут входить в состав одного и того же пакета.

Но принцип CRP способен сообщить намного больше информации, чем сведения о классах, входящих в пакет. В этом случае мы можем узнать о классах, которые *не* входят в пакет. Если один пакет использует другой, формируется зависимость между ними. Может случиться так, что использующий пакет применяет один класс внутри используемого пакета. Однако при этом не может ослабляться зависимость в целом. Использующий пакет по-прежнему зависит от используемого пакета. При каждом повторном выпуске пакета требуется повторная проверка и выпуск использующего его пакета. Это утверждение справедливо даже в том случае, если был повторно выпущен используемый пакет. Это связано с тем, что потребуется отразить изменения в классе, который использует пакет.

Общепринятой является физическое представление пакетов в виде общих библиотек, DLL и JAR. Если используемый пакет выпущен в качестве JAR, использующий код зависит от всего JAR. Любая модификация JAR (даже если она связана с классом, к которому использующий код не имеет отношения), по-прежнему приводит к выпуску новой версии JAR. Эта новая версия должна распространяться повторно, а использующий код должен быть повторно проверен.

Таким образом, следует убедиться в том, что если существует зависимость от пакета, должна устанавливаться зависимость от каждого класса, входящего в состав этого пакета. *Другими словами*, потребуется проверить, что классы, помещаемые в пакет, являются неотделимыми. В результате невозможно установить зависимость от других классов, но не сделать этого по отношению к другим классам. Иначе придется выполнить избыточный объем повторной проверки и распространения, что влечет за собой значительные трудозатраты.

Следовательно, принцип CRP сообщает больше информации о классах, которые не должны совместно применяться, а не о классах, которые будут совместно применяться. Этот принцип утверждает, что классы, которые не являются тесно связанными, не относятся к одному и тому же пакету.

## **Принцип общего закрытия (CCP, Common-Closure Principle)**

*Классы, входящие в состав пакета, должны быть закрыты по отношению к одним и тем же изменениям. Изменение, влияющее на пакет, оказывает воздействие на все классы, входящие в этот пакет (не затрагивая другие пакеты).*

На самом деле, в данном случае идет речь о принципе персональной ответственности (SRP, Single-Responsibility Principle), распространенном на пакеты. Подобно тому, как принцип SRP утверждает о том, что для изменения класса не должно быть несколько причин, рассматриваемый принцип утверждает, что для изменения пакета также не должно быть нескольких причин.

Для большинства приложений возможность поддержки является более важной, чем повторное использование. Если код приложения должен изменяться, лучше, когда эти изменения происходят в одном пакете, чем “распыляются” на множество пакетов. Если изменения сосредоточены в одном пакете, нам потребуется выпустить только один измененный пакет. Другие пакеты, которые не зависят от измененного пакета, не нужно повторно проверять и выпускать.

Принцип CCP предлагает собрать в одном месте все классы, которые могут изменяться в силу одних и тех же причин. Если два класса связаны достаточно сильно (физически или концептуально), они будут всегда изменяться вместе. Именно поэтому они относятся к одному пакету. Благодаря этому минимизируется рабочая нагрузка, связанная с повторными выпусками, проверкой, а также распределением ПО.

Этот принцип тесно связан с принципом открытия-закрытия (OCP, Open-Closed Principle). Рассмотрим “закрытие” с точки зрения принципа OCP. Принцип OCP утверждает, что классы должны быть закрыты для изменений, но открыты для расширения. Но, как изучалось раньше, стопроцентное закрытие просто недостижимо. Закрытие носит характер стратегии. Мы проектируем системы та-

ким образом, что они являются закрытыми для большинства практикуемых нами типов изменений.

Принцип CCP подчеркивает описанные моменты путем группировки классов, которые являются открытыми для определенных типов изменений, выполняемых в одних и тех же пакетах. Поэтому в случае изменения требований затрагивается минимальное количество пакетов.

## Сцепление пакетов: резюме

Ранее был распространенным более упрощенный взгляд на сцепление (по сравнению с тем, который предлагается описанными тремя принципами). Предполагалось, что сцепление — это просто атрибут модуля, который выполняет одну и только одну функцию. Оказалось, что три принципа сцепления пакетов описывают довольно богатый набор функций. В процессе выбора классов, группируемых в пакеты, следует учитывать “противоборствующие” силы, имеющие значение для способности к повторному применению и к разработке. Балансирование между этими силами с учетом потребностей приложения является нетривиальной задачей. Более того, подобное балансирование всегда носит динамический характер. Поэтому методы разбиения приложения с образованием пакетов, принятые сегодня, могут кардинальным образом измениться в следующем году. Поэтому композиция пакетов динамически развивается по мере того, как смещается акцент в процессе разработки проекта — от способности к разработке до возможности повторного использования.

## Устойчивость: принципы связывания пакетов

Три принципа описывают взаимосвязи между пакетами. И снова мы встречаем “натянутые отношения” между способностью к разработке и логическими методами проектирования. Силы, которые оказывают влияние на архитектуру структуры пакетов, бывают техническими, “политическими” и изменчивыми.

### Принцип ациклических зависимостей (ADP, Acyclic-Dependies Principle)

*Не допускается наличие циклов в графе, описывающем зависимости пакета.*

Наверняка вы сталкивались с такой ситуацией, когда вы приходите утром в офис и обнаруживаете, что коллектив находится в “нерабочем состоянии”? В чем причина? Наверное, все дело в том, что кое-кто, задержавшись на работе допоздна, изменил нечто важное, сказывающееся на всей дальнейшей работе! Подобная ситуация называется “утренним синдромом”.

Чаще всего подобный синдром проявляется в том случае, если доступ к среде разработки имеют многие разработчики, изменяющие одни и те же исходные



файлы. В относительно небольших проектах, где задействовано несколько разработчиков, особой проблемы в этом случае не возникает. Но по мере роста размера проекта и количества участников в нем разработчиков “утренние синдромы” могут плавно перерасти вочные кошмары. Не секрет, что во многих командах разработчиков на протяжении нескольких недель невозможно получить стабильную версию проекта. Это связано с тем, что каждый программист производит бесконечные модификации кода в надежде привести его в “работоспособное состояние” после того, как его коллега ранее изменил какой-либо модуль кода.

За истекшие десятилетия программистской практики сформировались два решения, призванные искоренить эту проблему. Оба решения позаимствованы из телекоммуникационной индустрии. Первое заключается в формировании “еженедельных выпусков”, второе — в использовании принципа ADP.

## Еженедельные выпуски

Эта практика является общепринятой при работе со средними по размеру проектами. В этом случае применяется следующий алгоритм. На протяжении первых четырех рабочих дней разработчики работают “в изоляции”. Они создают некие частные копии кода и не задумываются от интеграционных проблемах. По пятницам они интегрируют наработанные программы и выполняют построение системы.

Преимущество этого подхода заключается в том, что разработчики могут работать в уединении на протяжении четырех рабочих дней. Недостаток связан с тем, что приходится заниматься “интеграционной штурмовщиной” по пятницам.

К сожалению, по мере роста размера проекта завершение интеграционных операций в пятницу становится просто нереальным. Приходится выполнять сверхурочные работы по субботам. Несколько подобных рабочих суббот убеждают разработчиков в том, что такого рода работы лучше начинать по четвергам. В связи с этим выполнение интеграции плавно “переползает” на середину рабочей недели.

По мере увеличения доли интеграционных работ, эффективность команды разработчиков понижается. Обычно это ведет к подавленному настроению среди разработчиков или менеджеров проектов, которые склоняются к мысли о необходимости формирования выпусков программы раз в две недели. Это приводит к определенной экономии времени, но затраты времени на выполнение интеграции растут по мере увеличения размеров проектов.

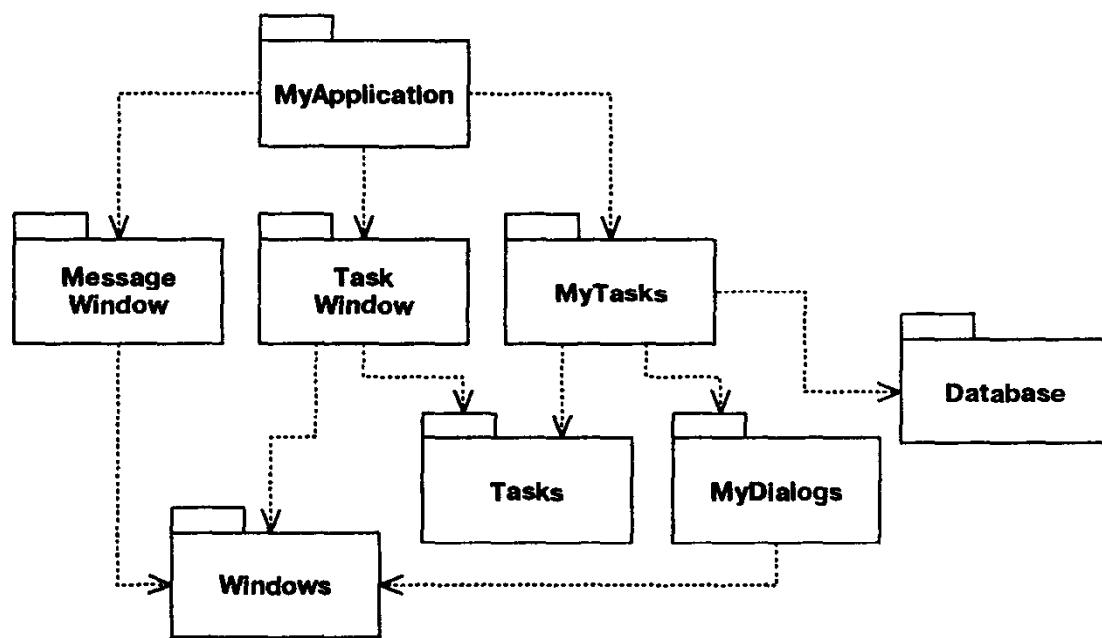
Результатом описанной ситуации является неизбежное наступление кризиса. В целях сохранения приемлемого уровня эффективности выполняется практически непрерывное “растягивание” календарного графика выпусков программы. В результате возрастают степень риска. Постоянно затрудняется выполнение операций, связанных с интеграцией и тестированием, что приводит к нивелированию преимуществ, обеспечиваемых быстрой обратной связью.

## **Исключение циклов зависимостей**

Разрешение описанной проблемы заключается в разбиении среды разработки, когда образуются повторно выпускаемые пакеты. Эти пакеты представляют собой рабочие модули, которые могут проверяться разработчиком или командой разработчиков. Как только разработчики получают рабочий пакет, они выпускают его для использования другими разработчиками. Данному пакету присваивается номер выпуска, после чего он перемещается в каталог, где доступен для использования другими командами разработчиков. Участники этих команд продолжают работы модификации пакетов в своих частных зонах. Другие пользователи работают с повторно выпущенными версиями.

По мере того как появляются новые выпуски пакетов, другие команды разработчиков принимают решение о том, стоит ли переходить к немедленной адаптации нового выпуска. Если было принято отрицательное решение, продолжается использование прежнего выпуска. Как только будет достигнуто состояние готовности, осуществляется переход к работе с новым выпуском.

В результате члены команды разработчиков “не страдают излишним милосердием” по отношению к другим пользователям. Изменения, выполненные в одном пакете, не приводят к непосредственному воздействию на другие команды разработчиков. Каждая команда решает для себя, когда следует приступить к адаптации используемых ими пакетов в соответствии с новыми выпусками этих же пакетов. Более того, интеграция осуществляется “мелкими шагками”. Не существует чет-



**Рис. 20.1.** Структура пакетов имитируется ациклическим ориентированным графом

ко определенного момента времени, когда все разработчики обязаны собраться вместе и интегрировать все, что “было наработано непосильным трудом”.

Описанный процесс является очень простым и рациональным, что способствовало его повсеместному распространению. Однако для приведения его в “рабочее состояние” требуется управление структурой зависимостей пакетов. При этом следует исключить циклы. Если в структуре зависимостей появляются циклы, избежать “утреннего синдрома” не удастся.

Обратите внимание на диаграмму пакетов, изображенную на рис. 20.1. Здесь мы имеем дело с типичной пакетной структурой, “собранной” в одно приложение. Выполняемые этим приложением функции несущественны с точки зрения назначения этого примера. Существенной является структура зависимостей между пакетами. Обратите внимание, что эта структура моделируется *ориентированным графом*. Пакеты имитируются вершинами графа, а связи между зависимостями — *ребрами графа*.

Теперь обратите внимание на следующее. Независимо от того, с какого пакета вы начинаете, невозможно следовать взаимосвязи между зависимостями, а потом вернуться к исходному пакету. В данном случае мы имеем дело с классическим примером *ациклического ориентированного графа*.

Если команда разработчиков, ответственная за формирование MyDialogs, разрабатывает новую версию этого пакета, легко проследить связанные с ними другие пакеты. Для этого достаточно следовать по ребру графа взаимосвязей между зависимостями в направлении, противоположном указанному стрелкой. В этом случае задействуются пакеты MyTasks и MyApplication. Разработчи-

ки, ответственные за эти пакеты, вольны сами принимать решение относительно того, когда следует выполнять интеграцию с новым выпуском MyDialogs.

Обратите внимание, что новые выпуски пакетов MyDialogs не оказывают немедленного воздействия на многие другие системные пакеты. Они ничего не “знают” о MyDialogs и не “заботятся” относительно его изменений. И этот момент является положительным, поскольку означает, что влияние выпущенных версий MyDialogs является относительно небольшим.

Как только разработчики, работающие над пакетом MyDialogs, переходят к его тестированию, им потребуется скомпилировать и скомпоновать MyDialogs с выпусками пакетов Windows. При этом не требуется использование каких-либо других системных пакетов. Это означает, что разработчикам, имеющим дело с MyDialogs, предстоит относительно небольшой объем работ по установке тестов. Именно поэтому требуется учитывать лишь несколько переменных.

Как только наступает время для выпуска всей системы, применяется подход “снизу вверх”. Сначала компилируется, тестируется и выпускается пакет Windows. Затем приходит очередь пакетов MessageWindow и MyDialogs. После этого обрабатываются пакеты Task, TaskWindow и Database. После чего наступает очередь пакетов MyTasks и MyApplication. Этот процесс является весьма “прозрачным”, а его реализация не представляет особого труда. Методы построения системы становятся известными после того, как мы постигаем взаимосвязи между отдельными ее компонентами.

## **Проявление циклического эффекта в графе, моделирующем взаимосвязи между пакетами**

Предположим, что вновь сформулированное требование приводит к тому, что изменяется один из классов пакета MyDialogs таким образом, что вовлекается класс из пакета MyApplication. При этом создается цикл зависимости, показанный на рис. 20.2.

После возникновения этого цикла появляется ряд проблем. Например, пусть разработчики работают с пакетом MyTasks, который будет выпущен в ближайшее время. Этот пакет должен быть совместим с пакетами Task, MyDialogs, Database и Windows. При наличии циклов требуется совместимость с классами MyApplication, TaskWindow и MessageWindow. При этом MyTasks зависит от *каждого другого пакета в системе*. В результате выпуск MyTasks весьма затруднен. Пакету MyDialogs уготована аналогичная судьба. Фактически наличие цикла приводит к тому, что выпуски MyApplication, MyTasks и MyDialogs всегда формируются в одно и то же время. По сути они образуют один большой пакет. И все разработчики, работающие с одним из этих пакетов, испытывают “утренний синдром”.

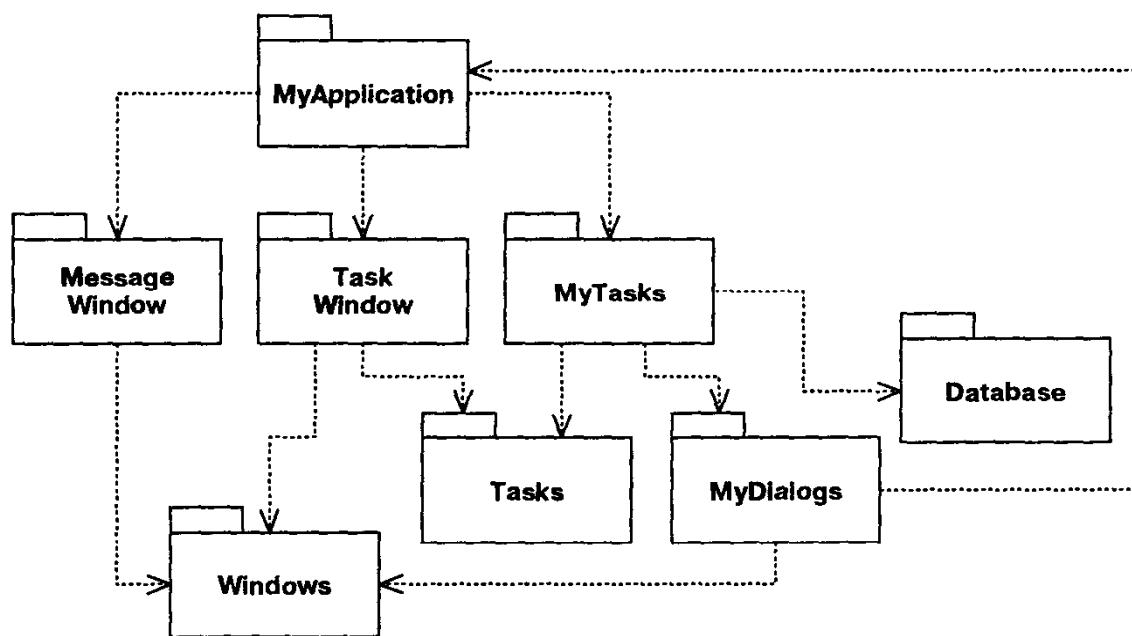


Рис. 20.2. Пакетная диаграмма, включающая цикл

Они будут испытывать затруднения в том случае, если придется использовать один и тот же выпуск для каждого другого пакета.

Но это лишь часть возможных затруднений. Рассмотрим, например, что будет происходить в том случае, если потребуется протестировать пакет MyDialogs. В этом случае придется выполнить связывание с каждым другим пакетом в системе, включая пакет Database. Это означает, что нам придется сформировать *завершенный выпуск только для того*, чтобы протестировать MyDialogs. Вряд ли этот факт будет приветствоваться.

Если у вас до сих пор вызывает удивление тот факт, что приходится выполнять компоновку с привлечением столь большого количества различных библиотек, а также необходимость работы с привлечением большого количества персонала, просто выполните простой модульной тест для одного из разработанных вами классов. Успех этого теста определяется наличием циклов в графе зависимостей. Но из-за этих же циклов весьма затрудняется изоляция модулей. Тестирование и выпуск модулей становятся весьма затруднительными и чреваты появлением ошибок. И, например, в C++ время компиляции растет в геометрической прогрессии с ростом количества модулей.

Наличие циклов в графе зависимостей весьма затрудняет работу по формированию пакетов. По крайней мере, становится затруднительным придерживаться корректного порядка следования пакетов. Следствием этого является ряд весьма неприятных проблем при работе с такими языками, как Java, которые считывают объявления пакетов из скомпилированных двоичных файлов.

## Разбиение цикла

Практически всегда можно разбить цикл на пакеты, а также повторно сформировать граф зависимостей (например, ациклический ориентированный граф).

1. Примените принцип инверсии зависимостей (DIP, Dependency-Inversion Principle). Как показано на рис. 20.3, можно создать абстрактный базовый класс, который служит интерфейсом для пакета MyDialogs. Затем можно поместить абстрактный базовый класс в пакет MyDialogs и наследовать его в MyApplication. При этом инвертируется зависимость между MyDialogs и MyApplication, разбивая тем самым цикл (рис. 20.3). И снова обратите внимание на то, что интерфейс был именован после клиента, а не после сервера. Это еще одно проявления действия правила, определяющего принадлежность интерфейсов клиентам.
2. Создайте новый пакет, от которого зависят пакеты MyDialogs и MyApplication. Переместите зависимые классы в новый пакет (рис. 20.4).

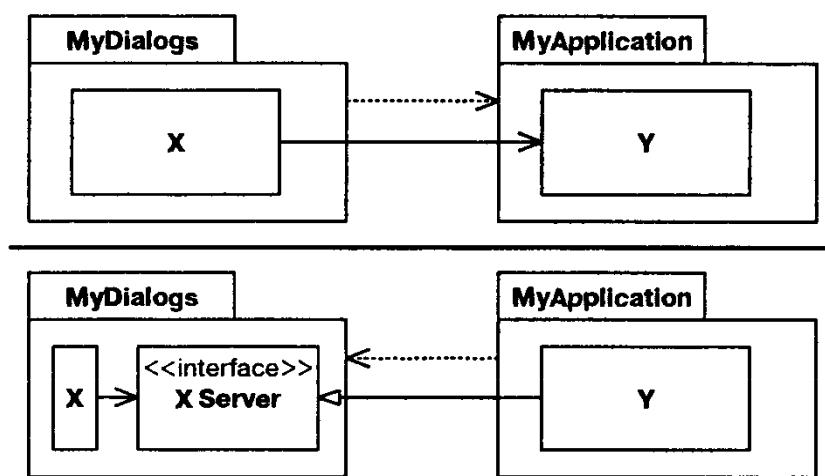


Рис. 20.3. Разбиение цикла с применением инверсии зависимостей

## Флуктуации

Следствием использования второго решения является то, что структура пакета “безразлична” к присутствию изменяющихся требований. Естественно, что, по мере роста объема приложений, структура зависимостей пакета “разбухает” и испытывает флуктуации. Именно поэтому структуру зависимостей следует отслеживать на предмет возможного наличия циклов. По мере обнаружения циклов, следует их немедленно устранять. Иногда это будет означать создание новых пакетов, приводящих к расширению структуры зависимостей.

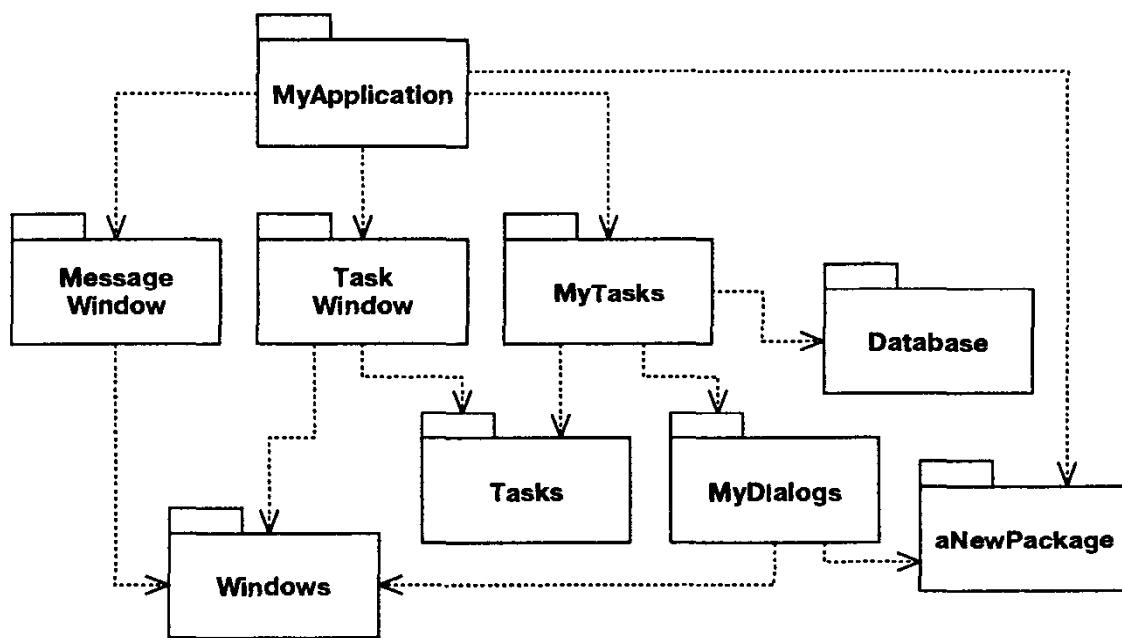


Рис. 20.4. Разбиение цикла с созданием нового пакета

## Проектирование “сверху вниз”

Вопросы, которые рассматривались до сих пор, неизбежно ведут к одному выводу: структура пакета не может быть спроектирована в соответствии с методикой разработки “сверху вниз”. Это означает, что изначально невозможно продумать все детали будущего проекта. Они появляются по мере роста и изменения системы.

Вас может смутить наличие противоречия с тем, что подсказывает интуиция. Мы вправе ожидать появление крупноячеистой декомпозиции (с образованием пакетов), а также высокоуровневой *функциональной* декомпозиции. Вместо этого мы наблюдаем крупноячеистую группировку, напоминающую структуру зависимостей пакетов. При этом появляется ощущение, что пакеты должны представлять некие весьма важные функции системы. Причем эти функции могут и не быть представленными в качестве неких атрибутов на диаграмме зависимостей пакетов.

На самом деле диаграммы зависимостей пакетов являются достаточно небольшими и не могут полностью описывать все функции приложения. Они являются своего рода картой, на которой отображена *возможность создания выпусков* приложения. Причина такого положения дел заключается в отсутствии многих определенных подробностей в начале осуществления проекта. Поскольку отсутствует ПО, выпуски которого необходимо готовить, формировать карту не нужно. Но по мере того как все больше и больше классов аккумулируют ранние стадии реализации и проектирования, появляется потребность в управлении зависимостями. В результате открывается возможность разработки проекта без “утреннего синдрома”. Более того, возникает желание сохранять по возможности локальный характер изменений, вследствие чего повышенное внимание начинает уделяться

соблюдению принципов SRP и CCP, а также взаимному расположению классов, которые будут “меняться ролями”.

По мере роста размера приложений, возникает потребность в создании совместно используемых элементов. В результате принцип CRP начинает диктовать необходимость выполнения композиции пакетов. И наконец, после отображения циклов применяется принцип ADP, в результате чего граф зависимостей пакетов подвергается флуктуации, а его размер растет.

Если вы попытаетесь спроектировать структуру зависимостей пакетов до того, как были разработаны какие-либо классы, скорее всего, вы потерпите неудачу. Мы не знакомы с подробностями общих принципов закрытия, с какими-либо повторно используемыми элементами, и практически наверняка создаваемые нами пакеты продуцируют циклы зависимостей. В результате происходит рост структуры зависимостей пакетов и формируется логический дизайн системы.

## **Принцип устойчивых зависимостей (SDP, Stable-Dependencies Principle)**

*Зависит от направления стабильности.*

Проекты не могут быть полностью статичными. Некая “степень свободы” просто жизненно необходима. Этого можно добиться, воспользовавшись принципом общего закрытия (CCP, Common-Closure Principle). Именно данный принцип позволяет создавать пакеты, которые чувствительны к определенным видам изменений. Эти пакеты по своей природе являются изменчивыми. Мы *ожидаем*, что они будут изменяться.

Изменчивые пакеты не должны зависеть от пакетов, изменение которых сопряжено со значительными трудностями! Иначе модифицировать изменчивые пакеты будет затруднительно.

Достаточно неприятной выглядит ситуация, когда модуль, который изначально должен легко модифицироваться, изменяется с большим трудом и только потому, что кто-то “навесил” на него зависимость. Причем речь идет не об одной строке кода, а о модуле в целом, который совершенно неожиданно приобрел статический характер. Воспользовавшись принципом SDP, мы сможем гарантировать, что изменяемые модули не зависят от тех модулей, модификация которых затруднена.

### **Устойчивость**

Поставьте монету на ребро. Устойчиво ли ее положение? Скорее нет, чем да. Но если монету не трогать, она может оставаться в этом положении сколь угодно долго. Поэтому понятие стабильности напрямую не связано с понятием частых изменений. Монете, стоящей на ребре, не свойственна склонность к изменениям, но ей также не присуща устойчивость.

В словаре Вебстера определяется устойчивость как “невозможность простого перемещения”<sup>1</sup>. Понятие устойчивости напрямую связано с величиной усилий, требуемых для выполнения изменений. С этой точки зрения, монета не считается устойчивой, поскольку незначительное усилие способно привести к ее падению. С другой стороны, стол является весьма устойчивым, поскольку для его перемещения требуются весьма значительные усилия.

Какое отношение сказанное имеет к разработке программ? Существует множество факторов, затрудняющих изменение программных пакетов: размер, сложность, “прозрачность” и так далее. В дальнейшем мы собираемся игнорировать все упомянутые факторы и сосредоточиться на кое-чем другом. Одним из способов затруднить изменение программного пакета является наличие множества других программных пакетов, которые зависят от него. Пакет, имеющий множество входящих зависимостей, является очень устойчивым. В этом случае потребуются серьезные усилия для согласования изменений во всех зависимых пакетах.

На рис. 20.5 показан устойчивый пакет, X. Этот пакет включает три зависимых пакета. В силу этого существуют, по крайней мере, три причины для устойчивости. В этом случае мы говорим, что пакет *ответственен* за три упомянутых пакета. С другой стороны, X ни от чего не зависит, поэтому он не оказывает внешнее воздействие, ведущее к появлению изменений. В этом случае говорят, что пакет является *независимым*.

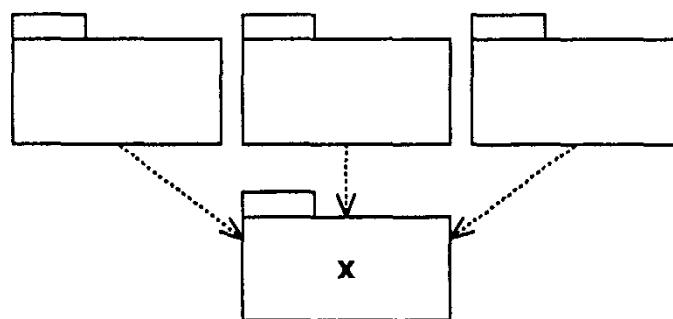


Рис. 20.5. X: устойчивый пакет

На рис. 20.6 демонстрируется неустойчивый пакет. Здесь для Y не существует зависимых пакетов. В этом случае говорят, что Y не несет какой-либо ответственности. Пакет Y также включает три пакета, от которых он зависит, поэтому изменения должны исходить из трех внешних источников. Говорят, что пакет Y является *зависимым*.

## Метрики устойчивости

Каким же образом следует измерять устойчивость пакета? Один из методов предусматривает подсчет количества зависимостей, которые входят и выходят

<sup>1</sup> Webster's Third New International Dictionary.

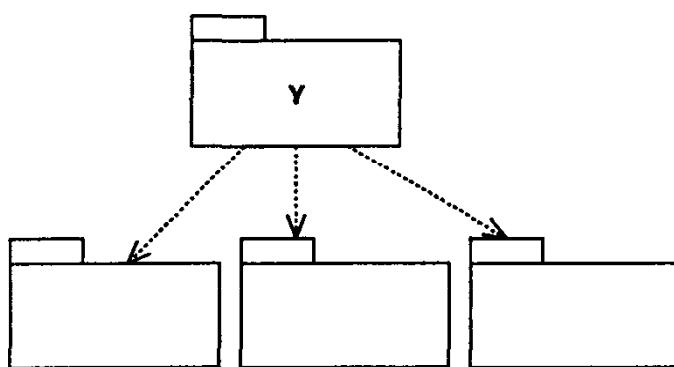
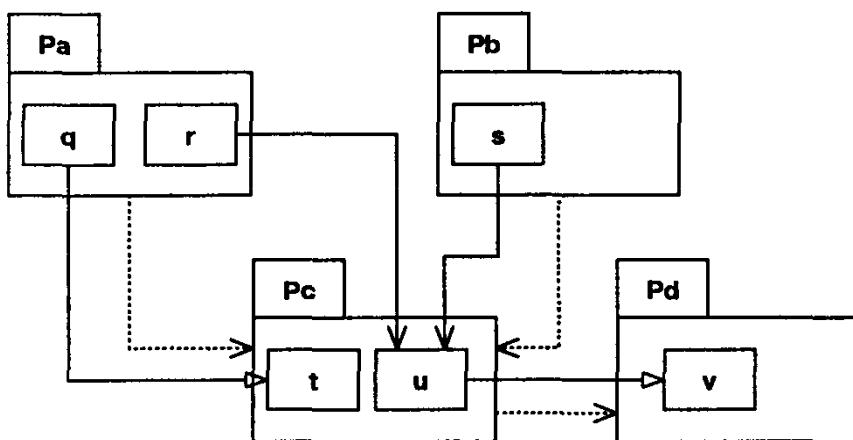


Рис. 20.6. Y: неустойчивый пакет

Рис. 20.7. Табуляция  $C_a$ ,  $C_e$  и  $I$ 

из пакета. Полученные в этом случае данные позволяют говорить об оценках *позиционной устойчивости* пакета.

- ( $C_a$ ) центростремительное связывание: количество классов вне пакета, которые зависят от классов, образующих пакет.
- ( $C_e$ ) центробежное связывание: количество классов внутри пакета, которые зависят от классов, не входящих в пакет.
- (Неустойчивость  $I$ )

$$I = \frac{C_e}{(C_a + C_e)}$$

Значения этой метрики попадают в диапазон  $[0,1]$ . Показатель  $I = 0$  свидетельствует о максимально устойчивом пакете. Показатель  $I = 1$  присущ максимально неустойчивому пакету.

Метрики  $C_a$  и  $C_e$  определяются путем подсчета количества *классов* вне данного пакета, которые имеют зависимости в классах внутри рассматриваемого пакета. Обратите внимание на пример, приведенный на рис. 20.7.

Пунктирные стрелки между пакетами демонстрируют зависимости. Взаимосвязи между классами в этих пакетах демонстрируют, каким образом эти зави-

симости фактически реализуются. Существуют ассоциативные и унаследованные взаимосвязи.

Предположим, что требуется оценить устойчивость пакета  $P_c$ . При этом три класса вне  $P_c$  зависят от классов в  $P_c$ . Поэтому  $C_a = 3$ . Существует один класс вне  $P_c$ , от которого зависят классы в  $P_c$ . Поэтому  $C_e = 1$ , а  $I = 1/4$ .

Средствами C++ описанные зависимости обычно представлены с помощью инструкций `#include`. В случае применения метрики  $I$  вычисления значительно облегчаются, если организовать исходный код таким образом, что в каждом файле исходного кода будет находиться один класс. В Java метрика  $I$  вычисляется путем подсчета количества инструкций импорта и уточненных имен.

Если метрика  $I$  равна 1, следовательно, от данного пакета не зависят какие-либо другие пакеты ( $C_a = 0$ ), но он зависит от других пакетов ( $C_e > 0$ ). Это означает неустойчивость пакета (*безответственность и зависимость*). Недостаток зависимости приводит к тому, что отсутствует повод для изменения, в то время как для пакетов, имеющих зависимые пакеты, причин для изменения будет предостаточно.

Если же метрика  $I$  равна нулю, это означает, что от данного пакета зависят другие пакеты ( $C_a > 0$ ), но он сам не зависит от каких-либо других пакетов ( $C_e = 0$ ). Говорят, что пакет будет *ответственным и независимым*. Подобные пакеты характеризуются устойчивостью. Зависимые от них пакеты изменять затруднительно, к тому же отсутствуют зависимости, которые стимулируют изменения.

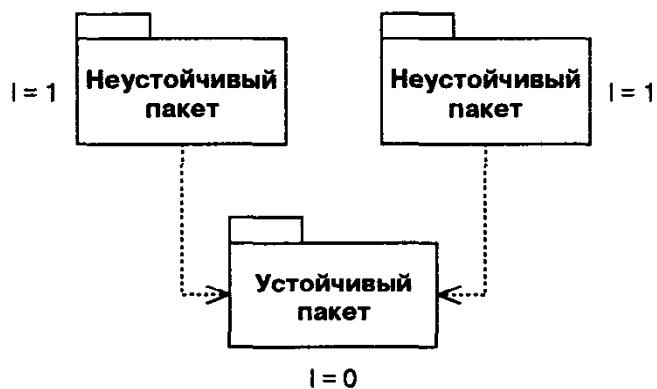
В соответствии с принципом SDP, значение метрики  $I$  базового пакета больше значения метрики  $I$  зависимого пакета (значение метрик должно уменьшаться в направлении изменения зависимости).

## Не все пакеты должны быть устойчивыми

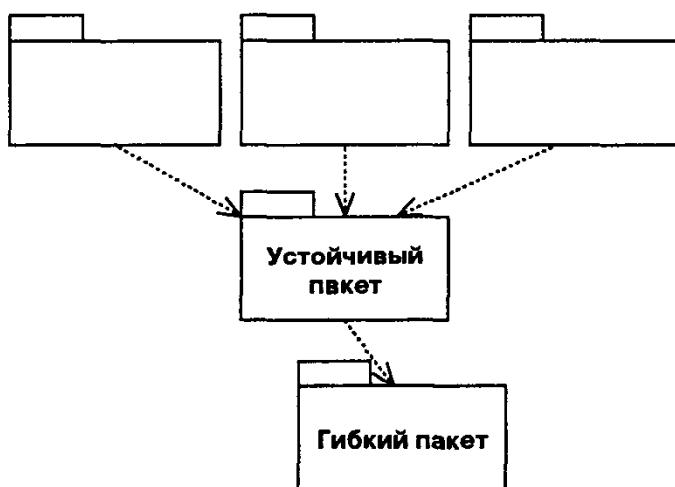
Если всем пакетам присуща максимальная степень устойчивости, система теряет способность к изменению. Подобной ситуации лучше избегать. Структура пакетов должна быть таковой, что некоторые из них являются устойчивыми, а некоторым это свойство не присуще. На рис. 20.8 демонстрируется идеальная конфигурация для системы, включающей три пакета.

Изменяемые пакеты размещены в верхней части рисунка и зависят от устойчивых пакетов, размещенных в нижней части рисунка. Подобная схема продиктована соображениями удобства, поскольку в этом случае указывающая *вверх* стрелка будет свидетельствовать о нарушении принципа SDP.

На рис. 20.9 приводится пример нарушения принципа SDP. Пакет *Flexible* характеризуется определенной легкостью изменения. В силу этого *Flexible* является неустойчивым. Некоторые разработчики, работающие с пакетом *Stable*, “навешивают” зависимость на *Flexible*. При этом нарушается принцип SDP, поскольку значение метрики  $I$  для *Stable* намного меньше, чем значение метри-



**Рис. 20.8.** Идеальная конфигурация пакетов



**Рис. 20.9.** Нарушение принципа SDP

ки  $I$  для **Flexible**. В результате **Flexible** не так то просто изменить. В случае наличия подобных попыток нам приходится иметь дело со **Stable** и всеми зависимыми от него пакетами.

В целях устранения этой проблемы следует так или иначе “разорвать” имеющую место зависимость между **Stable** и **Flexible**. Каково назначение подобной зависимости? Предположим, что существует класс **c** в составе **Flexible**, который использует класс **u** в составе **Stable** (рис. 20.10).

Устранение зависимости реализуется с помощью принципа DIP. Создадим интерфейсный класс **IU** и поместим его в пакет **UIinterface**. Убедитесь в том, что этот интерфейс объявляет все методы, используемые **u**. Затем на базе этого интерфейса будет наследоваться **c** (рис. 20.11). В результате устраняется зависимость **Stable** от **Flexible**, а также создается ситуация, когда оба пакета зависят от **UIinterface**. Пакет **UIinterface** является весьма устойчивым ( $I = 0$ ), а пакет **Flexible** сохраняет присущий ему уровень минимально необходимой неустойчивости ( $I = 1$ ). Теперь все зависимости “текут в русле” уменьшения  $I$ .

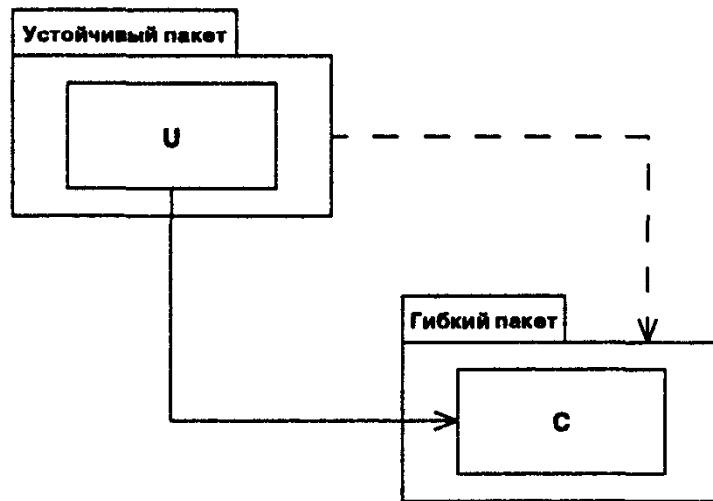


Рис. 20.10. Формирование “плохой” зависимости

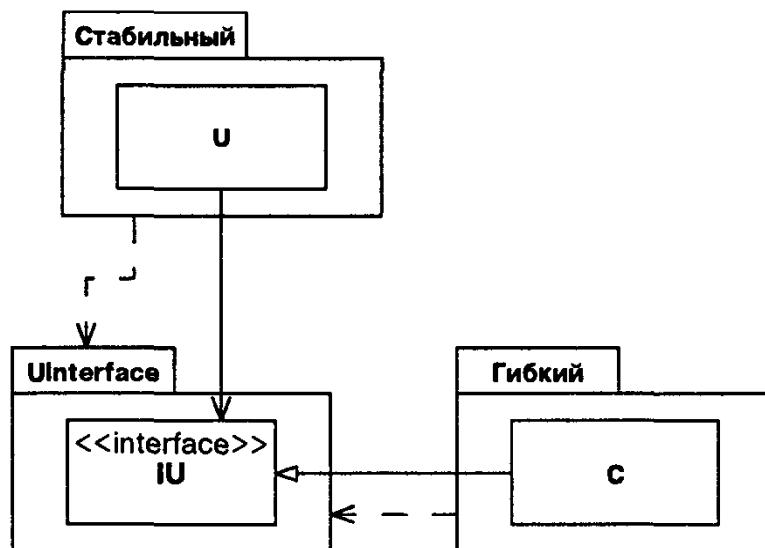


Рис. 20.11. Устранение неустойчивости с помощью принципа DIP

## Область применения высокоровневого проектирования

Некоторые программные системы не должны изменяться слишком часто. Эти программы представляют высокоровневую архитектуру и дизайнерские решения. Вряд ли мы захотим, чтобы архитектурные решения были изменчивыми. Поэтому программы, представляющие высокоровневый дизайн системы, должны помещаться в устойчивые пакеты ( $I = 0$ ). Неустойчивые пакеты ( $I = 1$ ) должны включать только те программы, которые должны легко изменяться.

Если же высокоровневый дизайн представлен устойчивыми пакетами, исходный код весьма трудно изменить. В результате дизайн теряет присущую ему гибкость. Каким же образом максимально устойчивые пакеты ( $I = 0$ ) должны быть достаточно устойчивыми, чтобы противостоять изменениям? Ответ на этот

вопрос кроется в принципе OCP. В соответствии с этим принципом, возможно (и даже желательно) создавать классы, которые являются достаточно гибкими для того, чтобы расширяться, не требуя при этом модификации. Причем этому принципу соответствуют *абстрактные* классы.

## Принцип устойчивости абстракций (SAP, Stable-Abstractions Principle)

*Пакет должен быть абстрактным в той же степени, в какой он является устойчивым.*

Благодаря применению этого принципа устанавливается взаимосвязь между устойчивостью и абстрактностью. В соответствии с ним утверждается, что устойчивые пакеты должны быть также абстрактными, а присущая им устойчивость не предотвращает их возможное расширение. Устойчивые же пакеты должны быть конкретными. Свойство неустойчивости приводит к тому, что конкретный код внутри него может быть легко изменен.

Если пакет является устойчивым, то при наличии в его составе абстрактных классов, он может расширяться. Устойчивые пакеты являются расширяемыми и гибкими и в связи с этим не ограничивают дизайн.

Принципы SAP и SDP комбинируют количественные показатели с принципом DIP, сформулированным для пакетов, так как в принципе SDP говорится о том, что зависимости должны эволюционировать в направлении устойчивости, а принцип SAP утверждает, что устойчивость влечет за собой абстракцию. Вывод: зависимости эволюционируют в направлении абстракции.

Вспомним, что принцип DIP применяется при работе с классами. Этим объектам присущ “черно-белый” характер. Классы могут быть абстрактными или неабстрактными. Комбинация принципов SDP и SAP рассчитана на работу с пакетами, которые могут быть частично абстрактными и частично устойчивыми.

## Оценки абстракций

А-метрика применяется для измерения “степени абстрактности” пакета. Ее значение рассчитывается как отношение количества абстрактных классов в пакете к общему количеству классов в пакете.

$N_c$  — количество классов в пакете.

$N_a$  — количество абстрактных классов в пакете. Помните о том, что абстрактным классом является тот класс, который содержит, как минимум, один чистый интерфейс, причем исключено создание экземпляра этого класса.

$A$  — мера абстрактности.

$$A = \frac{N_a}{N_c}$$

$A$ -метрика изменяется в диапазоне от 0 до 1. Нулевой показатель означает, что пакет не включает абстрактные классы. Значение, равное 1, свидетельствует о том, что пакет состоит исключительно из абстрактных классов.

## Главная последовательность

Теперь пришло время определить взаимосвязь между устойчивостью ( $I$ ) и абстрактностью ( $A$ ). Для этого нарисуем диаграмму, где  $A$  представляет ось ординат (по вертикали), а  $I$  — ось абсцисс (по горизонтали). Если нанести два типа “хороших” пакетов на диаграмму, нетрудно заметить, что максимально абстрактные и устойчивые пакеты находятся в левом верхнем углу (0,1). Пакеты, обладающие максимальной степенью нестабильности и конкретности, находятся в правом нижнем углу (1,0) (рис. 20.12).

Естественно, что не все пакеты попадают в описанные две категории. Могут быть промежуточные ситуации, когда пакеты характеризуются некоторой степенью абстрактности и устойчивости. Например, достаточно часто один абстрактный класс наследуется от другого абстрактного класса. Производный класс представляет собой абстракцию, которая включает зависимость. В этом случае максимизация абстракции не приводит к достижению максимальной устойчивости. Присущая классу зависимость ведет к ослаблению устойчивости.

Поскольку мы не можем добиться того, чтобы все пакеты попадали в точки с координатами (0,1) или (1,0), остается полагать, что существует область, занимаемая точками на диаграмме  $A/I$ , определяющая приемлемые позиции для размещения пакетов. Можно предположить, что эта область определяется зонами, в которых *не* допускается размещение пакетов (так называемые зоны исключения) (рис. 20.13).

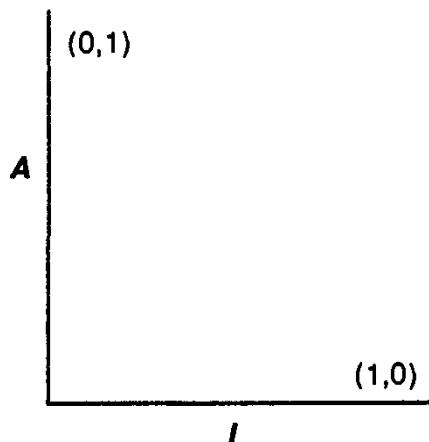


Рис. 20.12. Диаграмма A-I

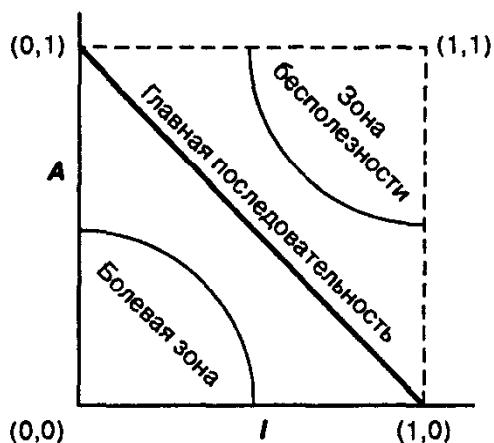


Рис. 20.13. Зоны исключения

Рассмотрим пакеты, которые попадают в область с координатами  $(0,0)$ . Этим пакетам присуща высокая степень устойчивости и конкретности. И далеко не всегда их применение оправдано из-за присущей им закрепощенности. Причем пакеты не могут расширяться, поскольку не являются абстрактными. А их изменение затруднено в силу характерной для них устойчивости. Поэтому в точке с координатами  $(0,0)$  не следует ожидать хорошо спроектированные пакеты. Эта область называется *болевой зоной*.

Конечно, бывают такие ситуации, когда пакеты неизбежно попадают в болевую зону. В качестве примера можно привести схему базы данных. Именно схемам баз данных присуща печально известная изменчивость, беспредельная конкретность, а также высокая степень зависимости. В силу этого реализация интерфейса между объектно-ориентированными приложениями и базами данных является столь непростой, а обновления схем баз данных затруднены.

В качестве другого примера пакета, попадающего в область с координатами  $(0,0)$ , может служить конкретная библиотека утилит. Хотя значение метрики  $I$  для подобного пакета равно 1, его можно рассматривать в качестве образца неизменности. Рассмотрим пример “строкового” пакета. Даже если все классы, входящие в состав этого пакета, являются конкретными (реальными), он не может изменяться. Поскольку подобные пакеты не изменяются, вполне естественным будет их размещение в зоне с координатами  $(0,0)$ . Можно рассмотреть возможность добавления третьей оси на диаграмме, служащей для оценки степени изменчивости. В этом случае к диаграмме, приведенной на рис. 20.13, добавляется плоскость изменчивости, равная 1.

Рассмотрим пакет, находящийся в районе точки с координатами  $(1,1)$ . Эта область нежелательна, поскольку пакету будет присуща максимальная степень абстрактности, причем отсутствуют зависимые от него пакеты. Пользы от подобных пакетов очень мало. Именно поэтому в этом случае выделяется так называемая *зона бесполезности*.

Очевидно, что требуемые нам изменчивые пакеты следует размещать как можно дальше от обеих зон исключения. Область максимального удаления представляет собой прямую, которая соединяет точки с координатами (1,0) и (0,1). Именно эта линия известна под названием *главная последовательность*<sup>2</sup>.

Пакет, находящийся на главной последовательности, не является “настолько абстрактным”, чтобы это повлияло на его устойчивость, но в то же время и не будет “слишком неустойчивым”, чтобы приобрести репутацию слишком абстрактного. Ему не присущи бесполезность или некая “болезненность”. Он зависит от расширений, которые являются абстрактными, а также от тех расширений, которые носят конкретный характер.

Очевидно, что наиболее “желанные” области нахождения пакетов представлены одной из двух конечных точек главной последовательности. Автор на основе своего личного опыта может сказать, что менее половины пакетов обладают столь идеальными характеристиками. Хорошие характеристики присущи пакетам, которые находятся либо на главной последовательности, либо возле нее.

## Оценка расстояния до главной последовательности

Теперь перейдем к рассмотрению последней в этой главе метрики. Поскольку желательно, чтобы пакеты находились в области главной последовательности, следует создать метрику, которая измеряет “расстояние от идеала”.

$D$  – расстояние.

$$D = \frac{|A + I - 1|}{\sqrt{2}}.$$

Диапазон изменения значений этой метрики находится в области  $[0, \sim 0.707]$ .  $D'$  – нормализованное расстояние.

$$D' = |A + I - 1|.$$

Эта метрика является наиболее удобной в применении, поскольку, в отличие от  $D$ , попадает в диапазон  $[0, 1]$ . Нулевое значение говорит о том, что пакет находится непосредственно на главной последовательности. Единица свидетельствует о максимальном расстоянии до главной последовательности.

Располагая этой метрикой, можно проанализировать проект на предмет “попадания” на главную последовательность. Для каждого пакета можно вычислить метрику  $D$ . Любой пакет,  $D$ -значение которого не приближается к нулю, может быть повторно проверен и реструктуризирован. Фактически именно благодаря такому анализу значительно облегчилась задача автора по определению пакетов,

---

<sup>2</sup>Причина появления термина “главная последовательность” объясняется интересом автора к астрономии и к HR-диаграммам.

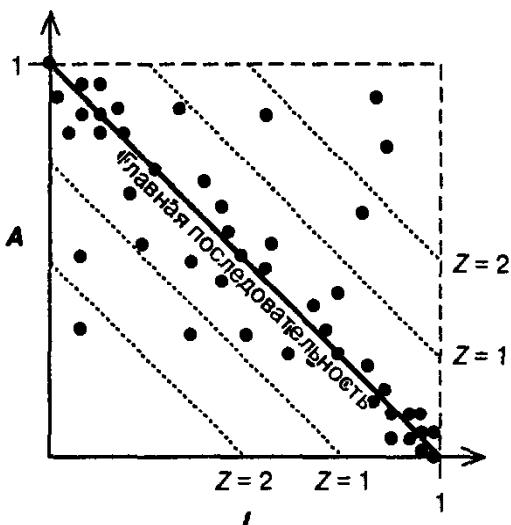


Рис. 20.14. График рассеяния D-метрик пакетов

которые лучше поддерживаются в дальнейшем и менее чувствительны к изменениям.

Также возможен статистический анализ проекта. Для любой  $D$ -метрики, соответствующей пакетам проекта, может вычисляться среднее значение и дисперсия. Вообще говоря, для любого проекта среднее значение и дисперсия должны приближаться к нулю. Значение дисперсии также может применяться для определения “контрольных границ”, с помощью которых идентифицируются пакеты, которые являются “исключительными” по сравнению с другими пакетами (рис. 20.14).

На приведенном графике рассеяния<sup>3</sup> видно, что большинство пакетов находятся на главной последовательности, но для некоторых из них характерно стандартное отклонение, превышающее единицу ( $Z = 1$ ), характеризующее смещение относительно среднего значения. Подобные аномальные пакеты нежелательны. В силу ряда причин они слишком абстрактны для некоторых зависимых пакетов и слишком конкретны для многих зависимых пакетов.

Другой метод использования метрик заключается в нанесении на график значений метрик  $D'$  для каждого пакета с течением времени. На рис. 20.15 приводится образец полученной в этом случае диаграммы. Изучая эту диаграмму, можно заметить некоторые странные зависимости, которые аккумулируются для нескольких последних выпусков пакета Payroll1. На диаграмме отображен контрольный порог, соответствующий значению  $D' = 0.1$ . Точка R2.1 иллюстрирует превышение контрольного порога, поэтому в этом случае следует найти причины, в силу которых этот пакет столь сильно отклонился от главной последовательности.

<sup>3</sup>Здесь не используются реальные данные.

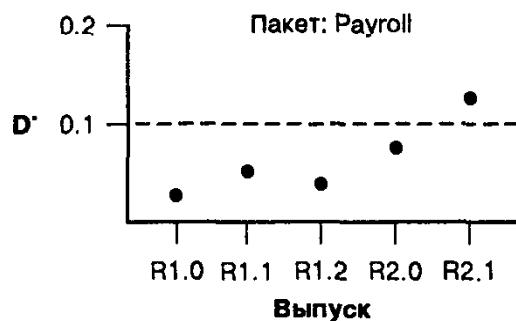


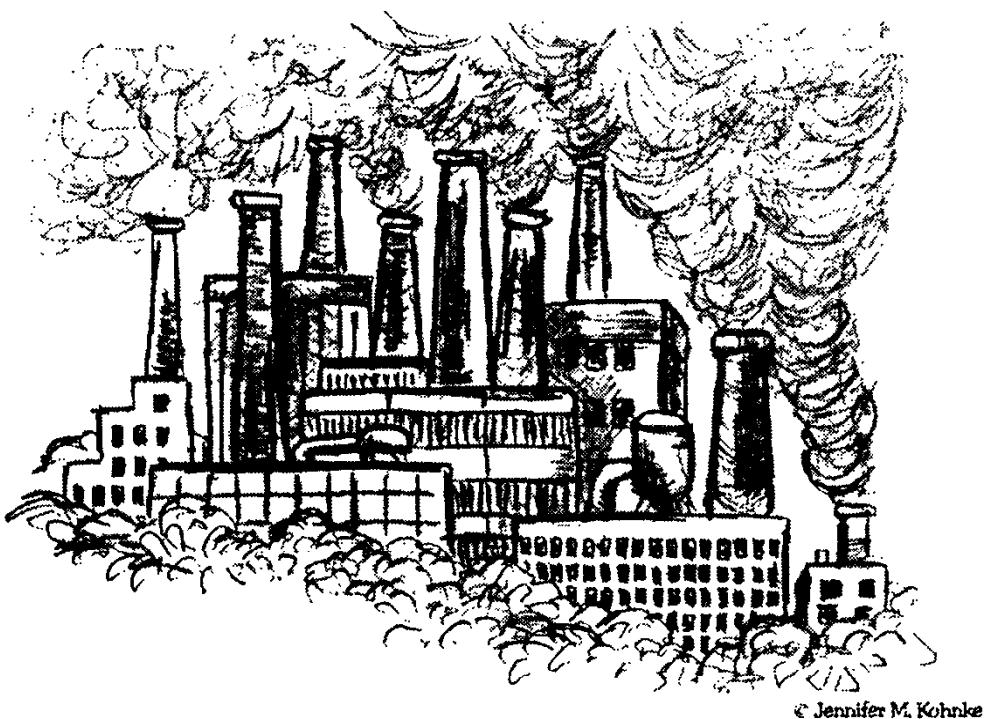
Рис. 20.15. Временная диаграмма  $D'$ -метрики для одного пакета

## Резюме

Описанные в главе *метрики управления зависимостями* позволяют оценить степень соответствия проекта шаблону зависимости и абстракции, который рассматривается в качестве “хорошего” шаблона. На основе личного опыта можно прийти к выводу о том, что некоторые зависимости являются “хорошими”, а некоторые — “не очень”. Поэтому рассматриваемый шаблон отражает личный опыт. Нельзя сказать, что метрика является очень хорошей, поскольку она просто позволяет выполнить оценки в соответствии со случным стандартом. Иногда бывает так, что выбранный в этой главе стандарт подходит только для определенных приложений, но не подходит для других приложений. Также могут существовать лучшие метрики, применяемые для оценки качества того или иного проекта.

# 21

## Шаблон Factory



Человек, строящий фабрику, сооружает храм.

---

Кэлвин Кулидж

Принцип инверсии зависимостей (DIP<sup>1</sup>, Dependency-Inversion Principle) гласит, что следует отдавать предпочтение зависимостям от абстрактных классов, избегая зависимости от конкретных классов, особенно, если такие классы не относятся к семейству неизменных. Следующий фрагмент кода вступает в явное противоречие с этим принципом.

`Circle c = new Circle(origin, 1);`

---

<sup>1</sup>Смотрите описания принципа инверсии зависимостей в гл. 11.

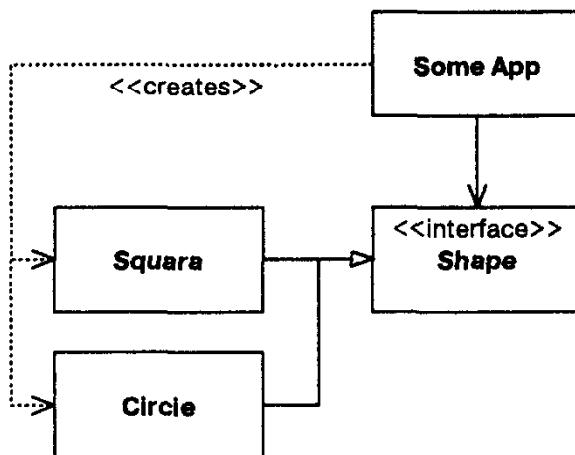
В данном случае `Circle` является конкретным классом. Следовательно, принцип DIP будут нарушать те модули, которые создают экземпляры класса `Circle`. Действительно, любая строка кода, в которой используется ключевое слово `new`, противоречит принципу DIP.

На практике имеет место ряд ситуаций, когда нарушение принципа DIP относительно безопасно<sup>2</sup>. Чем чаще изменяется конкретный класс, тем больше вероятность того, что зависимость от такого класса приведет к появлению определенных проблем.

Например, создание экземпляров класса `String` вряд ли приведет к появлению каких-либо проблем. Это связано с тем, что вероятность изменения этого класса в будущем практически равна нулю.

С другой стороны, в процессе активной разработки приложений мы сталкиваемся с большим количеством классов, которым присущ сверхвысокий уровень изменчивости. В случае появления зависимости от таких классов возникают определенные трудности. В целях предотвращения возможных неприятностей рекомендуется воспользоваться абстрактным интерфейсом.

Шаблон Factory позволяет создавать экземпляры конкретных объектов, причем в этом случае сохраняется зависимость от абстрактных интерфейсов. Следовательно, данный шаблон может в значительной мере пригодиться в процессе активной разработки приложений, при которой конкретные классы обладают высоким уровнем изменчивости.



**Рис. 21.1.** Класс App, нарушающий принцип DIP в процессе создания конкретных классов

На рис. 21.1. представлен некоторый “проблематичный” сценарий. Здесь вы видите класс `SomeApp`, который зависит от интерфейса `Shape`. Этот класс использует экземпляры `Shape` исключительно с помощью интерфейса `Shape`. В данном случае не применяются методы, присущие классам `Square` или `Circle`.

<sup>2</sup>Соответствующие примеры можно найти в данной книге.

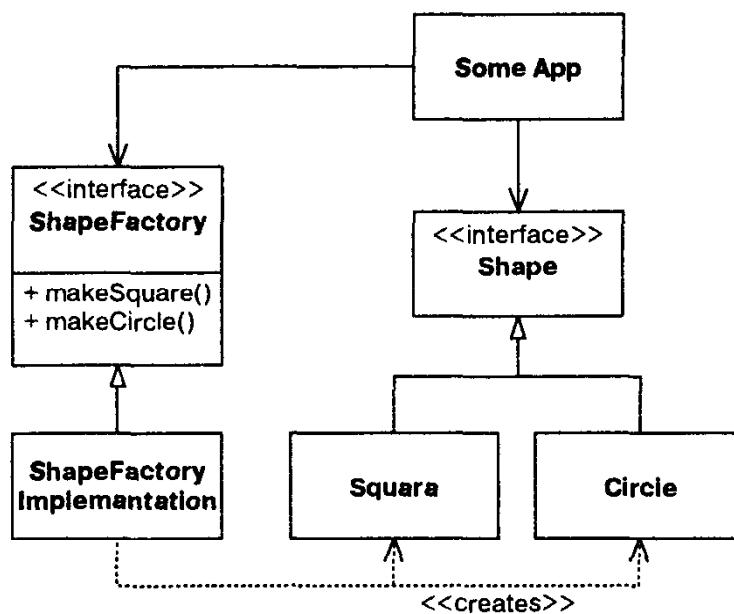


Рис. 21.2. Интерфейс ShapeFactory

Класс `SomeApp` также создает экземпляры классов `Square` и `Circle` и, следовательно, зависит от конкретных классов.

Описанную проблему можно решить, применив по отношению к классу `SomeApp` шаблон `Factory` так, как это изображено на рис. 21.2. Здесь демонстрируется интерфейс `ShapeFactory`, включающий два метода, `makeSquare` и `makeCircle`. Метод `makeSquare` возвращает экземпляр класса `Square`, а метод `makeCircle` возвращает экземпляр класса `Circle`. Обе функции возвращают тип данных `Shape`.

В листинге 21.1 приводится код интерфейса `ShapeFactory`, а в листинге 21.2 — код реализации `ShapeFactory`.

---

#### Листинг 21.1. ShapeFactory.java

---

```

public interface ShapeFactory
{
 public Shape makeCircle();
 public Shape makeSquare();
}

```

---



---

#### Листинг 21.2. ShapeFactoryImplementation.Java

---

```

public class ShapeFactoryImplementation implements ShapeFactory
{
 public Shape makeCircle()
 {
 return new Circle();
 }

 public Shape makeSquare()
 {
 return new Square();
 }
}

```

```
{
 return new Square();
}
}
```

Обратите внимание, что использованный в данном случае прием полностью устраняет проблему зависимости от конкретных классов. Код приложения уже больше не зависит от класса `Circle` или `Square`, обеспечивая в то же время создание экземпляров этих классов. При этом манипулирование экземплярами осуществляется посредством интерфейса `Shape`, и в этом случае не используются специфичные для классов `Square` и `Circle` методы.

Итак, проблема, связанная с зависимостью от конкретного класса, устранена. По-прежнему следует создавать класс `ShapeFactory-Implementation`, но зато не нужно формировать классы `Square` и `Circle`. Вероятней всего, создание `ShapeFactoryImplementation` будет осуществляться с помощью модуля `main` или функции инициализации, связанной с этим модулем.

## Цикл зависимости

Некоторые читатели, наверное, уже заметили, что с использованием описанной формы шаблона `Factory` связаны некоторые проблемы. Для каждого из производных модулей `Shape` в классе `ShapeFactory` предусмотрен отдельный метод. В результате возникает цикл зависимости, который усложняет процесс добавления в `Shape` новых производных модулей. В случае добавления каждого нового производного модуля `Shape` в интерфейс `ShapeFactory` необходимо включать соответствующий метод. Как правило, это означает необходимость рекомпиляции и повторного развертывания пользовательского интерфейса `ShapeFactory`<sup>3</sup>.

Можно избавиться от подобного цикла в ущерб безопасности использования типов. Вместо того чтобы передавать в распоряжение интерфейса `ShapeFactory` по одному методу для каждого производного модуля `Shape`, можно сопоставить ему всего лишь одну функцию `make`, для которой определен тип данных `String`. Пример применения подобной методики демонстрируется в листинге 21.3. В этом случае требуется, чтобы в `ShapeFactoryImplementation` использовалась цепь `if/else`, с помощью которой определяется производный модуль класса `Shape`, для которого и формируется экземпляр. Именно эта методика проиллюстрирована в листингах 21.4 и 21.5.

<sup>3</sup>При работе с языком Java этой рекомендации можно не следовать. Можно изменить интерфейс, не выполняя рекомпиляцию и повторное развертывание клиентов, хотя это чревато разного рода

**Листинг 21.3. Фрагмент кода, создающий цикл**

```
public void testCreateCircle() throws Exception
{
 Shape s = factory.make("Circle");
 assert(s instanceof Circle);
}
```

**Листинг 21.4. ShapeFactory.java**

```
public interface ShapeFactory
{
 public Shape make(String shapeName) throws Exception;
}
```

**Листинг 21.5. ShapeFactoryImplementation.java**

```
public class ShapeFactoryImplementation implements ShapeFactory
{
 public Shape make(String shapeName) throws Exception
 {
 if (shapeName.equals("Circle"))
 return new Circle();
 else if (shapeName.equals("Square"))
 return new Square();
 else
 throw new Exception("ShapeFactory cannot create " + shapeName);
 }
}
```

Подобное решение чревато появлением проблем (ошибка времени выполнения). Данная ситуация аналогична неправильному указанию имени формы (shape) в качестве аргумента вызывающей программы, в результате чего отображается сообщение об ошибке времени компиляции. Эта проблема достаточно неприятна. Однако, если воспользоваться достаточным количеством модульных тестов, а также выполнять разработку, основанную на тестировании, можно будет перехватить ошибки времени выполнения задолго до того, как они превратятся в серьезные проблемы.

## **Замещаемые фабрики**

Одно из наибольших преимуществ фабрик заключается в возможности выполнения подстановки одного экземпляра фабрики вместо другого. Благодаря этому можно изменять семейства объектов в рамках одного приложения.

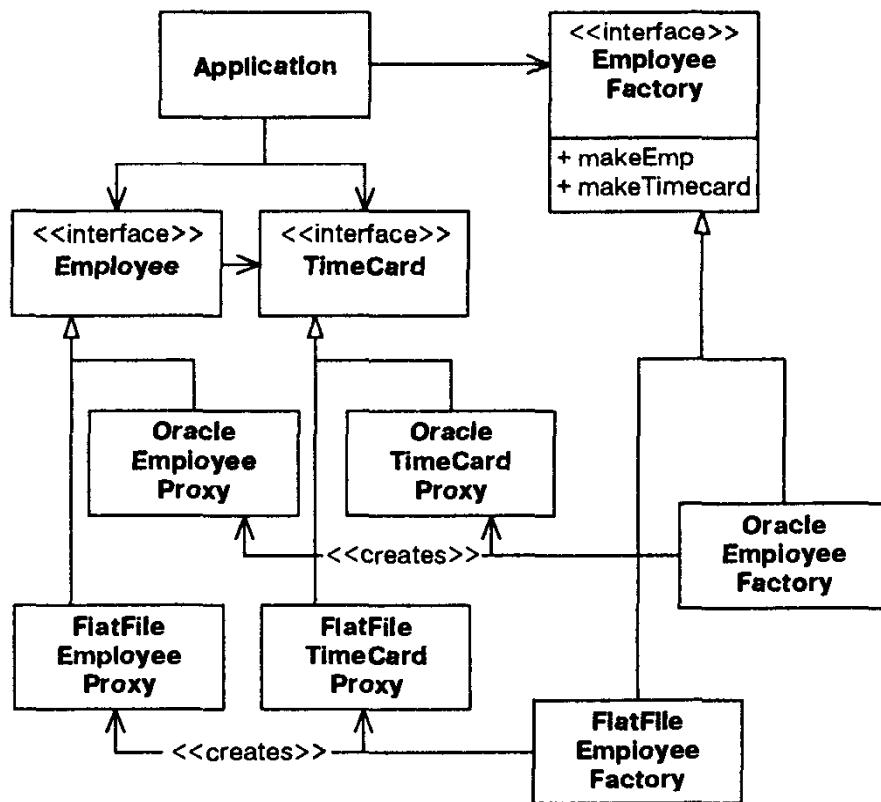


Рис. 21.3. Замещаемая фабрика

Например, представьте себе ситуацию, когда приходится адаптировать приложение с учетом множества реализаций баз данных. В нашем примере предполагается, что пользователи могут работать с простыми файлами таблиц либо воспользоваться конвертором Oracle<sup>TM</sup>. В этом случае можно применить шаблон Proxy<sup>4</sup> в целях изоляции приложения от реализации базы данных. Также можно использовать фабрики для создания экземпляров прокси-объектов. Применяемая в этом случае структура представлена на рис. 21.3.

Обратите внимание, что существует две реализации EmployeeFactory. Одна из них создает прокси-объекты, которые работают с простыми табличными файлами, а вторая приводит к образованию прокси-объектов, предназначенных для работы с конвертором Oracle<sup>TM</sup>. Также следует отметить, что самому приложению “не известно”, какая именно из реализаций используется в данный момент времени. Также обратите внимание, что тип используемой реализации не играет решающей роли.

<sup>4</sup>Этот шаблон будет подробнее рассмотрен в гл. 26. Имейте в виду, что прокси представляет собой класс, который “знает” о правилах считывания конкретных объектов из определенных видов баз данных.

## Применение фабрик для формирования схем тестов

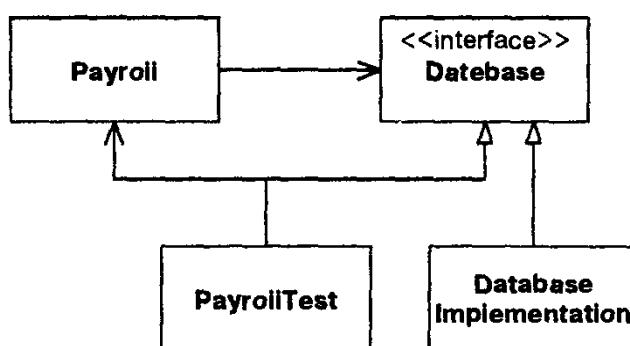
В процессе разработки модульных тестов зачастую возникает потребность протестировать поведение определенного модуля отдельно от используемых им модулей. Например, предположим, что нам приходится работать с приложением Payroll, в котором используется база данных (рис. 21.4.) В данном случае следует протестировать функциональные свойства модуля Payroll отдельно от базы данных.



**Рис. 21.4.** Использование базы данных в модуле Payroll

Подобное тестирование можно выполнить путем использования абстрактного интерфейса для базы данных. Одна из реализаций этого интерфейса использует реальную базу данных. Другая реализация представляет собой тестовый код, задача которого заключается в имитации поведения базы данных и проверки правильности текущих вызовов базы данных. Подобная структура представлена на рис. 21.5. Модуль PayrollTest тестирует модуль PayrollModule путем его вызовов. Он также реализует интерфейс базы данных Database, обеспечивающий перехват вызовов базы данных, выполняемых модулем Payroll. В результате модуль PayrollTest может убедиться в “безупречном поведении” модуля Payroll. Это также позволяет модулю PayrollTest имитировать многие виды сбоев и проблем, возникающих в активной базе данных. Имитации подобного рода достаточно трудно добиться другими методами. Иногда подобный метод тестирования называют *спуфингом* (имитацией соединения).

В связи с вышеописанным возникает вопрос: каким образом модуль Payroll получает экземпляр модуля PayrollTest, используемый в качестве базы дан-



**Рис. 21.5.** Модуль PayrollTest имитирует соединение с базой данных

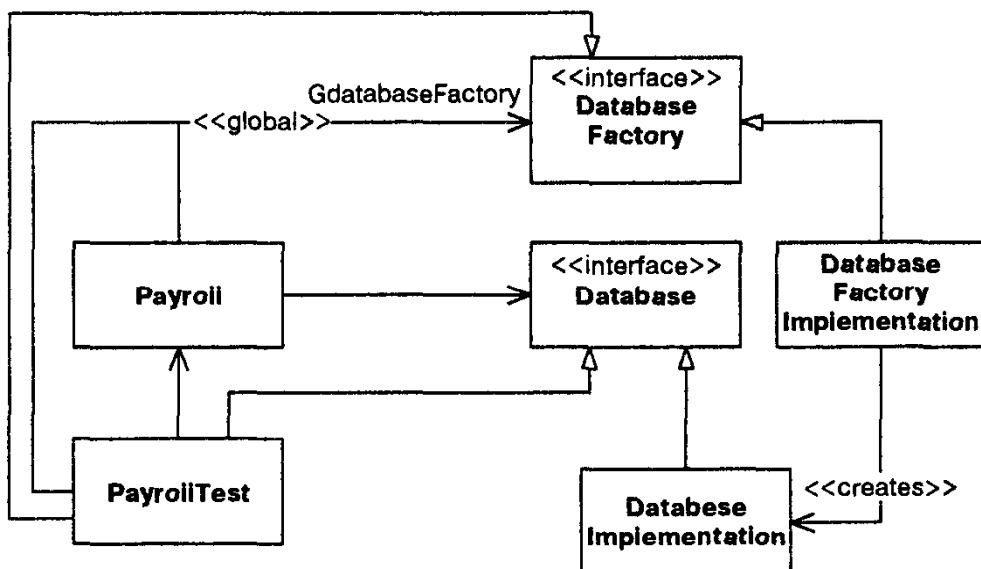


Рис. 21.6. Имитация соединения с Factory

ных `Database`? Модуль `Payroll` не создает сам модуль `PayrollTest`. Также понятно, что модуль `Payroll` каким-то образом получает ссылку на реализацию базы данных `Database`, которую он будет использовать в будущем.

В некоторых случаях `PayrollTest` естественным образом передает ссылку на базу данных `Database` модулю `Payroll`. В других случаях в модуле `PayrollTest` определяется глобальная переменная, которая будет устанавливать ссылку на базу данных `Database`. В ряде других случаев следует ожидать, что экземпляр `Database` будет создан модулем `Payroll`. В последнем случае, для того чтобы “вынудить” `Payroll` создать тестовую версию базы данных `Database`, можно воспользоваться шаблоном `Factory`, передав альтернативную фабрику модулю `Payroll`.

На рис. 21.6 показана возможная структура, реализующая применение данной методики на практике. Модуль `Payroll` получает фабрику посредством глобальной переменной (или статической переменной в глобальном классе) с именем `GdatabaseFactory`. Модуль `PayrollTest` реализует `DatabaseFactory`, а также устанавливает ссылку на самого себя в переменной `GdatabaseFactory`. Если `Payroll` использует фабрику для создания `Database`, модуль `PayrollTest` перехватывает вызов, а затем возвращает ссылку на самого себя. Таким образом, модуль `Payroll` удостоверяется в том, что был создан `PayrollDatabase`, но при этом `PayrollTest` вполне может сымитировать соединение с модулем `Payroll`, перехватывая при этом все вызовы базы данных.

## Преимущества, связанные с использованием фабрик

Строгая интерпретация принципа DIP предполагает, что фабрики должны использоваться для каждого изменяемого класса системы. Более того, шаблон *Factory* обладает достаточно широкими возможностями. Эти два фактора в некоторых случаях могут побуждать разработчиков к использованию фабрик во всех возможных ситуациях. Конечно, это — крайность, поэтому автор книги категорически возражает против подобного решения.

Не следует начинать с использования фабрик в самом начале процесса разработки. Их следует включать в состав системы только тогда, когда возникает настоятельная потребность в этом. Например, если применяется шаблон *Proxy*, скорее всего, потребуется воспользоваться фабрикой в целях создания постоянных объектов. Или же в процессе модульного тестирования может возникнуть ситуация, когда потребуется сымитировать соединение с создателем объекта. Скорее всего, и в этом случае будут задействованы фабрики. Но не стоит начинать разработку системы, ориентируясь изначально на использование исключительно одних фабрик.

Фабрики представляют достаточно сложные объекты, использования которых следует по возможности избегать, особенно на ранних фазах осуществления проекта. Если фабрики применяются повсеместно, в этом случае возможности по расширению проекта сводятся к минимуму. Для формирования нового класса могут потребоваться, как минимум, четыре новых класса. Эти классы включают два класса интерфейсов, которые представляют новый класс и его фабрику, а также два конкретных класса, которые реализуют эти интерфейсы.

## Резюме

Фабрики относятся к мощным инструментальным средствам. Они могут принести огромную пользу в плане соблюдения принципа DIP. Благодаря им стратегические модули высшего уровня могут создавать экземпляры классов, причем в этом случае отсутствует зависимость от конкретных реализаций этих классов. Фабрики также позволяют осуществлять переходы между абсолютно различными семействами реализаций в рамках одной группы классов. Тем не менее, работа с фабриками влечет свои сложности, которых лучше избегать. Использование этих объектов во всех ситуациях не всегда является оправданным.

## Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

# 22

## Практическое занятие: программа расчета зарплаты (часть 2)



© Jennifer M. Kohnke

Если что-либо представляется разумным и совершенным, остерегайтесь — вероятнее всего, это — самообман.

---

Дональд Э. Норман (*The Design of Everyday Things*, Donald A. Norman, Doubleday, 1990)

Мы уже проделали громадный объем работы по анализу, созданию проекта и реализации программы расчета зарплаты. Однако осталось решить еще некоторые вопросы. Обратите внимание, что над данным проектом вместо группы программистов трудился лишь один человек — его автор. Текущая структура среды разработки характеризуется стабильностью. Все программные файлы размещаются в единственном каталоге. Отсутствует структура более высокого порядка. Нет ни пакетов, ни подсистем, выпускаемых модулей, отличных от приложения в целом. Ничто не препятствует продвижению вперед.

Следует учесть, что по мере увеличения объема программы, растет количество программистов, занятых ее совершенствованием. Чтобы облегчить работу над программой, необходимо разбить исходный код на пакеты, удобные для проверки, обновления и тестирования.

В настоящее время приложение по расчету зарплаты содержит 3280 строк кода, разделенных на 50 различных классов и 100 файлов исходного кода. И хотя это количество не слишком велико, можно представить организационные трудности. Как же можно управлять подобным множеством файлов?

При обработке аналогичных строк кода необходимо так организовать работу по реализации программы, чтобы все разработчики трудились согласованно. Желательно разбить классы на группы, которые легко проверять и поддерживать как отдельным программистам, так и группам.

## Структура пакетов и запись

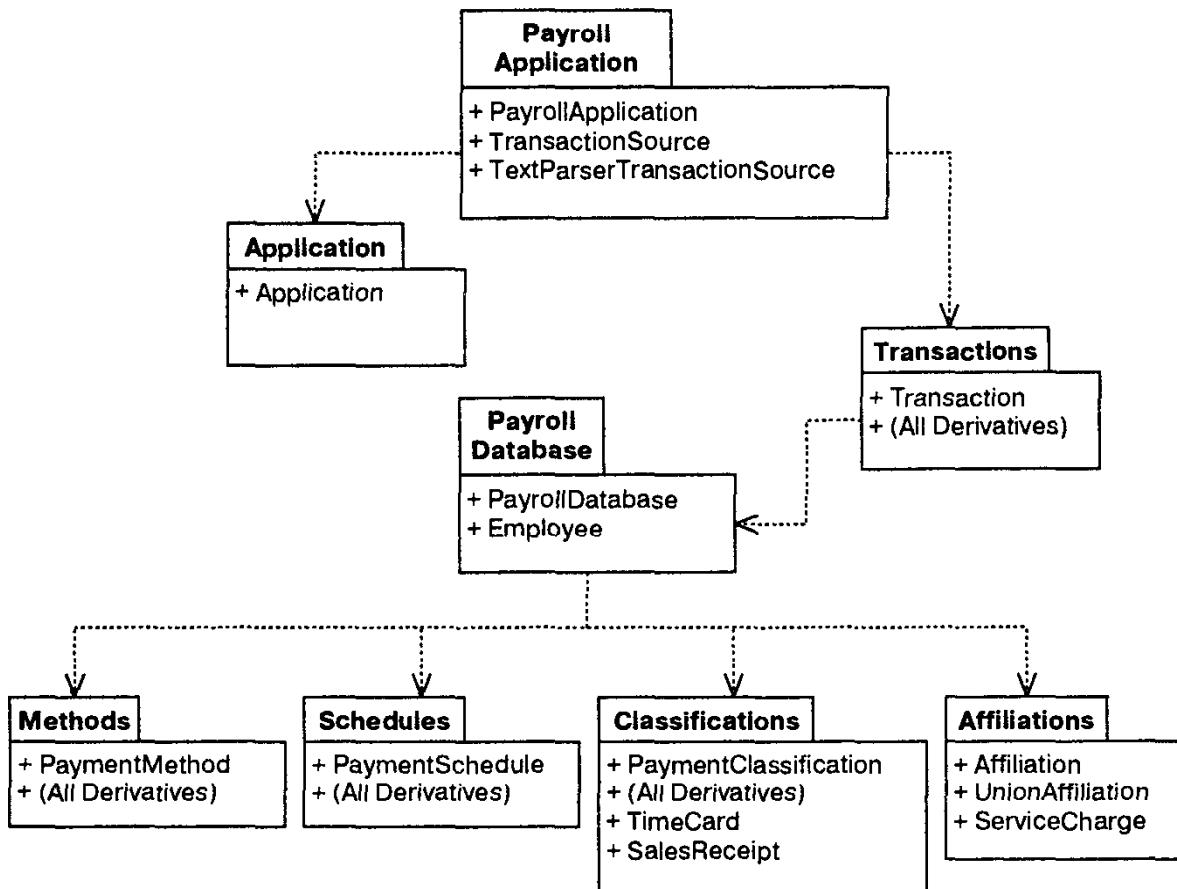
Диаграмма на рис. 22.1 отображает возможную структуру пакетов для приложения по расчету зарплаты. На данном этапе мы ограничимся вопросами документирования и применения этой структуры.

В приложении А представлено описание UML-записи для пакетов. Диаграммы пакетов отражают систему зависимостей “по нисходящей”.

На рис. 22.1 изображено приложение по расчету зарплаты, разделенное на восемь пакетов. Пакет `PayrollApplication` включает классы `PayrollApplication`, `TransactionSource` и `TextParserTransactionSource`. Пакет `Transactions` включает полную иерархию классов `Transaction`. При внимательном изучении диаграммы четко просматривается содержание других пакетов.

Зависимости также определены довольно четко. Пакет `PayrollApplication` зависит от пакета `Transactions`, поскольку класс `PayrollApplication` вызывает метод `Transaction::Execute`. Пакет `Transactions` зависит от пакета `PayrollDatabase`, поскольку каждый из производных модулей `Transaction` непосредственно связан с `PayrollDatabase`.

По какому критерию эти классы сгруппированы в пакеты? Вместе группировались те классы, которые на первый взгляд можно отнести в “общий” пакет. Но, как показано в главе 20, данный подход не является оптимальным.



**Рис. 22.1.** Вариант пакетной диаграммы приложения по расчету зарплаты

Предположим, что в пакет **Classifications** необходимо внести изменения. Вследствие этого предстоит повторно компилировать и протестировать пакет **EmployeeDatabase**. Затем нужно выполнить повторную компиляцию и тестирование пакета **Transactions**. Естественно, **ChangeClassification-Transaction** и три его производных модуля, показанные на рис. 19.3, должны быть повторно скомпилированы и протестированы, но зачем этой процедуре подвергать остальные модули?

С технической точки зрения, остальные транзакции не нуждаются в повторной компиляции и тестировании. Однако они входят в пакет **Transactions**, и если данный пакет повторно выпускается с учетом изменений в пакете **Classifications**, вполне логично повторно выполнить компиляцию и тестирование для всего пакета. Даже если не все транзакции повторно компилируются и тестируются, весь пакет должен быть заново подготовлен к выпуску и развертыванию. Все клиенты, по крайней мере, нуждаются в повторном подтверждении и возможной компиляции.

К классам пакета **Transactions** данные рассуждения не относятся. Каждый из этих классов чувствителен только к собственным изменениям. **ServiceChargeTransaction** открыт для изменений, выполняемых в классе **ServiceCharge**, в то же время **TimeCardTransaction** открыт для изменений,

реализуемых в классе TimeCard. Фактически, как следует из диаграммы на рис. 22.1, определенная часть пакета Transactions зависит почти от каждого компонента программы. Поэтому данный пакет требует существенных переработок перед выпуском. Каждый раз, когда вносятся изменения в зависящие от него структуры, пакет Transactions должен заново проходить подтверждение и подготовку перед выпуском.

Пакет PayrollApplication обладает даже большей чувствительностью к изменениям, поэтому уровень переработок этого пакета перед выпуском существенно возрастает. Может сложиться ошибочное мнение, что чем выше расположен пакет в иерархии зависимостей, тем выше уровень переработок этого пакета перед выпуском (release rate). К счастью, это не так, и именно благодаря принципу OOD.

## Применение принципа общего закрытия (CCP, Common-Closure Principle)

Рассмотрим рис. 22.2. На данной диаграмме классы приложения по расчету зарплаты компактно сгруппированы. Например, пакет PayrollApplication содержит классы PayrollApplication и TransactionSource. Оба класса зависят от абстрактного класса Transaction, содержащегося в пакете PayrollDomain.

Заметим, что класс TextParserTransactionSource находится в другом пакете, зависящем от абстрактного класса PayrollApplication. Формируется нисходящая структура, где подробности реализации зависят от более общих понятий, а обобщенные представления являются независимыми. Такой подход вполне соответствует принципу DIP.

Пакет PayrollDomain наилучшим образом демонстрирует процесс достижения обобщенности и независимости. Этот пакет выражает *сущность* всей системы, и он ни от чего не зависит! Внимательно рассмотрим его. В него входят Employee, PaymentClassification, PaymentMethod, PaymentSchedule, Affiliation и Transaction. Этот пакет содержит все основные абстракции нашей модели, однако он не находится в состоянии зависимости. В чем же дело? Причина в том, что почти все включенные в него классы являются абстрактными.

Рассмотрим пакет Classifications, включающий три производных модуля PaymentClassification. Также в него входят класс ChangeClassificationTransaction и три соответствующих производных модуля, наряду с TimeCard и SalesReceipt. Обратите внимание, что любое изменение, внесенное в эти девять классов, носит изолированный характер; ни один пакет, кроме TextParser, не будет изменен! Подобная особенность также присуща пакетам

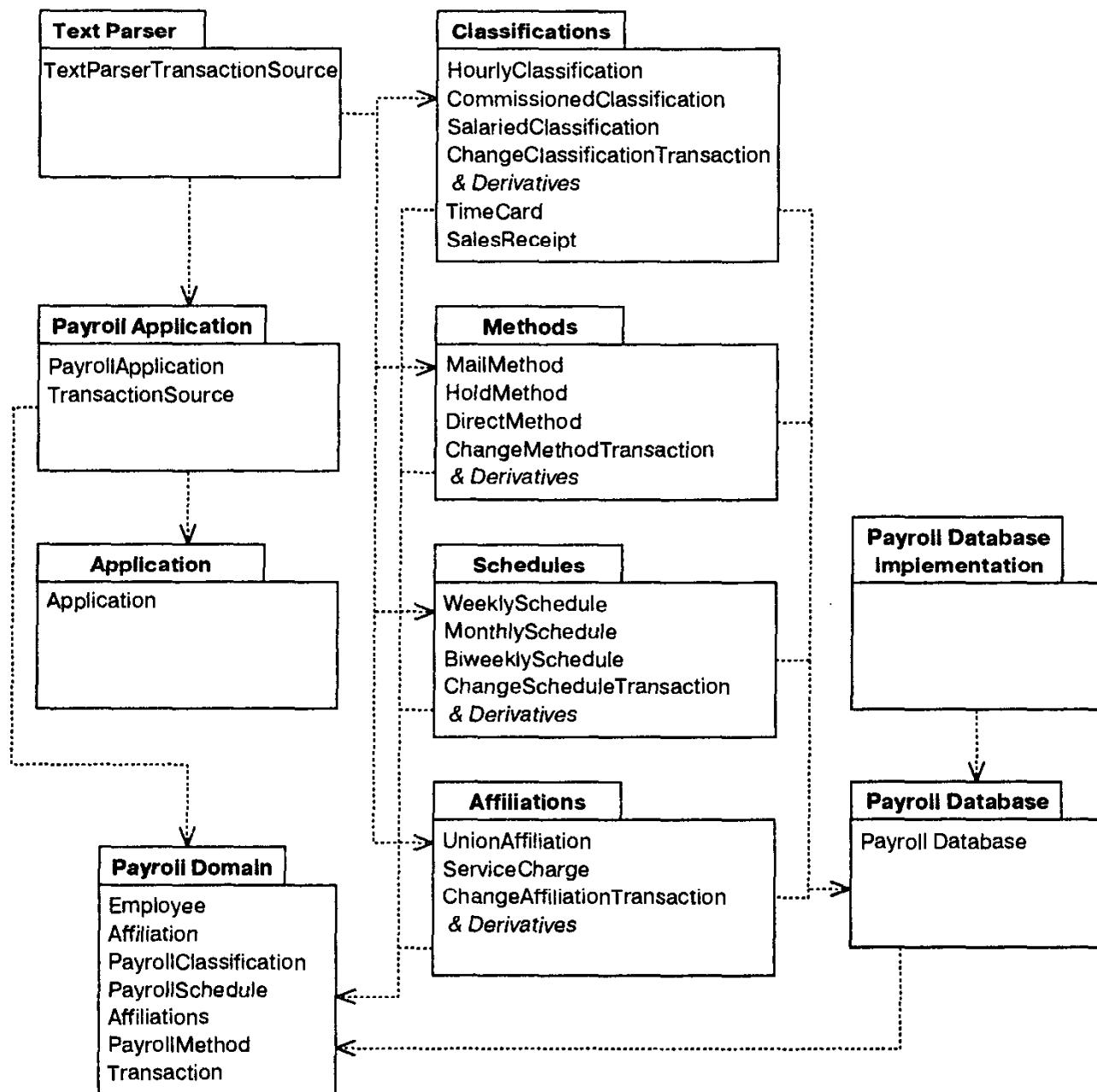


Рис. 22.2. Компактная иерархия пакетов для приложения по расчету зарплаты

**Methods**, **Schedules** и **Affiliations**. Все они отличаются определенной степенью изолированности.

Обратите внимание, что довольно большая часть исполняемого кода содержится в пакетах, имеющих немного зависимостей или же вообще лишенных таковых. Поскольку они почти ни с чем не связаны, можно назвать их *лишними ответственности* (*безответственными*, *irresponsible*). Код в этих пакетах обладает повышенной гибкостью; можно вносить в него изменения, и это не повлияет на функционирование подавляющего большинства других частей проекта. Также отметим, что большинство общих пакетов системы содержит минимальный объем исполняемого кода. От этих пакетов исходят многочисленные зависимости, но сами пакеты ни от чего не зависят. Поскольку от них зависит большое число пакетов, можно назвать их *ответственными* (*responsible*), а поскольку они ни

от чего не зависят, их также называют *независимыми*. Таким образом, очень мал объем кода, отличающегося ответственностью (то есть код, изменения в котором оказывают влияние на большую часть другого кода). Более того, этот небольшой объем ответственного кода также является независимым, изменения в этом коде не повлекут изменений в других модулях. Подобная нисходящая структура, где в нижней части располагаются строго независимые и ответственные обобщения, а в верхней части — лишенные ответственности и зависимые детали, является признаком реализации объектно-ориентированной проекта (*object-oriented design*).

Обратите внимание на отличия между схемами, представленными на рис. 22.1 и 22.2. Детали, расположенные в нижней части рис. 22.1, являются независимыми и строго ответственными. Но данное место не предназначено для подобных деталей! Подробности должны зависеть от основных архитектурных решений системы и не должны находиться во взаимной зависимости. Также отметим, что обобщения, а именно пакеты, определяющие архитектуру системы, лишены ответственности и строго зависимы. Поэтому пакеты, уточняющие архитектурные решения, находятся в состоянии зависимости и возможности их использования ограничиваются пакетами, содержащими детали по реализации. Это является грубым нарушением принципа SAP. Желательно, чтобы архитектура оказывала влияние на детали, а не наоборот!

## **Принцип эквивалентности повторного использования и выпусков (REP, Reuse-Release Equivalency Principle)**

Какие компоненты приложения по расчету зарплаты можно использовать заново? Если разработанную здесь систему ведения платежей применить в другом подразделении компании, работающем согласно совершенно иным политикам, то там не смогут воспользоваться такими объектами, как *Classifications*, *Methods*, *Schedules* или *Affiliations*. Но можно повторно применять *PayrollDomain*, *PayrollApplication*, *Application*, *PayrollDatabase* и, возможно, *PDIImplementation*. С другой стороны, если какой-либо отдел пожелает создать программу, анализирующую текущую базу данных служащих, можно заново воспользоваться классами *PayrollDomain*, *Classifications*, *Methods*, *Schedules*, *Affiliations*, *PayrollDatabase* и *PDIImplementation*. В любом случае в качестве объекта повторного использования фигурируют пакеты.

Иногда, но довольно редко, заново применяется только один класс из пакета. Причина этого довольно проста: классы в пакете связаны друг с другом. Они зависят друг от друга, причем их нельзя легко и “безболезненно” отделить друг

от друга. Не имеет смысла, например, применять класс `Employee` без использования класса `PaymentMethod`. Фактически, чтобы этого добиться, необходимо модифицировать класс `Employee` таким образом, чтобы он не включал класс `PaymentMethod`. Естественно, автор не поощряет повторное использование, в результате которого необходимо обновлять повторно применяемые компоненты. Поэтому объектом для повторного использования служит пакет. Но тогда при группировке классов в пакеты необходимо опираться на другой критерий связности: классы не только следует связывать друг с другом, классы должны вместе повторно применяться в полном согласии с принципом REP.

Снова рассмотрим исходную пакетную диаграмму на рис. 22.1. На первый взгляд, эти пакеты вполне пригодны для повторного использования, например, `Transactions` или `PayrollDatabase`, но, в действительности, при повторном применении возникнут трудности, связанные с большим объемом дополнительного “багажа” в этих пакетах. Пакет `PayrollApplication` отличается высокой степенью зависимости (он зависит абсолютно от всего). При создании нового приложения по расчету зарплаты, где используется иной набор, состоящий из расписания, метода, структуры объединения и квалификационных политик, данный пакет нельзя применить целиком. Вместо этого следует использовать отдельные классы из `PayrollApplication`, `Transactions`, `Methods`, `Schedules`, `Classifications` и `Affiliations`. Но при подобном демонтаже пакетов разрушается их исходная структура. В этом случае нельзя заключить, что версия 3.2 `PayrollApplication` применяется повторно.

Структура, отображенная на рис. 22.1, по сути иллюстрирует нарушение принципа CRP. Получая доступ к повторно используемым фрагментам различных пакетов, пользователь при повторном применении столкнется со сложной проблемой, связанной с контролем функционирования этих фрагментов. Нарушаются зависимости в структуре версии. Новая версия `Methods` приведет к неожиданным результатам, хотя речь идет о повторном использовании класса `PaymentMethod`. Большая часть изменений коснется тех классов, которые не применяются повторно, но все же необходимо отследить эти изменения, повторно скомпилировать и протестировать код.

Трудности, связанные с отслеживанием работы программы, могут оказаться столь существенными, что пользователь при повторном применении программы наверняка выберет следующую стратегию. Он скопирует повторно используемые компоненты и развернет данную копию отдельно от нашего проекта. Однако эту часть нельзя применять повторно. Две части кода имеют много отличий, потребуют дополнительной поддержки и дублирования работы, проведенной ранее.

Эти проблемы устранены в структуре, представленной на рис. 22.2. Пакеты из этой структуры просто применяются повторно. `PayrollDomain` не ассоциируется с большим багажом зависимостей. Каждый из производных модулей, например,

`PaymentMethod`, `PaymentClassification`, `PaymentSchedule`, используется повторно вполне независимо от других.

Внимательный читатель заметит, что диаграмма пакетов, показанная на рис. 22.2, не полностью соответствует принципу CRP. А именно: классы в `PayrollDomain` не образуют наименьшего повторно используемого модуля. Класс `Transaction` при повторном применении не использует оставшуюся часть пакета. Можно создать большое число приложений, получающих доступ к объекту `Employee` и его полям, но не использующих `Transaction`.

Это предположение приводит к изменению диаграммы пакетов, как показано на рис. 22.3. Транзакции отделяются от элементов, которыми они манипулируют. Например, классы в пакете `MethodTransactions` манипулируют классами, находящимися в пакете `Methods`. Класс `Transaction` перемещается

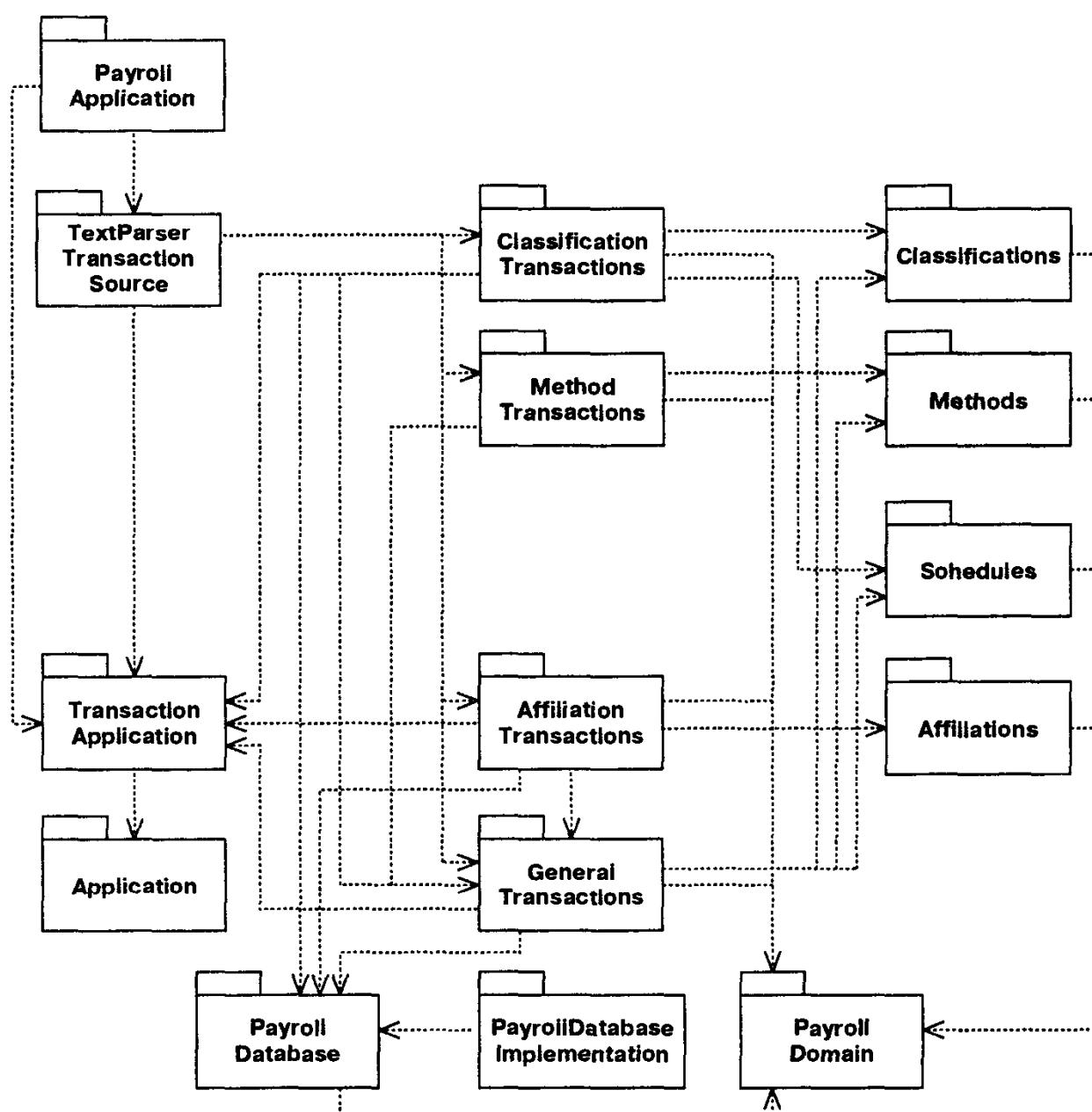


Рис. 22.3. Обновленная диаграмма пакетов приложения по расчету зарплаты

в новый пакет под названием `TransactionApplication`, куда также входит `TransactionSource` и класс под названием `TransactionApplication`. Эти три класса образуют повторно используемый модуль. Класс `PayrollApplication` становится доминирующим. В него входит основная программа, а также производный модуль для `TransactionApplication` под названием `PayrollApplication`. Этот производный модуль связывает `TextParser-TransactionSource` с `TransactionApplication`.

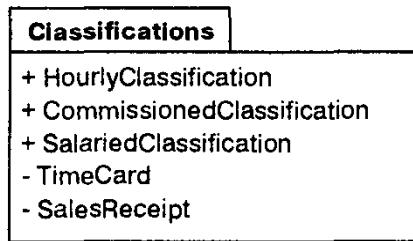
Эти манипуляции выводят абстракции проекта на новый уровень. Пакет `TransactionApplication` теперь можно повторно использовать с помощью любого приложения, получающего доступ к `Transactions` из `Transaction-Source`, а затем `Executes`. Пакет `PayrollApplication` не является больше повторно используемым, поскольку отличается слишком большой зависимостью. Но его место занимает пакет `TransactionApplication`, являясь более общим по своему характеру. Теперь пакет `PayrollDomain` можно применять повторно, не обращаясь к `Transactions`.

Подобный подход существенно улучшает возможности по повторному использованию и поддержке проекта, но в структуре теперь фигурируют пять дополнительных пакетов, причем архитектура зависимостей существенно усложняется. Для оценки компромисса необходимо обратить внимание на тип предполагаемого повторного использования, а также на перспективы, открывающиеся для данного приложения. Если приложение остается стабильным, а повторно его применяют немногие клиенты, эти изменения не оправдывают себя. С другой стороны, если данная структура повторно используется во многих приложениях либо ожидается, что приложение со временем существенно изменится, тогда новая структура вполне жизнеспособна и целесообразна. Лучше начать с несложного варианта и наращивать пакетную структуру по мере необходимости. Пакетные структуры всегда следует формировать с более высокой степенью аккуратности, чем это необходимо для исходного проекта.

## Связывание и инкапсуляция

Аналогично тому, как связывание классов при работе в языке Java и C++ достигается путем инкапсуляции границ, связывание пакетов образуется путем экспорта дополнительных элементов UML-диаграммы.

Если класс одного из пакетов используется в другом пакете, этот класс следует экспортировать. При работе с UML-диаграммами классы экспортируются по умолчанию, но можно дополнить пакет указанием на то, что определенные классы не следует экспортировать. На рис. 22.4, где представлена структура пакета `Classification`, показано, что экспортованы три производных класса `PaymentClassification`, а классы `TimeCard` и `SalesReceipt` не экспорттировались. Это означает, что другие пакеты не могут применять `TimeCard`



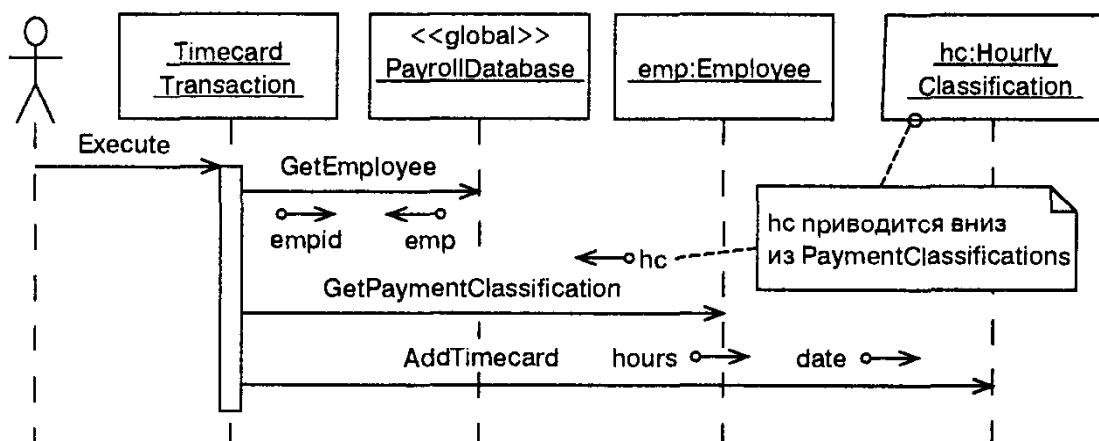
**Рис. 22.4.** Частные классы пакета Classifications

и SalesReceipt; эти классы являются частными для пакета Classifications.

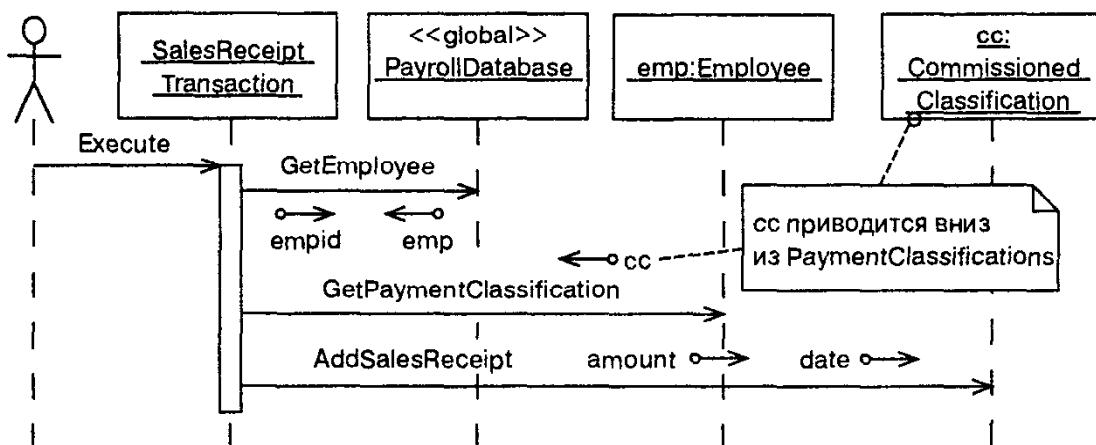
Чтобы воспрепятствовать нежелательному связыванию, можно в пределах пакета скрывать определенные классы. Classifications представляет собой детализированный пакет, содержащий реализации нескольких платежных политик. Чтобы сохранить этот пакет в основной последовательности, следует ограничить нежелательное связывание. Поэтому скрываются те классы, о наличии которых не следует ставить в известность другие пакеты.

Классы TimeCard и SalesReceipt являются удачными примерами частных классов. Они содержат подробности относительно механизма подсчета выплат работникам. Необходимо располагать относительной свободой при изменении этих подробностей, поэтому нежелательно связывать с ними зависимые посторонние объекты.

При просмотре схем, представленных на рис. 19.7–19.10, а также кодом листинга 19.15 можно заключить, что классы TimeCardTransaction и SalesReceiptTransaction уже зависят от TimeCard и SalesReceipt. Однако эту проблему можно легко разрешить, что и показано на рис. 22.5 и 22.6.



**Рис. 22.5.** Продукт пересмотра TimeCardTransaction в целях защиты структуры TimeCard



**Рис. 22.6.** Продукт пересмотра SalesReceiptTransaction в целях защиты SalesReceipt

## Метрики

Как показано в главе 20, с помощью нескольких несложных метрик можно провести количественную оценку атрибутов компактности, связывания, устойчивости, обобщенности и соответствия основной последовательности. Но к чему эти расчеты? Перефразируя Тома Де-Марко (Tom DeMarco), можно заключить: “Нет никакой возможности управлять тем, что нельзя проконтролировать, и нет никакой возможности проконтролировать то, что нельзя измерить”<sup>1</sup>. Чтобы эффективно трудиться в должности инженера или менеджера программного обеспечения, необходимо держать под контролем практические аспекты разработки ПО. Если нельзя их измерить, то вряд ли удастся их контролировать.

Применяя изложенные ниже эвристические соображения и некоторые фундаментальные метрики, связанные с объектно-ориентированными проектами, можно скоррелировать эти метрики для оценки скорости выполнения ПО и эффективности работы команды по ее созданию. Чем больше метрик окажется в нашем распоряжении, тем большим объемом информации мы будем располагать и тем в большей степени мы будем влиять на ход работы.

Приведенные ниже метрики с успехом применялись во многих проектах начиная с 1994 года. Для вычисления этих метрик предусмотрено несколько автоматизированных инструментов, и в то же время нетрудно выполнить эти вычисления вручную. Также не представит особого труда написание простого shell-, python- или ruby-сценария, проводящего вычисления, начиная с файлов исходного кода<sup>2</sup>.

- (*H*) **Относительное сцепление.** Один из аспектов сцепления пакета описывает среднее число внутренних связей для каждого класса. Пусть  $R$  пред-

<sup>1</sup>[DeMarco82], с. 3.

<sup>2</sup>В качестве примера shell-сценария можно загрузить файл depend.sh из свободно доступного раздела Web-узла [www.objectmentor.com](http://www.objectmentor.com) или обратиться к JDepend на Web-узле [www.clarkware.com](http://www.clarkware.com).

ставляет количество связей класса внутри пакета (т.е. связи не выходят за пределы пакета). Пусть  $N$  определяет количество классов в пакете. Дополнительное слагаемое 1 в числителе формулы нужно для исключения случая, когда  $H = 0$  и  $N = 1$ . Именно так можно представить отношение пакета ко всем своим классам.

$$H = \frac{R + 1}{N}.$$

- ( $C_a$ ) **Центростремительное связывание.** Его можно вычислить как количество классов из другого пакета, зависящих от классов данного пакета. В качестве зависимостей рассматриваются взаимоотношения классов, например, наследственность и ассоциативность.
- ( $C_e$ ) **Центробежное связывание.** Его можно вычислить как число классов из других пакетов, от которых зависят классы данного пакета. Как определялось ранее, в качестве зависимостей рассматриваются взаимоотношения классов.
- ( $A$ ) **Абстрактность или обобщенность.** Этот показатель можно вычислить как отношение количества абстрактных классов (или интерфейсов) в пакете к общему числу классов (и интерфейсов) пакета<sup>3</sup>. Диапазон изменения значений этой метрики — от 0 до 1.

$$A = \frac{\text{количество абстрактных классов}}{\text{общее количество классов}}.$$

- ( $I$ ) **Неустойчивость.** Этот показатель можно вычислить как отношение эффективного связывания к общему связыванию. Диапазон изменения значений этой метрики — от 0 до 1.

$$I = \frac{C_e}{C_e + C_a}.$$

- ( $D$ ) **Степень удаления от главной последовательности.** Главная последовательность связана с идеальной линией:  $A + I = 1$ . Формула для  $D$  вычисляет для любого пакета расстояние до главной последовательности.

---

<sup>3</sup>Можно предположить, что для вычисления  $A$  лучше воспользоваться отношением чисто виртуальных функций пакета к общему количеству функций. Но автор полагает, что данный подход не столь успешно отражает абстрактную природу измерения. Даже одна чисто виртуальная функция может формировать абстрактный класс. Значение этой абстракции очень велико по сравнению с тем фактом, что класс может располагать дюжинами конкретных функций, особенно в том случае, если соблюдается принцип DIP.

Диапазон изменения значений этой метрики – от  $\sim .7$  до  $0^4$ ; чем ближе к 0, тем лучше.

$$D = \frac{|A + I - 1|}{\sqrt{2}}.$$

- **( $D'$ ) Нормализованное расстояние до главной последовательности.** Эта метрика представляет метрику  $D$ , нормализованную в интервале [0,1]. Возможно, данная метрика не столь удобна для вычисления и интерпретации. Значение 0 представляет пакет, совпадающий с главной последовательностью. Значение 1 представляет пакет, наиболее удаленный от главной последовательности:

$$D' = |A + I - 1|.$$

## Применение метрик в программе по расчету зарплаты

В табл. 22.1 показано соотношение классов к пакетам в модели по расчету зарплаты. На рис. 22.7 отображена диаграмма пакетов для приложения по расчету зарплаты со всеми вычисленными метриками. А в табл. 22.2 показаны все метрики, вычисленные для каждого пакета.

Каждая зависимость пакетов, показанная на рис. 22.7, дополнена двумя числами. Число, расположенное ближе к зависимому пакету, представляет количество классов пакета, зависимых от основного пакета. Число, расположенное ближе к основному пакету, представляет количество классов пакета, от которых зависят классы зависимого пакета.

Каждый пакет, показанный на рис. 22.7, снабжен соответствующими метриками. Многие из них имеют рекомендательный характер. Например, PayrollApplication, PayrollDomain и PayrollDatabase отличаются высокой степенью относительной связности и располагаются либо на основной последовательности, либо в непосредственной близости от нее. Пакеты Classifications, Methods и Schedules отличаются невысокой степенью относительной связности и располагаются как можно дальше от главной последовательности!

<sup>4</sup>Не представляется возможным расположить все пакеты вне квадрата со стороной, равной 1, на графике от  $A$  до  $I$ . Дело в том, что ни значения  $A$ , ни значения  $I$  не превышают 1. Основная последовательность “проходит” по диагонали от (0,1) до (1,0). Внутри квадрата имеются две наиболее удаленные от основной последовательности точки, а именно углы (0,0) и (1,1). Расстояние этих точек до основной последовательности:

$$\frac{\sqrt{2}}{2} = 0,70710678\dots$$

Таблица 22.1.

| Пакет                          | Классы в пакете                  |                                |                            |
|--------------------------------|----------------------------------|--------------------------------|----------------------------|
| Affiliations                   | ServiceCharge                    | UnionAffiliation               |                            |
| AffiliationTransactions        | ChangeAffiliation-Transaction    | ChangeUnaffiliated-Transaction | ChangeMember-Transaction   |
|                                | ServiceCharge-Transaction        |                                |                            |
| Application                    | Application                      |                                |                            |
| Classifications                | CommissionedClassification       | HourlyClassification           | SalariedClassification     |
|                                | SalesReceipt                     | Timecard                       |                            |
| ClassificationTransaction      | ChangeClassification-Transaction | ChangeCommissionedTransaction  | ChangeHourlyTransaction    |
|                                | ChangeSalariedTransaction        | SalesReceiptTransaction        | TimecardTransaction        |
| GeneralTransactions            | AddCommissionedEmployee          | AddEmployeeTransaction         | AddHourlyEmployee          |
|                                | AddSalariedEmployee              | ChangeAddressTransaction       | ChangeEmployee-Transaction |
|                                | ChangeNameTransaction            | DeleteEmployee-Transaction     | PaydayTransaction          |
| Methods                        | DirectMethod                     | HoldMethod                     | MailMethod                 |
| MethodTransactions             | ChangeDirectTransactions         | ChangeHoldTransactions         | ChamgeMailTransactions     |
|                                | ChangeMethod-Transaction         |                                |                            |
| PayrollApplication             | PayrollApplication               |                                |                            |
| PayrollDatabase                | PayrollDatabase                  |                                |                            |
| PayrollDatabase-Implemetations | PayrollDatabase-Implemetations   |                                |                            |
| PayrollDomain                  | Affiliation                      | Employee                       | Payment Classification     |
|                                | PaymentMethod                    | PaymentSchedule                |                            |
| Schedules                      | BiweeklySchedule                 | MonthlySchedule                | WeeklySchedule             |
| TextParserTransactionSource    | TextParserTransactionSource      |                                |                            |
| TransactionApplication         | TransactionApplication           | Transaction                    | TransactionSource          |

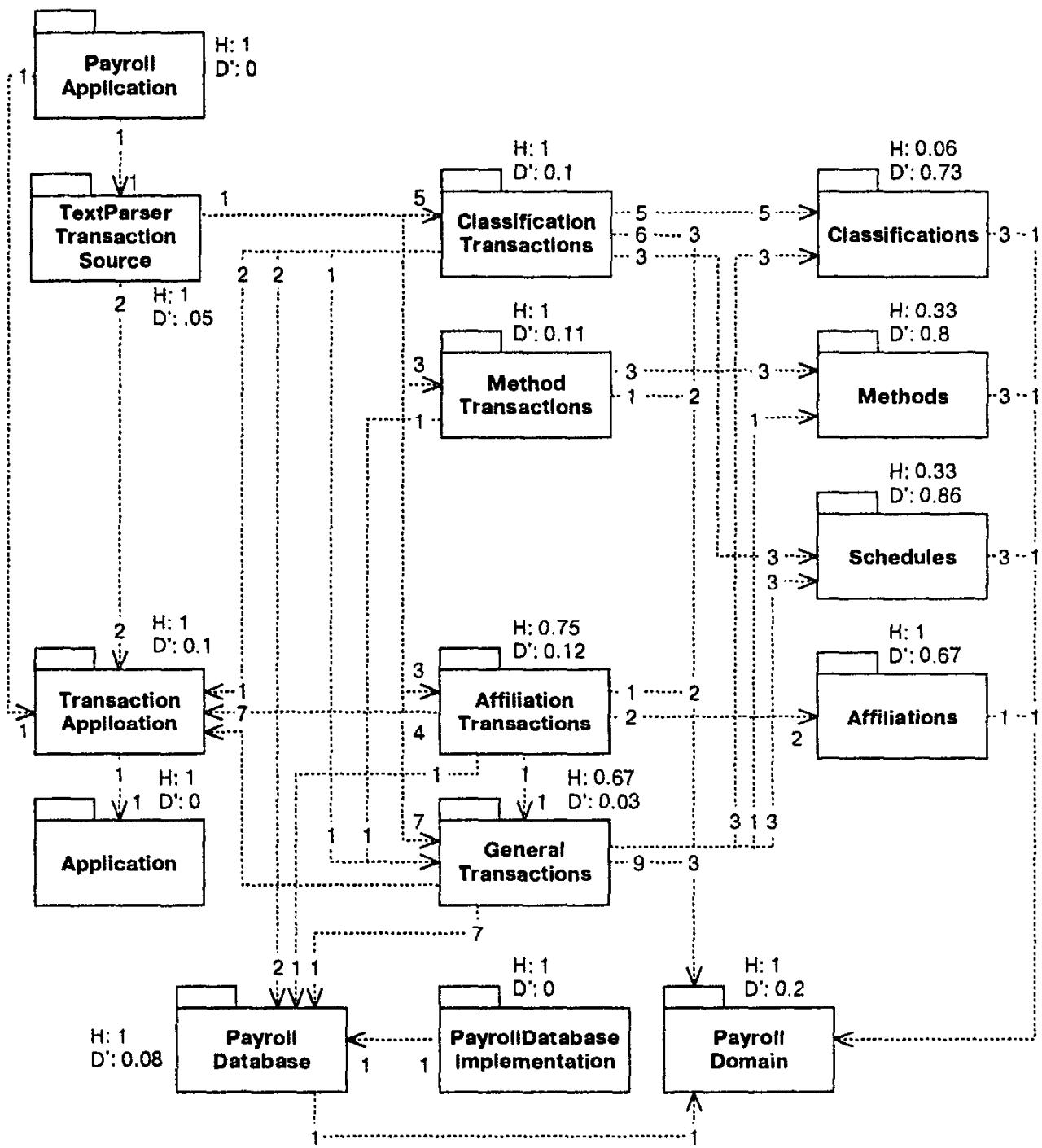


Рис. 22.7. Диаграмма пакетов с метриками

Эти числа наводят на мысль о том, что разбиение классов на пакеты не лишено недостатков. Если не будет указана возможность по совершенствованию приведенных значений, среда разработки окажется чувствительной к изменениям, что в свою очередь приведет к повторным выпускам и тестированиям. Точнее говоря, пакеты с невысокой степенью абстракции (типа *Classification-Transactions*), существенно зависят от других пакетов с невысокой степенью абстракции (типа *Classifications*). Классы с невысокой степенью абстракции содержат большую часть детализированного кода и чаще всего подвергаются изменениям, что приводит к повторным выпускам зависящих от них пакетов.

Поэтому пакет *ClassificationTransactions* отличается высоким уровнем переработки перед выпуском (release rate), поскольку отличается как собственной чувствительностью к изменениям (own high change rate), так и изменчивостью в связи с *Classifications*. Насколько это возможно, желательно ограничить восприимчивость к изменениям нашей среды разработки.

Ясно, что при наличии двух-трех разработчиков можно контролировать среду разработки, умозрительно представляя все связи. В этом случае не столь трудно поддерживать пакеты на основной последовательности. Если над проектом трудится большее количество разработчиков, довольно затруднительно рационально сформировать среду разработки. Более того, трудозатраты на получение этих метрик минимальны по сравнению с тем объемом работы, который требуется для выполнения хотя бы однократного повторного тестирования и выпуска<sup>5</sup>. Поэтому вычисление данных метрик вполне целесообразно.

**Таблица 22.2.**

| Название пакета               | N | A | C <sub>a</sub> | C <sub>e</sub> | R | H    | I    | A    | D    | D'   |
|-------------------------------|---|---|----------------|----------------|---|------|------|------|------|------|
| Affiliations                  | 2 | 0 | 2              | 1              | 1 | 1    | 0,33 | 0    | 0,47 | 0,67 |
| AffiliationTransactions       | 4 | 1 | 1              | 7              | 2 | 0,75 | 0,88 | 0,25 | 0,09 | 0,12 |
| Application                   | 1 | 1 | 1              | 0              | 0 | 1    | 0    | 1    | 0    | 0    |
| Classifications               | 5 | 0 | 8              | 3              | 2 | 0,06 | 0,27 | 0    | 0,51 | 0,73 |
| ClassificationTransaction     | 6 | 1 | 1              | 14             | 5 | 1    | 0,93 | 0,17 | 0,07 | 0,10 |
| GeneralTransactions           | 9 | 2 | 4              | 12             | 5 | 0,67 | 0,75 | 0,22 | 0,02 | 0,03 |
| Methods                       | 3 | 0 | 4              | 1              | 0 | 0,33 | 0,20 | 0    | 0,57 | 0,80 |
| MethodTransactions            | 4 | 1 | 1              | 6              | 3 | 1    | 0,86 | 0,25 | 0,08 | 0,11 |
| PayrollApplication            | 1 | 0 | 0              | 2              | 0 | 1    | 1    | 0    | 0    | 0    |
| PayrollDatabase               | 1 | 1 | 11             | 1              | 0 | 1    | 0,08 | 1    | 0,06 | 0,08 |
| PayrollDatabaseImplementation | 1 | 0 | 0              | 1              | 0 | 1    | 1    | 0    | 0    | 0    |
| PayrollDomain                 | 5 | 4 | 26             | 0              | 4 | 1    | 0    | 0,80 | 0,14 | 0,20 |
| Schedules                     | 3 | 0 | 6              | 1              | 0 | 0,33 | 0,14 | 0    | 0,61 | 0,86 |
| TextParserTransactionSource   | 1 | 0 | 1              | 20             | 0 | 1    | 0,95 | 0    | 0,03 | 0,05 |
| TransactionApplication        | 3 | 3 | 9              | 1              | 2 | 1    | 0,1  | 1    | 0,07 | 0,10 |

## Фабрики объектов

Пакеты *Classifications* и *ClassificationTransactions* являются жестко зависимыми, поскольку для находящихся внутри них классов должны создаваться экземпляры. Например, класс *TextParserTransactionSource* должен создавать объекты *AddHourlyEmployeeTransaction*; и поэтому необходимо центростремительное связывание по направлению из пакета *Text-*

<sup>5</sup> Для компиляции статистик и вычисления метрик для примера, связанного с программой расчета зарплаты, автору понадобилось около двух часов. При использовании одного из широко доступных инструментов вся эта работа выполняется практически моментально.

`ParserTransactionSource` к пакету `ClassificationTransactions`. Кроме того, класс `ChangeHourlyTransaction` должен создавать объекты `HourlyClassification`, поэтому имеет место центростремительное связывание по направлению из пакета `ClassificationTransactions` к пакету `Classification`.

Практически любое применение объектов в пределах этих пакетов осуществляется с помощью абстрактного интерфейса. Если нет необходимости создавать каждый конкретный объект, центростремительное связывание для этих пакетов не применяется. Например, если `TextParserTransactionSource` не создает другие транзакции, отсутствует и зависимость от четырех пакетов, содержащих реализации транзакций.

Эта проблема преодолевается с помощью шаблона `Factory`. Каждый пакет поддерживает фабрику объектов, ответственную за создание всех общедоступных объектов в пределах данного пакета.

## Фабрики объектов для `TransactionImplementation`

На рис. 22.8 показан процесс формирования фабрики объектов для пакета `TransactionImplementation`. Пакет `TransactionFactory` содержит абстрактный базовый класс, определяющий чисто виртуальные функции, которые представляют конструкторы для конкретных объектов транзакции. Пакет `TransactionImplementation` содержит конкретный производный модуль из класса `TransactionFactory` и применяет в процессе создания все конкретные транзакции.

Класс `TransactionFactory` включает статический член, объявленный как указатель `TransactionFactory`. Этот член должен инициализироваться основной программой для указания экземпляра конкретного объекта `TransactionFactoryImplementation`.

## Инициализация фабрик

Чтобы с помощью объектных фабрик создавать объекты, необходимо инициализировать статические члены абстрактных объектных фабрик для указания соответствующей конкретной фабрики. Это следует выполнить перед тем, как пользователь попытается воспользоваться данной фабрикой. Лучше всего выполнить это в основной программе, поэтому главная программа зависит от всех фабрик и всех конкретных пакетов. Таким образом, каждый конкретный пакет является центростремительно связанным с главной программой. Поэтому конкретный пакет располагается на небольшом удалении от главной последовательности, что не всегда можно реализовать<sup>6</sup>. Следовательно, при каждом внесении изменений в любой

<sup>6</sup>На практике автор обычно игнорирует связывания, относящиеся к основной программе.

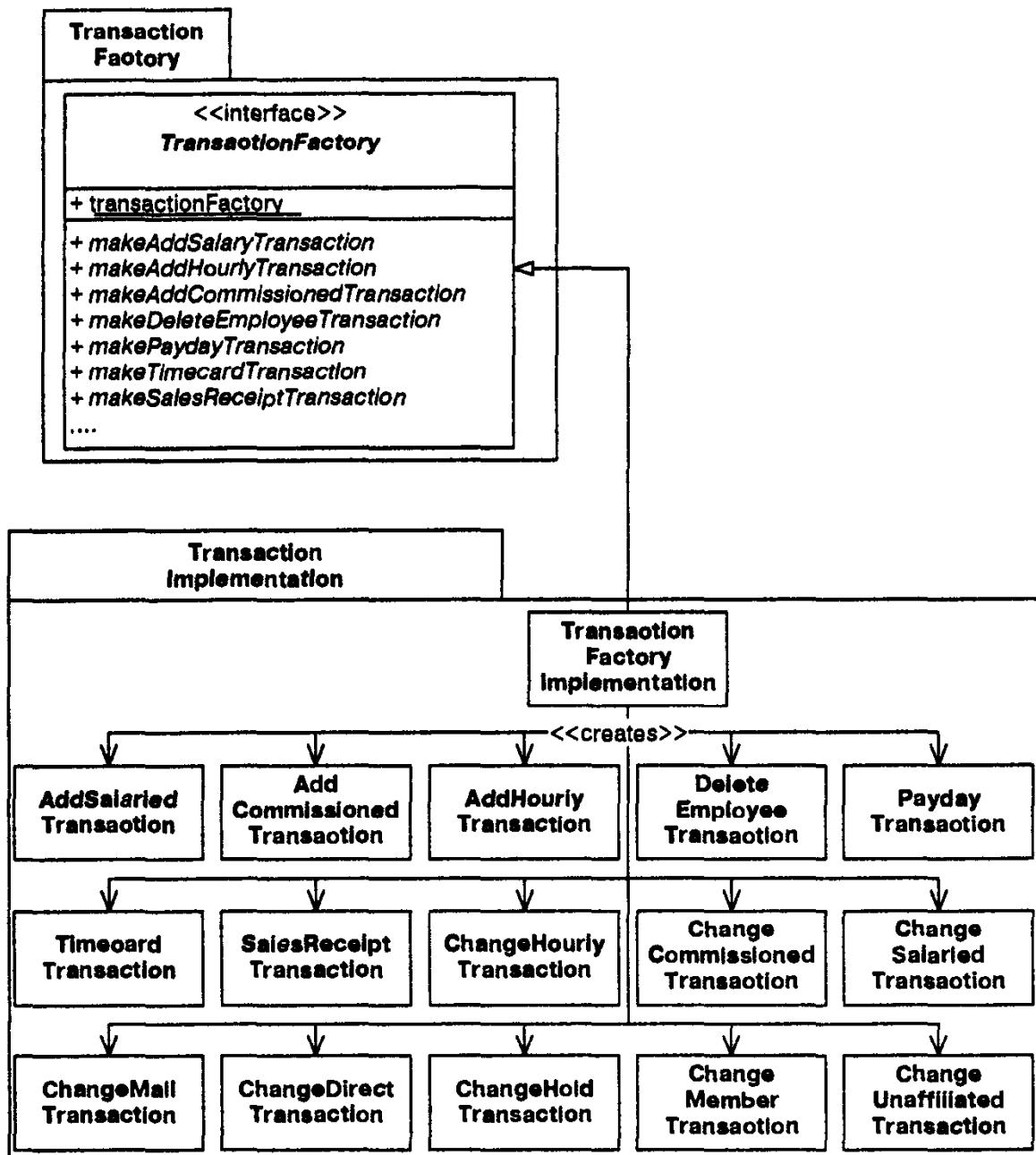


Рис. 22.8. Фабрика объектов для транзакций

из конкретных пакетов, необходимо выполнять повторный выпуск главной программы. Конечно, при внесении каких-либо изменений все равно выполняется повторный выпуск основной программы и проводится ее тестирование.

На рисунках 22.9 и 22.10 показана статическая и динамическая структуры главной программы и их объектные фабрики.

## Переосмысление ограничений на связывание

На рис. 22.1 первоначально разделяются между собой **Classifications**, **Methods**, **Schedules** и **Affiliations**. В то же время может возникать потребность в рациональном разбиении. В конце концов, другие пользователи могут повторно применять эти классы расписаний без использования классов, отно-

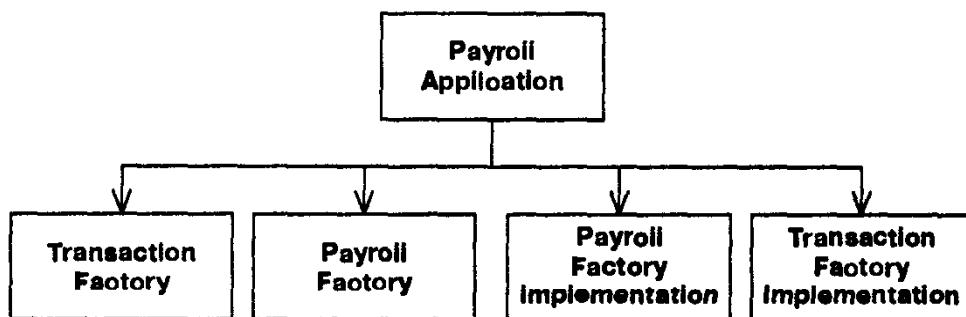


Рис. 22.9. Статическая структура главной программы и объектных фабрик

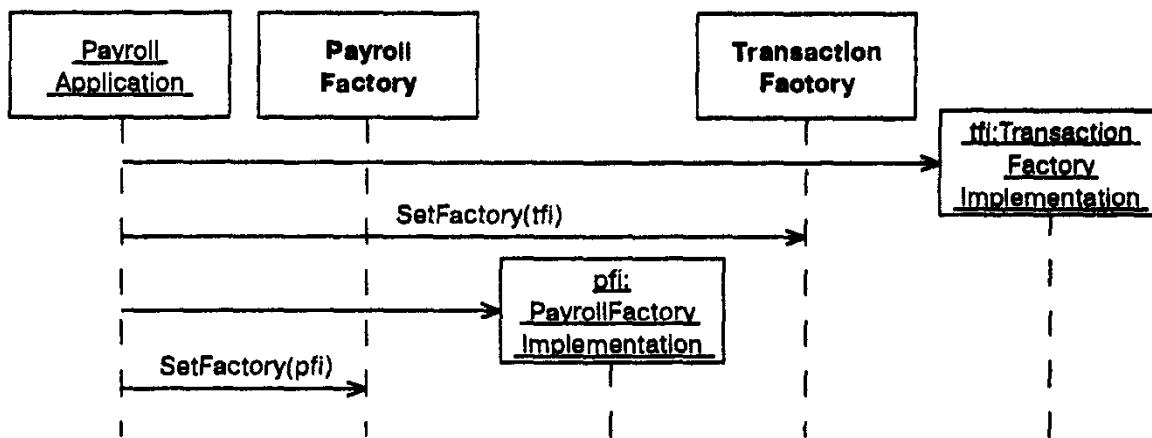


Рис. 22.10. Динамическая структура главной программы и объектных фабрик

сящихся к объединениям. Это разбиение поддерживалось после распределения транзакций по пакетам, что приводило к дуальной иерархии. Возможно, этот подход способствовал усложнению всей структуры. Диаграмма на рис. 22.7 отражает эту перегруженность деталями.

Перегруженная деталями диаграмма пакетов затрудняет производимый вручную контроль за выпусками программ. Несмотря на то, что пакетная диаграмма успешно функционирует при использовании автоматизированного инструмента по планированию проектов, большинство разработчиков лишены таких возможностями. Поэтому пакетная диаграмма должна иметь несложный вид, удобный для практического применения.

По мнению автора, разбиение с помощью транзакций имеет большее значение, чем разбиение с помощью функций. Транзакции можно объединить в единый пакет **TransactionImplementation**, что и показано на рис. 22.11. Также можно объединить пакеты **Classifications**, **Schedules**, **Methods** и **Affiliations** в один пакет **PayrollImplementation**.

## Заключительная структура пакетов

В табл. 22.3 показан заключительный вариант расположения классов в пакетах. В табл. 22.4 представлена электронная таблица метрик. На рис. 22.11 по-

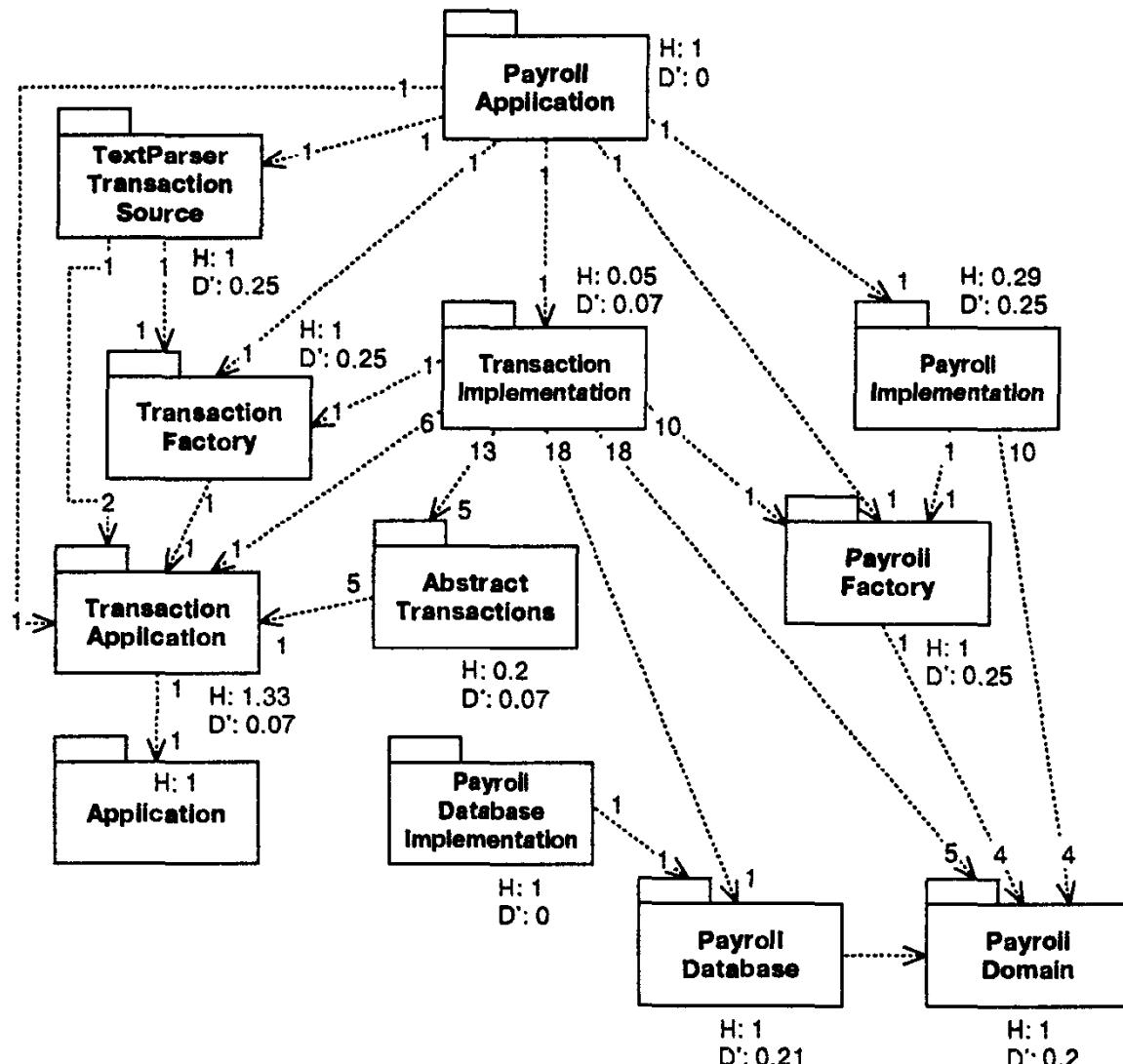
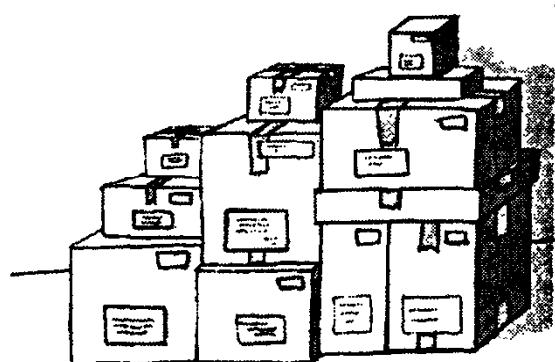


Рис. 22.11. Заключительная структура пакетов приложения по расчету зарплаты

казана заключительная пакетная структура, использующая объектные фабрики для расположения конкретных пакетов в непосредственной близости от основной последовательности.

Метрики, представленные на этой диаграмме, носят рекомендательный характер. Степень относительного связывания во всех случаях довольно высока (благодаря отношениям к конкретным факторам создаваемых объектов), отсутствуют серьезные отклонения от основной последовательности. Поэтому соединения между пакетами вполне соответствуют правилам формирования среды разработки. Рассматриваемые абстрактные пакеты замкнуты, могут применяться повторно и находятся в жесткой зависимости, имея небольшое количество собственных зависимостей. Конкрет-



ные пакеты подразделяются в целях повторного применения, они очень сильно зависят от абстрактных пакетов и в гораздо меньшей степени зависят друг от друга.

Таблица 22.3.

| Пакеты                             | Классы в пакетах                     |                                    |                                |
|------------------------------------|--------------------------------------|------------------------------------|--------------------------------|
| AbstractTransactions               | AddEmployeeTrans-<br>action          | ChangeAffiliation-<br>Transaction  | ChangeEmployee-<br>Transaction |
|                                    | ChangeClassification-<br>Transaction | ChangeMethodTrans-<br>action       |                                |
| Application                        | Application                          |                                    |                                |
| PayrollApplication                 | PayrollApplication                   |                                    |                                |
| PayrollDatabase                    | PayrollDatabase                      |                                    |                                |
| PayrollDatabase-<br>Implementation | PayrollDatabase-<br>Implementation   |                                    |                                |
| PayrollDomain                      | Affiliation                          | Employee                           | PaymentClassifica-<br>tion     |
|                                    | PaymentMethod                        | PaymentSchedule                    |                                |
| PayrollFactory                     | PayrollFactory                       |                                    |                                |
| PayrollImplemen-<br>tation         | BiweeklySchedule                     | CommissionedClas-<br>sification    | DirectMethod                   |
|                                    | HoldMethod                           | HourlyClassification               | MailMethod                     |
|                                    | MonthlySchedule                      | PayrollFactoryImple-<br>mentation  | SalariedClassification         |
|                                    | SalesReceipt                         | ServiceCharge                      | Timecard                       |
|                                    | UnionAffiliation                     | WeeklySchedule                     |                                |
|                                    | TextParserTransac-<br>tionSource     |                                    |                                |
| TransactionAppli-<br>cation        | Transaction                          | TransactionAppli-<br>cation        | TransactionSource              |
| TransactionFactory                 | TransactionFactory                   |                                    |                                |
| TransactionImple-<br>mentation     | AddCommissioned-<br>Employee         | AddHourlyEmplo-<br>yee             | AddSalariedEmplo-<br>yee       |
|                                    | ChangeAddress-<br>Transaction        | ChangeCommissio-<br>nedTransaction | ChangeDirectTrans-<br>action   |
|                                    | ChangeHoldTrans-<br>action           | ChangeHourlyTrans-<br>action       | ChangeMailTrans-<br>action     |
|                                    | ChangeMember-<br>Transaction         | ChangeNameTrans-<br>action         | ChangeSalaried-<br>Transaction |
|                                    | ChangeUnaffiliated-<br>Transaction   | DeleteEmployee                     | PaydayTranaction               |

Окончание табл. 22.3

| Пакеты | Классы в пакетах                   |                                         |  |  |  |  |  |  |  |  |  |
|--------|------------------------------------|-----------------------------------------|--|--|--|--|--|--|--|--|--|
|        | SalesReceiptTrans-                 | ServiceChargeTrans- TimecardTransaction |  |  |  |  |  |  |  |  |  |
|        | action                             | action                                  |  |  |  |  |  |  |  |  |  |
|        | TransactionFactory- Implementation |                                         |  |  |  |  |  |  |  |  |  |

Таблица 22.4.

| Название пакета             | N  | A | C <sub>a</sub> | C <sub>e</sub> | R | H    | I    | A    | D    | D'   |
|-----------------------------|----|---|----------------|----------------|---|------|------|------|------|------|
| AbstractTransactions        | 5  | 5 | 13             | 1              | 0 | 0,20 | 0,07 | 1    | 0,05 | 0,07 |
| Application                 | 1  | 1 | 1              | 0              | 0 | 1    | 0    | 1    | 0    | 0    |
| PayrollApplication          | 1  | 0 | 0              | 5              | 0 | 1    | 1    | 0    | 0    | 0    |
| PayrollDatabase             | 1  | 1 | 19             | 5              | 0 | 1    | 0,21 | 1    | 0,15 | 0,21 |
| PayrollDatabaseImplemen-    | 1  | 0 | 0              | 1              | 0 | 1    | 1    | 0    | 0    | 0    |
| ation                       |    |   |                |                |   |      |      |      |      |      |
| PayrollDomain               | 5  | 4 | 30             | 0              | 4 | 1    | 0    | 0,80 | 0,14 | 0,20 |
| PayrollFactory              | 1  | 1 | 12             | 4              | 0 | 1    | 0,25 | 1    | 0,18 | 0,25 |
| PayrollImplementation       | 14 | 0 | 1              | 5              | 3 | 0,29 | 0,83 | 0    | 0,12 | 0,17 |
| TextParserTransactionSource | 1  | 0 | 1              | 3              | 0 | 1    | 0,75 | 0    | 0,18 | 0,25 |
| TransactionApplication      | 3  | 3 | 14             | 1              | 3 | 1,33 | 0,07 | 1    | 0,05 | 0,07 |
| TransactionFactory          | 1  | 1 | 3              | 1              | 0 | 1    | 0,25 | 1    | 0,18 | 0,20 |
| TransactionImplementation   | 19 | 0 | 1              | 14             | 0 | 0,05 | 0,93 | 0    | 0,05 | 0,07 |

## Резюме

Вопросы, связанные с управлением структурами пакетов, возникают при увеличении размеров программы и численности коллектива разработчиков. Но даже в случае привлечения к проекту небольшой команды программистов следует разбивать исходный код на составные части, чтобы обрабатывать его независимым образом. Крупные программы, не обладающие структурой разбиения, могут превращаться в непонятное нагромождение исходных файлов.

## Литература

1. Benjamin/Cummings. *Object-Oriented Analysis and Design with Applications*, 1994.
2. DeMarco T. *Controlling Software Projects*. Yourdon Press, 1982.

# ЧАСТЬ V

## Практическое занятие: моделирование метеостанции

Следующие главы содержат подробное описание практического занятия, смысл которого заключается в моделировании простой системы мониторинга погоды. Хотя это практическое занятие является воображаемым, тем не менее, оно достаточно хорошо отражает реальную ситуацию. Здесь будут смоделированы проблемы нехватки времени, наследственного кода, плохих и изменяющихся спецификаций, новых неиспытанных технологий и т.п. Нашей целью является демонстрация того, как принципы, шаблоны и практики, которые были изучены ранее, используются в реальной разработке программ.

В процессе формирования программы, моделирующей работу метеостанции, будут рассмотрены несколько полезных шаблонов проектирования.

# 23

## Шаблон Composite



Шаблон **Composite** является очень простым, но его применение влечет за собой серьезные последствия. Фундаментальная структура шаблона **Composite** показана на рис. 23.1. Здесь можно видеть иерархию, основанную на формах. Базовый класс **Shape** включает две производные формы **Circle** и **Square**. Третья производная форма является составной. Форма **CompositeShape** включает список нескольких экземпляров **Shape**. При вызове метода **draw()** для

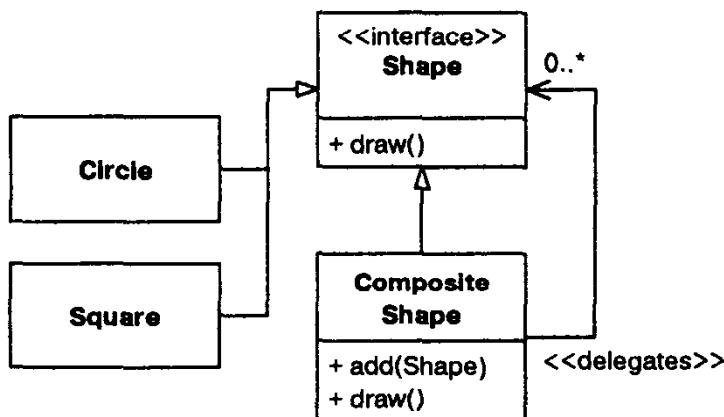


Рис. 23.1. Шаблон Composite

**CompositeShape** производится его делегирование всем экземплярам **Shape**, находящимся в списке.

Таким образом, экземпляр **CompositeShape** появляется в системе в качестве одинарной **Shape**. Этот экземпляр может быть передан любой функции или объекту, которые используют **Shape**, и будет вести себя как **Shape**. Однако в действительности это прокси-объект<sup>1</sup> для группы экземпляров **Shape**.

В листингах 23.1 и 23.2 показана одна возможная реализация **CompositeShape**.

---

#### Листинг 23.1. Shape.java

---

```

public interface Shape
{
 public void draw();
}

```

---



---

#### Листинг 23.2. CompositeShape.java

---

```

import Java.util.Vector;

public class CompositeShape implements Shape
{
 private Vector itsShapes = new Vector();
 public void add(Shape s)
 {
 itsShapes.add(s);
 }

 public void draw()
 {
 for (int i = 0; i < itsShapes.size(); i++)
 {

```

<sup>1</sup>Обратите внимание на схожесть со структурой в шаблоне Proxy.

```
 Shape shape = (Shape) itsShapes.elementAt(i);
 shape.draw();
}
}
```

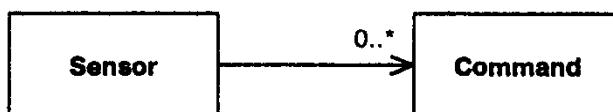
## **Пример: составные команды**

Обратимся к рассмотрению объектов `Sensors` и `Command`, о которых уже говорилось в главе 13. Там на рис. 13.3 приводился класс `Sensor`, использующий класс `Command`. Когда `Sensor` обнаруживает сигнал вызова, в `Command` вызывается метод `do()`.

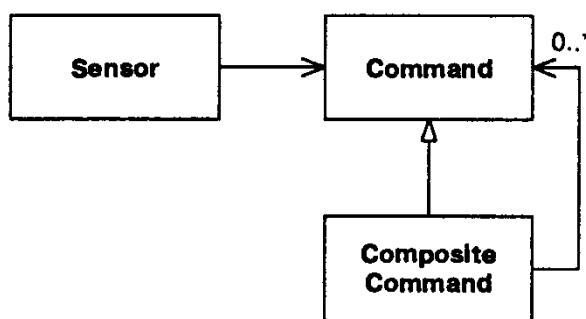
В приведенном ранее рассмотрении не было упомянуто, что существует много случаев, когда Sensor должен выполнять более чем одну Command. Например, когда бумага достигает определенного места в механизме протяжки, она должна отключить оптический датчик. Этот датчик затем останавливает один двигатель, запускает другой двигатель и включает захваты.

Сначала было принято во внимание, что каждый класс `Sensor` должен обслуживать список объектов `Command` (рис. 23.2). Однако вскоре было обнаружено, что всякий раз, когда `Sensor` нуждался в выполнении более чем одной `Command`, он всегда обращался с этими объектами `Command` идентично. То есть, `Sensor` только итерировал по списку и вызывал метод `do()` для каждой `Command`. И это было идеально в случае использования шаблона `Composite`.

Таким образом, класс `Sensor` был оставлен и создан `CompositeCommand`, как показано на рис. 23.3.



**Рис. 23.2.** Sensor включает несколько команд



**Рис. 23.3.** Составная команда

Это значит, что не требуется изменять `Sensor` или `Command`. Можно добавить множественность `Commands` к `Sensor` без их изменения. Это приложение соответствует принципу ОСР.

## Множественное и единственное число

А теперь мы приходим к интересному выводу. Была возможность определять поведение `Sensors` так, как будто бы они содержали много `Commands`, причем без необходимости изменять `Sensors`. Существует множество других подобных ситуаций в обычном программном проекте, т.е. должны быть случаи, когда можно использовать шаблон `Composite` вместо того, чтобы формировать список или вектор объектов.

Это можно объяснить по-другому. Взаимосвязь между `Sensor` и `Command` является однозначной. Изменив эту взаимосвязь на “один ко многим”, мы неожиданно получили способ имитировать поведение взаимосвязи “один ко многим” без фактического наличия этого типа связи. Однозначная взаимосвязь проще для понимания, программирования и поддержки, чем взаимосвязь “один ко многим”. Т.е. мы получили идеальный проект. Таким образом шаблон `Composite` позволяет заменять в проекте взаимосвязи “один ко многим” однозначными взаимосвязями.

Конечно, не все взаимосвязи “один ко многим” могут быть заменены однозначными взаимосвязями с помощью шаблона `Composite`. Только те, в которых каждый объект в списке обрабатывается идентично, являются “кандидатами” на подобное изменение. Например, пусть поддерживается список работников, в котором следует определить тех, день выплаты зарплаты которых настал сегодня. В этом случае шаблон `Composite` непригоден, поскольку различные работники трактуются различным образом.

Тем не менее, существует довольно много взаимосвязей “один ко многим”, которые могут быть преобразованы с помощью шаблона `Composite`. И получаемые в этом случае преимущества существенны. Вместо дублированной обработки списка и выполнения итерационного кода для каждого из клиентов, достаточно выполнить требуемый код один раз в составном классе.

# 24

## Обратно к шаблонам: Observer



В процессе написания этой главы преследовались особые цели. Здесь рассматривается шаблон *Observer*<sup>1</sup>, но описание этого шаблона не является в данном случае главной задачей. Данная глава посвящена проблеме применения шаблона в процессе разработки и в кодировании.

В предыдущих главах уже было описано достаточное количество шаблонов, но вопрос о предварительной обработке кодов почти не обсуждался. В этой ситуации может показаться, что готовые шаблоны просто включаются в код или проект. Но именно так поступать вовсе не рекомендуется. Желательно постепенно адаптировать код в соответствии с изменяющимися требованиями. В процессе рефакторинга кода с учетом связывания, упрощения и повышения действенности, обнаруживается, что окончательный вариант становится похожим на определенный шаблон. Если имеет место подобная ситуация, следует изменить названия классов и переменных с учетом названия шаблона, модифицировать структуру

<sup>1</sup>[GOF95], с. 293

кода, что позволит применять шаблон более регулярным образом. Таким образом, код “возвращается к шаблону”.

В данной главе на примере несложной проблемы показано, каким образом совершенствуются проект и код. Результатом эволюционных преобразований является переход к шаблону *Observer*.

## Электронные часы

Предположим, что мы имеем дело с объектом, моделирующим часы. Этот объект воспринимает миллисекундные прерывания операционной системы (известные как такты) и преобразовывает их в соответствующее показание часов. Объект определяет секунды на основе миллисекунд, минуты — на основе секунд, часы — на основе минут, дни — на основе часов и т.д. Также он располагает информацией о количестве дней в месяце и количестве месяцев в году. Имеются данные о високосных годах, а также о времени их наступления. Этот объект полностью “осведомлен” о текущем времени (рис. 24.1).

Наша задача заключается в создании электронных часов, отображающихся на рабочем столе, которые показывали бы текущее время. Для этого можно воспользоваться следующим кодом.

```
public void DisplayTime
{
 while(1)
 {
 int sec = clock.getSeconds();
 int min = clock.getMinutes();
 int hour = clock.getHours();
 showTime(hour,min,sec);
 }
}
```

Очевидно, что подобное решение не является оптимальным. В случае его реализации все свободные циклы ЦПУ будут использоваться для отображения текущего времени. В большинстве случаев это является излишним, поскольку показания часов не изменяются столь часто. Конечно, подобную методику можно использовать в случае “конструирования” каких-нибудь настенных цифровых

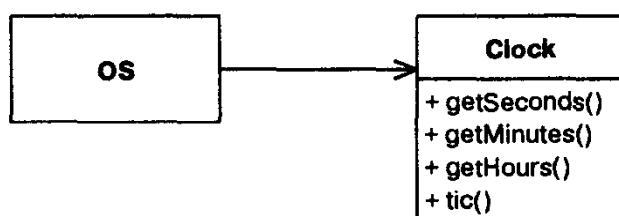


Рис. 24.1. Объект Clock

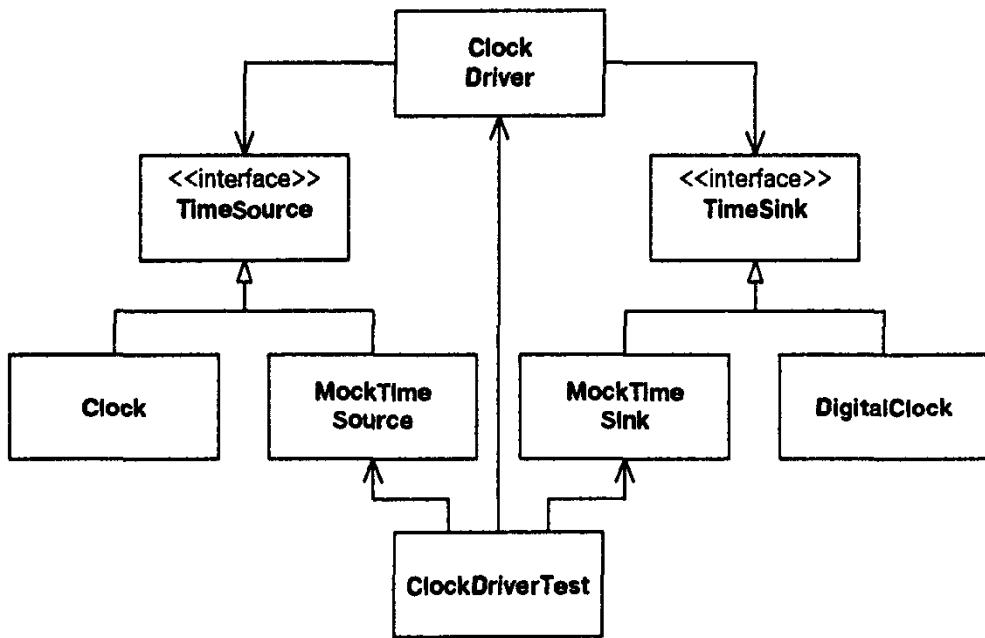


Рис. 24.2. Тестирование объекта DigitalClock

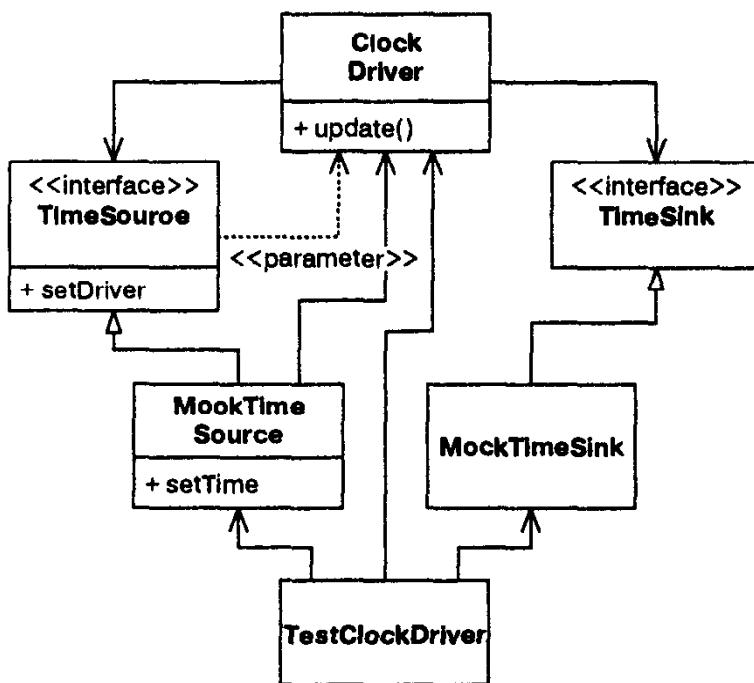
часов. Но для формирования изображений часов на рабочем столе использовать циклы ЦПУ весьма нежелательно.

Основная проблема состоит в реализации эффективного перенаправления данных от *Clock* к *DigitalClock*. Предполагается, что существует как объект *Clock*, так и объект *DigitalClock*. Между этими объектами необходимо установить связь. Это соединение можно протестировать, что позволит удостовериться, что данные, получаемые из *Clock*, совпадают с данными, пересылаемыми объекту *DigitalClock*.

Проще всего в процессе написания этого теста создать один интерфейс, претендующий на отображение функций *Clock*, а также другой интерфейс, имеющий отношение к *DigitalClock*. Затем определяются специальные тестовые объекты, реализующие эти интерфейсы, а также удостоверяющие корректную поддержку установленных между ними связей (рис. 24.2).

Объект *ClockDriverTest* связывает *ClockDriver* с двумя фиктивными объектами с помощью интерфейсов *TimeSource* и *TimeSink*. Затем производится проверка каждого из фиктивных объектов, чтобы удостовериться в том, что *ClockDriver* контролирует передачу времени из источника целевому объекту. При необходимости *ClockDriverTest* также гарантирует сохранение приемлемого уровня эффективности.

Представляет интерес включение интерфейсов в проект после завершения тестирования. Чтобы протестировать модуль, необходимо изолировать его от других модулей системы, подобно тому, как объект *ClockDriver* изолируется от *Clock* и *Digitalclock*. Рассматриваемые тесты помогают минимизировать количество связей в проектах.



**Рис. 24.3.** Получение значения TimeSource для обновления ClockDriver

Каким же образом функционирует *ClockDriver*? В целях повышения степени эффективности *ClockDriver* должен проверять каждое изменение значения времени, получая доступ к объекту *TimeSource*. И только тогда значение времени передается объекту *TimeSink*. Каким же образом *ClockDriver* узнает о том, что значение времени изменилось? Можно запрашивать об этом объект *TimeSource*, но это приведет к повторному возникновению проблемы с “громоздкими” циклами ЦПУ.

Проще всего сведения об изменении времени направлять объекту *ClockDriver* с помощью объекта *Clock*. Объект *ClockDriver* можно связать с *Clock* с помощью интерфейса *TimeSource*, тогда при изменении значения времени *Clock* обновит значение *ClockDriver*. Объект же *ClockDriver*, в свою очередь, установит значение времени для *ClockSink* (рис. 24.3).

Обратите внимание на зависимость, возникающую между *TimeSource* и *ClockDriver*. Ее причина заключается в том, что аргументом метода *setDriver* является *ClockDriver*. Это очень удобно, поскольку объекты *TimeSource* должны в любом случае использовать объекты *ClockDriver*. Но следует с осторожностью относиться к любым преобразованиям этой зависимости, чтобы не потерять функциональные свойства программы.

Код из листинга 24.1 демонстрирует тестовый случай для *ClockDriver*. Обратите внимание, что в данном случае создается *ClockDriver*, а *MockTimeSource* и *MockTimeSink* привязываются к нему. Затем в исходном объекте устанавливается значение времени и ожидается, что оно “магическим образом продвинется дальше”. Остальная часть кода демонстрируется в листингах 24.2–24.6.

---

**Листинг 24.1. ClockDriverTest.java**

---

```
import junit.framework.*;

public class ClockDriverTest extends TestCase
{
 public ClockDriverTest(String name)
 {
 super(name);
 }

 public void testTimeChange()
 {
 MockTimeSource source = new MockTimeSource();
 MockTimeSink sink = new MockTimeSink();
 ClockDriver driver = new ClockDriver(source,sink);
 source.setTime(3,4,5);
 assertEquals(3, sink.getHours());
 assertEquals(4, sink.getMinutes());
 assertEquals (5, sink.getSeconds());

 source.setTime(7,8,9);
 assertEquals(7, sink.getHours());
 assertEquals(8, sink.getMinutes());
 assertEquals(9, sink.getSeconds());
 }
}
```

---

---

**Листинг 24.2. TimeSource.java**

---

```
public interface TimeSource
{
 public void setDriver(ClockDriver driver);
}
```

---

---

**Листинг 24.3. TimeSink.java**

---

```
public interface TimeSink
{
 public void setTime(int hours, int minutes, int seconds);
}
```

---

---

**Листинг 24.4. ClockDriver.java**

---

```
public class ClockDriver
{
 private TimeSink itsSink;
```

```
public ClockDriver(TimeSource source, TimeSink sink)
{
 source.setDriver(this);
 itsSink = sink;
}

public void update(int hours, int minutes, int seconds)
{
 itsSink.setTime(hours, minutes, seconds);
}
}
```

---

**Листинг 24.5. MockTimeSource.java**

---

```
public class MockTimeSource implements TimeSource
{
 private ClockDriver itsDriver;

 public void setTime(int hours, int minutes, int seconds)
 {
 itsDriver.update(hours, minutes, seconds);
 }

 public void setDriver(ClockDriver driver)
 {
 itsDriver = driver;
 }
}
```

---

**Листинг 24.6. MockTimeSink.java**

---

```
public class MockTimeSink implements TimeSink
{
 private int itsHours;
 private int itsMinutes;
 private int itsSeconds;

 public int getSeconds()
 {
 return itsSeconds;
 }

 public int getMinutes()
 {
 return itsMinutes;
 }

 public int getHours()
 {
 return itsHours;
 }
}
```

```

 return itsHours;
}

public void setTime(int hours, int minutes, int seconds)
{
 itsHours = hours;
 itsMinutes = minutes;
 itsSeconds = seconds;
}
}

```

Теперь попробуем усовершенствовать разработанный код. Сначала следует устранить зависимость `TimeSource` от `ClockDriver`, поскольку интерфейс `TimeSource` должен применяться любыми объектами, а не только объектами `ClockDriver`. Для этого создадим интерфейс, применяющий `TimeSource`, который сможет реализовать `ClockDriver`. Этот интерфейс назовем `ClockObserver`. Обратитесь к листингам 24.7-24.10. Измененный код выделен с помощью полужирного шрифта.

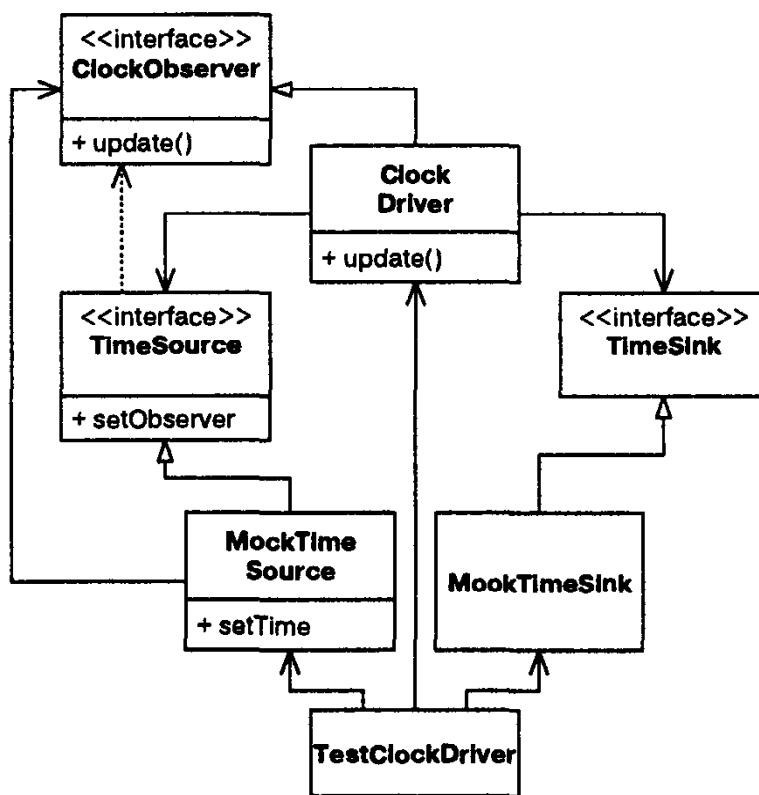


Рис. 24.4. Устранение зависимости между `TimeSource` и `ClockDriver`

---

**Листинг 24.7. ClockObserver.java**

---

```
public interface ClockObserver
{
 public void update(int hours, int minutes, int seconds);
}
```

---

---

**Листинг 24.8. ClockDriver.java**

---

```
public class ClockDriver implements ClockObserver
{
 private TimeSink itsSink;

 public ClockDriver(TimeSource source, TimeSink sink)
 {
 source.setObserver(this);
 itsSink = sink;
 }
 public void update(int hours, int minutes, int seconds)
 {
 itsSink.setTime(hours, minutes, seconds);
 }
}
```

---

---

**Листинг 24.9. TimeSource.java**

---

```
public interface TimeSource
{
 public void setObserver(ClockObserver observer);
}
```

---

---

**Листинг 24.10. MockTimeSource.java**

---

```
public class MockTimeSource implements TimeSource
{
 private ClockObserver itsObserver;

 public void setTime(int hours, int minutes, int seconds)
 {
 itsObserver.update(hours, minutes, seconds);
 }
 public void setObserver(ClockObserver observer)
 {
 itsObserver = observer;
 }
}
```

Получилось значительно лучше. Никто не сможет воспользоваться объектом `TimeSource`. Для этого следует создать копию `ClockObserver` и вызвать `SetObserver`, передавая в качестве аргумента значение.

Желательно для отсчета времени использовать более одного объекта `TimeSink`. Одно значение используется электронными часами. Другое значение применяется для поддержки времени службы напоминания. Еще одно предназначено для запуска службы ночного резервного копирования. В общем, один объект `TimeSource` должен поддерживать значение времени для нескольких объектов `TimeSink`.

Итак, изменим конструктор `ClockDriver`, обращающийся к объекту `TimeSource`, затем включим метод `addTimeSink`, который позволит в любое удобное для вас время создавать экземпляры `TimeSink`.

В этом случае также получаются два нежелательных следствия. При вызове `setObserver` необходимо указывать `TimeSource` для объекта `ClockObserver`. Также необходимо указывать `ClockDriver` для экземпляров `TimeSink`. Можно ли исключить эти моменты?

Рассматривая `ClockObserver` и `TimeSink`, замечаем, что они применяют метод `setTime`. То есть `TimeSink` образует копию `ClockObserver`. Если выполнить это, тестовая программа создает `MockTimeSink` и вызывает `setObserver` для `TimeSource`. Можно избавиться сразу и от `ClockDriver` (и от `TimeSink`)! Код из листинга 24.11 демонстрирует изменения в `ClockDriverTest`.

---

#### Листинг 24.11. `ClockDriverTest.java`

---

```
import junit.framework.*;

public class ClockDriverTest extends TestCase
{
 public ClockDriverTest(String name)
 {
 super(name);
 }

 public void testTimeChange()
 {
 MockTimeSource source = new MockTimeSource();
 MockTimeSink sink = new MockTimeSink();
 source.setObserver(sink);

 source.setTime(3,4,5);
 assertEquals(3, sink.getHours());
 assertEquals(4, sink.getMinutes());
 assertEquals(5, sink.getSeconds());

 source.setTime(7,8,9);
```

```
 assertEquals(7, sink.getHours());
 assertEquals(8, sink.getMinutes());
 assertEquals(9, sink.getSeconds());
 }
}
```

Следовательно, `MockTimeSink` должен создавать копию `ClockObserver` вместо `TimeSink`. Обратите внимание на листинг 24.12. Изменения оказались вполне функциональными. Почему `ClockDriver` необходимо разместить на первом месте? На рис. 24.5 показана соответствующая UML-диаграмма.

#### Листинг 24.12. `MockTimeSink.java`

```
public class MockTimeSink implements ClockObserver
{
 private int itsHours;
 private int itsMinutes;
 private int itsSeconds;

 public int getSeconds()
 {
 return itsSeconds;
 }

 public int getMinutes()
 {
 return itsMinutes;
 }

 public int getHours()
 {
 return itsHours;
 }

 public void update(int hours, int minutes, int seconds)
 {
 itsHours = hours;
 itsMinutes = minutes;
 itsSeconds = seconds;
 }
}
```

Очевидно, что данный вариант кода значительно проще.

Теперь несколько объектов `TimeSink` можно обрабатывать путем изменения функции `setObserver` на `registerObserver`. При этом гарантируется, что все зарегистрированные экземпляры `ClockObserver` отобразятся в списке, а также модифицируются соответствующим образом. Теперь в тестовую программу следует внести другие изменения. Код из листинга 24.13 демонстрирует эти

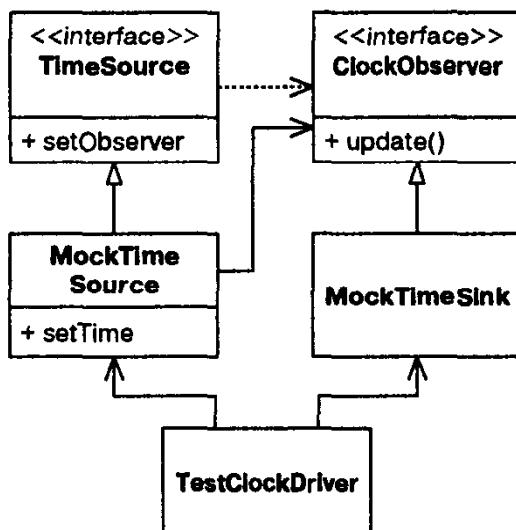


Рис. 24.5. Результат устранения ClockDriver и TimeSink

изменения. Также в тестовой программе выполнен небольшой рефакторинг, что значительно сократит ее размеры и облегчит просмотр кода.

---

#### Листинг 24.13. ClockDriverTest.java

---

```

import junit.framework.*;

public class ClockDriverTest extends TestCase
{
 private MockTimeSource source;
 private MockTimeSink sink;

 public ClockDriverTest(String name)
 {
 super(name);
 }

 public void setUp()
 {
 source = new MockTimeSource();
 sink = new MockTimeSink();
 source.registerObserver(sink);
 }

 private void assertSinkEquals(
 MockTimeSink sink, int hours, int minutes, int seconds)
 {
 assertEquals(hours, sink.getHours());
 assertEquals(minutes, sink.getMinutes());
 assertEquals(seconds, sink.getSeconds());
 }

 public void testTimeChange()
 {
 source.setTime(12, 30, 0);
 assertEquals(12, sink.getHours());
 assertEquals(30, sink.getMinutes());
 assertEquals(0, sink.getSeconds());
 source.setTime(13, 0, 0);
 assertEquals(13, sink.getHours());
 assertEquals(0, sink.getMinutes());
 assertEquals(0, sink.getSeconds());
 }
}

```

```
{
 source.setTime(3,4,5);
 assertEquals(sink, 3,4,5);
 source.setTime(7,8,9);
 assertEquals(sink, 7,8,9);
}

public void testMultipleSinks()
{
 MockTimeSink sink2 = new MockTimeSink();
 source.registerObserver(sink2);

 source.setTime(12,13,14);
 assertEquals(sink, 12,13,14);
 assertEquals(sink2, 12,13,14);
}
}
```

Для выполнения рассматриваемых заданий следует внести небольшие изменения. Преобразуем `MockTimeSource` так, чтобы в него вошли все зарегистрированные наблюдатели `Vector`. Затем, когда изменится значение времени, выполним итерацию с использованием `Vector` и вызовем `update` для всех зарегистрированных `ClockObservers`. Код из листингов 24.14 и 24.15 демонстрирует эти изменения. На рис. 24.6 показана соответствующая UML-диаграмма.

---

#### Листинг 24.14. `TimeSource.java`

---

```
public interface TimeSource
{
 public void registerObserver(ClockObserver observer);
}
```

---

---

#### Листинг 24.15. `MockTimeSource.java`

---

```
import java.util.*;

public class MockTimeSource implements TimeSource
{
 private Vector itsObservers = new Vector();

 public void setTime(int hours, int minutes, int seconds)
 {
 Iterator i = itsObservers.iterator();
 while (i.hasNext())
 {
 ClockObserver observer = (ClockObserver) i.next();
 observer.update(hours, minutes, seconds);
 }
 }
}
```

---

```

public void registerObserver(ClockObserver observer)
{
 itsObservers.add(observer);
}
}

```

---

Все отлично, но программиста не устраивает, что `MockTimeSource` выполняет регистрацию и обновление. Отсюда следует, что `Clock` и все остальные производные классы `TimeSource` должны дублировать код, относящийся к регистрации и обновлению. Автор полагает, что `Clock` не должен иметь отношения к регистрации и обновлению. Также не вполне уместно дублирование кода. Итак, желательно переместить весь подготовительный материал в `TimeSource`. Естественно, `TimeSource` превратится из интерфейса в класс.

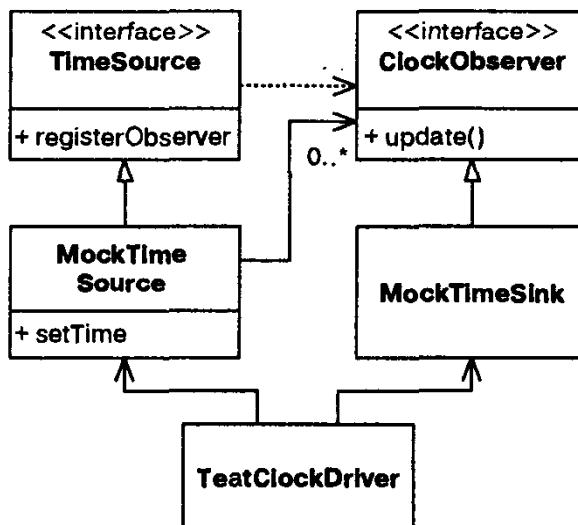


Рис. 24.6. Обработка нескольких объектов `TimeSink`

В результате получим резкое сокращение размеров `MockTimeSource`. Имеющие место изменения отображаются в листингах 24.16 и 24.17, а также на рис. 24.7.

---

#### Листинг 24.16. `TimeSource.java`

```

import java.util.*;

public class TimeSource
{
 private Vector itsObservers = new Vector();

 protected void notify(int hours, int minutes, int seconds)
 {
 Iterator i = itsObservers.iterator();
 while (i.hasNext())

```

```

ClockObserver observer = (ClockObserver) i.next();
observer.update(hours, minutes, seconds);
}

public void registerObserver(ClockObserver observer)
{
 itsObservers.add(observer);
}
}

```

---

**Листинг 24.17. MockTimeSource.java**

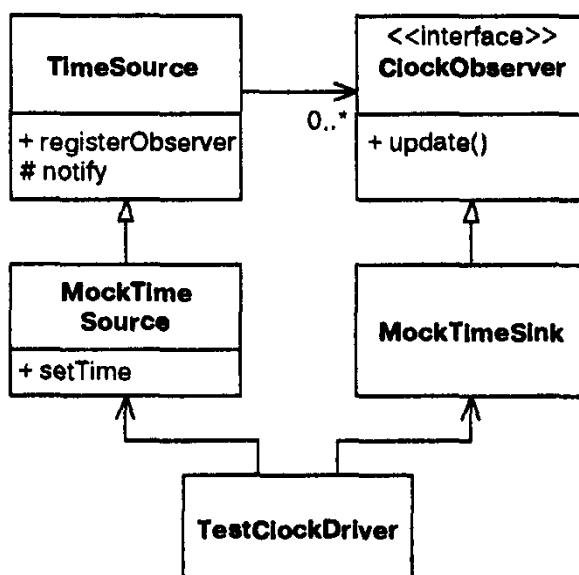
```

public class MockTimeSource extends TimeSource
{
 public void setTime(int hours, int minutes, int seconds)
 {
 notify(hours, minutes, seconds);
 }
}

```

---

Это уже небольшое достижение. Теперь можно создавать производные классы на основе TimeSource. Для этого необходимо только получить наблюдателей, обновленных с учетом обращения к `notify`. Но именно этого желательно избегать. MockTimeSource наследуется непосредственно из TimeSource. Значит, Clock также должен быть производным классом от TimeSource. Почему Clock обязательно должен зависеть от процессов регистрации и обновления? Он является лишь классом, который получает информацию о времени. Зависимость от TimeSource кажется обязательной и в то же время нежелательной.



**Рис. 24.7. Перемещение процедур регистрации и обновления в TimeSource**

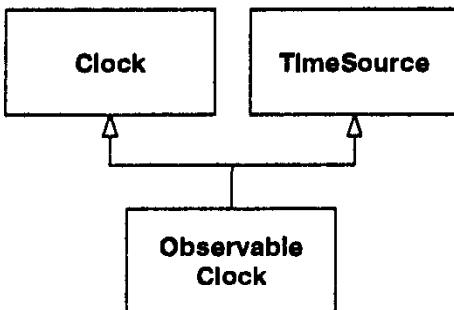


Рис. 24.8. Применение множественного наследования в C++ для отделения Clock от TimeSource

Известен способ разрешения этих затруднений при работе в C++. Как для TimeSource, так и для Clock создается подкласс под названием ObservableClock. Перекрывается использование `tic` и `setTime` в ObservableClock для вызова `tic`, либо `setTime` в Clock и затем вызывается `notify` из TimeSource. Обратите внимание на листинг 24.18 и рис. 24.8.

---

#### Листинг 24.18. ObservableClock.cc (C++)

---

```

class ObservableClock : public Clock, public TimeSource
{
public:
 virtual void tic()
 {
 Clock::tic();
 TimeSource::notify(getHours(), getMinutes(), getSeconds());
 }

 virtual void setTime(int hours, int minutes, int seconds)
 {
 Clock::setTime(hours, minutes, seconds);
 TimeSource::notify(hours, minutes, seconds);
 }
};

```

---

К сожалению, подобная возможность отсутствует в Java, поскольку в этом языке не используется множественное наследование классов. Итак, при работе в Java либо оставим все без изменения, либо применим прием делегирования. Реализация этого метода показана в листингах 24.19–24.21 и на рис. 24.9.

---

#### Листинг 24.19. TimeSource.java

---

```

public interface TimeSource
{
 public void registerObserver(ClockObserver observer);

```

---

**Листинг 24.20. TimeSourceImplementation.java**

---

```
import java.util.*;

public class TimeSourceImplementation
{
 private Vector itsObservers = new Vector();
 public void notify(int hours, int minutes, int seconds)
 {
 Iterator i = itsObservers.iterator();
 while (i.hasNext())
 {
 ClockObserver observer = (ClockObserver) i.next();
 observer.update(hours, minutes, seconds);
 }
 }

 public void registerObserver(ClockObserver observer)
 {
 itsObservers.add(observer);
 }
}
```

---

---

**Листинг 24.21. MockTimeSource.java**

---

```
public class MockTimeSource implements TimeSource
{
 TimeSourceImplementation tsImp =
 new TimeSourceImplementation();

 public void registerObserver(ClockObserver observer)
 {
 tsImp.registerObserver(observer);
 }

 public void setTime(int hours, int minutes, int seconds)
 {
 tsImp.notify(hours, minutes, seconds);
 }
}
```

---

Обратите внимание, что класс `MockTimeSource` реализует `TimeSource` и содержит ссылку на экземпляр `TimeSourceImplementation`. Заметим также, что все вызовы, направляемые методу `registerObserver` из `MockTimeSource`, делегируются объекту `TimeSourceImplementation`. Итак, `MockTimeSource.setTime` вызывает `notify` с помощью экземпляра `TimeSourceImplementation`.

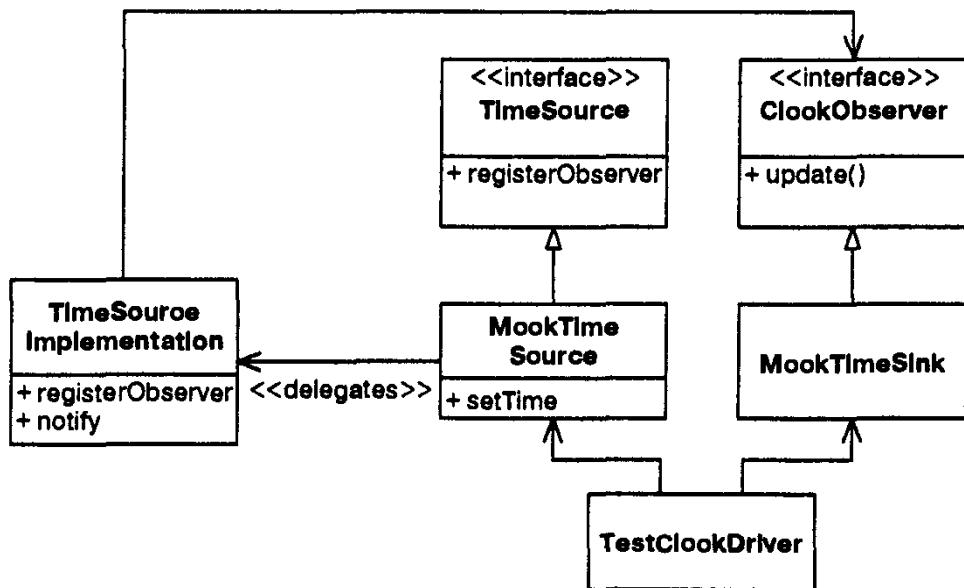


Рис. 24.9. Выполнение делегирования с помощью Observer в Java

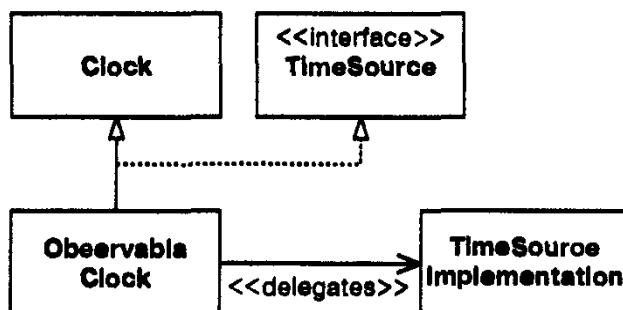
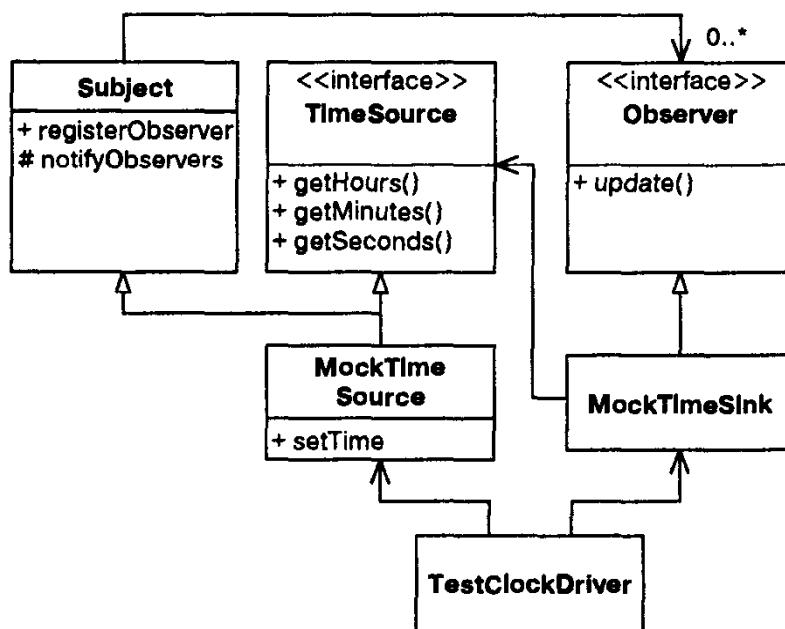


Рис. 24.10. Выполнение делегирования для ObservableClock

Хотя это и не совсем правильно, но преимущество в данном случае состоит в том, что `MockTimeSource` не расширяет класс. То есть при создании `ObservableClock` можно расширить `Clock`, реализовать `TimeSource` и делегировать его `TimeSourceImplementation` (рис. 24.10.) Подобный подход решает проблему с `Clock`, связанную с регистрацией и обновлением, но цена этого решения достаточно велика.

Итак, вернемся к тому “положению вещей”, которое отображено на рис. 24.7, до формирования рассматриваемых представлений. Просто согласимся с тем, что `Clock` зависит от всего процесса регистрации и обновления.

При этом `TimeSource` – не совсем удачное название для данного класса. По сравнению с тем периодом, когда применялся `ClockDriver`, появилось множество изменений. Необходимо заменить название с учетом процессов регистрации и обновления. Шаблон `Observer` вызывает класс `Subject`. Поскольку следует учесть специфику задачи, назовем его `TimeSubject`, но это название не отве-



**Рис. 24.11.** Заключительная версия *Observer*, примененная к *MockTimeSource* и *MockTimeSink*

чает интуитивным представлениям. Можно применить известный моникер Java *Observable*, но неужели *TimeObservable* звучит гармоничнее? Ничуть.

Возможно, затруднения вытекают из специфики наблюдателя “модели для толчка”<sup>2</sup>. При переходе к “модели для толчка” можно обобщить данный класс. Затем название *TimeSource* изменяется на *Subject*, и каждый, кто знаком с шаблоном *Observer*, будет лучше осведомлен о том, с чем он имеет дело.

Рассматриваемую возможность можно считать весьма неплохой. Вместо того чтобы передавать значение времени с помощью методов *notify* и *update*, можно устроить так, что *TimeSink* запросит *MockTimeSource* о показаниях времени. Нежелательно, чтобы *MockTimeSink* содержал сведения о *MockTimeSource*, поэтому создадим интерфейс, применяемый *MockTimeSink* для получения информации о времени. *MockTimeSource* (и *Clock*) реализует этот интерфейс. Данный интерфейс можно назвать *TimeSource*.

Заключительная версия кода и соответствующая UML-диаграмма представлены на рис. 24.11 и в листингах 24.22–24.27.

#### Листинг 24.22. ObserverTest.java

```

import junit.framework.*;

public class ObserverTest extends TestCase
{

```

<sup>2</sup>Наблюдатели “выталкивающей модели” перемещают данные от субъекта к наблюдателю, используя для этого методы *notify* и *update*. Наблюдатели “выталкивающей модели” не передают каких-либо сведений с помощью методов *notify* и *update*. Они зависят от объекта наблюдения, запрашивая наблюдаемый объект при получении обновления. См. [GOF95].

```
private MockTimeSource source;
private MockTimeSink sink;

public ObserverTest(String name)
{
 super(name);
}

public void setUp()
{
 source = new MockTimeSource();
 sink = new MockTimeSink(source);
 source.registerObserver(sink);
 private void assertSinkEquals(
}

private void assertSinkEquals(
 MockTimeSink sink, int hours, int minutes, int seconds)
{
 assertEquals(hours, sink.getHours());
 assertEquals(minutes, sink.getMinutes());
 assertEquals(seconds, sink.getSeconds());
}

public void testTimeChange()
{
 source.setTime(3,4,5);
 assertSinkEquals(sink, 3,4,5);

 source.setTime(7,8,9);
 assertSinkEquals(sink, 7,8,9);
}

public void testMultipleSinks()
{
 MockTimeSink sink2 = new MockTimeSink(source);
 source.registerObserver(sink2);

 source.setTime(12,13,14);
 assertSinkEquals(sink, 12,13,14);
 assertSinkEquals(sink2, 12,13,14);
}
```

---

---

**Листинг 24.23. Observer.java**

---

```
public interface Observer
{
 public void update();
```

---

**Листинг 24.24. Subject.java**

---

```
import java.util.*;

public class Subject
{
 private Vector itsObservers = new Vector();

 protected void notifyObservers()
 {
 Iterator i = itsObservers.iterator();
 while (i.hasNext())
 {
 Observer observer = (Observer) i.next();
 observer.update ();
 }
 }

 public void registerObserver(Observer observer)
 {
 itsObservers.add(observer);
 }
}
```

---

---

**Листинг 24.25. TimeSource.java**

---

```
public interface TimeSource
{
 public int getHours();
 public int getMinutes();
 public int getSeconds();
}
```

---

---

**Листинг 24.26. MockTimeSource.java**

---

```
public class MockTimeSource extends Subject
 implements TimeSource
{
 private int itsHours;
 private int itsMinutes;
 private int itsSeconds;

 public void setTime(int hours, int minutes, int seconds)
 {
 itsHours = hours;
 itsMinutes = minutes;
 itsSeconds = seconds;
 notifyObservers();
 }
```

```
}

public int getHours()
{
 return itsHours;
}

public int getMinutes()
{
 return itsMinutes;
}

public int getSeconds()
{
 return itsSeconds;
}
}
```

---

---

**Листинг 24.27. MockTimeSink.java**

---

```
public class MockTimeSink implements Observer
{
 private int itsHours;
 private int itsMinutes;
 private int itsSeconds;
 private TimeSource itsSource;

 public MockTimeSink(TimeSource source)
 {
 itsSource = source;
 }

 public int getSeconds()
 {
 return itsSeconds;
 }

 public int getMinutes()
 {
 return itsMinutes;
 }

 public int getHours()
 {
 return itsHours;
 }

 public void update ()
 {
 itsHours = itsSource.getHours();
 }
}
```

```
 itsMinutes = itsSource.getMinutes();
 itsSeconds = itsSource.getSeconds();
}
}
```

## Резюме

Мы начали рассмотрение с проблемы, возникшей при разработке, и в результате вполне обоснованной эволюции остановились на использовании “канонического” шаблона *Observer*. Читатель может удивиться, что обращение к шаблону было предопределено автором.

Если вы знакомы с шаблонами проектирования, то при появлении проблемы мысль об их использовании сразу же всплывает в вашем представлении. Вопрос в том, реализовать ли шаблон непосредственно или же предпочесть постепенное развитие в этом направлении. Данная глава подтверждает большую действенность второго варианта. Постепенно становится ясно, что код эволюционирует в направлении шаблона *Observer*, изменились названия, и код принял “каноническую” форму.

## Применение диаграмм в данной главе

Часть диаграмм создана для удобства читателей.

Обычно эти диаграммы описывали промежуточные шаги. Имеет ли смысл детализировать подобные диаграммы? Если вам необходимо обосновать свое решение, как это делает автор данной книги, применение диаграмм весьма удобно. Но обычно нет необходимости документировать эволюционный путь, отнявший несколько рабочих часов. Как правило, эти диаграммы отражают “мимолетные состояния” и не сохраняются на долгое время, т.е. код выполняет роль документации. На более высоких уровнях разработки этот вывод не всегда справедлив.

## Шаблон *Observer*

“Каноническая” форма этого шаблона показана на рис. 24.12. В этом примере *Clock* наблюдается с помощью *DigitalClock*. Объект *DigitalClock* поддерживает регистрацию с помощью интерфейса *Subject* из *Clock*. Объект *Clock* вызывает метод *notify* из *Subject*, когда бы не изменилось значение времени. Метод *notify* из *Subject* вызывает метод *update* для каждого зарегистрированного *Observer*. Поэтому *DigitalClock* получит и обновит сообщение при любом изменении значения времени. Используется возможность, состоящая в запросе *Clock* по поводу времени и отображении этого значения.

Шаблон *Observer* используется повсеместно. Наблюдателей можно зарегистрировать с помощью объектов всех видов, а не записывать эти объекты исключительно в виде строковых литералов.

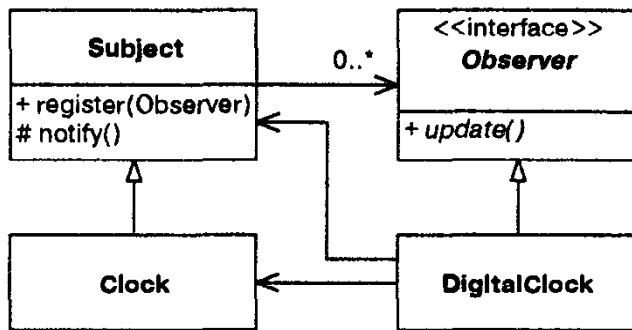


Рис. 24.12. “Каноническая” модель “вытягивания” Observer

чительно с применением вызовов. Это обстоятельство помогает управлять зависимостями, которые “легко переходят в крайности”. Чрезмерное использование шаблона **Observer** приводит к усложнению систем, что затрудняет осмысление их функциональных возможностей и отслеживание их работы.

**“Тяни-толкай”.** Шаблон **Observer** характеризуется двумя основными моделями. На рис. 24.13 показана модель “вытягивания” **Observer**. Свое название она получила на основании того факта, что после получения обновленного сообщения **DigitalClock** “вытягивает” сведения о времени из объекта **Clock**.

Преимуществом модели “вытягивания” является простота реализации, а также тот факт, что классы **Subject** и **Observer** могут быть стандартными, повторно используемыми элементами библиотеки. Но представьте, что запись служащего содержит тысячу полей, а вы только что получили сообщение **update**. В какое из тысяч полей внести это сообщение?

Когда **update** вызывается для **ClockObserver**, ответ очевиден. **ClockObserver** “тянет” значение времени из **Clock** и отображает его. Но если обновление вызывается на **EmployeeObserver**, ответ не столь очевиден. Неизвестно, как поступить. Возможно, изменилось имя служащего, или он получил свою зарплату. Может быть, он станет новым боссом. Или изменится счет в банке. Тут необходим дополнительный ориентир.

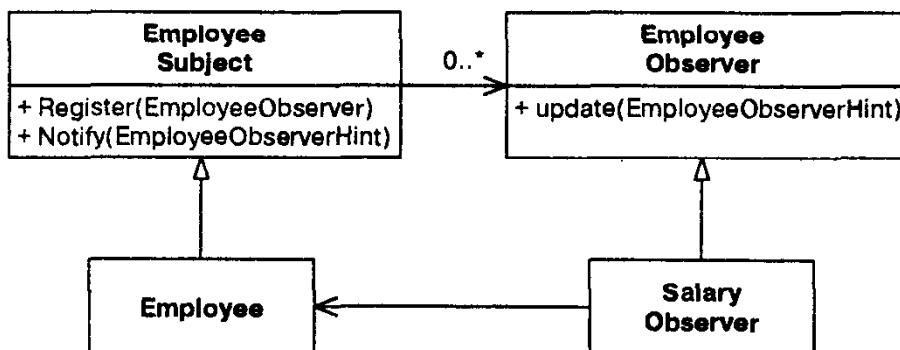


Рис. 24.13. “Толкающая” модель **Observer**

Такую помощь можно получить на основе формы модели “тяни-толкай” шаблона *Observer*. Структура модели “толчка” для наблюдателя показана на рис. 24.13. Обратите внимание, что аргумент получают оба метода: как *notify*, так и *update*. Аргумент передается из *Employee* к *SalaryObserver* с помощью методов *notify* и *update*. Также *SalaryObserver* получает подсказку об изменении записи *Employee*.

Аргумент *EmployeeObserverHint* методов *notify* и *update* может представлять собой определенное численное значение, являться строкой или более сложной структурой данных, включающей старое и новое значение некоторого поля. Какую бы природу не имел этот аргумент, он передается наблюдателю.

Выбор между двумя различными моделями *Observer* полностью зависит от уровня сложности наблюдаемого объекта. Если наблюдаемый объект имеет достаточно сложную природу, а наблюдатель должен располагать дополнительными возможностями, применяется модель “толчка”. Если наблюдаемый объект достаточно прост, удобно обращаться к “тянущей” модели.

## Каким образом шаблон *Observer* учитывает принципы ООП?

Шаблон *Observer* тесно связан с использованием принципа Open-Closed Principle (OCP). Обоснованием для применения шаблона служит тот факт, что новые наблюдаемые объекты можно добавлять без внесения изменений в исходный объект наблюдения. Объект наблюдения остается закрытым.

Возвращаясь к рис. 24.12, можно заключить, что *Clock* заменяется на *Subject*, а *DigitalClock* — на *Observer*. Поэтому соблюдается принцип подстановки Лискоу (Liskov Substitution Principle, LSP).

Класс *Observer* является абстрактным, причем от него зависит конкретный *DigitalClock*. Также от него зависят конкретные методы *Subject*. Следовательно, в этом случае применяется принцип инверсии зависимостей (DIP, Dependency-Inversion Principle). Можно предположить, что поскольку *Subject* не содержит абстрактных классов, зависимость между *Clock* и *Subject* нарушает принцип DIP. Но *Subject* является классом, на основе которого не формируются экземпляры. Он имеет смысл только в контексте производного класса. Поэтому *Subject* является логически абстрактным, даже если он не располагает абстрактными методами. Абстрактность *Subject* можно усилить, придавая ему чисто виртуальный деструктор в C++ или превращая конструкторы в защищенные.

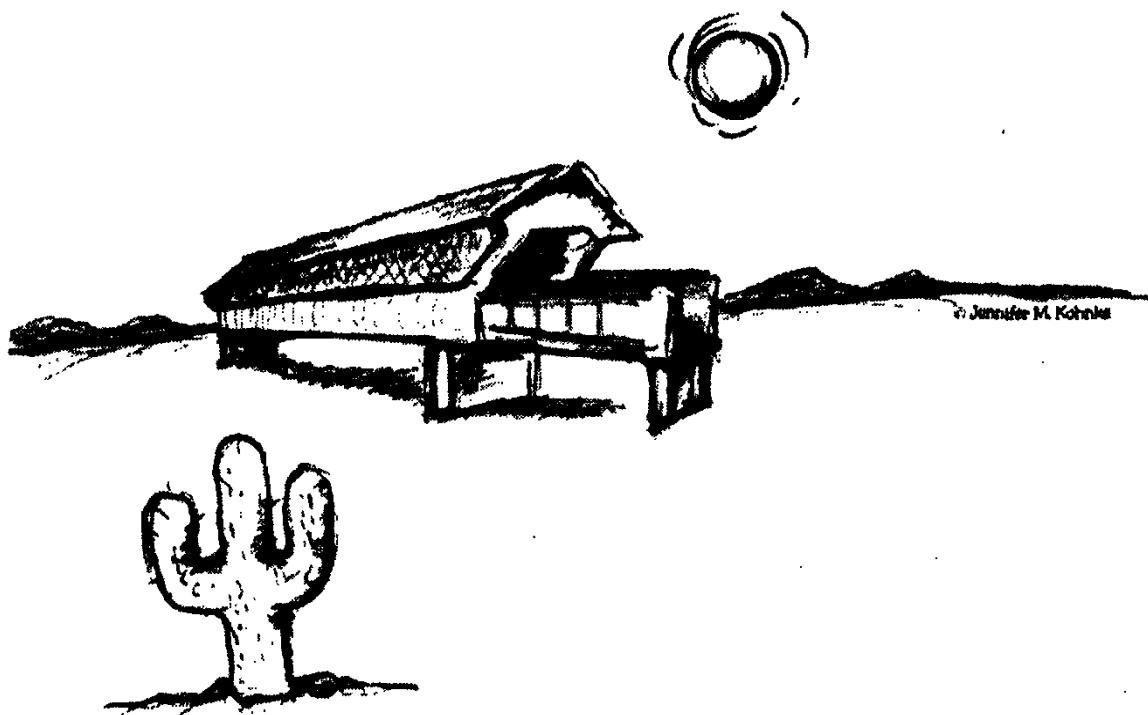
На рис. 24.11 приведены подсказки, связанные с принципом отделения интерфейса (ISP, Interface-Segregation Principle). Классы *Subject* и *TimeSource* производят разделение клиентов *MockTimeSource*, поддерживая для каждого из них специализированные интерфейсы.

## Литература

1. Gamma и др. *Design Patterns*. Addison-Wesley, 1995.
2. Martin C. и др. *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.

# 25

## Некоторые примеры из практики: шаблоны Abstract Server, Adapter и Bridge



Политики все одинаковы. Они обещают построить мост даже там, где нет реки.

---

Никита Хрущев

В середине 90-х годов прошлого века я принимал участие в дискуссиях, развернувшихся в группе новостей `comp.object`. Программисты, отправлявшие сообщения в эту группу новостей, вели оживленные споры о различных стратегиях анализа и проектирования. Мы полагали, что с помощью конкретного примера

можно было бы оценить позицию каждого, кто принимал участие в этих дискуссиях. Итак, мы остановили свой выбор на простейшей проектной проблеме, чтобы представлять свои наиболее приемлемые решения.

Сама проблема была чрезвычайно простой. Мы решили разработать программу, моделирующую работу простой настольной лампы. Настольная лампа состоит из выключателя и лампочки. Можно сделать запрос выключателю о том, был ли он включен, а лампочке можно дать команду включиться или выключиться. Довольно простая проблема, не правда ли?

Дискуссии о методах решения данной проблемы длились несколько месяцев. Каждый из участников дебатов настаивал на превосходстве своего собственного стиля проекта. Некоторые предлагали простой подход, который заключался в наличии только объектов выключателя и лампочки. Другие полагали, что в качестве объекта выступает электричество. Один программист даже предложил ввести объект сетевого шнура.

Несмотря на нелепость некоторых из приведенных аргументов, модель проекта на самом деле представляет собой весьма интересный предмет исследования. Обратите внимание на рис. 25.1. Можно, конечно же, сделать этот проект действующим. Объект `Switch` опрашивает состояние текущего выключателя и отправляет соответствующие сообщения `turnOn` и `turnOff` объекту `Light`.



Рис. 25.1. Простая настольная лампа

Каковы недостатки описанного проекта?

Очевидно, что здесь нарушаются два принципа проектирования: принцип инверсии зависимостей (DIP) и принцип “открытия-закрытия” (OCP). Нарушение принципа инверсии зависимостей легко прослеживается; зависимость от `Switch` до `Light` на самом деле представляет собой зависимость от конкретного класса. В соответствии с принципом инверсии зависимостей, лучше выбирать зависимости из абстрактных классов. Нарушение принципа “открытия-закрытия” имеет немного опосредованный характер, но в то же время его последствия более существенны. Нас не устраивает этот проект в том плане, что приходится перетаскивать объект `Light` туда, где требуется объект `Switch`. Не так просто расширить объект `Switch` таким образом, чтобы управлять объектами, отличными от `Light`.

## Шаблон Abstract Server

Может возникнуть мысль о возможности наследования подкласса из объекта `Switch`, который контролирует объекты, отличные от `Light`, как показано на

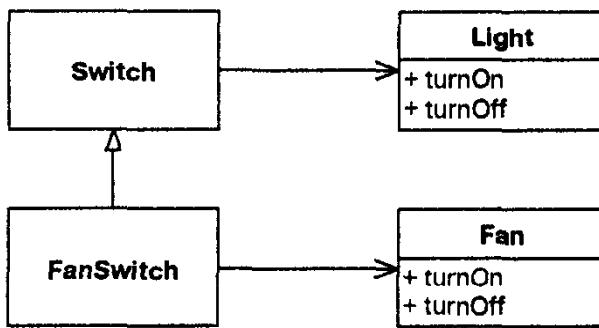


Рис. 25.2. Некорректный способ расширения объекта Switch

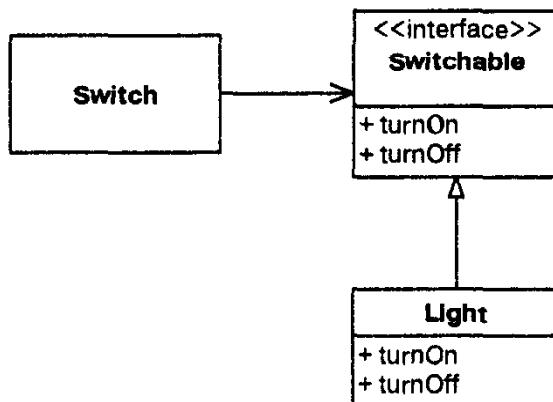


Рис. 25.3. Решение проблемы настольной лампы с помощью Abstract Server

рис. 25.3. Однако проблема этим не решается, поскольку FanSwitch все еще наследует зависимость от объекта Light. При любом использовании FanSwitch понадобится объект Light. В любом случае именно эти отношения зависимостей также приводят к нарушению принципа инверсии зависимостей.

В целях разрешения описанной проблемы активизируется простейший из всех шаблонов проектирования: *Abstract Server*. (рис. 25.2). Путем формирования интерфейса между объектами Switch и Light можно управлять всем, что реализует этот интерфейс. Благодаря этому сразу же достигается соответствие принципу инверсии зависимостей и принципу “открытия-закрытия”.

## Кому принадлежит интерфейс?

Обратите внимание, что интерфейс именуется в соответствии с применяемым клиентом. В данном случае он получил имя *Switchable*, а не *ILight*. Мы уже говорили об этом и, возможно, еще раз обратимся к этому вопросу. Интерфейсы принадлежат клиенту, а не производному классу. Логическая связь между клиентом и интерфейсом сильнее, чем логическая связь между интерфейсом и его производными классами. Она настолько сильна, что нет смысла развертывать *Switchable* без *Light*. Степень логической взаимосвязи превышает степень

физической взаимосвязи. Наследование обеспечивает более сильную физическую взаимосвязь, чем ассоциация.

В начале 90-х годов прошлого века физическая взаимосвязь обычно считалась доминирующей. Написанные в то время авторитетные книги рекомендовали размещать иерархии наследования вместе в одном и том же физическом пакете. Такое решениеказалось оправданным, поскольку наследование представляет собой очень прочную физическую связь. Однако за последнее десятилетие мы узнали, что физическая сила наследования не настолько реальна, как казалось, и что иерархии наследования не должны упаковываться вместе. Вместо этого формируется тенденция упаковки клиентов вместе с контролируемыми ими интерфейсами.

Различная сила, присущая логическим и физическим связям, характерна для языков, обладающих статическими данными типа C++ и Java. Языки с динамическими данными типа Smalltalk, Python и Ruby не приводят к расхождениям, поскольку они не используют наследование в целях достижения полиморфного поведения.

## Шаблон Adapter

На рис. 25.3 представлена небольшая проектная проблема. Эта проблема заключается в потенциальном нарушении принципа персональной ответственности (SRP). Мы связали вместе два объекта `Light` и `Switchable`, которые могут не изменяться в силу одних и тех же причин. Что случится, если мы не сможем добавить отношения зависимостей в `Light`? Что, если бы мы приобрели объект `Light` у стороннего производителя, не располагая при этом исходным кодом? Или что было бы, если бы какой-то другой класс контролировал объект `Switch`, но который невозможно наследовать от объекта `Switchable`? Рассмотрим шаблон `Adapter`<sup>1</sup>.

На рис. 25.4 показаны возможные пути применения шаблона `Adapter` для разрешения проблемы. Причем `Adapter` является производным от `Switchable` и передается объекту `Light`. В результате проблема решается однозначно. Теперь любой объект может включаться или выключаться, контролируясь объектом `Switch`. Все, что необходимо сделать — создать соответствующий адаптер. На самом деле, объекту даже не нужны методы `turnOn` и `turnOff`, принадлежащие объекту `Switchable`. Адаптер может быть приспособлен к интерфейсу объекта.

Не так уж просто разрабатывать адаптеры. Прежде всего необходимо создать новый класс, создать экземпляр адаптера, связав с ним адаптированный объект. Затем при каждой активизации адаптера необходимо “платить дань” за время и пространство, необходимые для делегирования. Очевидно, что нет смысла по-

<sup>1</sup>Шаблон `Adapter` уже рассматривался (рис. 10.2 и 10.3).

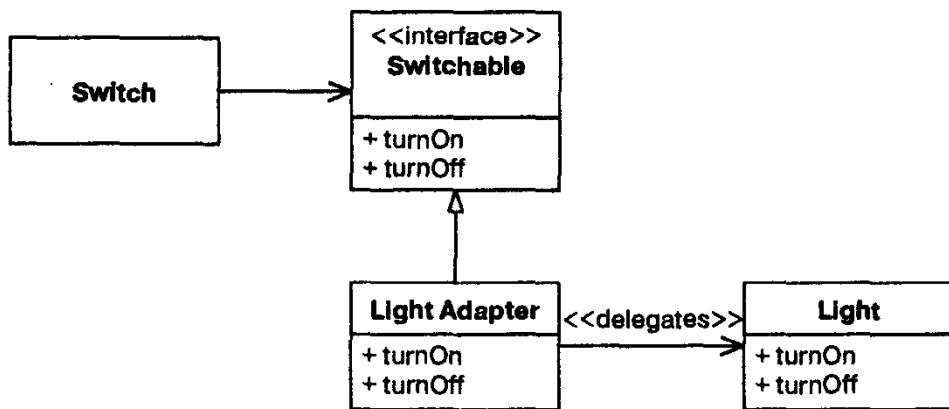


Рис. 25.4. Решение проблемы настольной лампы с помощью шаблона Adapter

стоянно прибегать к помощи адаптеров. Решение в виде *Abstract Server* вполне приемлемо во многих ситуациях. На самом деле, довольно неплохим является даже исходное решение, представленное на рис. 25.1, но только в том случае, если вы *не знаете* о существовании других объектов, контролирующих объект **Switch**.

## Классовая форма шаблона Adapter

Класс **LightAdapter**, представленный на рис. 25.4, известен как *объектная форма адаптера*, представленного на рис. 25.5. В этой форме объект адаптера наследуется как из интерфейса **Switchable**, так и из класса **Light**. Эта форма немного более эффективна, чем объектная форма, а также проще в применении за счет использования высокой степени связывания для наследования.

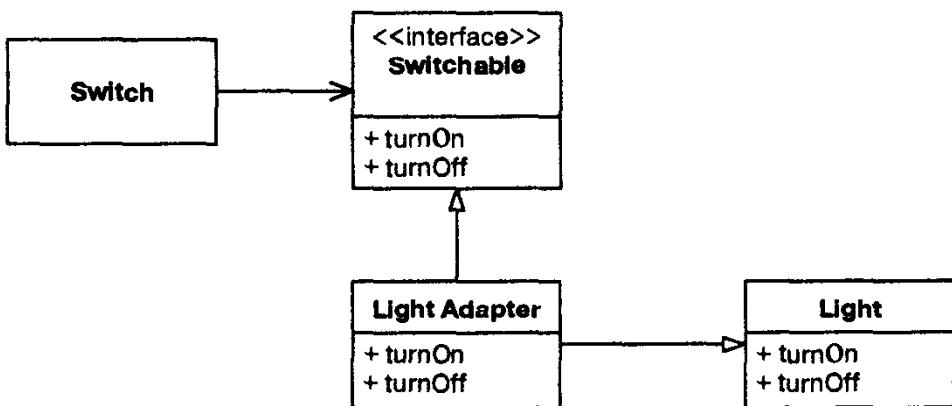


Рис. 25.5. Решение проблемы настольной лампы с помощью шаблона Adapter

## Модемная проблема, шаблон Adapter и принцип LSP

Рассмотрим ситуацию, представленную на рис. 25.6. Мы имеем большое количество клиентов модемов, использующих интерфейс **Modem**. Интерфейс **Modem**

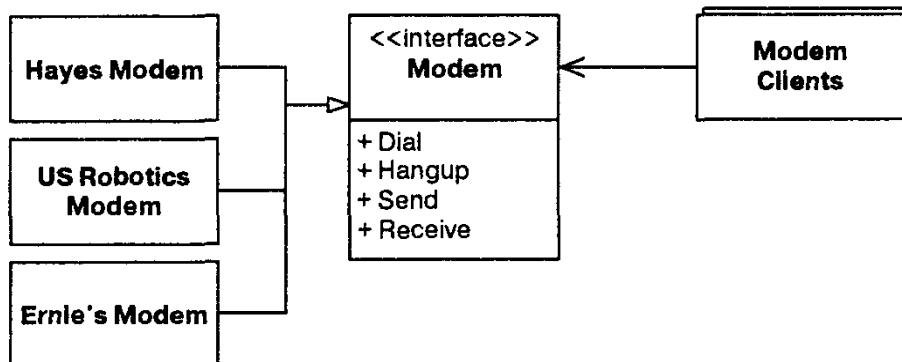


Рис. 25.6. Модемная проблема

применяется несколькими производными классами, включая `HayesModem`, `USRoboticsModem` и `EarniesModem`. Это довольно обычная ситуация. Она неплохо соответствует принципам “открытия-закрытия”, наследования зависимостей и персональной ответственности (OCP, LSP, и DIP). Клиенты модемов, имея дело с новыми видами модемов, не подвергаются их влиянию. Предположим, сотни клиентов модемов успешно использовали интерфейс `Modem`.

Предположим теперь, что заказчики сформулировали новое требование. Существуют модемы без функции набора номера. Они называются выделенными, поскольку расположены на обоих концах выделенной линии<sup>2</sup>. Разработано несколько новых приложений, которые используют выделенные модемы, не поддерживая коммутируемый режим. Будем называть их `DedUsers`. Тем не менее, наши заказчики хотели бы, чтобы все текущие клиенты модемов могли использовать выделенные модемы. Заказчики объясняют, что не хотят изменять сотни приложений клиентских модемов, поэтому будет выделена отдельная команда дозвона по фиктивным телефонным номерам.

Если есть возможность выбора, мы можем воспользоваться ею после завершения проектирования системы, как показано на рис. 25.7. Мы могли бы воспользоваться принципом ISP, чтобы разделить функции набора номера и соединения на два отдельных интерфейса. Устаревший модем будет выполнять оба интерфейса, а клиенты модемов будут их использовать. Класс `DedUsers` будет использовать только интерфейс `Modem`, а `DedicatedModem` — лишь интерфейс `Modem`. К сожалению, в связи с этим нам потребуется внести изменения во все клиенты модемов — именно то, что запретили нам заказчики.

Как бы нам ни хотелось, мы не можем разделить интерфейсы, но можно предложить такой способ, чтобы все клиенты модемов могли пользоваться классом `DedicatedModem`. Одно из возможных решений — вывести `DedicatedModem`

<sup>2</sup>Все модемы могут использоваться в режиме выделенной линии. Этой способности не было у устаревших модемов, т.е. раньше применялись модемы, специально предназначенные для работы с выделенными линиями. Эти модемы могли работать в коммутируемом режиме только в случае подключения отдельного устройства, называемого автокоммутатором.

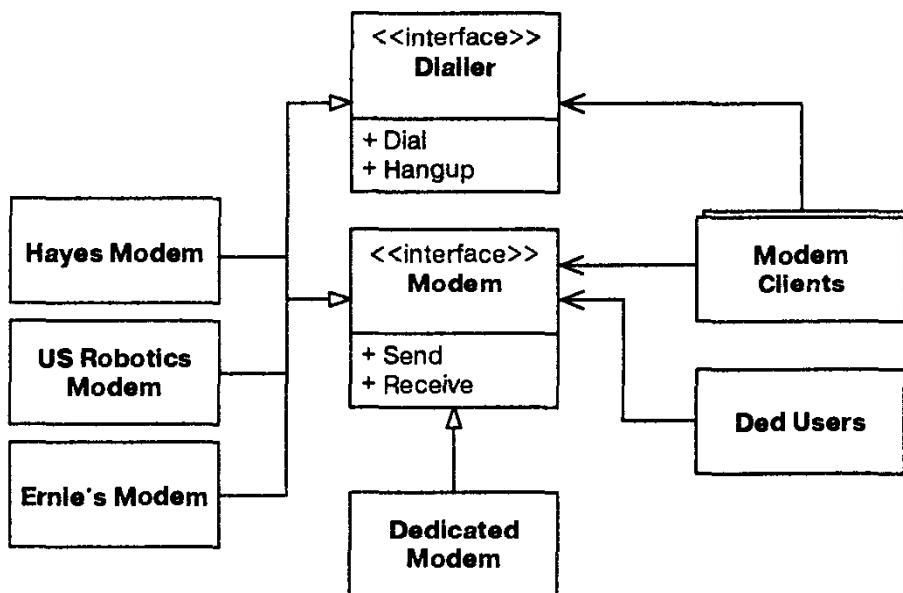


Рис. 25.7. Идеальное решение модемной проблемы

из интерфейса `Modem` и применить функции `dial` и `hangup` для выполнения следующих действий.

```

class DedicatedModem public : Modem
{
public:
 virtual void dial(char phoneNumber[10]) {}
 virtual void hangup() {}
 virtual void send(char c)
 {...}
 virtual char receive()
 {...}
};

```

Дегенерация функций может означать то, что мы нарушаем принцип LSP. Пользователи основного класса могут ожидать, что функции `dial` и `hangup` значительно изменяют состояние модема. Дегенерация применений в `DedicatedModem` может не оправдать эти ожидания.

Допустим, что при создании клиентов модемов предполагалось, что их модемы должны находиться в состоянии ожидания до момента начала набора номера и вернуться в это состояние при вызове функции `hangup`. Другими словами, не предполагается, что модемы будут передавать какие-либо символы, не введенные ранее. Класс `DedicatedModem` приводит к обратным результатам. Он вернет символы до вызова функции `hangup` и будет продолжать возвращать их после вызова функции `hangup`. Таким образом, `Dedicated-Modem` может повредить некоторые из модемных клиентов.

Теперь читатель может справедливо полагать, что вся проблема заключается в клиентах модемов. Они разработаны недостаточно корректно, если повреждаются при непредвиденном вводе данных. Я бы согласился с этим утверждением.

Однако трудно будет убедить программистов, занимающихся поддержкой клиентов модемов, внести изменения в свои программы по причине добавления нового вида модема. Этим нарушается не только принцип “открытия-закрытия” (OCP), но и ставится под вопрос функционирование всей программы. К тому же, наши заказчики запретили вносить изменения в клиенты модемов.

### Решение проблемы: клудж

Можно имитировать состояние соединения в методах `dial` и `hangup` объекта `DedicatedModem`. Можно не возвращать символы, если не вызвана функция `dial` или вызвана функция `hangup`. Прибегнув к такому решению, нам не придется изменять все клиенты модемов. *Все, что от нас требуется, — убедить `DedUsers` вызывать функции `dial` и `hangup`* (рис. 25.8).

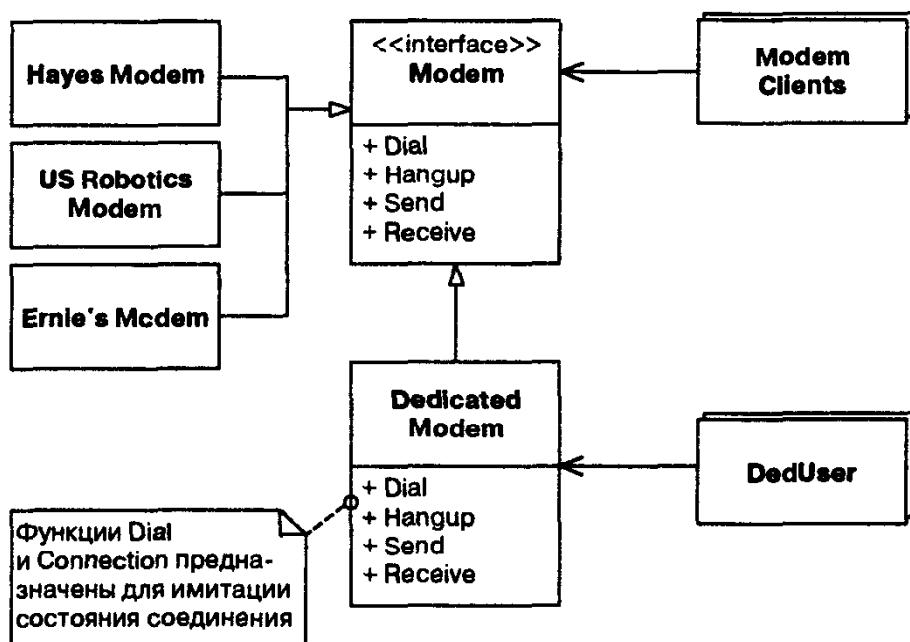


Рис. 25.8. Разрешение модемной проблемы путем включения клуджа `DedicatedModem` для имитации состояния соединения

Представьте себе, что программисты, создающие `DedUsers`, находят такие действия довольно разрушающими. Они явно используют `DedicatedModem`. Зачем же вызывать `dial` и `hangup`? Тем не менее, они пока еще не создали свою программу, поэтому их проще убедить сделать то, чего хотим мы.

### Запутанная сеть зависимостей

Спустя месяцы, когда появляются сотни `DedUsers`, наши заказчики предлагают внести новое изменение. Создается впечатление, что все эти годы нашим программистам не приходилось набирать международные телефонные номера.

Теперь мы отказались от `char[10]` в функции `dial`. Наши заказчики должны получать возможность набирать телефонные номера произвольной длины.

Понятно, что все клиенты модемов должны изменяться. Они создавались с расчетом на использование массива `char [10]` для набора телефонного номера. Заказчики разрешают внести изменения в модемные клиенты. Как известно, классы в модемной иерархии должны измениться с тем, чтобы подстроиться под новый формат телефонного номера. *К сожалению, нам теперь необходимо обратиться к разработчикам DedUsers с просьбой об изменении кода.* Можете только себе представить, как они “обрадуются”! Они не вызывали функцию `dial`, поскольку в этом нет необходимости. Вызов этой функции осуществлялся после нашего указания. А теперь им необходимо проделать дорогостоящую работу по поддержке, поскольку они выполнили наши указания.

Это пример неприятной путаницы зависимостей, с которой сталкиваются многие проекты. Клудж в одной части системы создает неприятную угрозу появления зависимости, которая фактически является причиной возникновения проблем в тех частях системы, которые должны быть полностью несвязанными.

## На помощь придет шаблон Adapter

Можно было бы предотвратить эту неприятную ситуацию, если бы мы воспользовались шаблоном **Adapter** для разрешения исходной проблемы, как показано на рис. 25.9. В этом случае `DedicatedModem` не наследует из интерфейса `Modem`. Клиенты модемов используют `DedicatedModem` опосредованным образом через `DedicatedModemAdapter`. Этот адаптер использует функции `dial` и `hangup` для имитации состояния соединения. Он передает исходящие и входящие звонки `DedicatedModem`.

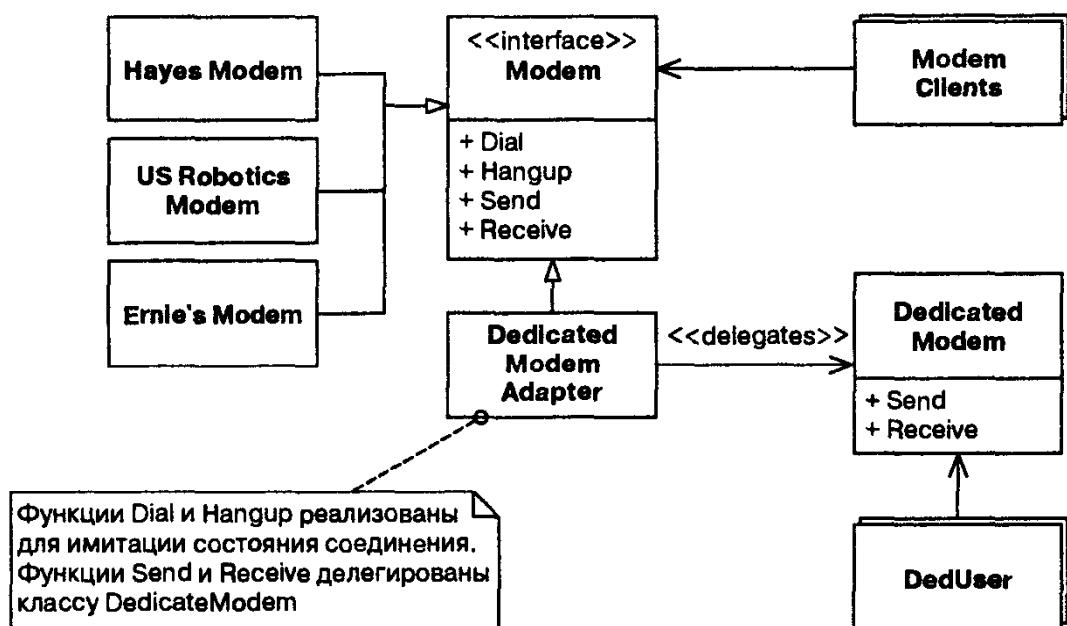


Рис. 25.9. Разрешение модемной проблемы с помощью шаблона Adapter

Обратите внимание, что благодаря такому решению устраняются все имевшиеся ранее трудности. Клиенты объекта `Modem` наблюдают ожидаемый тип поведения, а `DedUsers` не могут работать с функциями `dial` или `hangup`. При изменении требований к набору и формату телефонного номера `DedUsers` не изменяется. Таким образом, благодаря использованию адаптера устранились проблемы с нарушением принципов LSP и OCP.

Обратите внимание, что клудж пока еще существует. Адаптер по прежнему имитирует состояние соединения. Вам это покажется ужасным, и я, конечно же, соглашусь с вами. Тем не менее, обратите внимание, что все зависимости указывают направление, противоположное от адаптера. Клудж изолирован от системы и “скрыт” в адаптере, о котором едва кто-то знает. Устанавливается только жесткая зависимость до тех пор, пока адаптер не реализует здесь некоторую фабрику<sup>3</sup>.

## Шаблон Bridge

Сформулированную проблему можно рассмотреть и с другой точки зрения. Необходимость в выделенном модеме стимулирует добавление новой степени свободы в типовую иерархию `Modem`. Изначально тип `Modem` был задуман как интерфейс для набора различных аппаратных средств.

Известно, что `HayesModem`, `USRModem` и `ErniesModem` являлись производными классами от основного класса `Modem`. Оказывается, что есть еще один способ сокращения иерархии `Modem`. Можно было бы вывести `DialModem` и `DedicatedModem` из объекта `Modem`.

Слияние этих двух независимых иерархий можно выполнить по схеме, представленной на рис. 25.10. Каждый из уровней иерархии типов включает поведение `dialup` (коммутируемая линия) или `dedicated` (выделенная линия) в контролируемое им аппаратное обеспечение. Объект `DedicatedHayesModem` контролирует модем Hayes, применяющийся в выделенном контексте.

Эта структура не является идеальной. Всякий раз, добавляя что-то новое из аппаратного обеспечения, мы должны создать *два* новых класса — один для выделенного случая и один для коммутируемой линии. Каждый раз, добавляя новый тип соединения, следует создавать *три* новых класса — по одному для каждого отдельного аппаратного компонента. Если две степени свободы обеспечивают высшую степень изменчивости, мы можем развернуть большое количество производных классов (до определенного предела, конечно).

Шаблон `Bridge` часто помогает в таких ситуациях, когда типовая иерархия имеет более одной степени свободы. Вместо слияния иерархий можно разделить их, а затем соединить посредством моста.

---

<sup>3</sup>См. гл 21.

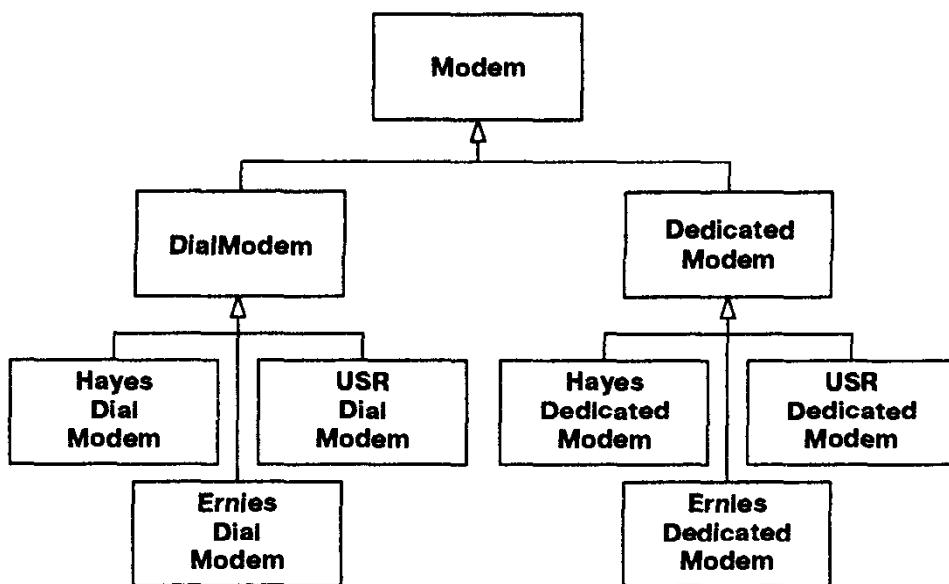


Рис. 25.10. Разрешение модемной проблемы путем слияния иерархии типов

На рис. 25.11 представлена рассматриваемая структура. Иерархия модема разделяется на две подчиненные иерархии. Одна представляет метод соединения, а другая — аппаратное обеспечение.

Пользователи модемов продолжают использовать интерфейс `Modem`. Объект `ModemConnectionController` использует интерфейс `Modem`. Производные классы для `ModemConnectionController` контролируют механизм соединения.

Объект `DialModemController` просто передает методы `dial` и `hangup` объектам основного класса `dialImp` и `hangImp`. Эти методы затем передаются классу `ModemImplementation`, где они разворачиваются в соответствующем контроллере аппаратного обеспечения. `DedModemController` использует методы `dial` и `hangup` для имитации состояния соединения. Он передает методы `send` и `receive` объектам `sendImp` и `receiveImp`, которые, как и раньше, передаются иерархии `ModemImplementation`.

Обратите внимание, что четыре функции `imp` в основном классе `ModemConnectionController` защищены. Это происходит в силу того, что они должны строго использоваться производными `ModemConnectionController`. Никто больше не должен их вызывать.

Описанная структура сложная, но интересная. Ее можно создать, не затрагивая интересы пользователей модемов, и, кроме того, она позволяет полностью отделить политики соединения от применения аппаратных средств. Каждый производный класс объекта `ModemConnectionController` представляет новую политику соединения. Эта политика может использовать методы `sendimp`, `receiveimp`, `dialimp` и `hangimp`. Можно создать новые функции `imp`, не затрагивая при этом интересы пользователей. Принцип ISP может применяться для

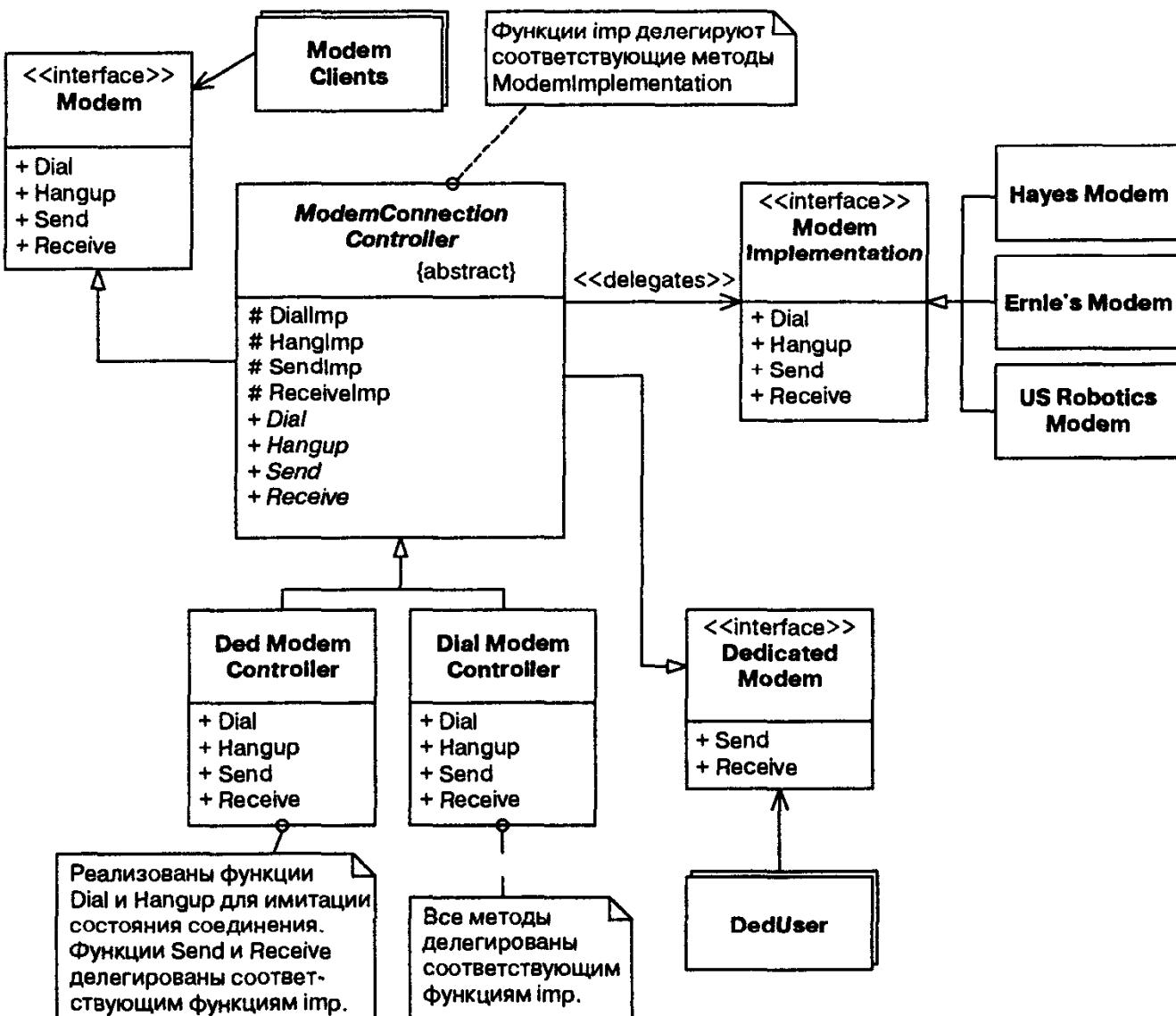


Рис. 25.11. Решение модемной проблемы с помощью шаблона Bridge

добавления новых интерфейсов в классы контроллеров соединений. Это приводит к образованию пути перемещения, по которому клиенты модема могут переходить на уровень API. Этот уровень находится выше того уровня, на котором находятся объекты `dial` и `hangup`.

## Резюме

Может показаться, что настоящая проблема со сценарием `Modem` заключается в некорректной разработке исходного проекта. Разработчики должны были знать, что понятия соединения и коммуникации весьма различны. Проведя небольшой анализ, они обнаружили бы эту проблему и устранили ее. Поэтому вся суть проблемы заключается в недостаточном анализе.

Все это нонсенс! Нет такого понятия, как *достаточный* анализ. Не важно, сколько времени проходит в попытках создать идеальную структуру. Всегда обнаружится, что заказчик вносит изменения, нарушающие исходную структуру.

От этого никуда не денешься. Идеальных структур не существует. Есть только структуры, которые пытаются уравновесить текущие затраты и получаемые доходы. С течением времени эти структуры должны изменяться по мере изменений требований системы. Мастерство управления этим изменением состоит в поддержании системы как можно более простой и гибкой.

Решение в виде шаблона *Adapter* является простым и непосредственным. Благодаря этому решению все зависимости продолжают указывать правильное направление. Решение в виде шаблона *bridge* немного сложнее. Я не буду предлагать вам эту методику до тех пор, пока вы в достаточной степени не убедитесь в необходимости полностью разграничить политики соединения и коммуникации и добавить новые политики соединения.

Вытекающий из этого вывод, как всегда, состоит в том, что применение шаблона влечет за собой определенные затраты, но в то же время обеспечивает определенные преимущества. Следует использовать такие шаблоны, которые наилучшим образом соответствуют рассматриваемой проблеме.

## Литература

1. Gamma и др. *Design Patterns*, Reading, MA: Addison-Wesley, 1995.

# 26

## Шаблоны Proxy и Stairway to Heaven: управление API от независимых производителей



Неужели смех не забыт?

---

Роберт Плант, The Song Remains the Same

В процессе передачи данных из программы в базу данных всегда преодолевается барьер, связанный с последними структурами. При пересылке сообщения с одного компьютера на другой преодолевается сетевой барьер.

Преодоление этих барьеров может представлять значительные трудности. Если не уделить достаточно внимания этому обстоятельству, задача программы будет заключаться не в решении поставленной проблемы, а во “взятии” барьеров. Рассматриваемые в этой главе шаблоны помогают преодолевать подобные барьеры, отвлекающие программу от решения основной задачи.

## Шаблон Proxy

Допустим, что для Web-узла разрабатывается система по поддержке электронной коммерции. В подобной системе определенные объекты предназначены для заказчика, для заказа (принцип “тележки”), а также для товаров, фигурирующих в данном заказе. На рис. 26.1 представлен один из вариантов этой структуры. Эта структура является упрощенной, но вполне пригодна для наших практических целей.

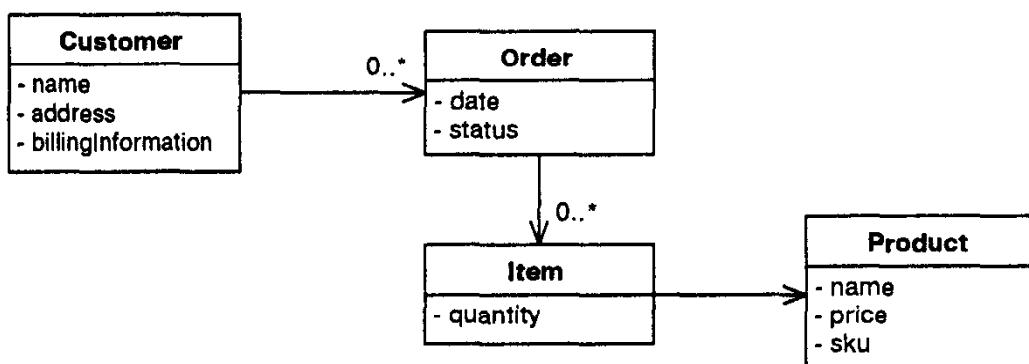


Рис. 26.1. Простая объектная модель “тележки для закупок”

Если в заказ следует добавить новый пункт, можно обратиться к коду, представленному в листинге 26.1. Метод `addItem` класса `Order` просто создает новый `Item` для соответствующего `Product` и заданного количества. Затем `Item` добавляется к `Vector` из `Items`.

---

### Листинг 26.1. Добавление пункта к объектной модели

---

```

public class Order
{
 private Vector itsItems = new Vector();
 public void addItem(Product p, int qty)
 {
 Item item = new Item(p, qty);
 itsItems.add(item);
 }
}

```

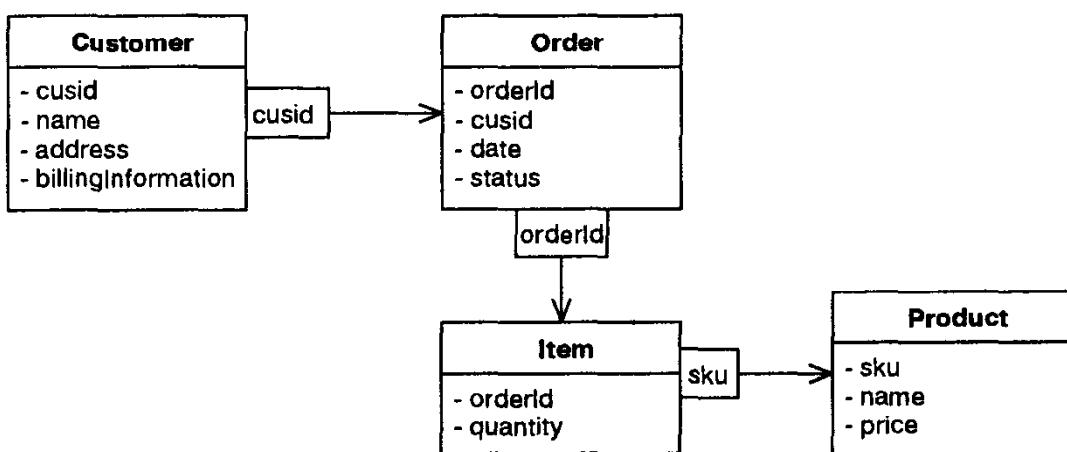


Рис. 26.2. Модель реляционной базы данных “тележки для закупок”

Пусть эти объекты представляют данные, содержащиеся в реляционной базе данных. На рис. 26.2 показаны таблицы и ключи, которые могут представлять эти объекты. Чтобы найти заказы данного клиента, разыскиваются все заказы, для которых задан клиентский `cusid`. Чтобы обнаружить все пункты данного заказа, разыскиваются пункты с указанным `orderId` для заказа. При обнаружении продуктов, указанных в пунктах, используется идентификатор `sku` товара.

Если к определенной строке необходимо добавить строку, состоящую из пунктов, применяется код, подобный показанному в листинге 26.2. Этот код формирует JDBC-вызовы, позволяющие непосредственным образом манипулировать моделью реляционных данных.

---

#### Листинг 26.2. Добавление элемента в реляционную модель

---

```

public class AddItemTransaction extends Transaction
{
 public void addItem(int orderId, String sku, int qty)
 {
 Statement s = itsConnection.createStatement();
 s.executeUpdate("insert into items values(" +
 orderId + "," + sku + "," + qty + ")");
 }
}

```

---

Два приведенных фрагмента кода различны, но выполняют одну логическую функцию. Они связывают пункт с заказом. В первом фрагменте кода игнорируется наличие базы данных, а во втором — сполна используются все доступные при этом преимущества.

Очевидно, что программа “тележки для закупок” основана на обработке заказов, пунктов и продуктов. К сожалению, при использовании кода из листинга 26.2 применяются SQL-операторы, устанавливаются соединения с базами данных и строками запросов. А это является грубым нарушением принципа SRP и, возможно, принципа CCP. В коде из листинга 26.2 совмещаются два понятия,

зависящие от различных причин. Понятие о пунктах и заказах совмещается с понятием о реляционных схемах и SQL. Если какое-либо понятие непредвидено изменяется, это скажется и на другом понятии. Код из листинга 26.2 также нарушает принцип DIP, поскольку политика данной программы зависит от деталей механизма по организации хранения данных.

Шаблон *Proxy* позволяет устраниить эти особенности. Чтобы убедиться в этом, установим тестовую программу, демонстрирующую процесс формирования заказа и подсчета общей его стоимости. Значительная часть кода этой программы приведена в листинге 26.3.

---

**Листинг 26.3. Тестовая программа формирует заказ и проверяет его общую стоимость**

---

```
public void testOrderPrice()
{
 Order o = new Order("Bob");
 Product toothpaste = new Product("Toothpaste", 129);
 o.addItem(toothpaste, 1);
 assertEquals(129, o.total());
 Product mouthwash = new Product("Mouthwash", 342);
 o.addItem(mouthwash, 2);
 assertEquals (813, o.total());
}
```

---

Простой код, проходящий описанный тест, показан в листингах 26.4–26.6. При написании этих программных кодов использовалась простая объектная модель, приведенная на рис. 26.1. Не требуется наличия базы данных. Следует отметить, что данные коды не являются завершенными. Но для тестирования они вполне достаточны.

---

**Листинг 26.4. order.java**

---

```
public class Order
{
 private Vector itsItems = new Vector();

 public Order(String cusid)
 {
 }

 public void addItem(Product p, int qty)
 {
 Item item = new Item(p,qty);
 itsItems.add(item);
 }

 public int total()
```

```
int total = 0;
for (int i = 0; i < itsItems.size (); i++)
{
 Item item = (Item) itsItems.elementAt(i);
 Product p = item.getProduct();
 int qty = item.getQuantity();
 total += p.getPrice() * qty;
}
return total;
}
```

---

#### Листинг 26.5. product.java

```
public class Product
{
 private int itsPrice;

 public Product(String name, int price)
 {
 itsPrice = price;
 }

 public int getPrice()
 {
 return itsPrice;
 }
}
```

---

#### Листинг 26.6. item.java

```
public class Item
{
 private Product itsProduct;
 private int itsQuantity;

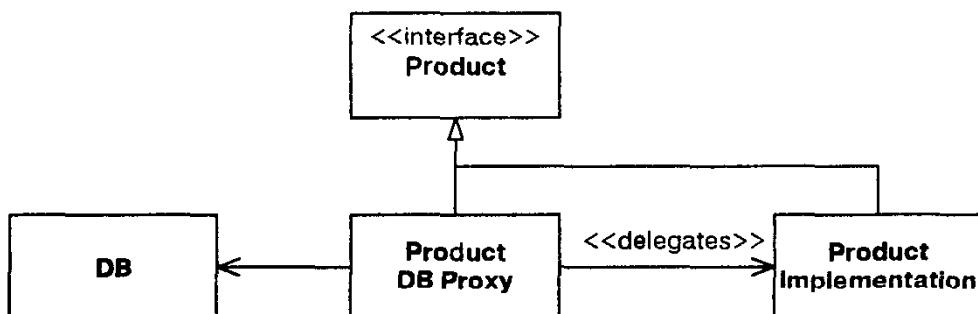
 public Item(Product p, int qty)
 {
 itsProduct = p;
 itsQuantity = qty;
 }

 public Product getProduct()
 {
 return itsProduct;
 }

 public int getQuantity()
 {
```

```
 return itsQuantity;
}
```

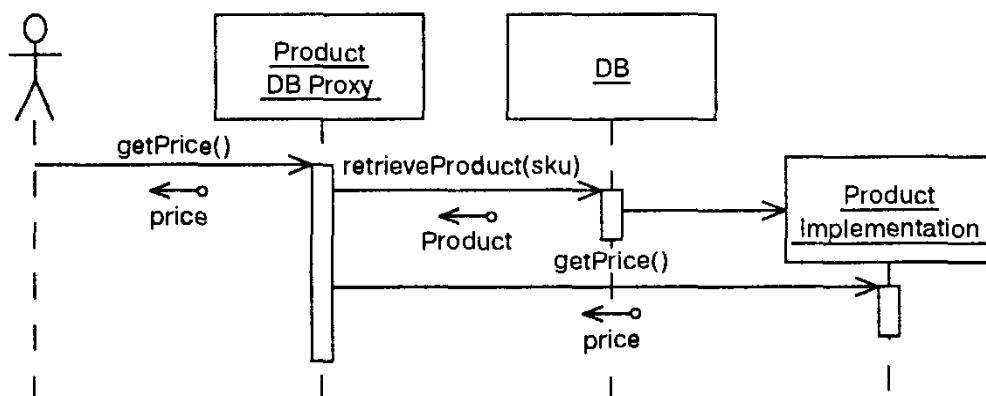
На рис. 26.3 и 26.4 показано, как функционирует шаблон Proxy. Каждый объект, к которому применяется это шаблон, разделяется на три части. Первая часть представляет собой интерфейс, объявляющий все методы, которые в последствии вызываются клиентом. Вторая часть является классом, реализующим эти методы без использования информации о базе данных. Третья часть представляет собой прокси-объект, содержащий сведения о базе данных.



**Рис. 26.3.** Статическая модель Proxy

Рассмотрим класс `Product`. Он превращается в прокси-объект путем замены его интерфейсом. Данный интерфейс включает все методы, которыми располагает `Product`. Класс `ProductImplementation` реализует этот интерфейс почти так же, как и раньше. `ProductDBProxy` реализует все методы `Product`, ответственные за доставку сведений о товаре из базы данных, создает экземпляр `ProductImplementation` и затем делегирует ему сообщение.

Последовательная диаграмма, изложенная на рис. 26.4, демонстрирует, каким образом все функционирует. Клиенты направляют сообщение `getPrice` объекту, который воспринимается ими как `Product`, но в действительности является `ProductDBProxy`. Класс `ProductDBProxy` отвечает за выборку `ProductImplementation` из базы данных. Затем ему делегируется метод `getPrice`.



**Рис. 26.4.** Динамическая модель Ргоху

Но ни клиент, ни `ProductImplementation` не уведомляются о происходящих событиях. База данных вставляется в приложение, но ни одна из сторон об этом не уведомляется. В этом и состоит существенное отличие шаблона `Proxy`. Теоретически, данный шаблон можно включать между двумя объектами, находящимися в состоянии сотрудничества, даже если этим объектам и неизвестно об этом. Этот шаблон можно использовать для преодоления барьера базы данных или сети, причем ни один из участников может и не знать об этом.

На практике применение возможностей прокси-объекта не столь тривиально. Чтобы ознакомиться с возможными затруднениями, попытаемся добавить шаблон `Proxy` к простому приложению “тележка для закупок”.

## Применение шаблона `Proxy` в приложении “тележка для закупок”

Проще всего шаблон `Proxy` создается для класса `Product`. Для удобства таблица товаров представляет простой словарь. Она загружается в то же место, что и перечень всех продуктов. С этой таблицей не производится никаких манипуляций, что приводит к относительно тривиальному использованию шаблона `Proxy`.

Для начала необходимо применить простую утилиту базы данных, которая сохраняет и выбирает данные о продуктах. Прокси-объект использует этот интерфейс для манипулирования с базой данных. Листинг 26.7 представляет тестовую программу, иллюстрирующую рассуждения автора. Тестовый код реализован в листингах 26.8 и 26.9.

---

### Листинг 26.7. DBTest.java

---

```
import junit.framework.*;
import junit.swingui.TestRunner;

public class DBTest extends TestCase
{
 public static void main(String[] args)
 {
 TestRunner.main(new String[]{"DBTest"});
 }

 public DBTest(String name)
 {
 super(name);
 }

 public void setUp() throws Exception
 {
 DB.init();
 }
```

```
public void tearDown() throws Exception
{
 DB.close();
}

public void testStoreProduct() throws Exception
{
 ProductData storedProduct = new ProductData();
 storedProduct.name = "MyProduct";
 storedProduct.price = 1234;
 storedProduct.sku = "999";
 DB.store(storedProduct);
 ProductData retrievedProduct = DB.getProductData("999");
 DB.deleteProductData("999");
 assertEquals(storedProduct, retrievedProduct);
}
}
```

---

**Листинг 26.8. ProductData.java**

---

```
public class ProductData
{
 public String name;
 public int price;
 public String sku;

 public ProductData()
 {
 }

 public ProductData(String name, int price, String sku)
 {
 this.name = name;
 this.price = price;
 this.sku = sku;
 }

 public boolean equals(Object o)
 {
 ProductData pd = (ProductData)o;
 return name.equals(pd.name) &&
 sku.equals(pd.sku) &&
 price==pd.price;
 }
}
```

---

**Листинг 26.9. DB.java**

```
import java.sql.*;

public class DB
{
 private static Connection con;

 public static void init() throws Exception
{
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 con = DriverManager.getConnection(
 "j dbc:odbc:PPP Shopping Cart");
 }

 public static void store(ProductData pd) throws Exception
{
 PreparedStatement s = buildInsertionStatement(pd);
 executeStatement(s);
 }

 private static PreparedStatement
 buildInsertionStatement(ProductData pd) throws SQLException
{
 PreparedStatement s = con.prepareStatement(
 "INSERT into Products VALUES (?, ?, ?)");
 s.setString(1, pd.sku);
 s.setString(2, pd.name);
 s.setInt(3, pd.price);
 return s;
 }

 public static ProductData getProductData(String sku) throws Exception
{
 PreparedStatement s = buildProductQueryStatement(sku);
 ResultSet rs = executeQueryStatement(s);
 ProductData pd = extractProductDataFromResultSet(rs);
 rs.close();
 s.close();
 return pd;
 }

 private static PreparedStatement
 buildProductQueryStatement(String sku) throws SQLException
{
 PreparedStatement s = con.prepareStatement(
 "SELECT * FROM Products WHERE sku = ?");
 s.setString(1, sku);
 return s;
 }
}
```

```
private static ProductData
extractProductDataFromResultSet(ResultSet rs) throws SQLException
{
 ProductData pd = new ProductData();
 pd.sku = rs.getString(1);
 pd.name = rs.getString(2);
 pd.price = rs.getInt(3);
 return pd;
}

public static void deleteProductData(String sku) throws Exception
{
 executeStatement(buildProductDeleteStatement(sku));
}

private static PreparedStatement
buildProductDeleteStatement(String sku) throws SQLException
{
 PreparedStatement s = con.prepareStatement(
 "DELETE from Products where sku = ?");
 s.setString(1, sku);
 return s;
}

private static void executeStatement(PreparedStatement s)
throws SQLException
{
 s.execute();
 s.close();
}

private static ResultSet executeQueryStatement(PreparedStatement s)
throws SQLException
{
 ResultSet rs = s.executeQuery();
 rs.next();
 return rs;
}

public static void closed throws Exception
{
 con.close();
}
```

Следующим шагом в реализации прокси-объекта является запись теста, отображающего принцип его функционирования. В результате проведения этого теста запись о товаре вносится в базу данных. Затем с помощью идентификатора `sku` для сохраняемого продукта создается `ProductProxy` и предпринимаются

попытки применить средства доступа Product для ознакомления с данными, поступающими от прокси-объекта. (листинг 26.10.)

---

#### Листинг 26.10. ProxyTest.java

---

```

import junit.framework.*;
import junit.swingui.TestRunner;

public class ProxyTest extends TestCase
{
 public static void main(String[] args)
 {
 TestRunner.main(new String[]{"ProxyTest"});
 }

 public ProxyTest(String name)
 {
 super(name);
 }

 public void setUp() throws Exception
 {
 DB.init();
 ProductData pd = new ProductData();
 pd.sku = "ProxyTest1";
 pd.name = "ProxyTestName1";
 pd.price = 456;
 DB.store(pd);
 }

 public void tearDown() throws Exception
 {
 DB.deleteProductData("ProxyTest1");
 DB.close();
 }

 public void testProductProxy() throws Exception
 {
 Product p = new ProductProxy("ProxyTest1");
 assertEquals(456, p.getPrice());
 assertEquals("ProxyTestName1", p.getName());
 assertEquals("ProxyTest1", p.getSku());
 }
}

```

---

Чтобы выполнить описанную работу, следует отделить интерфейс Product от его реализаций. Поэтому Product изменяется и принимает вид интерфейса, а для его реализации создается ProductImp (листинги 26.11 и 26.12).

Обратите внимание, что к интерфейсу Product добавлены исключения. Дело в том, что в процессе создания ProductProxy (листинг 26.13) формирова-

лись `Product`, `ProductImp` и `ProxyTest`. При реализации применялось одно средство обеспечения доступа. Как видите, класс `ProductProxy` вызывает базу данных, которая генерирует исключения. Нежелательно, чтобы эти исключения перехватывались и скрывались в процессе функционирования прокси-объекта, поэтому они исключены из интерфейса.

---

**Листинг 26.11. `Product.java`**

---

```
public interface Product
{
 public int getPrice() throws Exception;
 public String getName() throws Exception;
 public String getSKU() throws Exception;
}
```

---

---

**Листинг 26.12. `ProductImp.java`**

---

```
public class ProductImp implements Product
{
 private int itsPrice;
 private String itsName;
 private String itsSKU;

 public ProductImp(String sku, String name, int price)
 {
 itsPrice = price;
 itsName = name;
 itsSKU = sku;
 }

 public int getPrice()
 {
 return itsPrice;
 }

 public String getName()
 {
 return itsName;
 }

 public String getSKU()
 {
 return itsSKU;
 }
}
```

---

**Листинг 26.13. ProductProxy.java**

```
public class ProductProxy implements Product
{
 private String itsSku;
 public ProductProxy(String sku)
 {
 itsSku = sku;
 }

 public int getPrice() throws Exception
 {
 ProductData pd = DB.getProductData(itsSku);
 return pd.price;
 }

 public String getName() throws Exception
 {
 ProductData pd = DB.getProductData(itsSku);
 return pd.name;
 }

 public String getSKU() throws Exception
 {
 return itsSku;
 }
}
```

---

В данном случае реализация прокси-объекта тривиальна. Фактически, нет полного соответствия с “канонической” формой шаблона, показанного на рис. 26.3 и 26.4. А это приводит к неожиданному сюрпризу. Автор намеревался реализовать шаблон *Proxy*. Но на момент окончательной реализации следование “каноническому” шаблону потеряло смысл.

Как показано ниже, данный “канонический” шаблон приводит к тому, что *ProductProxy* с помощью какого-либо метода создает *ProductImp*. Затем этот метод делегируется *ProductImp*.

```
public int getPrice() throws Exception
{
 ProductData pd = DB.getProductData(itsSku);
 ProductImp p = new ProductImp(pd.sku, pd.name, pd.price);
 return p.getPrice();
}
```

Создание *ProductImp* приводит к излишним тратам рабочего времени и вычислительных ресурсов. Класс *ProductProxy* уже располагает данными, которые возвращаются средствами доступа *ProductImp*. Поэтому нет необходимости создавать *ProductImp*, а затем делегировать метод *ProductImp*. Это уже

другой пример нерационального использования кода при работе с шаблонами и предполагаемыми моделями.

Обратите внимание, что метод `getSku` из `ProductProxy` (листинг 26.13) развивает эту схему далее. Он не заботится о выборе базы данных для `sku`. Почему так происходит? Вообще, это задача, решаемая `sku`.

Можно предположить, что реализация `ProductProxy` не является эффективной. В данном случае выбирается база данных для каждого средства доступа. Не предпочтительнее ли кэшировать пункт `ProductData`, что позволит избежать необходимости выбора базы данных?

Это изменение тривиально, но для этого шага нужна определенная смелость. На этом этапе отсутствуют данные, которые могут привести к снижению эффективности при выполнении программы. Кроме того, известно, что механизм базы данных также выполняет определенное кэширование. Непонятно, зачем формировать собственное кэширование. Следует подождать, пока появятся признаки снижения эффективности, а затем приступать к усовершенствованиям.

### Применение прокси-объекта по отношению к взаимосвязям

На следующем шаге создается прокси-объект для объекта `Order`. Каждый экземпляр `Order` включает большое число экземпляров `Item`. В реляционной схеме (рис. 26.2), подобное отношение отражено в таблице `Item`. Каждая строка таблицы `Item` содержит ключ для включающего ее объекта `Order`. Но в объектной модели отношение реализуется с помощью `Vector` в пределах `Order` (листинг 26.4). Иногда прокси-объект выполняет трансляцию между этими двумя формами.

Начнем с формулирования тестового случая, рассматриваемого как пример для прокси-объекта. Этот тест добавляет в базу данных записи о нескольких товарах. Затем для этих товаров используется прокси-объект, а `addItem` вызывается для `OrderProxy`. Наконец, `OrderProxy` запрашивается в целях получения сведений о суммарной стоимости (листинг 26.14). Данный тестовый случай должен показать, что `OrderProxy` ведет себя точно так же, как `Order`, но данные получает из базы данных, а не из содержащихся в памяти объектов.

---

#### Листинг 26.14. `ProxyTest.java`

```
public void testOrderProxyTotal() throws Exception
{
 DB.store(new ProductData("Wheaties", 349, "wheaties"));
 DB.store(new ProductData("Crest", 258, "crest"));
 ProductProxy wheaties = new ProductProxy("wheaties");
 ProductProxy crest = new ProductProxy("crest");
 OrderData od = DB.newOrder("testOrderProxy");
 OrderProxy order = new OrderProxy(od.orderId);
 order.addItem(crest, 1);
 order.addItem(wheaties, 2);
```

```
 assertEquals(956, order.total());
}
```

---

Чтобы данный тестовый случай функционировал, необходимо реализовать несколько новых классов и методов. Во-первых, обратимся к методу `newOrder` из `db`. Этот метод возвращает экземпляр под названием `OrderData`. `OrderData` подобен `ProductData`. Данная простая структура данных представляет строку из таблицы данных `Order`. Она показана в листинге 26.15.

---

#### Листинг 26.15. OrderData.java

---

```
public class OrderData
{
 public String customerId;
 public int orderId;
 public OrderData()
 {
 }

 public OrderData(int orderId, String customerId)
 {
 this.orderId = orderId;
 this.customerId = customerId;
 }
}
```

---

Всегда следует осторожно подходить к использованию общедоступных данных (public data members). Это не объект в истинном понимании этого термина. Речь идет просто о контейнере для данных. Его поведение не представляет интереса, поскольку он нуждается в инкапсуляции. Придание переменным данным частного характера, а также поддержка методов получения и установки данных (getters и setters) приводит к ненужному усложнению программы.

Теперь следует сформировать функцию `newOrder` из DB. Заметим, что при вызове ее в коде из листинга 26.14 поддерживается ID заказчика, но не поддерживается `orderId`. Каждый `Order` требует наличия `orderId` в качестве соответствующего ключа. Более того, в схеме зависимостей каждый `Item` включает ссылку на `orderId`, что позволяет отображать связь с `Order`. Ясно, что `orderId` должен быть уникальным. Каким образом он формируется? Просто напишем соответствующий тест (листинг 26.16).

---

#### Листинг 26.16. DBTest.java

---

```
public void testOrderKeyGeneration() throws Exception
{
 OrderData o1 = DB.newOrder("Bob");
 OrderData o2 = DB.newOrder("Bill");
 int firstOrderId = o1.orderId;
```

```
 int secondOrderId = o2.orderId;
 assertEquals(firstOrderId+1, secondOrderId);
}
```

Этот тест показывает, что orderId в некотором смысле автоматически увеличивается при добавлении нового Order. Это нетрудно реализовать, запрашивая базу данных на предмет выборки максимального orderId, который используется в текущее время, а затем добавляется в эту базу данных (листинг 26.17).

---

#### Листинг 26.17. DB.java

---

```
public static OrderData newOrder(String customerId) throws Exception
{
 int newMaxOrderId = getMaxOrderId() + 1;
 PreparedStatement s = con.prepareStatement(
 "Insert into Orders(orderId,cusid) Values(?,?);");
 s.setInt(1, newMaxOrderId);
 s.setString(2,customerId);
 executeStatement(s);
 return new OrderData(newMaxOrderId, customerId);
}

private static int getMaxOrderId() throws SQLException
{
 Statement qs = con.createStatement();
 ResultSet rs = qs.executeQuery(
 "Select max(orderId) from Orders;");
 rs.next ();
 int maxOrderId = rs.getInt(1);
 rs.close();
 return maxOrderId;
}
```

---

Теперь можно приступить к записи кода OrderProxy. Аналогично тому, как это происходило с товаром, следует разделить Order на интерфейс и реализацию. Тогда Order превратится в интерфейс, а OrderImp станет реализацией (листинги 26.18 и 26.19).

---

#### Листинг 26.18. Order.java

---

```
public interface Order
{
 public String getCustomerId();
 public void addItem(Product p, int quantity);
 public int total();
```

---

**Листинг 26.19. OrderImp.java**

```
import java.util.Vector;

public class OrderImp implements Order
{
 private Vector itsItems = new Vector();
 private String itsCustomerId;

 public String getCustomerId()
 {
 return itsCustomerId;
 }

 public OrderImp(String cusid)
 {
 itsCustomerId = cusid;
 }

 public void addItem(Product p, int qty)
 {
 Item item = new Item(p,qty);
 itsItems.add(item);
 }

 public int total()
 {
 try
 {
 int total = 0;
 for (int i = 0; i < itsItems.size(); i++)
 {
 Item item = (Item) itsItems.elementAt(i);
 Product p = item.getProduct();
 int qty = item.getQuantity();
 total += p.getPrice() * qty;
 }
 return total;
 }
 catch (Exception e)
 {
 throw new Error(e.toString());
 }
 }
}
```

---

Необходимо добавить к OrderImp обработку исключений, поскольку Product перехватывает исключения. Правда, эти исключения добавляют лишнюю работу. Реализации прокси-объектов, находящиеся за интерфейсом, должны да-

вать определенный эффект при обращении к этому интерфейсу, а при использовании прокси-объекта перехватываются исключения, распространяющиеся с помощью этого интерфейса. Итак, принято решение изменить все Exceptions на Errors, тогда интерфейсы не привлекаются к обработке фраз throws, а пользователи этих интерфейсов не обращаются к блокам try/catch.

Как можно реализовать для прокси-объектов addItem? Ясно, что прокси-объект не может делегировать его OrderImp.addItem! Вместо этого прокси-объект пытается включить в базу данных строку Item. С другой стороны, желательно реально делегировать OrderProxy.total к OrderImp.total, поскольку необходимо инкапсулировать в OrderImp определенные бизнес-правила (т.е. политику, определяющую подсчет итогов). При формировании прокси-объектов обычно реализация базы данных отделяется от бизнес-правил.

Чтобы делегировать функцию total, прокси-объект потребуется сгенерировать завершенный объект Order вместе со всеми содержащимися Items. Поэтому в OrderProxy.total следует просмотреть все пункты items из базы данных, вызвать addItem для пустого OrderImp и для каждого обнаруженного item, а затем вызвать total для этого OrderImp. Тогда реализация OrderProxy будет выглядеть так, как показано в листинге 26.20.

---

#### Листинг 26.20. OrderProxy.java

---

```
import java.sql.SQLException;

public class OrderProxy implements Order
{
 private int orderId;

 public OrderProxy(int orderId)
 {
 this.orderId = orderId;
 }

 public int total()
 {
 try
 {
 OrderImp imp = new OrderImp(getCustomerId());
 ItemData[] itemdataArray = DB.getItemsForOrder(orderId);
 for (int i = 0; i < itemdataArray.length; i++)
 {
 ItemData item = itemdataArray[i];
 imp.addItem(new ProductProxy(item.sku), item.qty);
 }
 return imp.total();
 }
 catch (Exception e)
```

```

 throw new Error(e.toString());
 }
}

public String getCustomerId()
{
 try
 {
 OrderData od = DB.getOrderData(orderId);
 return od.customerId;
 }
 catch (SQLException e)
 {
 throw new Error(e.toString());
 }
}

public void addItem(Product p, int quantity)
{
 try
 {
 ItemData id =
 new ItemData(orderId, quantity, p.getSku());
 DB.store(id);
 }
 catch (Exception e)
 {
 throw new Error(e.toString());
 }
}

public int getOrderId()
{
 return orderId;
}
}

```

---

Отсюда следует факт существования класса `ItemData` и нескольких `DB` при манипулировании строками `ItemData`. Эти операции продемонстрированы в листингах 26.21–26.23.

---

#### Листинг 26.21. `ItemData.java`

---

```

public class ItemData
{
 public int orderId;
 public int qty;
 public String sku = "junk";

 public ItemData()

```

```

{
}

public ItemData(int orderId, int qty, String sku)
{
 this.orderId = orderId;
 this.qty = qty;
 this.sku = sku;
}

public boolean equals(Object o)
{
 ItemData id = (ItemData)o;
 return orderId == id.orderId &&
 qty == id.qty &&
 sku.equals(id.sku);
}

```

---

**Листинг 26.22. DBTest.java**

```

public void testStoreItem() throws Exception
{
 ItemData storedItem = new ItemData(1, 3, "sku");
 DB.store(storedItem);
 ItemData[] retrievedItems = DB.getItemsForOrder(1);
 assertEquals(1, retrievedItems.length);
 assertEquals(storedItem, retrievedItems[0]);
}

public void testNoItems() throws Exception
{
 ItemData[] id = DB.getItemsForOrder(42);
 assertEquals(0, id.length);
}

```

---

**Листинг 26.23. DB.java**

```

public static void store(ItemData id) throws Exception
{
 PreparedStatement s = buildItemInsersionStatement(id);
 executeStatement(s);
}

private static PreparedStatement
buildItemInsersionStatement(ItemData id) throws SQLException
{
 PreparedStatement s = con.prepareStatement(
 "Insert into Items(orderId,quantity,sku) " +

```

```
 "VALUES (?, ?, ?);");
s.setInt(1,id.orderId);
s.setInt(2,id.qty);
s.setString(3, id.sku);
return s;
}

public static ItemData[] getItemsForOrder(int orderId)
throws Exception
{
 PreparedStatement s =
 buildItemsForOrderQueryStatement(orderId);
 ResultSet rs = s.executeQuery();
 ItemData[] id = extractItemDataFromResultSet(rs);
 rs.close();
 s.close();
 return id;
}

private static PreparedStatement
buildItemsForOrderQueryStatement(int orderId)
throws SQLException
{
 PreparedStatement s = con.prepareStatement(
 "SELECT * FROM Items WHERE orderid = ?;");
 s.setInt(1, orderId);
 return s;
}

private static ItemData[] extractItemDataFromResultSet(ResultSet rs)
throws SQLException
{
 LinkedList l = new LinkedList();
 for (int row = 0; rs.next(); row++)
 {
 ItemData id = new ItemData ();
 id.orderId = rs.getInt("orderid");
 id.qty = rs.getInt("quantity");
 id.sku = rs.getString("sku");
 l.add(id);
 }
 return (ItemData[]) l.toArray(new ItemData[l.size()]);
}

public static OrderData getOrderData (int orderId)
throws SQLException
{
 PreparedStatement s = con.prepareStatement(
 "Select cusid from orders where orderid = ?;");
 s.setInt(1, orderId);
```

```
ResultSet rs = s.executeQuery();
OrderData od = null;
if (rs.next())
 od = new OrderData(orderId, rs.getString("cusid"));
rs.close ();
s.close();
return od;
}
```

## Резюме: шаблон Proxy

Рассмотренный пример рассеивает иллюзии по поводу удобства и простоты применения прокси-объектов. Применять их не столь легко и просто. Простая модель делегирования, связанная с каноническим шаблоном, редко материализуется в чистом виде. Вместо этого получаем замкнутый круг делегирований для тривиальных методов получения и назначения данных (getters и setters). Для методов, контролирующих взаимосвязи типа 1 : N, обнаруживается наличие *запаздывания* на этапе делегирования и смещения к другим методам, например, делегирование для addItem смещается к total. Наконец, мы имеем дело со спектром кэширования.

В данном примере кэширование не выполнялось. Все описанные тесты реализуются довольно быстро, поэтому не сказываются на эффективности выполняемой программы. Но в реальном приложении обычно возникают вопросы, связанные именно с эффективностью программы и необходимостью выполнения кэширования в разумных объемах. Желательно реализовывать стратегию кэширования в автоматическом режиме, поскольку производительность программы может резко ухудшиться. Естественно, добавление возможности кэширования на раннем этапе приводит к снижению производительности. Если вас беспокоят вопросы, связанные с эффективностью функционирования программы, желательно провести эксперименты, выявляющие причины снижения эффективности. После выявления причины и только после выявления причины снижения эффективности, можно переходить к реализации соответствующих мер.

## Преимущества, связанные с использованием Proxy

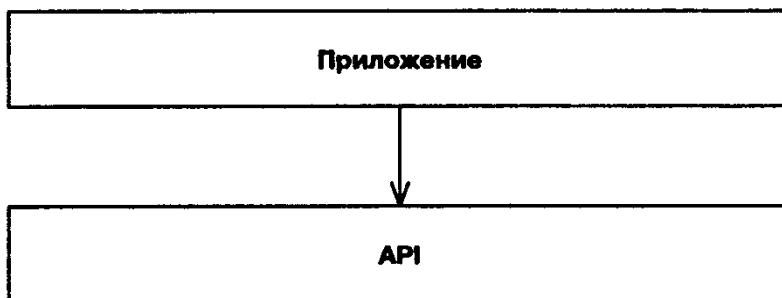
Наряду со многими проблемами, связанными с прокси-объектами, есть и одно существенное преимущество: *разделение ссылок*. В данном примере бизнес-правила и базы данных полностью отделены. Класс OrderImp совершенно не зависит от базы данных. Если необходимо изменить схему базы данных или механизм ее формирования, этого можно достичь, не изменяя Order, OrderImp или какой-либо иной класс деловой предметной области.

В тех экземплярах объектов, где отделение бизнес-правил от реализации базы данных имеет чрезвычайно важное значение, шаблон Proxy сможет весьма успешно применяться. С этой точки зрения, шаблон может использоваться для от-

деления бизнес-правил от любой другой реализации. Этот шаблон может использоваться для предохранения бизнес-правил от влияния со стороны COM, CORBA, EJB и т.д. Таким образом, содержимое бизнес-правил проекта можно отделить от механизмов реализации, которые подвержены быстротечной моде.

## Обработка баз данных, промежуточного ПО и других интерфейсов от сторонних производителей

Интерфейсы API от сторонних производителей пользуются популярностью среди программистов. Повсеместно приобретаются механизмы баз данных, промежуточного ПО, библиотеки классов, потоковые библиотеки и т.д. Первоначально именно к этим API непосредственно формировались обращения из кода приложения (рис. 26.5).



**Рис. 26.5. Исходные взаимосвязи между приложением и API от сторонних производителей**

Со временем, однако, обнаружилось, что код приложения искажается этими API-вызовами. В приложении базы данных, например, можно встретить большое число SQL-строк, беспорядочно распределяющих части кода, которые содержат бизнес-правила.

Также возникают проблемы при внесении изменений в API от сторонних производителей. При изменении схемы появляются проблемы и в случае баз данных. При выпуске новых версий API или схем необходимо для учета изменений перерабатывать все большее количество частей кода приложения.

Постепенно разработчики решили избавить себя от подобных проблем. Для этого был создан слой, отделяющий бизнес-правила приложения от API сторонних производителей (рис. 26.6). В этом слое концентрируется весь код, применяемый API от сторонних производителей, а также все понятия, относящиеся к API, а не к бизнес-правилам приложения.

Подобные слои могут приобретаться отдельно. Примерами могут служить ODBC или JDBC. Они отделяют код приложения от реального механизма базы данных. Конечно, они также являются API сторонних производителей или разрабатываются отдельно; поэтому приложение должно изолироваться даже от них.

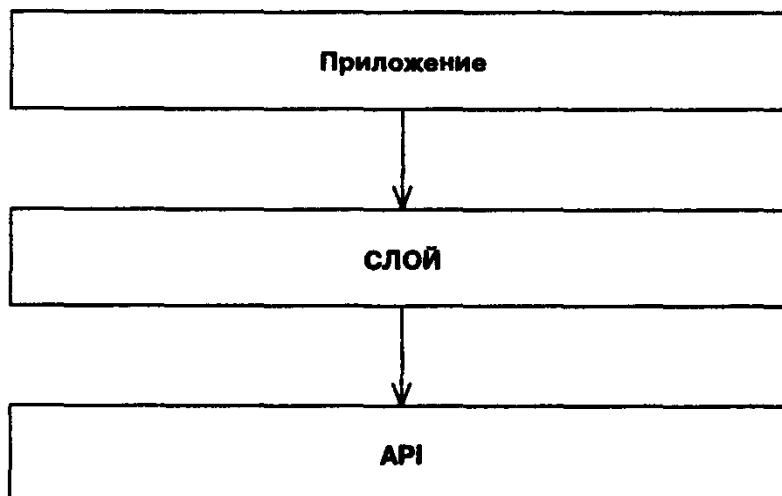


Рис. 26.6. Добавление изолирующего слоя

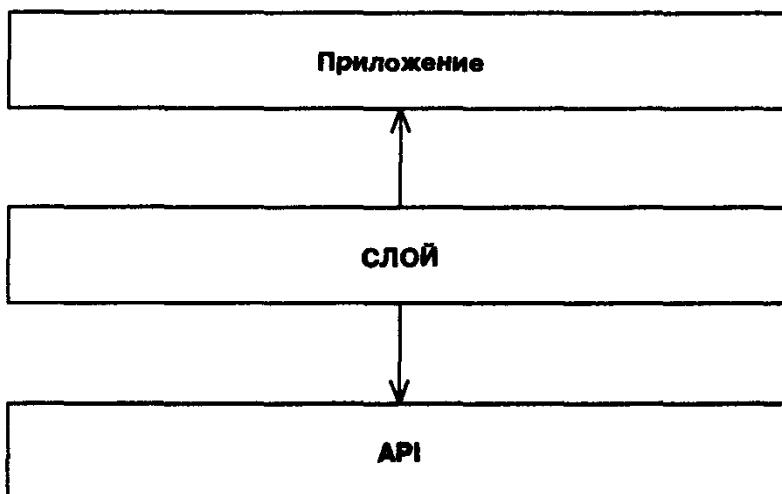


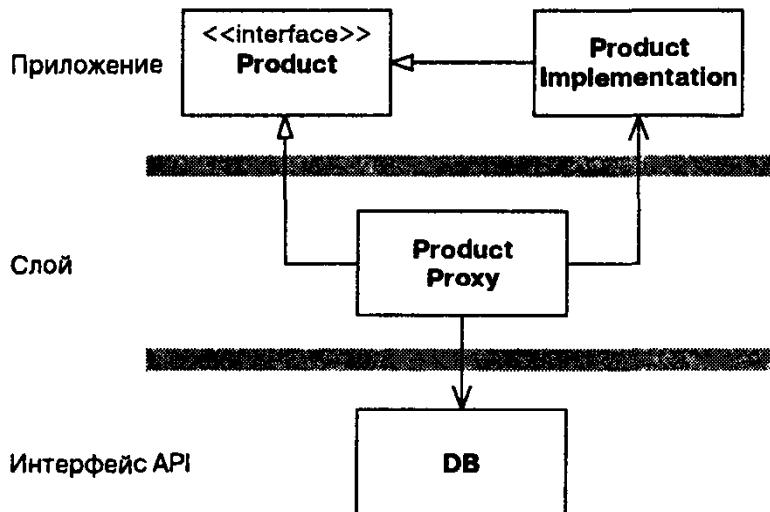
Рис. 26.7. Инвертирование зависимости между приложением и слоем

Заметим, что появляется транзитивная зависимость Application от API. В некоторых приложениях подобная непрямая зависимость также приводит к затруднениям. JDBC, к примеру, не изолирует приложение от деталей реализации схемы.

Чтобы успешно поддерживать изолированность, следует инвертировать зависимость между приложением и слоем (рис. 26.7). Тогда приложение не получает никаких сведений об API от сторонних производителей. В случае с базой данных приложение не содержит прямых указаний на используемую схему.

В случае с механизмом промежуточного ПО (middleware engine) приложение не получает информации о типах данных, применяемых процессором промежуточного ПО (middleware processor).

Подобный набор зависимостей и является заслугой шаблона Proxy. Это приложение не зависит вообще от прокси-объекта. Вместо этого прокси-объекты за-



**Рис. 26.8.** Способ, с помощью которого Proxy инвертирует зависимость между приложением и слоем

висят от приложения и от API. Все сведения о соответствии между приложением и API сосредотачиваются в прокси-объектах.

Подобная насыщенность сведениями делает прокси-объекты поистине могущественными. Когда бы ни изменился API, изменяются и прокси-объекты. Как бы ни изменилось приложение, изменения касаются и прокси-объектов. В этом плане с прокси-объектами достаточно сложно иметь дело.

Хорошо, что изменения, связанные с прокси-объектами, сосредоточены в определенных местах программы, а не разбросаны по всему коду приложения.

Большинство приложений не нуждаются в использовании прокси-объектов, поскольку они представляет собой довольно серьезное решение. Если приходится встречать прокси-решения в деле, обычно советуют устраниить их и применить что-либо попроще. Но иногда интенсивное разделение приложения и API, поддерживаемое с помощью прокси-объектов, приносит много пользы. Обычно подобная ситуация складывается в системах очень большого размера, где используется частотная схема (frequent schema) или “пробуксовка” API (API thrashing). Применение прокси-объектов также целесообразно в системах, находящихся в верхней части большого числа различных механизмов баз данных или механизмов промежуточного ПО.

## Шаблон Stairway to Heaven<sup>1</sup>

Шаблон Stairway to Heaven поддерживает такую же инверсию зависимостей, что и шаблон Proxy. В данном случае реализуется вариация класса форм, описанная с помощью шаблона Adapter (рис. 26.9).

<sup>1</sup>[Martin97]

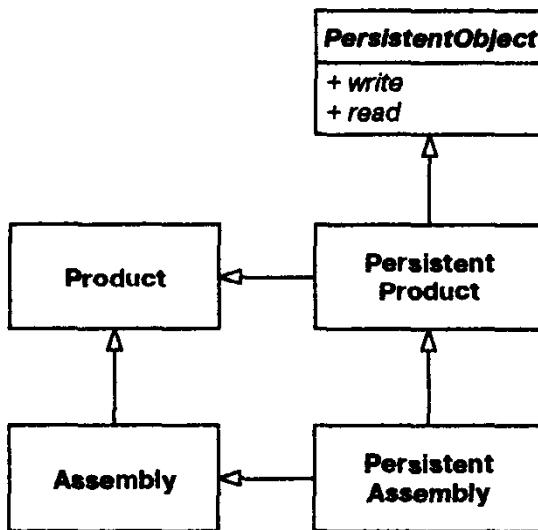


Рис. 26.9. Шаблон Stairway to Heaven

Класс `PersistentObject` является абстрактным и содержит информацию о базе данных. Он поддерживает два абстрактных метода: `read` и `write`. Также поддерживается набор реализованных методов, которые обеспечивают работу инструментов, необходимых для реализации функций `read` и `write`. `PersistentProduct`, например, применяет эти инструменты для реализации методов `read` и `write`, что позволяет просматривать и записывать в базе данных всю информацию, содержащуюся в полях для `Product`. Аналогично, `PersistentAssembly` реализует методы `read` и `write` для выполнения этих операций при работе с дополнительными полями в пределах `Assembly`. Наследуется возможность по просмотру и ведению записей в полях `Product` из `PersistentProduct` и структур методов `read` и `write`.

Данный шаблон удобно применять только при работе с языками, поддерживающими множественное наследование. Обратите внимание, что как `PersistentProduct`, так и `PersistentAssembly` наследуются из двух реализованных базовых классов. Более того, `PersistentAssembly` находится в ромбическом отношении наследования с `Product`. При работе в C++ используется виртуальное наследование, не разрешающее двум экземплярам из `Product` являться наследниками в `PersistentAssembly`.

Необходимость обращения к виртуальному наследованию или другой подобной взаимосвязи в других языках означает, что данный шаблон несколько несовершенен. Это ощущается при обращении к иерархии `Product`, но степень “вторжения” (*intrusion*) шаблона будет минимальной.

Преимуществом этого шаблона является полное отделение сведений о базе данных от бизнес-правил приложения. Небольшие части приложения, которые нуждаются в обращении к методам `read` и `write`, могут воспользоваться следующим приемом:

```
PersistentObject* o = dynamic_cast<PersistentObject*>(product);
if (o)
 o->write();
```

Другими словами, объект приложения запрашивается на предмет того, согласован ли он с интерфейсом `PersistentObject`, и если это так, вызывается метод `read` либо `write`. Часть приложения, которая не нуждается в использовании методов `read` либо `write`, полностью не зависит от наследования со стороны `PersistentObject`.

## Пример использования шаблона Stairway to Heaven

Листинги 26.24–26.34 демонстрируют пример использования шаблона `Stairway to Heaven` при работе в C++. Как обычно, лучше всего начать с тестового случая. Программа `CppUnit`<sup>2</sup> является достаточно объемной, поэтому в листинг 26.24 включены только методы, относящиеся к тестовому случаю. Первый тестовый случай подтверждает, что `PersistentProduct` может передаваться в систему, как и `Product`, и затем конвертироваться в `PersistentObject` и записываться (при необходимости). Определим, чтобы `PersistentProduct` записывался в простом XML-формате. Второй тестовый случай удостоверяет аналогичную ситуацию для `PersistentAssembly`, лишь с тем одним отличием, что реализует добавление данных во второе поле объекта `Assembly`.

---

### Листинг 26.24. `productPersistenceTestCase.cpp` {сокращенный вариант}

---

```
void ProductPersistenceTestCase::testWriteProduct()
{
 ostrstream s;
 Product* p = new PersistentProduct("Cheerios");
 PersistentObject* po = dynamic_cast<PersistentObject*>(p);
 assert(po);
 po->write(s);
 char* writtenString = s.str();
 assert(strcmp("<PRODUCT><NAME>Cheerios</NAME>/PRO DUCT>",
 writtenString) == 0);
}

void ProductPersistenceTestCase::testWriteAssembly()
{
 ostrstream s;
 Assembly* a = new PersistentAssembly("Wheaties", "7734");
 PersistentObject* po = dynamic_cast<PersistentObject*>(a);
 assert(po);
 po->write(s);
```

---

<sup>2</sup>Одно из семейств XUnit для схем тестовых модулей. Обратитесь на Web-узлы [www.junit.org](http://www.junit.org) и [www.xprogramming.com](http://www.xprogramming.com) для получения более подробной информации.

---

```

char* writtenString = s.str();
assert (strcmp ("<ASSEMBLY><NAME>Wheaties"
 "</NAME><ASSYCODE>7734</ASSYCODE></ASSEMBLY>",
 writtenString) == 0);
}

```

---

Далее, в листингах 26.25–26.28 приведены определения и реализации как для **Product**, так и для **Assembly**. В обычном приложении эти классы включают методы, реализуемые с помощью бизнес-правил. Заметим, что в каждом из этих классов нет и намека на неизменность. Отсутствует какое-либо постоянство при формулировании бизнес-правил.

Если характеристики зависимости вполне удовлетворительны, проявляется артефакт, приведенный в листинге 26.27. Он существует исключительно благодаря шаблону **Stairway to Heaven**. Класс **Assembly** наследуется из **Product** с помощью ключевого слова **virtual**. Это необходимо для предотвращения дублирования наследования **Product** от **PersistentAssembly**. Если вернуться к рис. 26.9, можно заметить, что **Product** является вершиной ромба наследования<sup>3</sup>, включающего **Assembly**, **PersistentProduct** и **PersistentObject**. Чтобы предотвратить дублирование наследования **Product**, наследование должно поддерживаться виртуально.

---

#### Листинг 26.25. product.h

---

```

#ifndef STAIRWAYTOHEAVENPRODUCT_H
#define STAIRWAYTOHEAVENPRODUCT_H

#include <string>
class Product
{
public:
 Product(const string& name);
 virtual ~Product();
 const string& getName() const {return itsName;}
private:
 string itsName;
};

#endif

```

---



---

#### Листинг 26.26. product.cpp

---

```

#include "product.h"

Product::Product(const string& name)
 : itsName(name)
{

```

<sup>3</sup>Его иногда шутливо называют “ромбом смерти”.

```

}

Product::~Product()
{
}

```

---

**Листинг 26.27. assembly.h**

```

#ifndef STAIRWAYTOHEAVENASSEMBLY_H
#define STAIRWAYTOHEAVENASSEMBLY_H

#include <string>
#include "product.h"

class Assembly : public virtual Product
{
public:
 Assembly(const string& name, const string& assyCode);
 virtual ~Assembly();
 const string& getAssyCode() const {return itsAssyCode;}
private:
 string itsAssyCode;
};

#endif

```

---

**Листинг 26.28. assembly.cpp**

```

#include "assembly.h"

Assembly::Assembly(const string& name, const string& assyCode)
 : Product(name), itsAssyCode(assyCode)
{
}

Assembly::~Assembly()
{
}

```

---

Листинги 26.29 и 26.30 показывают определение и реализацию `PersistentObject`. Заметим, что поскольку `PersistentObject` не имеет сведений об иерархии `Product`, необходимо получить информацию о порядке записи XML-кода. По крайней мере, следует знать, что объекты записываются начиная с заголовка, затем следуют поля и промежуточные итоги.

Метод `write` из `PersistentObject` использует шаблон `Template method`<sup>4</sup>, контролирующий запись всех производных модулей. Поэтому в неиз-

<sup>4</sup>За дополнительными сведениями обращайтесь к главе 14.

менной части шаблона *Stairway to Heaven* используются возможности базового класса *PersistentObject*.

---

#### Листинг 26.29. persistentObject.h

---

```
#ifndef STAIRWAYTOHEAVENPERSISTENTOBJECT_H
#define STAIRWAYTOHEAVENPERSISTENTOBJECT_H

#include <iostream>

class PersistentObject
{
public:
 virtual ~PersistentObject () ;
 virtual void write(ostream&) const;

protected:
 virtual void writeFields(ostream&) const = 0;

private:
 virtual void writeHeader(ostream&) const = 0;
 virtual void writeFooter(ostream&) const = 0;
};

#endif
```

---

---

#### Листинг 26.30. persistentObject.cpp

---

```
#include "persistentObject.h"

PersistentObject::~PersistentObject()
{
}

void PersistentObject::write(ostream& s) const
{
 writeHeader(s);
 writeFields(s);
 writeFooter(s);
 s << ends;
}
```

---

Листинги 26.31 и 26.32 демонстрируют реализацию *PersistentProduct*. Этот класс в целях создания соответствующего XML-кода для *Product* реализует функции *writeHeader*, *writeFooter* и *writeField*. Он наследует поля и средства доступа из *Product* и управляется с помощью метода *write* из базового класса *PersistentObject*.

---

**Листинг 26.31. persistentProduct.h**

---

```
#ifndef STAIRWAYTOHEAVENPERSISTENTPRODUCT_H
#define STAIRWAYTOHEAVENPERSISTENTPRODUCT_H

#include "product.h"
#include "persistentObject.h"

class PersistentProduct : public virtual Product
 , public PersistentObject
{
public:
 PersistentProduct(const string& name);
 virtual ~PersistentProduct();

protected:
 virtual void writeFields(ostream& s) const;

private:
 virtual void writeHeader(ostream& s) const;
 virtual void writeFooter(ostream& s) const;
};

#endif
```

---

---

**Листинг 26.32. persistentProduct.cpp**

---

```
#include "persistentProduct.h"

PersistentProduct::PersistentProduct(const string& name)
:Product(name)
{
}

PersistentProduct::~PersistentProduct()
{
}

void PersistentProduct::writeHeader(ostream& s) const
{
 s << "<PRODUCT>";
}

void PersistentProduct::writeFooter(ostream& s) const
{
 s << "</PRODUCT>";
}

void PersistentProduct::writeFields(ostream& s) const
```

---

```
s << "<NAME>" << getName() << "</NAME>";
}
```

---

Наконец, листинги 26.33 и 26.34 демонстрируют, каким образом `PersistentAssembly` унифицирует `Assembly` и `PersistentProduct`. Подобно тому, как это происходит с `PersistentProduct`, в данном случае перекрываются `writeHeader`, `writeFooter` и `writeFields`. Но для вызова `PersistentProduct:::writeFields` реализуется `writeFields`. Поэтому наследуется возможность записи из `PersistentProduct` части `Product` из `Assembly`, а также из `Assembly` наследуется `Product`, поля `Assembly` и средства доступа.

---

#### Листинг 26.33. persistentAssembly.h

---

```
#ifndef STAIRWAYTOHEAVENPERSISTENTASSEMBLY_H
#define STAIRWAYTOHEAVENPERSISTENTASSEMBLY_H

#include "assembly.h"
#include "persistentProduct.h"

class PersistentAssembly : public Assembly, public PersistentProduct
{
public:
 PersistentAssembly(const string& name,
 const string& assyCode);
 virtual ~PersistentAssembly();

protected:
 virtual void writeFields(ostream& s) const;

private:
 virtual void writeHeader(ostream& s) const;
 virtual void writeFooter(ostream& s) const;
};

#endif
```

---



---

#### Листинг 26.34. persistentAssembly.cpp

---

```
#include "PersistentAssembly.h"

PersistentAssembly::PersistentAssembly(const string& name,
 const string& assyCode)
: Assembly(name, assyCode)
, PersistentProduct(name)
, Product(name)
{

}

PersistentAssembly::~PersistentAssembly()
```

```

{
}

void PersistentAssembly::writeHeader(ostream& s) const
{
 s << "<ASSEMBLY>";
}

void PersistentAssembly::writeFooter(ostream& s) const
{
 s << "</ASSEMBLY>";
}

void PersistentAssembly::writeFields(ostream& s) const
{
 PersistentProduct::writeFields(s);
 s << "<ASSYCODE>" << getAssyCode() << "</ASSYCODE>";
}

```

---

## **Краткое резюме**

Известно, что шаблон *Stairway to Heaven* используется во многих сценариях. Этот шаблон довольно легко установить. Он оказывает минимальное влияние на объекты, включающие бизнес-правила. С другой стороны, нужно использовать язык программирования типа C++, если при реализации поддерживается множественное наследование.

## **Другие шаблоны, применяемые при работе с базами данных**

### **Шаблон Extension Object**

Вообразите себе шаблон *Extension Object*<sup>5</sup>, которому “известно”, как записать в базу данных расширенный объект. Чтобы записать этот объект, можно направить запрос о выборке расширенного объекта, соответствующего ключу “базы данных”, поместить его в *DatabaseWriterExtension*, а затем вызвать метод *write*.

```

Product p = /* некоторая функция, возвращающая Product */
ExtensionObject e = p.getExtension("Database");
if (e != null)
{
 DatabaseWriterExtension dwe = (DatabaseWriterExtension) e;
 e.write();
}

```

<sup>5</sup>Дополнительные сведения по этой теме приведены в главе 28.

## Шаблон Visitor<sup>6</sup>

Представим иерархию шаблона **Visitor**, которой “известно”, как записать в базу данных посетивший ее объект. Объект записывается в базу данных путем создания шаблона **Visitor** соответствующего типа, а затем для созданного объекта вызывается метод `accept`.

```
Product p = /* некоторая функция, возвращающая Product */
DatabaseWriterVisitor dwv = new DatabaseWriterVisitor();
p.accept (dwv);
```

## Шаблон Decorator<sup>7</sup>

Шаблон **Decorator** используется при обработке базы данных двумя способами. Можно “выполнить отделку” бизнес-объекта и передать ему методы `read` и `write`; или же можно “украсить” объект данных, которому “известно”, каким образом просматривать и записывать содержимое, и передать ему бизнес-правила. При работе с объектно-ориентированными базами данных последний подход используется редко. Бизнес-правила хранятся вне OODB-схемы и вносятся в нее с помощью шаблонов **Decorator**.

## Шаблон Facade

Автор предпочитает использовать этот шаблон в качестве отправной точки. Он прост и эффективен. С другой стороны, он соединяет объекты бизнес-правил и базой данных. На рис. 26.10 показана схема этого шаблона. Класс **DatabaseFacade** просто поддерживает методы, служащие для просмотра и записи всех необходимых объектов. В данном случае объекты соединяются с **DatabaseFacade** и наоборот. Эти объекты располагают сведениями о шаблоне **Facade**, поскольку довольно часто именно они вызывают методы `read` и `write`. Шаблон **Facade** получает информацию об этих объектах, поскольку для реализации методов `read` и `write` он должен применять средства доступа и мутаторы.

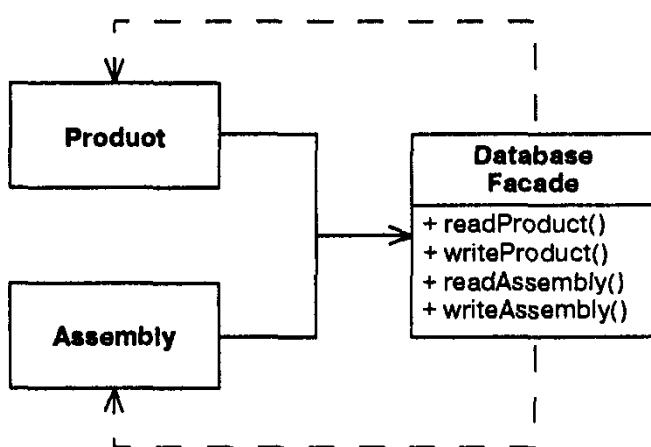


Рис. 26.10. Класс DatabaseFacade

<sup>6</sup>Дополнительные сведения представлены в главе 28.

<sup>7</sup>За дополнительной информацией обращайтесь к главе 28.

Наблюдаемое в этом случае соединение может привести к появлению множества проблем в более крупных приложениях; но при работе с небольшими приложениями или же с теми, которые только начинают разрастаться, данная методика довольно эффективна. Если вы только приступаете к использованию фасада, а позже захотите внести изменения в какой-либо шаблон для ослабления связности, шаблон *Facade* может легко рефакторизоваться.

## Резюме

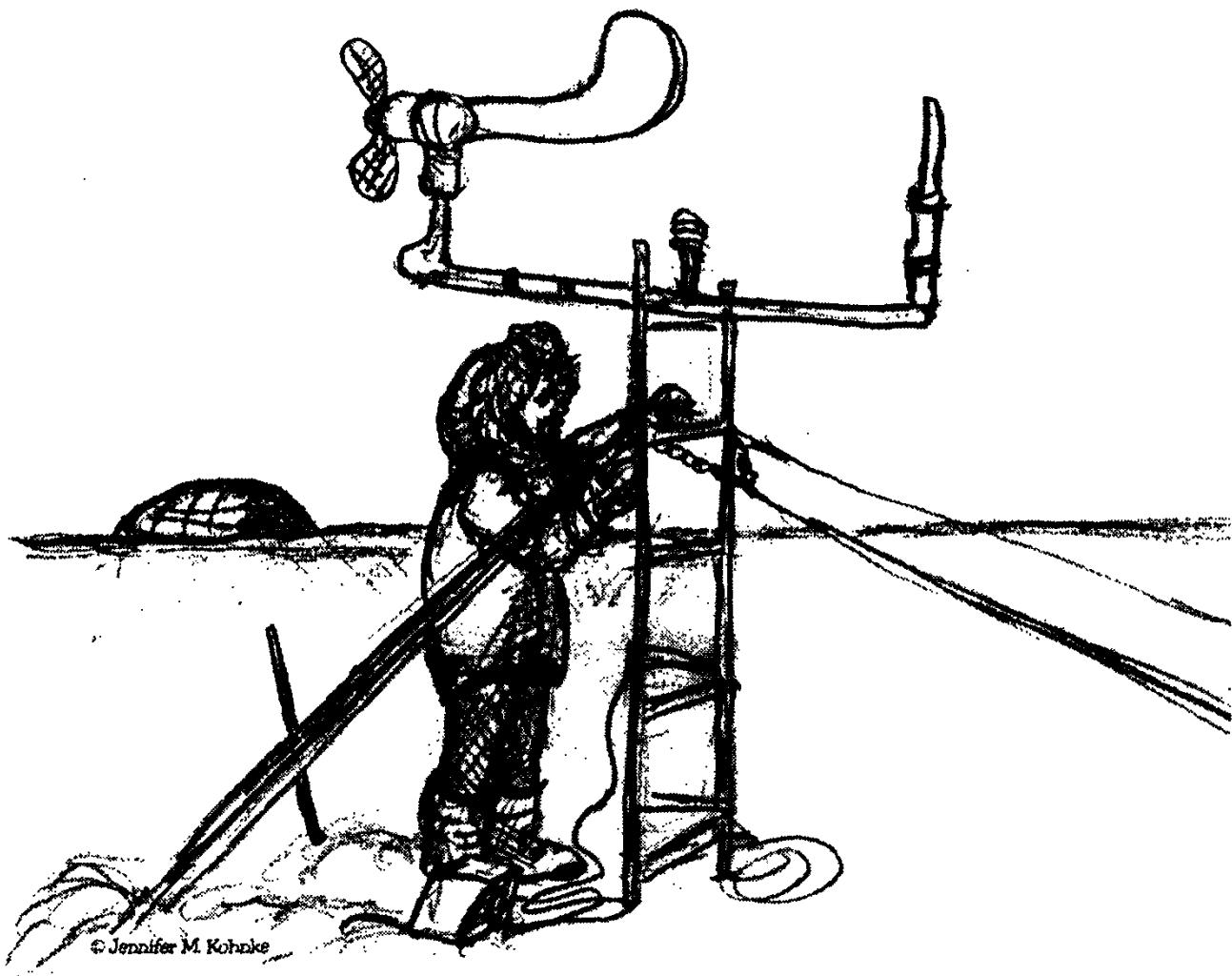
Обычно программисты с волнением ожидают, когда же возникнет необходимость в применении шаблонов *Proxy* или *Stairway to Heaven*. Но почти всегда этот процесс приводит к разочарованию, особенно если речь идет о шаблоне *Proxy*. Автор рекомендует начинать с использования шаблона *Facade*, а затем по мере необходимости применять рефакторинг. Таким образом вы сэкономите время и избежите дополнительных трудностей.

## Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
2. Martin R. C. Design Patterns for Dealing with Dual Inheritance Hierarchies. *C++ Report* (апрель): 1997.

# 27

## Практическое занятие: метеостанция



Глава написана совместно с Джимом Ньюкирком (Jim Newkirk).  
Следующая история вымыщена, но содержит много реальных подробностей

## Фирма Cloud Company

Фирма Cloud Company является лидером в производстве промышленных систем мониторинга погодных условий (WMS, Weather monitoring systems). Примером передового продукта этого производства служит система WMS, представляющая данные о температуре, влажности, атмосферном давлении, скорости, направлении ветра и т.д. Система отображает эти данные на дисплее в режиме реального времени. Поддерживается также хронология в почасовом и ежедневном режиме. По запросу пользователя данные выводятся на дисплей.

Основными заказчиками фирмы Cloud Company являются авиационные и морские службы, отрасли сельского хозяйства, телевидение и радиовещание. Для этих подразделений использование систем WMS имеет исключительное значение. Фирма Cloud Company хорошо зарекомендовала себя в качестве производителя высоконадежных программ, которые можно устанавливать в средах с низким уровнем контроля. Но подобные системы являются достаточно дорогостоящими.

Именно высокая стоимость программ от Cloud Company привела к сужению круга заказчиков подобных высоконадежных систем. Менеджеры из фирмы Cloud Company надеются расширить рынок продаж за счет этих клиентов.

### Сформулированная задача

Конкурирующая компания, Microburst, Inc., недавно объявила о создании линии программ, уровень надежности которых постоянно возрастает. Для компании Cloud Company возникла угроза потери определенного сегмента рынка. В данном случае идет речь о заказчиках, которые сначала обращаются к программным продуктам компании Microburst, а в дальнейшем планируют перейти к использованию программ, производимых фирмой Cloud Company.

Здесь настораживает то обстоятельство, что программные продукты компании Microburst могут составить конкуренцию программам компании Cloud Company и на более высоком уровне. То есть высокоуровневые обновления могут со временем превратиться в систему мониторинга, выявляющую изменения погодных условий на обширных территориях. Текущая база заказчиков компании Cloud Company при этом может серьезно сократиться.

### Стратегия

Несмотря на то, что компания Microburst успешно продемонстрировала подготовленные для распространения программные модули по выполнению конкретных задач, в течение последних шести месяцев качественная программная продукция в продажу не поступала. Это свидетельствует о том, что компания Microburst столкнулась с определенными инженерными или производственными проблемами. Более того, в настоящее время недоступны высоконадежные модификации программы, которые компания Microburst обещала выпустить в рамках разрекламированной линии программных продуктов. Складывается впечатление,

что компания Microburst преждевременно анонсировала выпуск рассматриваемой программы.

Если компания Cloud Company анносирует выпуск обновляемого низкоуровневого и подключаемого программного продукта, который будет предложен заказчикам *в течение* шести месяцев, можно будет сохранить лидирующее положение на рынке программных продуктов. В противном случае заказчики будут приобретать программные продукты компании Microburst. Упрочив свое положение на рынке, компания Cloud Company не позволит конкурирующей с ней компании Microburst опередить ее в разрешении инженерных и производственных проблем.

## Дilemma

Разработка новой обширной линии программных пакетов, распространяемых по относительно невысокой цене, потребует существенных затрат на инженерные изыскания. Инженеры, ответственные за разработку аппаратного обеспечения, категорически не согласны с тем, что на выполнение работ выделено только шесть месяцев. Они надеются, что срок разработки предназначенных для продажи программных модулей будет продлен до года.

Менеджеры, управляющие продажами на рынке, утверждают, что за год компания Microburst сможет создать конкурентоспособную программную продукцию, что приведет к безвозвратной потере заказов компанией Cloud Company.

## План

Менеджеры из фирмы Cloud Company пришли к выводу о том, что следует немедленно анонсировать новую линию программных продуктов, чтобы сохранить заказы, которые неминуемо будут потеряны по прошествии шести месяцев. Они назвали новую программу Nimbus-LC 1.0. Предполагается заново упаковать уже известное, недешевое и высоконадежное аппаратное обеспечение в новый корпус, оборудованный удобной для обзора жидкокристаллической панелью. Вследствие достаточно высоких затрат в связи с разработкой этих модулей, компания понесет определенные убытки на этапе продаж.

Инженеры, ответственные за разработку аппаратного обеспечения, приступят к созданию аппаратных средств, стоимость которых относительно невелика. Процесс разработки займет 12 месяцев. Новая конфигурация программного продукта получит название Nimbus-LC 2.0. В процессе наращивания производственных мощностей, направленных на выпуск новой продукции, Nimbus-LC 1.0 будет постепенно сворачиваться.

Если заказчики Nimbus-LC 1.0 пожелают обновить ПО с учетом более высокого уровня сервиса, приобретенные модули будут заменены модулями версии Nimbus-LC 2.0 без дополнительной оплаты. Компания понесет потери относительно доходности этого продукта в течение шести месяцев, но сможет удержать или стабилизировать отток заказчиков.

## Программа WMS-LC

Программный проект Nimbus-LC весьма сложен. Разработчикам предстоит создать программу, которая использует как существующее аппаратное обеспечение, так и дешевую аппаратную часть версии 2.0, имеющую низкую стоимость. Прототипные модули версии 2.0 вряд ли появятся в течение ближайших девяти месяцев. Более того, процессор, установленный на плате версии 2.0, не совместим с процессором, поддерживаемым в версии 1.0. Несмотря на это система должна функционировать аналогичным образом на двух аппаратных платформах.

Инженеры-электронщики разрабатывают низкоуровневые аппаратные драйверы, а создание API, необходимых для поддержки этих драйверов, — задача инженеров-программистов. Разработанные API будут доступны инженерам-электронщикам в течение следующих четырех месяцев. В соответствии с планом, подготовка ПО к стадии массового копирования займет 6 месяцев, а в течение 12 месяцев оно будет функционировать совместно с аппаратным обеспечением версии 2.0. По крайней мере, 6 недель уйдет на тестирование качества для устройств версии 1.0, поэтому инженеры-программисты располагают лишь 20 неделями, в течение которых должен быть получен работоспособный программный продукт. Тестирование качества для версии 2 займет от 8 до 10 недель (в силу новизны этой аппаратной платформы). При этом следует учитывать тот момент, что это время составляет большую часть трехмесячного периода между созданием первого прототипа и заключительной версией. В силу этого в распоряжении инженеров-программистов остается достаточно ограниченный период времени, в течение которого они могут разработать программу, обеспечивающую бесперебойное функционирование аппаратуры.

## Планы по разработке ПО

Разработчики и маркетологи создали несколько документов, в которых приводится описание проекта Nimbus-LC.

1. “Обзор требований к Nimbus-LC” (“Nimbus-LC Requirements Overview”) далее в этой главе. В документе описываются операционные требования системы Nimbus-LC в том виде, в котором они представляются в начальный момент работы над проектом<sup>1</sup>.
2. “Варианты использования Nimbus-LC” (“Nimbus-LC Use Cases”) далее в этой главе. В этом документе описаны исполнители и варианты использования, основанные на документе, в котором сформулированы требования.
3. “План выпусков версий Nimbus-LC” (“Nimbus-LC Release Plan”) далее в этой главе. В этом документе формулируется план версий данной программы.

<sup>1</sup>Как известно, документ, включающий требования к проекту, обычно представляет собой наиболее изменчивую часть любого программного проекта.

Здесь основные риски смещаются на ранние этапы жизненного цикла проекта, а также приводится обоснование, позволяющее сделать выводы о том, что разработка ПО будет завершена в оптимальный срок.

## Выбор языка программирования

Наиболее существенным ограничением при выборе языка программирования является требование переносимости. Небольшой срок разработки ПО, а также недостаточный уровень ознакомления инженеров-программистов с аппаратным обеспечением версии 2.0 приводят к тому, что в обеих версиях аппаратных платформ (1.0 и 2.0) должно использоваться одно и то же программное обеспечение. При этом исходные коды программ должны быть идентичными (или практически идентичными). Если в языке программирования не учтены ограничения, связанные с переносимостью, вряд ли аппаратное обеспечение версии 2.0 может быть выпущено в течение 12-ти месяцев.

Также следует учитывать некоторые другие ограничения. Сама программа невелика по объему, вследствие чего выдвигаются незначительные требования к занимаемому объему пространства на диске. Не требуется поддержка операций, которые выполняются в режиме реального времени за период, не превышающий секунды, поэтому быстродействие программы не имеет решающего значения. Конечно, операции, выполняемые в режиме реального времени, выполняются со столь низкой скоростью, что возможно применение языка программирования, обладающего свойством сравнительно быстрой “уборки мусора” (*moderately fast garbage-collecting language*). Наличие ограничений, связанных с переносимостью, а также некоторых других серьезных ограничений, говорит в пользу выбора языка программирования Java.

## Разработка проекта **Nimbus-LC**

Согласно плану выпуска версий, на фазе I необходимо разработать архитектуру, обеспечивающую независимость программы от аппаратуры. Конечно, желательно отделять абстрактное поведение метеостанции от конкретной реализации.

Например, программа должна отображать текущее значение температуры независимо от конкретной конфигурации аппаратного обеспечения. Схема проекта, учитывающая эту особенность, показана на рис. 27.1.

Абстрактный базовый класс `TemperatureSensor` поддерживает полиморфную функцию `read()`. Модули, производные от этого класса, позволяют выполнять обращение к отдельным реализациям функции `read()`.

## Тестовые испытания

Заметим, что для каждой из двух известных аппаратных платформ существует по одному унаследованному модулю. Также предусмотрен специальный унасле-

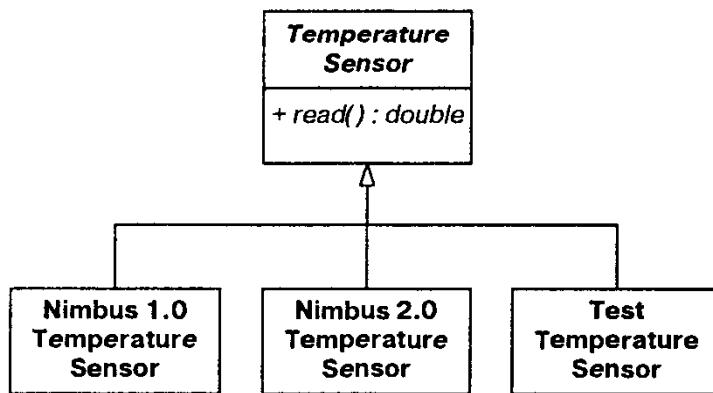


Рис. 27.1. Первоначальный проект температурного датчика

данный модуль `TestTemperatureSensor`. Этот класс применяется для тестирования ПО рабочей станции, не связанной с аппаратным обеспечением `Nimbus`. Инженеры-программисты могут выполнять модульное тестирование, а также тестирование приемлемости ПО даже в том случае, если они лишены доступа к системе `Nimbus`.

Выделено критически малое время на интеграцию аппаратной и программной части `Nimbus 2.0`. Из-за столь сжатых сроков появляется угроза срыва выпуска версии `Nimbus 2.0`. Поскольку ПО `Nimbus` функционирует как с аппаратным обеспечением `Nimbus 1.0`, так и с тестовым классом, необходимо, чтобы программа `Nimbus` выполнялась на нескольких платформах. В этом случае уменьшается риск проявления серьезных срывов в процессе переноса `Nimbus 2.0`.

Тестовые классы также позволяют выполнять тестирование свойств или условий, при выявлении которых в ПО обычно появляются затруднения. Например, тестовые классы можно эксплуатировать в режиме, выявляющем сбои, с трудом имитируемые с помощью аппаратного обеспечения.

## Проведение периодических измерений

Система `Nimbus` чаще всего применяется для отображения данных, соответствующих текущему состоянию погоды. Каждая из этих величин обновляется независимо, с учетом определенного диапазона значений. Значение температуры обновляется ежеминутно, величина атмосферного давления изменяется каждые пять минут. Внесение соответствующих изменений в записи и поддержка связи с пользователем проходит по определенному графику. На рис. 27.2 показан один из вариантов применяемых в этом случае структуры.

Представим, что `Scheduler` является базовым классом, имеющим большое число различных реализаций, по одному для каждой из аппаратных и тестовых платформ. Класс `Scheduler` включает функцию `tic`, которая вызывается каждые 10 миллисекунд. Производный класс несет ответственность за реализацию этих вызовов (рис. 27.3). Класс `Scheduler` включает счетчик вызовов `tic()`.

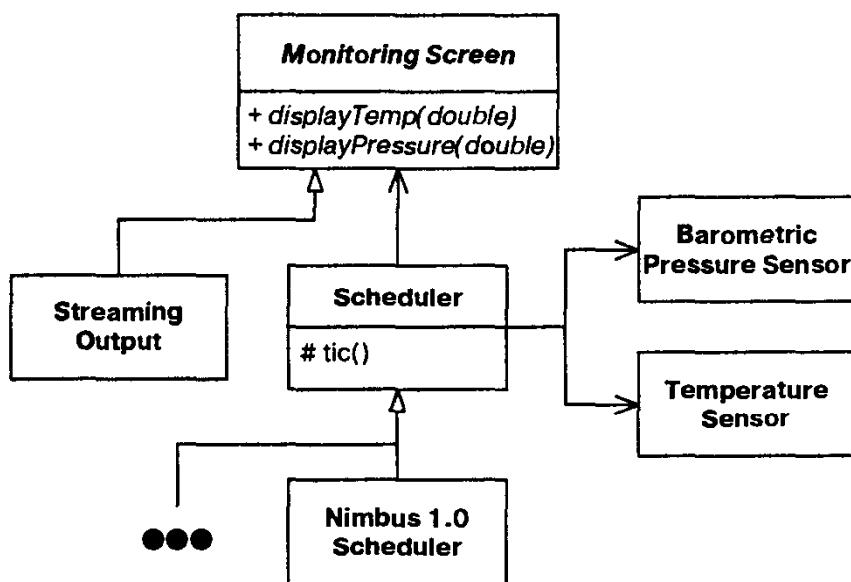


Рис. 27.2. Начальная архитектура Scheduler и Display

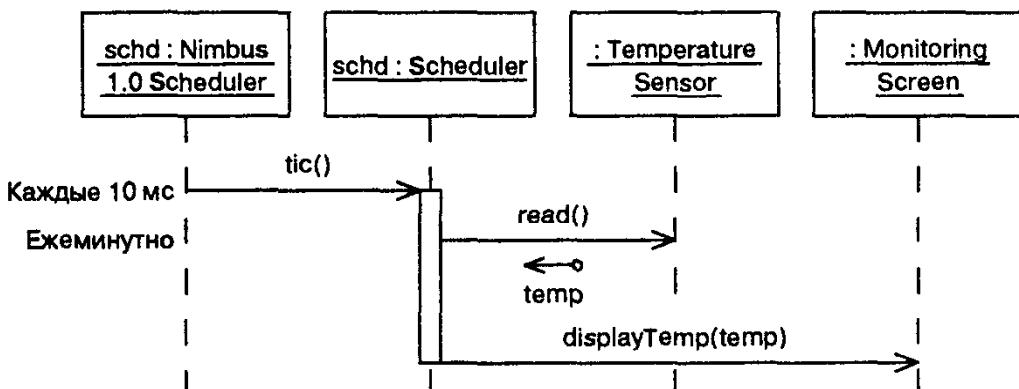


Рис. 27.3. Последовательная диаграмма для начального состояния Scheduler

Ежеминутно вызывается функция `read()` из класса `TemperatureSensor`, которая передает значение температуры классу `MonitoringScreen`. На фазе I не требуется отображать значение температуры средствами графического интерфейса пользователя, поэтому производный модуль `MonitoringScreen` просто пересыпает результат в выводной поток.

### Тренд атмосферного давления

Содержащий требования документ утверждает, что необходимо сообщать о тренде атмосферного давления. Эта переменная величина может быть: *повышенной, пониженной или стабильной*. Как же определить значение этой переменной?

Согласно Федеральному метеорологическому руководству (Federal Meteorological Handbook)<sup>2</sup>, тренд атмосферного давления устанавливается следующим образом.

<sup>2</sup>Federal Meteorologocal Handbook №1, глава 11б раздел 11.4.6 (<http://www.nws.noaa.gov>).

*Если давление повышается или понижается хотя бы на 0,06 дюйма в час, а давление изменяет свое значение на 0,02 дюйма или больше за время наблюдения [проводится один раз в три часа], должны сообщаться сведения об изменении давления.*

Где же находится описанный алгоритм? Если разместить его в классе BarometricPressureSensor, этот класс получает информацию о времени каждого просмотра, а также сохраняет сведения о предыдущих наблюдениях за последние три часа. Сформированный к настоящему времени проект не позволяет реализовывать эти функции. Можно внести значение текущего времени в качестве аргумента функции Read из класса BarometricPressureSensor, что позволяет гарантировать регулярный вызов этой функции.

В этом случае определение тренда связано с частотой пользовательских обновлений. Это достаточно неудобно, поскольку изменение схемы по обновлению пользовательского интерфейса может повлиять на алгоритм определения тренда атмосферного давления. Также это неблагоприятно оказывается на показателях датчика, так как для правильного использования функции показатели датчика следует снимать регулярно. Поэтому следует найти более подходящее решение.

В случае необходимости можно устроить так, чтобы класс Scheduler отслеживал колебания атмосферного давления за определенный промежуток времени, выдавая сведения об этих трендах. Но в этом случае потребуется провести аналогичные изыскания для температуры и скорости ветра, помещая соответствующие данные в класс Scheduler? Каждый новый ряд показателей датчика или исторические сведения приводят к существенным изменениям в классе Scheduler. Получаем “замкнутый круг”.

## **Повторный обзор класса Scheduler**

Еще раз обратите внимание на схему (рис. 27.2). Нетрудно заметить, что в классе Scheduler имеется связь с каждым из датчиков, а также с интерфейсом пользователя. Сколь бы много датчиков и экранов с интерфейсами пользователей не вносились в проект, все они должны отображаться также и в классе Scheduler. Поэтому в Scheduler не исключается возможность добавления новых датчиков или интерфейсов пользователей. А это приводит к нарушению принципа OCR. Желательно в разработке класса Scheduler учесть принцип независимости изменений, играющий немалую роль в процессе добавления датчиков и пользовательских интерфейсов.

## **Разделение интерфейса пользователя**

Пользовательские интерфейсы являются изменяемым элементом. Они зависят от требований заказчиков, маркетологов, а также почти каждого пользователя, контактирующего с данным продуктом. Складывается впечатление, что именно с этого элемента можно начинать процесс разделения требований.

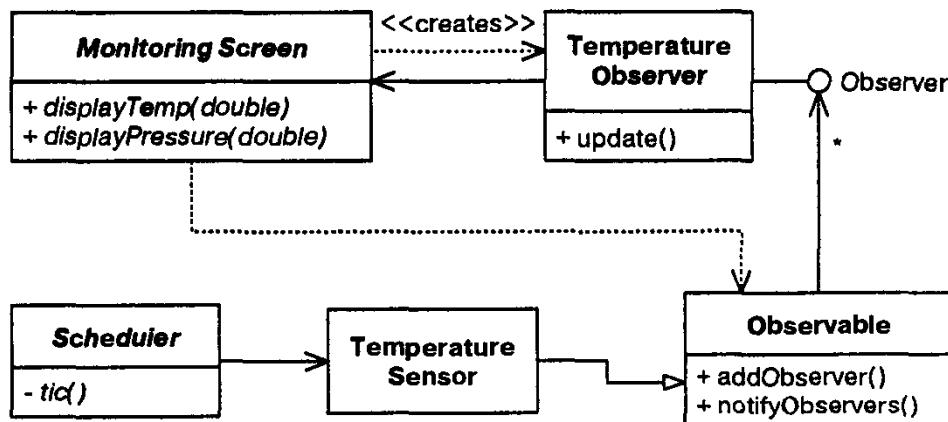


Рис. 27.4. Шаблон Observer отделяет UI от Scheduler

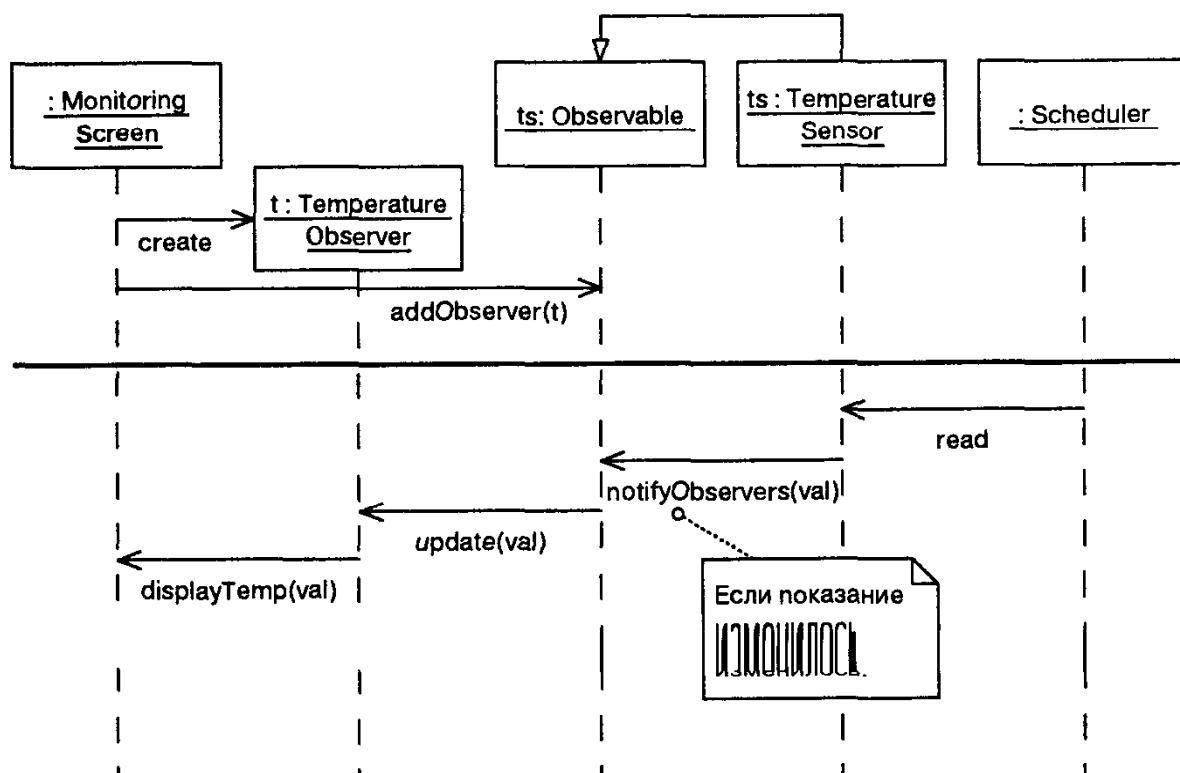


Рис. 27.5. Последовательная диаграмма разделенного интерфейса пользователя

На рис. 27.4 и 27.5 показан новый проект, в котором используется шаблон **Observer**. В этом случае пользовательский интерфейс становится зависимым от показаний датчика. Как только датчик регистрирует изменения, они автоматически отображаются интерфейсом. Обратите внимание, что эта зависимость является косвенной.

Реальным наблюдателем в данном случае выступает шаблон **Adapter<sup>3</sup>** под названием **TemperatureObserver**. Этот объект регистрируется в **TemperatureSensor** в случае изменения значения температуры. В ответ **TemperatureObserver** вызывает функцию **DisplayTemp**, относящуюся к объекту **MonitoringScreen**.

<sup>3</sup> Година с 120

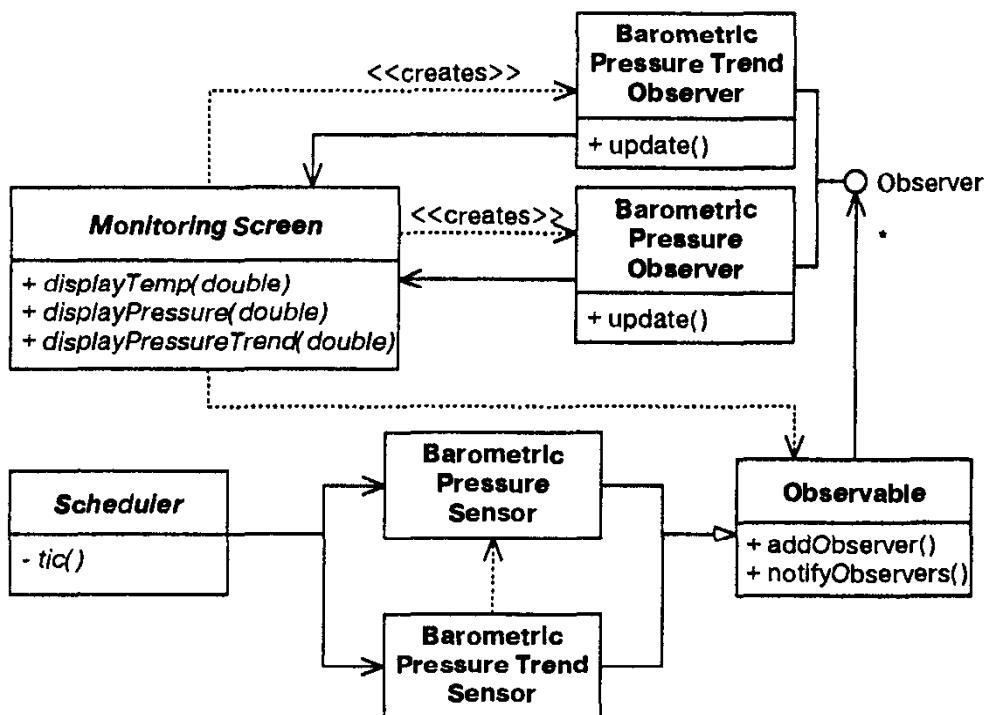


Рис. 27.6. Отслеживание атмосферного давления

Этот проект прекрасно отделяет интерфейс пользователя от **Scheduler**. Теперь **Scheduler** не получает никакой информации об UI и фокусируется исключительно на указании датчикам конкретного времени просмотра. Интерфейс пользователя связывается с датчиками и ожидает отчета об изменениях. Но он не получает данных от датчиков. Поступают сведения о наборе объектов, реализующих наблюдаемый интерфейс. Это позволяет добавлять датчики без внесения серьезных изменений в рассматриваемую часть интерфейса пользователя.

Аналогично решается проблема с трендом атмосферного давления. Эти данные можно уточнять с помощью отдельного модуля **BarometricPressureTrendSensor**, регистрирующего данные от **BarometricPressureSensor** (рис. 27.6).

## Новое представление о **Scheduler** – все сначала

Главная задача **Scheduler** состоит в отсылке команды каждому из датчиков, указывающую на то, когда именно следует обратиться к новому значению. Если же в дальнейшем добавляется или удаляется датчик, в **Scheduler** следует внести изменения. Конечно, в **Scheduler** вносятся изменения даже в том случае, если речь идет о модификации шкалы датчика. И вновь явно нарушается принцип OCR. Создается впечатление, что данные о шкале отсчета датчика имеют отношение только к датчику и не влияют на другие компоненты системы.

Можно отделить **Scheduler** от датчиков с помощью парадигмы **Listener**<sup>4</sup> из библиотеки классов Java. Она подобна шаблону **Observer**, который также

<sup>4</sup>[JAVA98], с. 360.

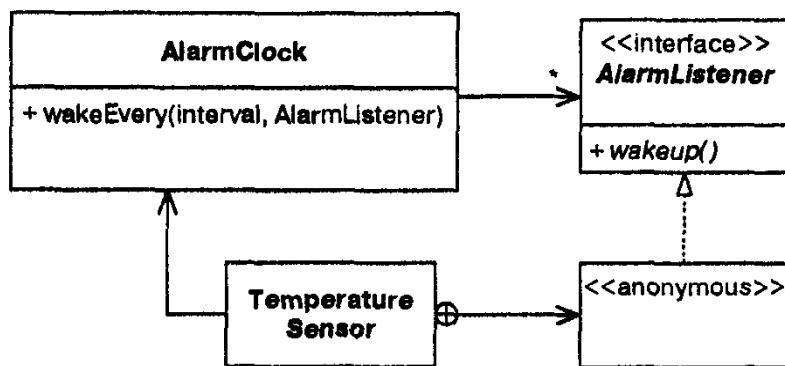


Рис. 27.7. Отделение часов (Alarm) от других модулей

проводит регистрацию событий; но в этом случае желательно зарегистрировать время наступления этого события (рис. 27.7).

Применение датчиков приводит к формированию анонимных классов *Adapter*, реализующих интерфейс *AlarmListener*. Затем датчики регистрируют эти адаптеры с помощью *AlarmClock* (этот класс используется для вызова *Scheduler*). В рамках регистрации *AlarmClock* получает указание, как часто ему следует “просыпаться” (т.е. каждую секунду или каждые пятьдесят миллисекунд). По завершении этого периода *AlarmClock* направляет адаптеру сообщение о “пробуждении”, а адаптер затем пересыпает датчику сообщение *read*.

Подобный подход полностью меняет структуру класса *Scheduler*. На рис. 27.2 класс *Scheduler* формирует центральное ядро нашей системы, а также получает сведения о большинстве других компонентов. Теперь же этот класс смешен на “периферию” системы.

Этот класс не располагает данными о других компонентах. Он согласуется с принципом SRP при выполнении одного вида деятельности (составление графика), что не имеет ничего общего с отслеживанием метеорологических данных. Конечно, этот класс можно применять повторно в других приложениях. Фактически, изменения столь существенны, что коснулись названия класса. Теперь он называется *AlarmClock*.

## Структура датчиков

При разделении датчиков, находящихся в других частях системы, следует учитывать их внутреннюю структуру. Теперь датчики включают три отдельные функции. Во-первых, они создают и регистрируют анонимные производные модули класса *AlarmListener*. Во-вторых, они уточняют, внесены ли изменения в их показания, и вызывают метод *notifyObservers* класса *Observable*. В-третьих, при просмотре соответствующих значений они взаимодействуют с аппаратным обеспечением *Nimbus*.

На рис. 27.1 показано, как можно разделить функции датчиков. На рис. 27.8 этот проект интегрирован с другими изменениями. Базовый класс *Tempera-*

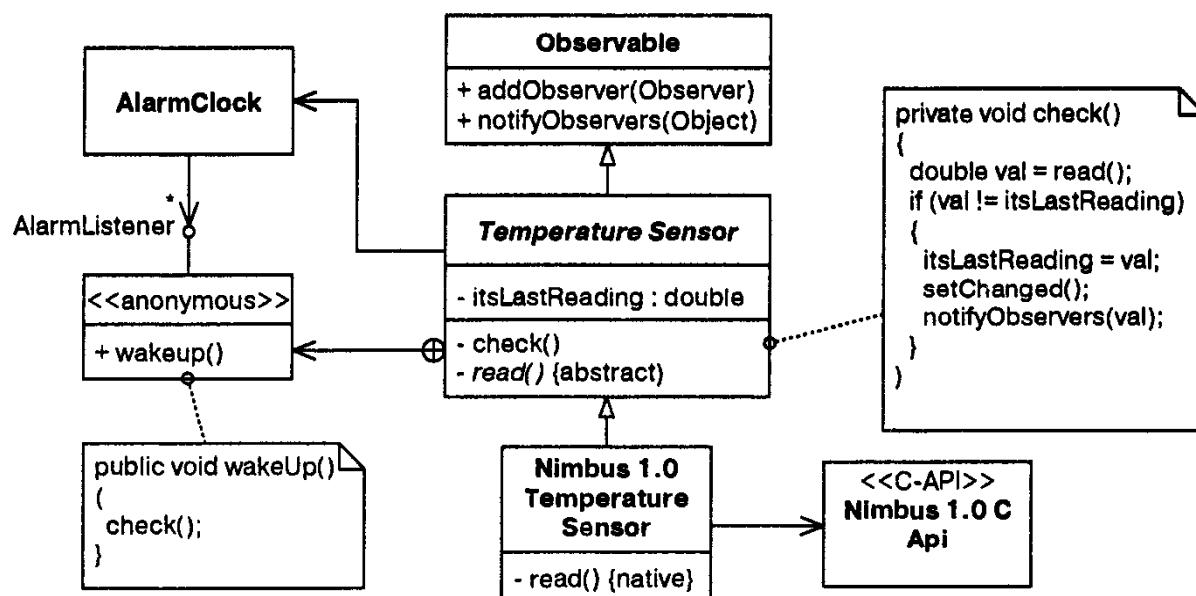


Рис. 27.8. Структура Sensor

tureSensor имеет дело с первыми двумя функциями, поскольку они носят общий характер. Производный модуль TemperatureSensor может реализовать связь с аппаратным обеспечением, а также реализовать просмотр в режиме реального времени.

На рис. 27.8 показана реализация шаблона Template Method, что позволяет разделить общие и специфические функции TemperatureSensor. Этот шаблон можно использовать частными функциями check и read из TemperatureSensor. Если AlarmClock вызывает wakeup для анонимного класса, который передает вызов функции check из TemperatureSensor. Затем функция check вызывает абстрактную функцию read из TemperatureSensor. Эта функция реализуется с помощью производного модуля, что позволяет должным образом взаимодействовать с аппаратным обеспечением и получать показания датчика. Затем функция check определяет, отличается ли новое показание от предыдущего. Если имеется отличие, об этом уведомляются ожидающие наблюдатели.

Таким образом удачно достигается искомое разделение функций. Для каждой новой аппаратной или тестовой платформы следует создавать функционирующий совместно с ней производный модуль TemperatureSensor. Более того, этот производный модуль практически перекрывает простую функцию: read(). Остальные функциональные возможности датчика остаются реализованными в соответствующем базовом классе.

### Где находится API?

Одной из целей разрабатываемой нами версии II является создание новой API для аппаратной платформы Nimbus 2.0. Эта API должно записываться в Java, иметь расширяемый характер и поддерживать простой и непосредственный до-

ступ к аппаратному обеспечению Nimbus 2.0. Более того, API должен поддерживать аппаратуру версии Nimbus 1.0. Без применения API все простые отладочные и классификационные инструменты, созданные для данного проекта, будут изменяться при вводе новой платы. Где же находится подобное API в нашем нынешнем проекте?

Следует отметить, что до сих пор проект не предусматривал разработку даже простого API. В этом случае можно воспользоваться следующим кодом:

```
public interface TemperatureSensor
{
 public double read();
}
```

Следует разрабатывать инструменты, располагающие непосредственным доступом к этому API, без вмешательства регистрирующих наблюдателей. Также нежелательно, чтобы на этом уровне датчики выполняли автоматическую регистрацию или взаимодействовали с `AlarmClock`. Разработчики заинтересованы в создании простого и изолированного механизма, который служит в качестве непосредственного аппаратного интерфейса.

Может сложиться мнение, что происходит возврат к старым аргументам. В конце концов, на рис. 27.1 показана требуемая схема. Но в эту схему последовательно вносились определенные изменения. Теперь необходимо получить “гибрид”, включающий лучшее из обеих схем.

Для исключения реального API из `TemperatureSensor` применяется шаблон `Bridge` (рис. 27.9). Этот шаблон отделяет реализацию от абстракции, вследствие чего обе могут варьироваться независимым образом. В нашем случае `TemperatureSensor` является абстрацией, а `TemperatureSensorImp` — реализацией. Заметим, что термин “реализация” применяется для описания абстрактного интерфейса, и “реализация” самореализуется с помощью класса `Nimbus1.0TemperatureSensor`.

## Вопросы, связанные с формированием проекта

Снова рассмотрим рис. 27.9. Чтобы данная схема носила функциональный характер, следует создать объект `TemperatureSensor` и привязать его к объекту `Nimbus1.0TemperatureSensor`. Кто позаботится об этом? Конечно, компонент программы, который отвечает за это, не является независимым от платформы, поскольку должен содержать точные сведения о зависимом от платформы классе `Nimbus1.0TemperatureSensor`.

Реализация описанных деталей осуществляется в основной программе. Ее код может выглядеть так, как показано в листинге 27.1.

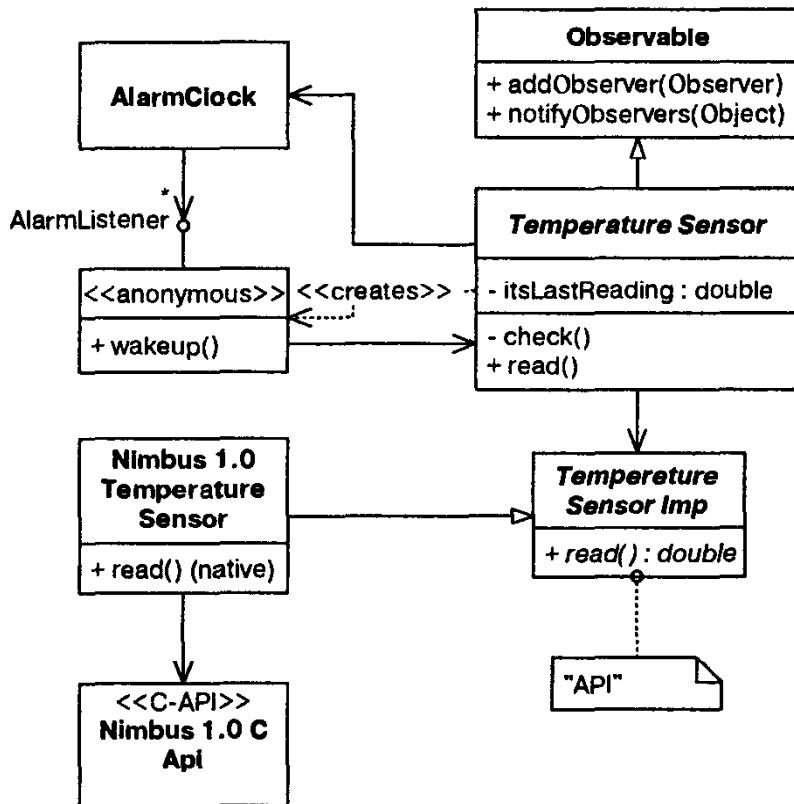


Рис. 27.9. Датчик температуры, для которого был разработан API

### Листинг 27.1. Программа Weatherstation

```

public class Weatherstation
{
 public static void main(String [] args)
 {
 AlarmClock ac = new AlarmClock(
 new Nimbus1_0AlarmClock);

 TemperatureSensor ts =
 new TemperatureSensor(ac,
 new Nimbus1_0TemperatureSensor);

 BarometricPressureSensor bps =
 new BarometricPressureSensor(ac,
 new Nimbus1_0BarometricPressureSensor);

 BarometricPressureTrend bpt =
 new BarometricPressureTrend(bps)
 }
}

```

Это вполне реальное решение, но требует много дополнительной канцелярской работы. Поэтому лучше воспользоваться шаблоном **Factory**. Эта структура показана на рис. 27.10.

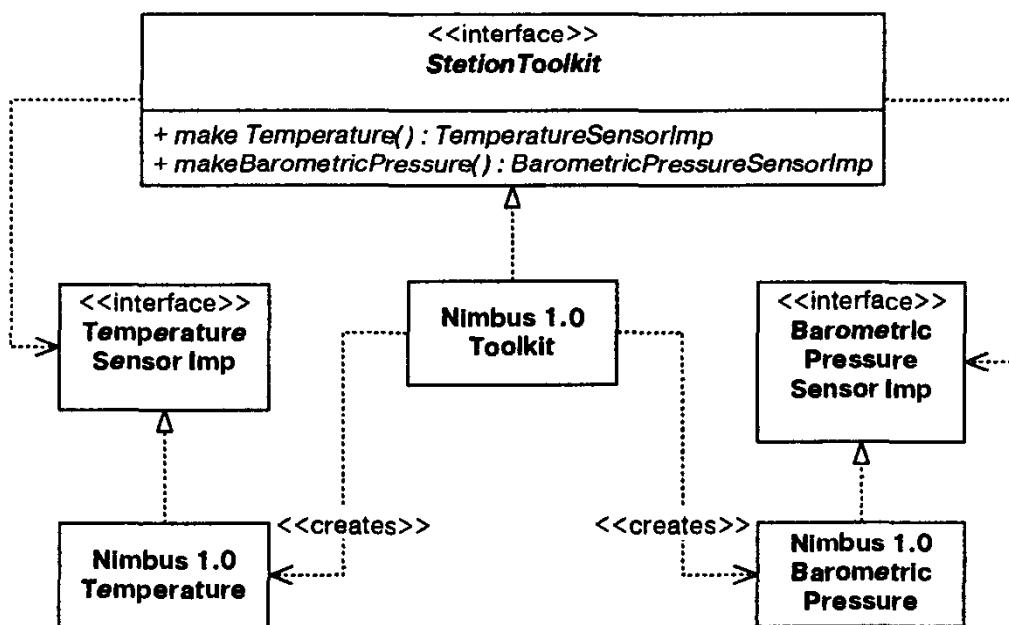


Рис. 27.10. Набор инструментов, моделирующих метеостанцию

В данном случае фабрика называется **StationToolkit**. Речь идет об интерфейсе, представляющем методы, предполагающие создание экземпляров API классов. Каждая платформа имеет собственный производный модуль из **StationToolkit**, и этот производный модуль создает соответствующие производные модули классов API.

Теперь можно переписать основную функцию, как показано в листинге 27.2. Обратим внимание на тот факт, что при выполнении основной программы на иной платформе необходимо вносить изменения в две строки кода, где формируются классы **Nimbus1\_0AlarmClock** и **Nimbus1\_0Toolkit**. Данное усовершенствование кода из листинга 27.1 имеет большое значение, поскольку раньше следовало вносить изменения в каждый создаваемый датчик.

### Листинг 27.2. Программа Weatherstation

```

public class Weatherstation
{
 public static void main(String[] args)
 {
 AlarmClock ac = new AlarmClock(
 new Nimbus1_0AlarmClock;

 StationToolkit st = new Nimbus1_0Toolkit();

 TemperatureSensor ts =
 new TemperatureSensor(ac,st);

 BarometricPressureSensor bps =
 new BarometricPressureSensor(ac,st);
 }
}

```

```

 BarometricPressureTrend bpt =
 new BarometricPressureTrend(bps)
 }
}

```

Заметим, что сигналы от `StationToolkit` передаются каждому датчику. В этом случае датчики создают собственные реализации. В листинге 27.3 показан конструктор для `TemperatureSensor`.

#### Листинг 27.3. Класс `TemperatureSensor`

```

public class TemperatureSensor extends Observable
{
 public TemperatureSensor(AlarmClock ac,
 StationToolkit st)
 {
 itsimp = st.makeTemperature();
 }
 private TemperatureSensorImp itsImp;
}

```

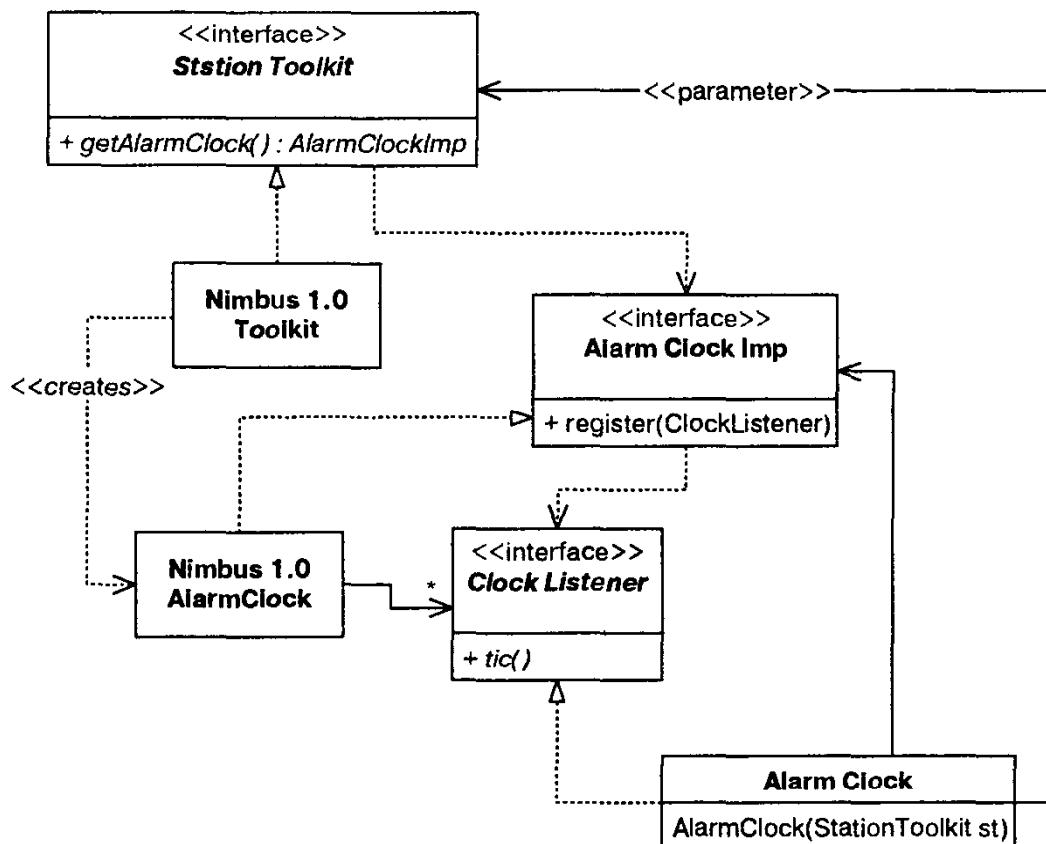
### Использование набора инструментов для создания `AlarmClock`

Можно продолжить совершенствование программы, с помощью `StationToolkit` создавая соответствующий производный модуль для `AlarmClock`. Снова обратимся к шаблону `Bridge`, используемому для изоляции абстракции `AlarmClock`, играющей важную роль в приложениях по отслеживанию изменений погодных условий, от реализации, которая поддерживает аппаратную платформу.

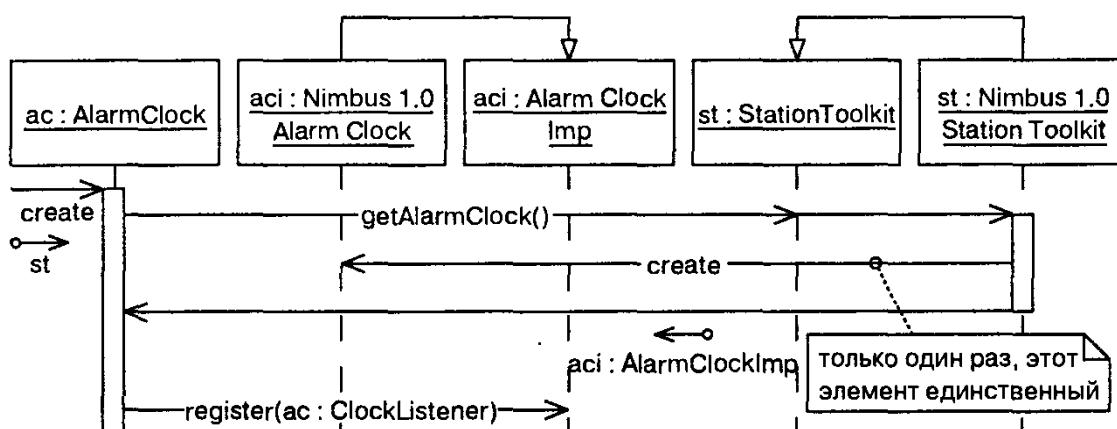
На рис. 27.11 показана новая структура `AlarmClock`. Теперь `AlarmClock` получает сообщения от функции `tic()` с помощью соответствующего интерфейса `ClockListener`. Эти сообщения пересылаются в API из соответствующего производного модуля класса `AlarmClockImp`.

На рис. 27.12 иллюстрируется процесс создания `AlarmClock`. Соответствующий производный модуль `StationToolkit` передается конструктору `AlarmClock`. Класс `AlarmClock` направляет его для создания подходящего производного модуля из `AlarmClockImp`. Этот модуль передается обратно, `AlarmClock`, и `AlarmClock` регистрируются с его помощью, что позволяет получать от него сообщения `tic()`.

Аналогично, изложенные соображения реализуются в основной программе, код которой представлен в листинге 27.4. Обратите внимание, что в настоящем коде есть лишь одна строка, определяющая зависимость от платформы. Изменим эту строку, после чего будет определена новая платформа, применяемая в системе.



**Рис. 27.11.** Набор инструментов, моделирующих метеостанцию, и Alarm Clock



**Рис. 27.12.** Процесс создания Alarm Clock

#### Листинг 27.4. Программа Weatherstation

```

public class Weatherstation
{
 public static void main(String[] args)
 {
 StationToolkit st = new Nimbus1_0Toolkit();
 AlarmClock ac = new AlarmClock(st);
 TemperatureSensor ts =
 new TemperatureSensor(ac, st);
 }
}

```

```
BarometricPressureSensor bps =
 new BarometricPressureSensor(ac,st);

BarometricPressureTrend bpt =
 new BarometricPressureTrend(bps)
}

}
```

---

Все это просто замечательно, но при использовании Java можно получить еще лучший результат. Средствами Java можно создавать объекты в соответствии с их именами. Основная программа, код которой приведен в листинге 27.5, не изменяется в случае применения новой платформы. Название производного модуля *StationToolkit* просто передается ей в качестве аргумента командной строки. Если название указано корректно, создается соответствующий класс *StationToolkit*, а оставшаяся часть системы ведет себя соответствующим образом.

---

#### Листинг 27.5. Программа Weatherstation

---

```
public class Weatherstation
{
 public static void main(String[] args)
 {
 try
 {
 Class tkClass = Class.forName(args[0]);
 StationToolkit st =
 (StationToolkit)tkClass.newInstance();

 AlarmClock ac = new AlarmClock(st);

 TemperatureSensor ts =
 new TemperatureSensor(ac,st);

 BarometricPressureSensor bps =
 new BarometricPressureSensor(ac,st);

 BarometricPressureTrend bpt =
 new BarometricPressureTrend(bps)
 }
 catch(Exception e)
 {
 }
 }
}
```

---

## Упаковка классов

Существует несколько компонентов разрабатываемого ПО, которые желательно выпускать и распределять отдельно. Интерфейс API и каждый из его экземпляров применяются повторно без использования оставшейся части приложения. К этим компонентам обращаются при тестировании и проведении проверок качества. Пользовательский интерфейс и датчики следует разделить, чтобы они могли изменяться независимо друг от друга. В конце концов, вновь создаваемые продукты могут располагать на вершине той же самой системной архитектуры более совершенными интерфейсами пользователя. Фактически, версия II служит первым таким примером.

На рис. 27.13 показана пакетная структура для фазы I. Эта структура состоит из классов, которые будут разрабатываться в дальнейшем. Для каждой платформы есть один пакет, а классы в этих пакетах являются производными от классов в пакете API. Единственным клиентом пакета API является пакет `WeatherMonitoringSystem`, содержащий все другие классы.

Хотя для версии I разработан весьма компактный интерфейс пользователя, к сожалению, имеет место смешивание с классами `WeatherMonitoringSystem`. Желательно поместить этот класс в отдельный пакет. Но тут возникает проблема. В этом случае объект `Weatherstation` создает объект `MonitoringScreen`, но объект `MonitoringScreen` должен располагать информацией обо всех датчиках, что позволит ему с помощью интерфейса `Observable`

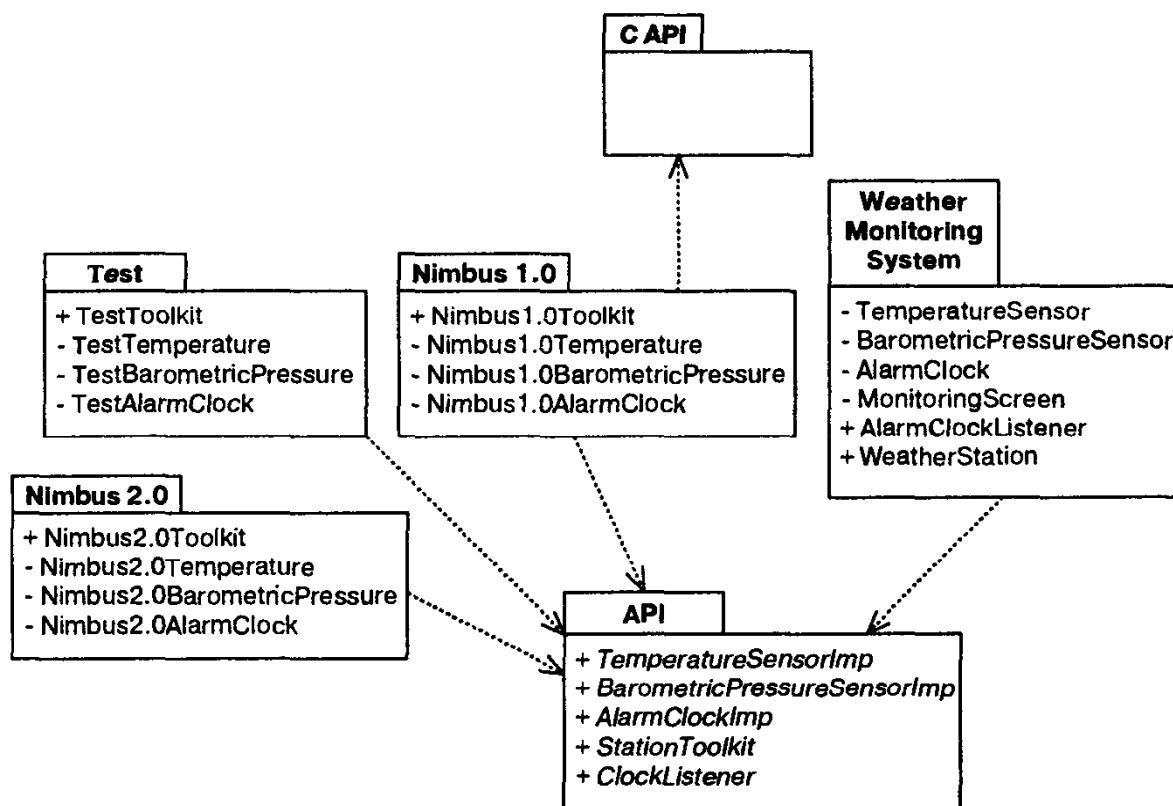


Рис. 27.13. Фаза I пакетной структуры

увеличивать количество соответствующих наблюдателей. Поэтому, если объект `MonitoringScreen` перемещается в собственный пакет, между этим пакетом и пакетом `WeatherMonitoringSystem` появляется циклическая зависимость. Это приводит к нарушению принципа ациклических зависимостей (ADP), вследствие чего два пакета не смогут существовать независимо друг от друга.

Чтобы устранить эту особенность, следует переместить основную программу из класса `Weatherstation`. `Weatherstation` по-прежнему создает `StationToolkit` и все датчики, но `MonitoringScreen` не формируется. Основная программа создает `MonitoringScreen` и `Weatherstation`. Затем основная программа передает `Weatherstation` к `MonitoringScreen`, и тогда `MonitoringScreen` может увеличивать количество наблюдателей за показаниями датчиков.

Каким образом `MonitoringScreen` получает показания датчиков от `Weatherstation`? Для этого необходимо к `Weatherstation` добавить некоторые методы (листинг 27.6).

---

#### Листинг 27.6. Программа `Weatherstation`

---

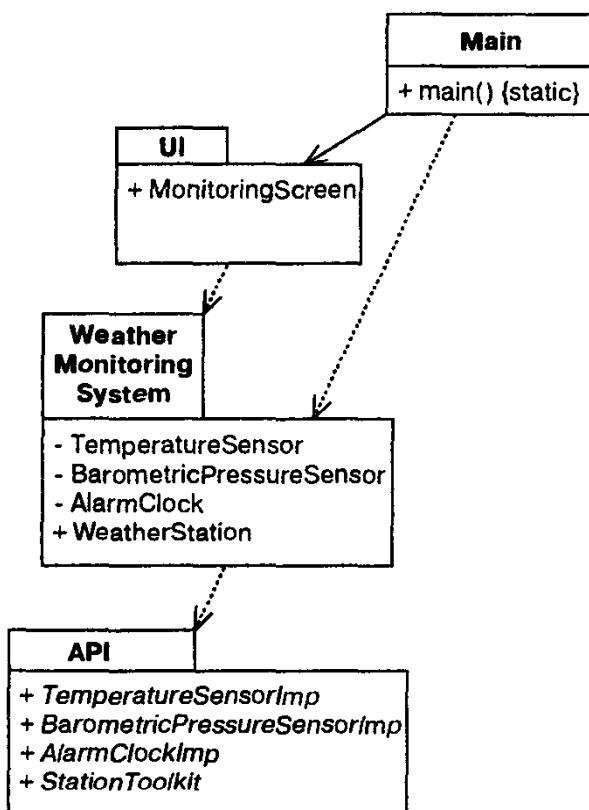
```
public class Weatherstation
{
 public Weatherstation(String tkName)
 {
 // создание набора инструментов станции и датчиков, как и раньше
 }

 public void addTempObserver(Observer o)
 {
 itsBPS.addObserver(o);
 }

 public void addBPObserver(Observer o)
 {
 itsBPS.addObserver(o);
 }

 public void addBPTrendObserver(Observer o)
 {
 itsBPT.addObserver(o);
 }

 // частные переменные...
 private TemperatuerSensor itsTS;
 private BarometricPressureSensor itsBPS;
 private BarometricPressureTrend itsBPT;
```



**Рис. 27.14.** Пакетная диаграмма с прерванным циклом

Теперь можно заново создать пакетную диаграмму, как показано на рис. 27.14. Опустим большинство пакетов, не имеющих отношения к `MonitoringScreen`. Все прекрасно получается. Конечно, интерфейс пользователя можно варьировать без внесения изменений в `WeatherMonitoringSystem`. Но зависимость интерфейса пользователя от `WeatherMonitoringSystem` приводит к проблемам при внесении изменений в `WeatherMonitoringSystem`.

Как `UI`, так и `WeatherMonitoringSystem` являются статичными. Если один статичный пакет зависит от другого, нарушается принцип инверсии зависимости (DIP, Dependency inversion principle). В этом случае желательно, чтобы `UI` зависел от некоторой абстракции, а не от `WeatherMonitoringSystem`.

Эту особенность можно устраниТЬ, создавая интерфейс, который может применяться `MonitoringScreen` наряду с производными модулями `WeatherStation` (рис. 27.15).

Теперь, если интерфейс `WeatherStationComponent` поместить в собственный пакет, получим желаемое разделение (рис. 27.16). Заметим, что теперь `UI` и `WeatherMonitoringSystem` полностью отделены. Они оба могут изменяться независимо друг от друга, что весьма целесообразно.

## 24-часовая история, а также свойство неизменности

В разделе производных модулей из версии I содержатся пункты, указывающие на необходимость поддержки данных, связанных с 24-часовой историей.

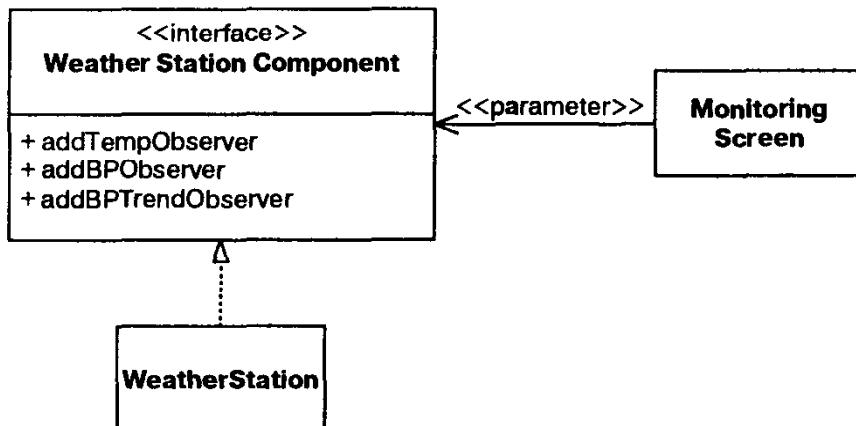


Рис. 27.15. Абстрактный интерфейс WeatherStation

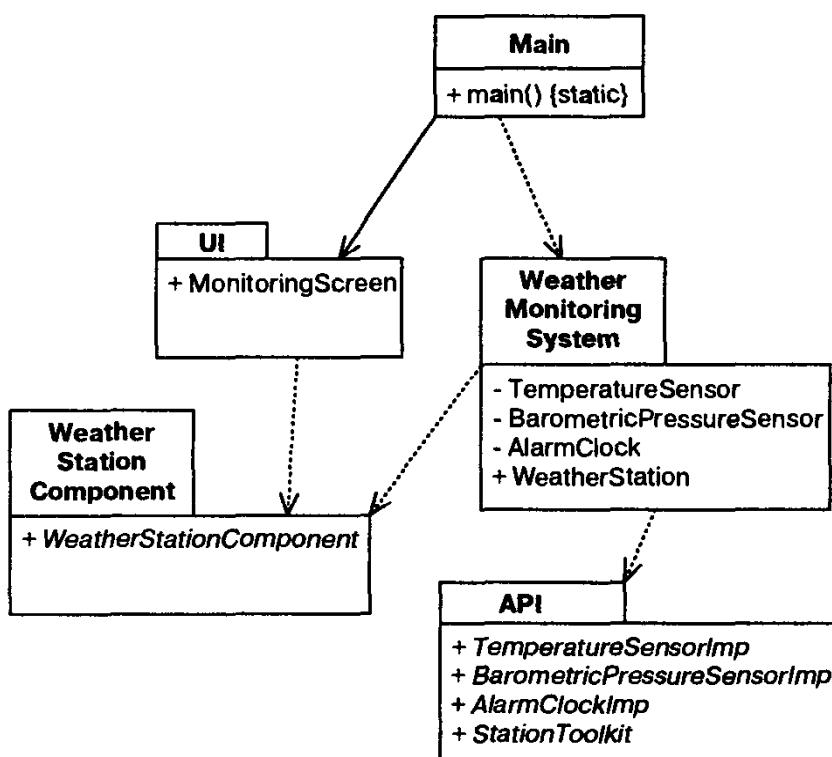


Рис. 27.16. Пакетная диаграмма компонента Weather Station

Известно, что аппаратное обеспечение версий Nimbus 1.0 и Nimbus 2.0 снабжено определенным видом энергонезависимой памяти (NVRAM, Nonvolatile memory). С другой стороны, тестовая платформа имитирует этот вид памяти с помощью жесткого диска.

Необходимо создать постоянный механизм, независимый от отдельных платформ, где поддерживаются необходимые функциональные возможности. Также следует соединить его с механизмами, которые поддерживают данные, накопленные за последние 24 часа.

Ясно, что механизм, поддерживающий на низком уровне неизменные данные, следует определить как интерфейс пакета API. Какой вид должен иметь этот

интерфейс? Nimbus I C-API поддерживает вызовы, позволяющие просматривать блоки байтов и вести записи с помощью определенных смещений ячеек энергонезависимой памяти. Это довольно эффективно, но несколько примитивно. Имеется ли более совершенный способ?

## Постоянные API

Среда Java поддерживает возможность, позволяющую выполнить быструю конвертацию каждого объекта в массив байтов. Этот процесс называется *серIALIZацией (serialization)*. Из подобного массива байтов можно воссоздать объект, воспользовавшись процессом *десериализации (deserialization)*. Этот подход удобно применять, если API низкого уровня позволяет указывать объект и его название. Этот процесс отображен в листинге 27.7.

---

### Листинг 27.7. Объект PersistentImp

---

```
package api;
import Java.io.Serializable;
import Java.util.AbstractList;
public interface PersistentImp
{
 void store(String name, Serializable obj);
 Object retrieve(String name);
 AbstractList directory(String regExp);
}
```

---

Интерфейс `PersistentImp` позволяет сохранять и полностью восстанавливать объекты по названию. Единственным ограничением служит то, что подобные объекты должны реализовать интерфейс `Serializable`, а это ограничение не столь значительно.

### 24-часовая история

Если возникает необходимость в использовании низкоуровневого механизма для хранения неизменных данных, обратите внимание на вид этих данных. Специалисты утверждают, что следует сохранять записи о максимальных и минимальных показаниях за предыдущий 24-часовой период. На рис. 27.23 приводится график с этими данными. Этот график производит впечатление плохо упорядоченного набора данных. Максимальное и минимальное значения кажутся излишними. Хуже того, они соответствуют 24-часовому промежутку времени, но не учитывают данные за предыдущий календарный день. Обычно, если речь идет о наибольшем и наименьшем значении за 24 часа, хотят получить данные с учетом прошедшего дня.

Сложившаяся ситуация возникла вследствие недоработок специалистов или служит укором для программистов? Не следует реализовывать то, что устраивает специалистов, но не подходит заказчикам.

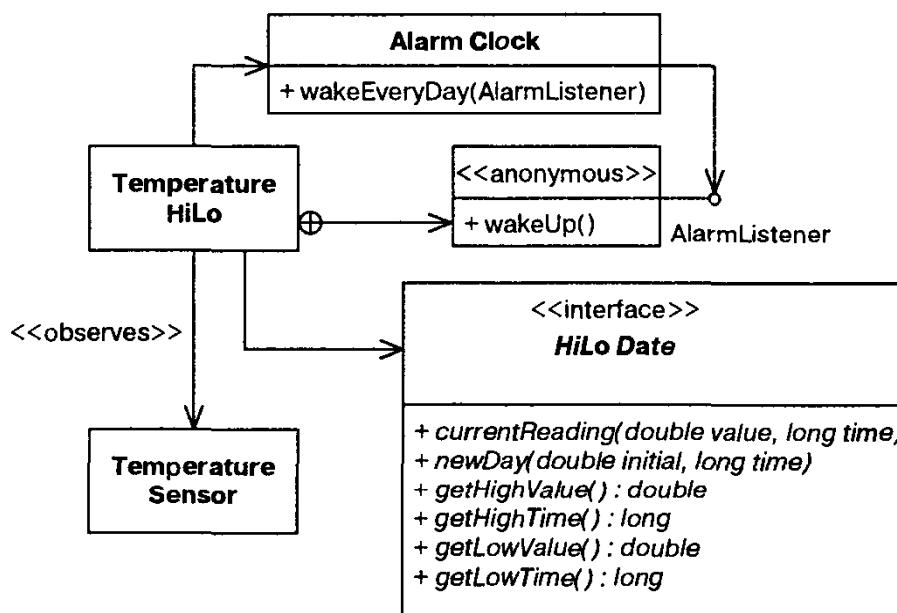
При поверхностной проверке с участием членов команды разработчиков приходим к выводу, что интуиция нас не подвела. Действительно, необходимо сохранять данные за последние 24 часа. Но наименьшие и наибольшие показатели должны определяться с учетом данных за предыдущий календарный день.

### **Максимальное и минимальное значение, фиксируемое за 24-часовый период**

Максимальное и минимальное значение определяются ежедневно на основе показаний датчиков, регистрируемых в реальном времени. Например, при изменении температуры соответствующим образом обновляются как минимальное, так и максимальное значение, фиксируемое за 24-часовой период. Ясно, что речь идет об отношении *observer*. На рис. 27.17 показана статическая структура, а на рис. 27.18 отображен соответствующий динамический сценарий.

При демонстрации шаблона *Observer* используется ассоциация, отмеченная как стереотип “наблюдения” (“*observes*”). Создается класс под названием *TemperatureHiLo*, который активизируется с помощью *AlarmClock* каждые сутки, в полночь. Обратите внимание, что к *AlarmClock* был добавлен метод *wakeEveryDay*.

Благодаря своей особой конструкции объект *TemperatureHiLo* регистрируется как *AlarmClock* и как *TemperatureSensor*. Всякий раз при изменении температуры объект *TemperatureHiLo* уведомляется об этом с помощью шаблона *observer*. Затем *TemperatureHiLo* информирует интерфейс *HiLoData* с помощью метода *currentReading*.



**Рис. 27.17. Структура *TemperatureHiLo***

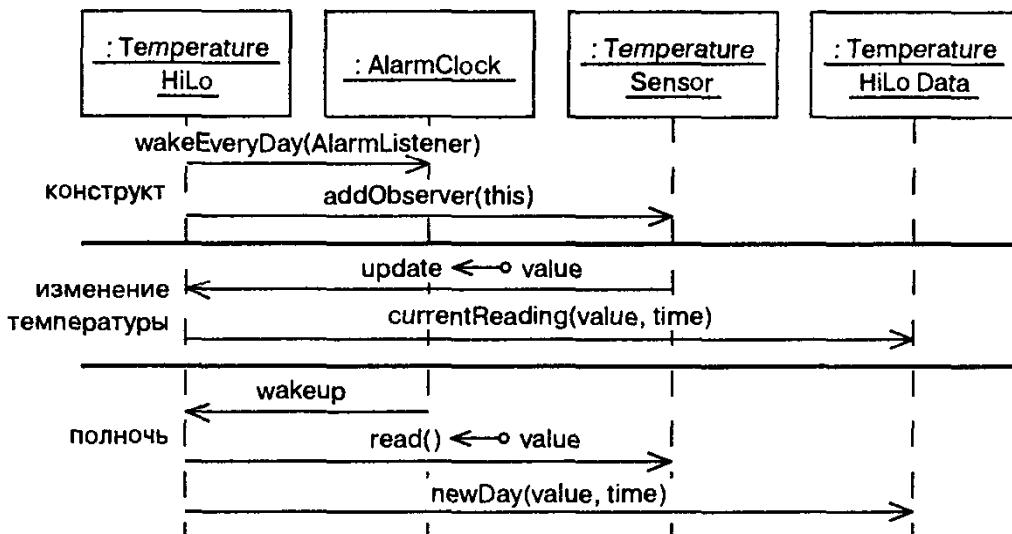


Рис. 27.18. Сценарии HiLo

Класс `HiLoData` реализуется с помощью определенного класса, которому известно, как сохраняется максимальное и минимальное значение для текущего 24-часового периода.

Класс `TemperatureHiLo` отделяется от класса `HiLoData` по двум причинам. Прежде всего необходимо отделить информацию об `TemperatureSensor` и `AlarmClock` от алгоритмов, определяющих минимальные и максимальные значения. Во-вторых, что более важно, алгоритм по определению ежедневных максимумов и минимумов может использоваться заново при уточнении значений атмосферного давления, скорости ветра, точки росы и т.д. Поэтому для наблюдения за соответствующими датчиками следует применять `BarometricPressureHiLo`, `DewPointHiLo`, `WindSpeedHiLo` и т.д., каждый из которых может использовать класс `HiLoData` для вычисления данных и их хранения.

В полночь `AlarmClock` пересыпает объекту `TemperatureHiLo` сообщение `wakeup`. `TemperatureHiLo` откликается, обращаясь к текущему значению температуры из `TemperatureSensor` и пересыпая это значение в интерфейс `HiLoData`. Реализация `HiLoData` с помощью интерфейса `PersistentImp` сохраняет значения, соответствующие прошедшему календарному дню, а также начинает новый календарный день, используя пересланное значение в качестве начального.

Класс `PersistentImp` с помощью строки получает доступ к объектам в неизменяемом хранилище. Эта строка служит ключом доступа. Объекты `HiLoData` сохраняются и выбираются с помощью строк, имеющих следующий формат: "<type>+HiLo+<MMxddхуууу>". Например "temperatureHiLo04161998".

## Реализация алгоритмов HiLo

Как же реализован класс HiLoData? Подобный подход может показаться несколько прямолинейным. Листинг 27.8 включает код Java для этого класса.

---

### Листинг 27.8. HiLoDataImp

---

```

public class HiLoDataImp implements HiLoData, Java.io.Serializable
{
 public HiLoDataImp(StationToolkit st, String type,
 Date theDate, double init, long initTime)
 {
 itsPI = st.getPersistentImpl();
 itsType = type;
 itsStorageKey = calculateStorageKey(theDate);
 try
 {
 HiLoData t = (HiLoData)itsPI.retrieve(itsStorageKey);
 itsHighTime = t.getHighTime();
 itsLowTime = t.getLowTime();
 itsHighValue = t.getHighValue();
 itsLowValue = t.getLowValue();
 currentReading(init, initTime);
 }
 catch (RetrieveException re)
 {
 itsHighValue = itsLowValue = init;
 itsHighTime = itsLowTime = initTime;
 }
 }

 public long getHighTime() {return itsHighTime;}
 public double getHighValue() {return itsHighValue;}
 public long getLowTime() {return itsLowTime;}
 public double getLowValue () {return itsLowValue;}

 // Уточните, изменяют ли новые показания значения
 // hi и lo, и возвратите значение true, если
 // показания изменились.
 public void currentReading(double current, long time)
 {
 if (current > itsHighValue)
 {
 itsHighValue = current;
 itsHighTime = time;
 store();
 }
 else if (current < itsLowValue)
 {
 itsLowValue = current;
 itsLowTime = time;
 }
 }
}

```

```
 store();
 }
}

public void newDay(double initial, long time)
{
 store();
 // теперь очистите и генерируйте новый ключ.
 itsLowValue = itsHighValue = intial;
 itsLowTime = itsHighTime = time;
 // вычислите новый ключ для хранения на основе
 // текущих данных и сохраните новую запись.
 itsStorageKey = calculateStorageKey(new Date());
 store()
}

private store()
{
 try
 {
 itsPI.store(itsStorageKey, this);
 }
 catch (StoreException)
 {
 // регистрация ошибки каким-либо образом.
 }
}

private String calculateStorageKey(Date d)
{
 SimpleDateFormat df = new SimpleDateFormat("MMddyyyy");
 return(itsType + "HiLo" + df.format(d));
}

private double itsLowValue;
private long itsLowTime;
private double itsHighValue;
private long itsHighTime;
private String itsType;
// следующие данные нежелательно сохранять.
transient private String itsStorageKey;
transient private api.PersistentImp itsPI;
}
```

Конечно, далеко не все в этом коде столь тривиально. Рассмотрим более подробно, какие действия здесь выполняются.

В нижней части класса можно заметить частные переменные-члены. Первые четыре переменные вполне прогнозируемы. Они хранят максимальное и минимальное значение, а также время фиксации. Переменная `itsType` запоминает тип

показания, сохраняемого с помощью `HiLoData`. Эта переменная имеет значение "Temp" для температуры, "BP" — для атмосферного давления, "DP" — для точки росы и т.д. Последние две переменные объявлены как транзитные (`transient`). Значит, они не сохраняются в постоянной (`persistent`) памяти. Они записывают текущий ключ хранения (`storage key`) и ссылаются на `PersistentImp`.

Конструктор располагает пятью аргументами. `StationToolkit` используется для получения доступа к `PersistentImp`. Аргументы `type` и `Date` применяются для формирования ключа хранения, который применяется для сохранения и восстановления объекта. Наконец, аргументы `init` и `initTime` применяются для инициализации объекта в том случае, если `PersistentImp` не обнаружит ключ хранения.

Конструктор пытается получить данные из `PersistentImp`. Если данные присутствуют, нетранзитивные данные копируются в свои переменные-члены. Затем с помощью начального значения и времени вызывается `currentReading`. Это позволяет удостовериться в том, что эти показания записаны. Наконец, если `currentReading` делает вывод об изменении максимального или минимального значения данных, возвращается значение `true`, а также вызывается функция `Store`, что позволяет удостовериться в обновлении постоянной памяти (`persistent memory`).

Метод `currentReading` находится в центре этого класса. Он сравнивает "старые" максимальное и минимальное значения с новыми, только что поступившими, показаниями. Если новое показание превышает "старое" максимальное или же является более низким, чем "старое" минимальное, величина заменяется, записывается соответствующее время, а изменения сохраняются в постоянной памяти.

Метод `newDay` вызывается в полночь. Во-первых, он сохраняет в постоянной памяти текущие значения `HiLoData`. Затем заново устанавливает значения `HiLoData` для начала нового дня. Ключ хранения вычисляется заново с учетом новых данных, и затем новые `HiLoData` сохраняются в постоянной памяти.

Функция `Store` просто применяет текущий ключ хранения для записи объекта `HiLoData` в постоянную память с помощью объекта `PersistentImp`.

Наконец, метод `calculateStorageKey` формирует ключ хранения на основе типа `HiLoData` и аргумента даты.

## Недостатки

Конечно, код из листинга 27.8 нельзя назвать трудным для понимания. Но есть в нем и определенные недостатки. Политика, реализованная с помощью функций `currentReading` и `newDay`, служит для обработки максимальных и минимальных значений данных и не зависит от постоянных значений (`independent of persistence`). С другой стороны, методы `store` и `calculateStorageKey`, конструктор и транзитивные переменные имеют специфическое отношение к по-

стостоянным значениям и не участвуют в обработке максимальных и минимальных значений, что является нарушением принципа SRP.

В текущем состоянии, при смешении различных объектов, данный класс может привести к сбоям в работе. Если в механизме постоянной памяти что-либо кардинально изменится, из-за чего функции `calculateStorageKey` и `store` потеряют свое положение, тогда новые возможности по реализации постоянной памяти могут повлиять на функционирование класса. Функции типа `newDay` и `currentReading` должны обязательно измениться для вызова новых возможностей постоянной памяти.

## Отделение постоянства от политики

Этих потенциальных проблем можно избежать, отделяя политику низкого уровня по обработке данных от механизма постоянства с помощью шаблона Proxy. Вернитесь к рис. 26.7. Обратите внимание на процесс разделения слоя политики (приложения) от слоя механизма (API).

На рис. 27.19 показано применение шаблона Proxy для выполнения отделения. Этот рисунок отличается от рис. 27.17 наличием класса `HiLoDataProxy`. Именно на этот прокси-класс сохраняет ссылки объект `TemperatureHiLo`. Прокси-класс, в свою очередь, сохраняет ссылки к объекту `HiLoDataImp` и делегирует к нему вызовы. Листинг 27.9 показывает реализацию критических функций как для `HiLoDataProxy`, так и для `HiLoDataImp`.

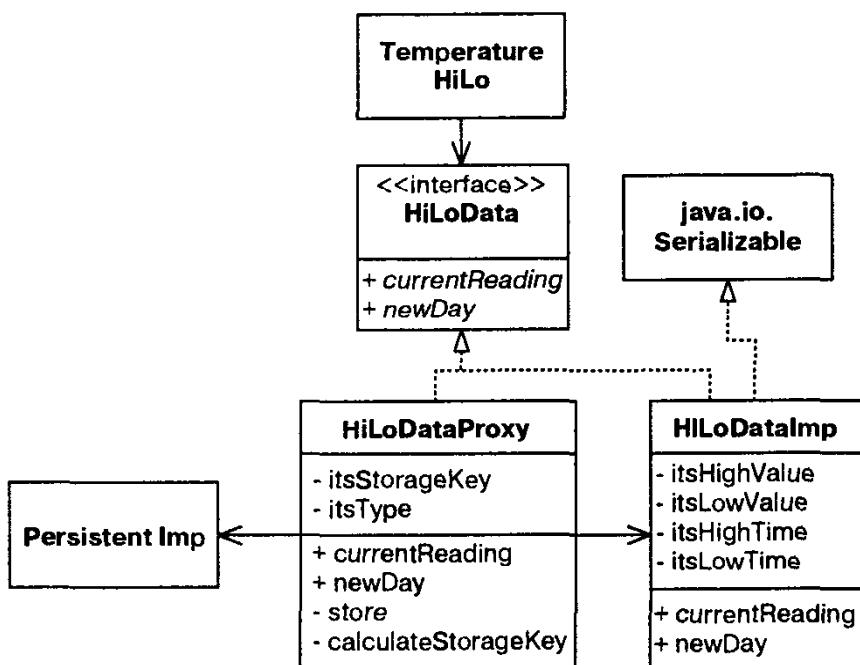


Рис. 27.19. Применение шаблона Proxy к постоянному HiLo

---

**Листинг 27.9. Фрагменты кода с применением Proxy**

---

```
class HiLoDataProxy implements HiLoData
{
 public boolean currentReading(double current, long time)
 {
 boolean change;
 change = itsImp.currentReading(current, time);
 if (change)
 store();
 return change;
 }

 public void newDay (double initial, long time)
 {
 store();
 itsImp.newDay(initial, time);
 calculateStorageKey(new Date(time));
 store();
 }

 private HiLoDataImp itsImp;
}

class HiLoDataImp implements HiLoData, java.io.Serializable
{
 public boolean currentReading(double current, long time)
 {
 boolean changed = false;
 if (current > itsHighValue)
 {
 itsHighValue = current;
 itsHighTime = time;
 changed = true;
 }
 else if (current < itsLowValue)
 {
 itsLowValue = current;
 itsLowTime = time;
 changed = true;
 }
 return changed;
 }

 public void newDay(double initial, long time)
 {
 itsHighTime = itsLowTime = time;
 itsHighValue = itsLowValue = initial;
```

Заметим, что класс `HiLoDataImp` обходится без сопровождения относительно принципа постоянства. Также отметим, что класс `HiLoDataProxy` проявляет “заботу” обо всех недочетах постоянной памяти и затем делегирует данные к `HiLoDataImp`, что весьма важно. Более того, заметим, каким образом прокси-класс зависит и от `HiLoDataImp` (уровень политики), и от `PersistentImp` (уровень механизма). Как раз это обстоятельство и вызывало обеспокоенность в процессе разработки. Опытный читатель обратит внимание на изменение, внесенное в метод `currentReading`. Этот метод возвращает значение `boolean`. Это значение необходимо использовать в прокси, чтобы последнее в нужный момент вызывало `store`. Почему `store` не вызывается всякий раз при вызове метода `currentReading`? Существует большое разнообразие видов NVRAM. Некоторые из них имеют верхний предел для количества обращений к ним для внесения записей. Чтобы “продлить жизнь” NVRAM, в ней сохраняются только те данные, которые соответствуют изменившимся значениям. Реальная жизнь требует компромиссных решений.

## Фабрики и инициализация

Ясно, что совершенно нежелательно, чтобы `TemperatureHiLo` имел сведения о прокси-объекте. Он содержит сведения только о `HiLoData`. (Рис. 27.19.) Пусть `HiLoDataProxy` создается для использования с объектом `TemperatureHiLo`. Также необходимо создать и `HiLoDataImp`, куда прокси делегирует данные.

Какой же способ следует избрать для создания объектов, если точно неизвестно, к какому типу относятся создаваемые объекты? Нужно создать `HiLoData` для `TemperatureHiLo`, причем не используя сведений о том, что в действительности создается `HiLoDataProxy` и `HiLoDataImp`. Снова обращаемся к шаблону `Factory` (рис. 27.20).

Класс `TemperatureHiLo` применяет интерфейс `DataToolkit` для создания объекта, согласованного с интерфейсом `HiLoData`. Метод `getTempHiLoData` развертывается к объекту `DataToolkitImp`, который, в свою очередь, создает `HiLoDataProxy`. Типом кода `HiLoDataProxy` служит “`Temp`”, и данные возвращаются как `HiLoData`.

Подобный прием прекрасно решает проблему формирования. `TemperatureHiLo` не нуждается в зависимости от `HiLoDataProxy` в процессе создания. Но каким же образом `TemperatureHiLo` получит доступ к объекту `DataToolkitImp`? Нежелательно, чтобы `TemperatureHiLo` располагал какими-либо сведениями об `DataToolkitImp`, поскольку это породит зависимость уровня политики от уровня механизма.

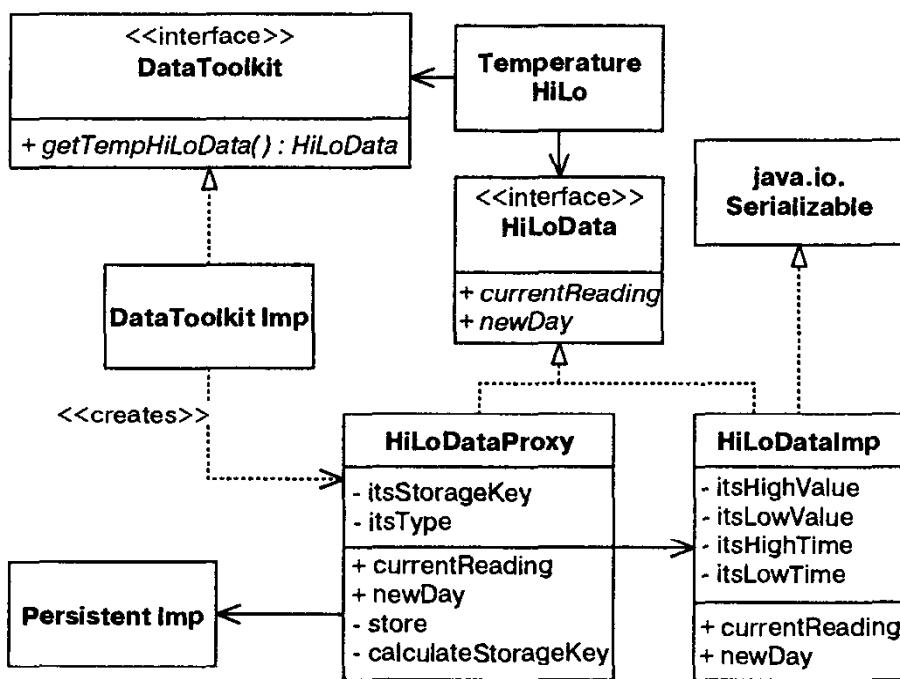


Рис. 27.20. Применение абстрактной фабрики для создания прокси-объекта

## Структура пакетов

Для ответа на этот вопрос рассмотрим пакетную структуру, показанную на рис. 27.21. Сокращение WMS означает Weather Monitoring System, который описан на рис. 27.16.

При просмотре рис. 27.21 можно заметить принудительную зависимость уровня с постоянным интерфейсом от уровней с механизмами. Также показано, каким образом классы разворачиваются в пакеты. Обратите внимание, что абстрактный фактор, **DataToolkit**, определяется в пакете **WMSData** наряду с **HiLoData**.

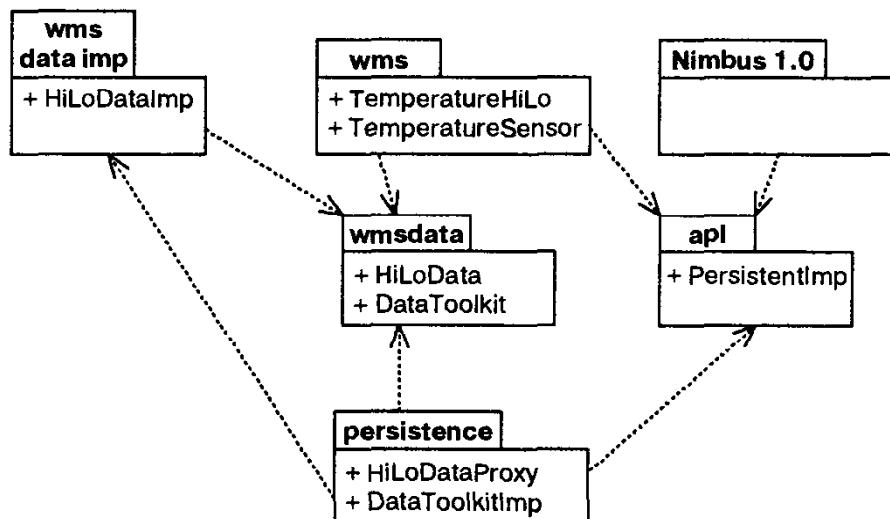


Рис. 27.21. Пакетная структура Proxy и Factory

HiLoData реализуется в пакете MMS-DataImp, в то время как DataToolkit реализуется в пакете persistence.

## Кто создает фабрику?

Теперь заново зададимся этим вопросом. Каким образом экземпляр wms.TemperatureHiLo получает доступ к экземпляру persistence.DataToolkitImpl, может вызывать метод getTempHiLoData и создавать экземпляры persistence.HiLoDataProxy?

Необходимо использовать статическую переменную, достижимую для классов в wmsdata, которая объявляется для сохранения wmsdata. DataToolkit, но инициализируется для хранения persistence.DataToolkitImpl. Поскольку все переменные в Java, включая статические переменные, должны объявляться в некотором классе, можно создать класс под названием Scope, содержащий необходимые нам переменные. Этот класс помещается в пакет wmsdata.

В листингах 27.10 и 27.11 показано, как все это функционирует. Класс Scope в wmsdata объявляет статическую переменную-член, содержащую ссылку DataToolkit. Класс Scope в пакете persistence объявляется функцию init(), которая создает экземпляр DataToolkitImpl и сохраняет его в переменной wmsdata.Scope.itsDataToolkit.

---

### Листинг 27.10. wmsdata.Scope

---

```
package wmsdata;
public class Scope
{
 public static DataToolkit itsDataToolkit;
```

---

---

### Листинг 27.11. persistence.Scope

---

```
package persistence;
public class Scope
{
 public static void init()
 {
 wmsdata.Scope.itsDataToolkit =
 new DataToolkit();
 }
}
```

---

Между пакетами и классами scope наблюдается интересная симметрия. Все классы в пакете wmsdata, отличные от Scope, являются интерфейсами, включающими абстрактные методы и не содержащими переменных. Но класс

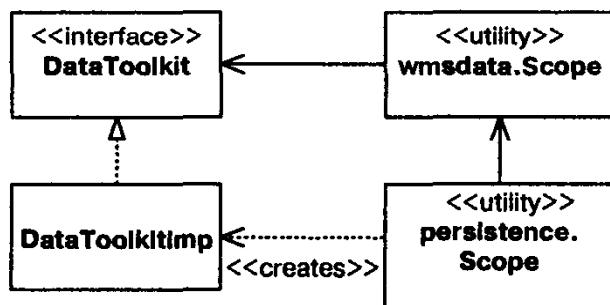


Рис. 27.22. Утилиты Scope

`wmsdata.Scope` включает переменную и не содержит функций. С другой стороны, все классы в пакете `persistence`, отличные от `Scope`, являются конкретными классами, содержащими переменные. Но `persistence.Scope` имеет функцию и не содержит переменных.

На рис. 27.22 показано, каким образом можно отобразить все это на диаграмме класса. Классы `Scope` являются классами “утилит”. Все члены этих классов, являются ли они переменными или функциями, статичны, т.е. являются заключительным элементом в симметрии. Оказывается, что пакеты, содержащие абстрактные интерфейсы, обычно включают утилиты с данными, но без функций, в то время как пакеты, содержащие конкретные классы, обычно включают утилиты с функциями, но без данных.

### Итак, кто обращался к `persistence.Scope.init()`?

#### Возможно, функция `main()`

Класс, содержащий основную функцию, должен находиться в пакете, который поддерживает зависимость от `persistence`. Часто вызывается пакет, который включает основной пакет `root`.

#### Однако, можно заметить...

Уровень постоянства должен зависеть от уровня политики. Но при внимательном рассмотрении рис. 27.21 можно заметить зависимость между `persistence` и `wmsDataImp`. Эта зависимость просматривается и на рис. 27.20, где `HiLoDataProxy` зависит от `HiLoDataImp`. Причина этой зависимости кроется в том, что `HiLoDataProxy` может создавать зависящий от него `HiLoDataImp`.

В большинстве случаев прокси-объект не создает дочерние объекты, поскольку выполняет их поиск в постоянном хранилище. То есть `HiLoDataImp` возвращается к прокси-объекту при вызове `PersistentImp.retrieve`. Но в тех редких случаях, когда функция восстановления не обнаруживает объект в постоянном хранилище, `HiLoDataProxy` должна создавать пустой `HiLoDataImp`.

Итак, похоже, что необходима другая фабрика, которой известно, как создаются экземпляры `HiLoDataImp` и которая может быть вызвана прокси. Значит, появляется больше пакетов и классов `Scope` и т.д.

## Так ли это необходимо?

Возможно, что в этом случае такой необходимости нет. Создается фабрика для прокси-объекта, поскольку желательно, чтобы `TemperatureHiLo` функционировал с большим числом различных механизмов. В результате получаем преимущество, состоящее в обоснованном использовании фабрики `DataToolkit`. Но какой смысл в расположении фабрики между `HiLoDataProxy` и `HiLoDataImpl`? Если возможно осуществить много различных реализаций `HiLoDataImpl` и необходимо, чтобы прокси-объект функционировал со всеми, тогда следует обратиться к обоснованию.

Но нет уверенности в том, что требования не изменятся. Пакет `wmsDataImpl` включает политики по отслеживанию погодных условий и бизнес-правила, которые должны оставаться неизменными некоторое время. Кажется неправдоподобным, чтобы они часто изменялись в дальнейшем. Необходимо определиться с выбором позиции. В данном случае мы пришли к выводу, что зависимость между прокси-объектом и дочерним объектом не представляет большого риска, и поэтому решили обойтись без использования фабрики.

## Резюме

Джим Ньюкирк (Jim Newkirk) и автор данной книги написали эту главу в начале 1998 года. Джим Ньюкирк выполнял кодирование, автор транслировал код в UML-диаграммы и дополнял их описаниями. Объем кода значительно увеличился. На основе этого кода и создавалась глава. Большинство диаграмм сформировано после завершения кода.

В 1998 не было известно об экстремальном программировании. Поэтому описанная здесь разработка не рассматривалась в среде парного программирования и разработки, управляемой тестами. Плодом успешного сотрудничества авторов и явился этот проект. Внимательно прослеживался код, вносились изменения, формировались UML-диаграммы.

## Литература

1. Gamma и др . *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
2. Meyer B. *Object-oriented Software Construction*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997.
3. Arnold K., and Gosling J. *The Java Programming Language*, 2nd ed. Reading, MA: Addison-Wesley, 1998.

## Обзор требований к **Nimbus-LC**

### Требования к применению

Данная система поддерживает автоматический мониторинг различных погодных условий. В частности, измеряются следующие переменные значения:

- скорость ветра и его направление;
- температура;
- атмосферное давление;
- относительная влажность;
- степень охлаждения ветром;
- температура точки росы.

Система также поддерживает отслеживание текущего тренда показаний атмосферного давления, которое может быть стабильным, повышенным и пониженным. Например, атмосферное давление, равное 29,95 дюймам ртутного столба (inches of mercury, IOM) является пониженным.

Система имеет дисплей, где непрерывно отображаются все измерения, а также текущее время и дата.

### 24-часовая история

С помощью сенсорного экрана (touch screen) пользователь может отображать предысторию любого из следующих измерений за последние 24 часа:

- температура;
- атмосферное давление;
- относительная влажность.

История изменения этих данных представляется в виде линейного графика (рис. 27.23).

### Установка пользователем

Система поддерживает следующие возможности, разрешающие конфигурирование станции в процессе инсталляции:

- установка текущего времени, даты и времени зоны;
- установка единиц измерения (английская или метрическая).

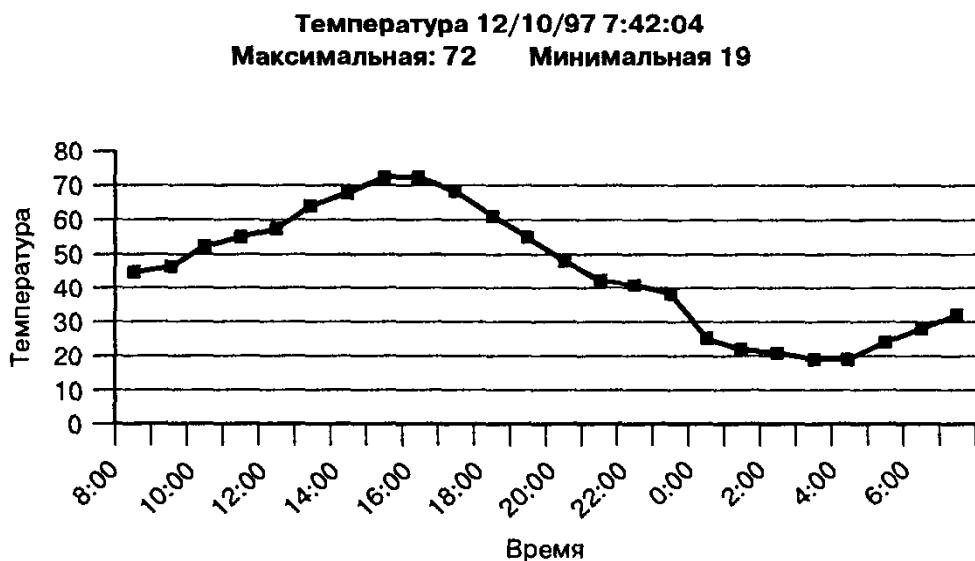


Рис. 27.23. Изменения показаний температуры

## Административные требования

Система поддерживает механизм по обеспечению безопасности, позволяющий получить доступ к административным функциям метеорологической станции. Эти функции включают следующее:

- калибровка датчиков с учетом известных значений;
- повторная установка станции.

## Примеры применения Nimbus-LC

### Исполнители

В этой системе пользователи играют две роли.

### Пользователь

Пользователь просматривает полученные станцией данные о погодных условиях в реальном времени. В результате взаимодействия с системой также отображаются предварительные данные, ассоциированные с отдельными датчиками.

### Администратор

Администратор контролирует соблюдение мер безопасности системы, отвечает за калибровку отдельных датчиков, устанавливает время/дату, уточняет единицы измерения и заново устанавливает станцию (в случае необходимости).

## **Варианты использования**

### **Вариант использования #1. Мониторинг погодных условий**

Система отображает текущие значения температуры, атмосферного давления, относительной влажности, скорости ветра, направления ветра, температуру охлаждения ветром, точку росы и тренд атмосферного давления.

## **История измерений**

Система отображает линейный график, где отмечаются показания датчиков системы за последние 24 часа. Кроме графика, система отображает текущее время и дату, а также максимальное и минимальное значение параметров за последние 24 часа.

### **Вариант использования #2. Обзор изменений температуры**

### **Вариант использования #3. Обзор изменений атмосферного давления**

### **Вариант использования #4. Обзор изменений относительной влажности**

## **Установка**

### **Вариант использования #5. Выбор единиц измерения**

Пользователь устанавливает тип отображаемых единиц измерения. Можно выбрать между английской и метрической системами. По умолчанию используется метрическая система.

### **Вариант использования #6. Установка даты**

Пользователь устанавливает текущую дату.

### **Вариант использования #7. Установка времени**

Пользователь устанавливает текущее время, а также зону времени для системы.

## **Администрирование**

### **Вариант использования #8. Повторная установка метеостанции**

Администратор имеет возможность повторно устанавливать на станции заданные по умолчанию установки. Важно отметить, что при этом удаляются все данные по предыстории, сохраняемые на станции, а также устраняется калибровка. Администратор информируется об этих последствиях, и поступает запрос о продолжении процесса вторичной установки станции.

### **Вариант использования #9. Калибровка датчика температуры**

Администратор, используя хорошо известный источник данных о температуре, вводит значение в систему. Система получает доступ к этому значению и использует его для калибровки. Действительное показание сопоставляется с показаниями текущих измерений. Подробно процесс калибровки датчиков рассмотрен в документе, описывающем аппаратное обеспечение.

- Вариант использования #10. Калибровка датчика атмосферного давления
- Вариант использования #11. Калибровка датчика относительной влажности
- Вариант использования #12. Калибровка датчика скорости ветра
- Вариант использования #13. Калибровка датчика направления ветра
- Вариант использования #14. Калибровка датчика точки росы
- Вариант использования #15. Журнал регистрации калибровок

Система показывает администратору историю процессов калибровки единиц измерения. Эта история включает указание времени и даты калибровки, название датчика, для которого выполнялась калибровка, а также значение, примененное для калибровки датчика.

## План выпуска версий Nimbus-LC

### Введение

Реализация проекта метеорологической станции выполняется с помощью набора итераций. Каждая итерация базируется на том, что сделано до начала поддержки функциональных возможностей, необходимых для выпуска заказанной версии. Этот документ выделяет три версии этого проекта.

### Версия I

При разработке этой версии преследовались две цели. Во-первых, создается архитектура, поддерживающая основу приложения таким образом, чтобы оно оставалось независимым от аппаратной платформы Nimbus. Во-вторых, контролируются значения двух самых крупных рисков.

1. Применение устаревших Nimbus 1.0 API для платы процессора под управлением новой операционной системы. Это выполнимо, но довольно трудно предвидеть, сколько времени займет этот процесс, поскольку трудно предугадать все затруднения.
2. Java Virtual Machine. До сих пор JVM не использовалась совместно со встроенными платами. Неизвестно, будет ли она функционировать с нашей операционной системой и будут ли корректно реализованы все байтовые коды Java. В подобной ситуации следует учитывать немалый риск.

Интеграция JVM с применением сенсорного экрана и графических подсистем проходит параллельно с формированием данной версии. Ожидается, что оно полностью завершится перед началом второй фазы.

### Риски

1. Обновление операционной системы. В настоящее время применяется более старая версия этой ОС. Чтобы применять JVM, следует выполнить обновле-

ние с учетом самой последней версии OS. Также следует применять самую последнюю версию инструментов разработки.

2. Поставщики ОС поддерживают для этой версии OS самую последнюю версию JVM. Чтобы учесть реалии, необходимо применять версию 1.2 для JVM. Но VI.2 сейчас проходит тестирование, и во время реализации проекта наверняка изменится.
3. Собственный интерфейс Java на уровне платы "С" API должен проверяться в новой архитектуре.

## **Поставляемые модули**

1. Наше аппаратное обеспечение реализует новую ОС наряду с последней версией JVM.
2. Потоковый вывод, отображающий текущие показания температуры и атмосферного давления (*устраненный код не используется в заключительной версии*).
3. При наличии изменений в атмосферном давлении система информирует нас о тренде для этих данных - является ли давление повышенным, пониженным или стабильным.
4. Каждый час система отображает значения температуры и атмосферного давления, зафиксированные за последние 24 часа. Эти данные являются постоянными, так что можно поместить их в цикл и сохранить.
5. Каждый полдень (12:00 А.М.) система отображает максимальное и минимальное значение температуры и атмосферного давления.
6. Все измерения выполняются в метрической системе мер и весов.

## **Версия II**

Во время фазы проектирования базис для пользовательского интерфейса добавляется к первой версии. Никакие дополнительные измерения не вносятся. Единственным изменением в процесс измерений служит добавление калибровочного механизма. На этой фазе основное внимание уделяется презентации системы. Основной риск представляет интерфейс программного обеспечения для жидкокристаллического/сенсорного экрана. Кроме того, поскольку в этой версии впервые интерфейс пользователя отображается в таком же виде, как его видит пользователь, можно сделать небольшую коррекцию требований. Для нового аппаратного обеспечения также можно предусмотреть спецификацию. Именно по этой причине на этой фазе проекта добавляется калибровка. Это API определяется в Java.

## Реализованные варианты использования

- #2: просмотр предыстории изменений температуры;
- #3: просмотр предыстории изменений атмосферного давления;
- #5: установка единиц измерения;
- #6: установка даты;
- #7: установка времени/временной зоны;
- #9: калибровка датчика температуры;
- #10: калибровка датчика атмосферного давления.

## Риски

1. Интерфейс жидкокристаллического/сенсорного экрана для виртуальной машины Java (virtual machine) необходимо протестировать с помощью реального аппаратного обеспечения.
2. Изменения требований.
3. Изменения в JVM, наряду с изменениями в фундаментальных классах Java, обрабатываются в предварительной и в окончательной формах.

## Поставляемые модули

1. Система, выполняющая и обеспечивающая все функциональные возможности, указанные в приведенных выше примерах применения.
2. Также реализуются компоненты примера применения #1, имеющие отношение к температуре, атмосферному давлению и времени/дате.
3. Часть программной архитектуры, относящаяся к GUI, завершается как часть этой фазы.
4. Административная часть программного обеспечения реализуется для поддержки калибровок показаний температуры и атмосферного давления.
5. Спецификация для нового аппаратного обеспечения API указывается с использованием Java вместо C.

## Версия III

Эта версия предваряет развертывание готового продукта у заказчика.

## Реализованные варианты использования

- #1: отслеживание погодных условий;
- #4: просмотр данных по изменениям относительной влажности;

- #8: повторная установка метеорологической станции;
- #11: калибровка датчика относительной влажности;
- #12: калибровка датчика скорости ветра;
- #13: калибровка датчика направления ветра;
- #14: калибровка датчика точки росы;
- #15: журнал регистрации калибровок.

## Риски

1. Изменения требований. По завершении большей части работы над продуктом могут потребоваться некоторые изменения.
2. После завершения всей работы над продуктом может возникнуть необходимость в изменениях аппаратного обеспечения API (речь о которых шла в конце описания версии II).
3. Ограничения на аппаратное обеспечение. По завершении разработки программного продукта следует учесть ограничения, связанные с аппаратным обеспечением (память, ЦПУ и т.д.).

## Поставляемые модули

1. Новое ПО, выполняемое на старой аппаратной платформе.
2. Спецификация для нового аппаратного обеспечения, которая удостоверяется в данной реализации.

# ЧАСТЬ VI

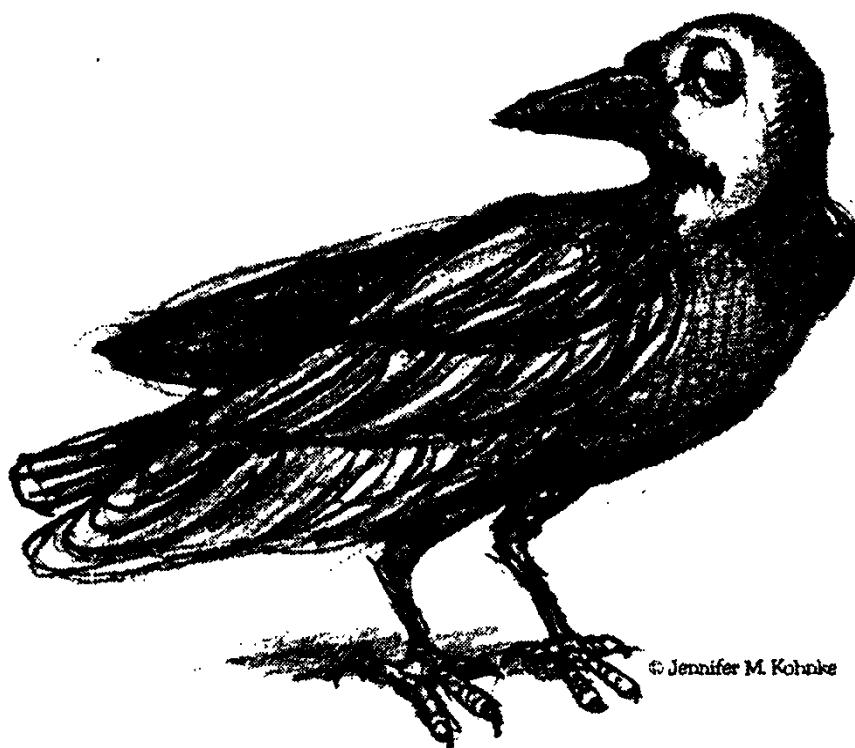
## Практическое занятие: система ETS

Для получения в США или Канаде лицензии на право ведения архитектурных работ необходимо сдать экзамен. Если он будет успешно сдан, государственная комиссия (licensing board) предоставит вам лицензию на выполнение практических работ по архитектуре. Экзаменационные задания разрабатываются службой Educational Testing Service (ETS) под эгидой совета National Council of Architectural Registration Boards (NCARB) и в настоящее время администрируются представителями Chauncey Group International.

В прошлом кандидаты выполняли экзаменационные задания с помощью карандаша и бумаги. Компетентное жюри рассматривало предложенные проекты. Жюри состояло из опытных архитекторов, которые выносили свое заключение о результатах экзамена после внимательного изучения предложенных материалов.

В 1989 NCARB обратилась к ETS за заключением, возможно ли хотя бы частично автоматизировать процедуру экзамена. Представленные в этом разделе главы частично описывают заключительный проект. Как и раньше, при разработке этого программного обеспечения вы познакомитесь с большим числом полезных шаблонов, поэтому главы, описывающие эти шаблоны, предшествуют практическому занятию.

## Шаблон Visitor



**Проблема.** К иерархии классов необходимо добавить новый метод, но в процессе добавления наносится вред всему проекту.

Данная проблема встречается довольно часто. Например, рассмотрим иерархию объектов `Modem`. Основной класс располагает методами генерирования, общими для всех модемов. Производные модули представляют драйверы, относящиеся к большому числу различных производителей и типов модемов. Также предположим, что к иерархии необходимо добавить новый метод под названием `configureForUnix`. Этот метод позволяет конфигурировать модем для работы с операционной системой UNIX. Каждый производный модуль класса модема обрабатывается иным образом, поскольку каждый модем имеет свои особенности при установках конфигураций и работе с UNIX.

К сожалению, добавление метода `configureForUnix` порождает много вопросов. Как ведет себя Windows? Что можно сказать о поведении MacOs? Как

это отразится на функционировании Linux? Следует ли добавлять новый метод к иерархии Modem при использовании любой новой операционной системы? Ясно, что это не так. В противном случае нам не удалось бы закрыть интерфейс Modem. Всякий раз, когда речь заходит о новой операционной системе, необходимо изменять данный интерфейс и заново разворачивать все программное обеспечение модема.

## Семейство Visitor в шаблонах проектирования

*Семейство Visitor позволяет добавлять к существующим иерархиям новые методы без обновления этих иерархий. Ниже приводится перечень шаблонов из этого семейства:*

- Visitor;
- Acyclic Visitor;
- Decorator;
- Extension object.

## Шаблон Visitor<sup>1</sup>

Рассмотрим иерархию Modem, представленную на рис. 28.1. Интерфейс Modem включает методы генерирования, которые можно реализовать для любых модемов. Здесь показаны три производных модуля: один управляет модемом Hayes, другой — модемом Zoom, а третий — модемом Ernie, разработанным одним из наших инженеров-электронщиков.

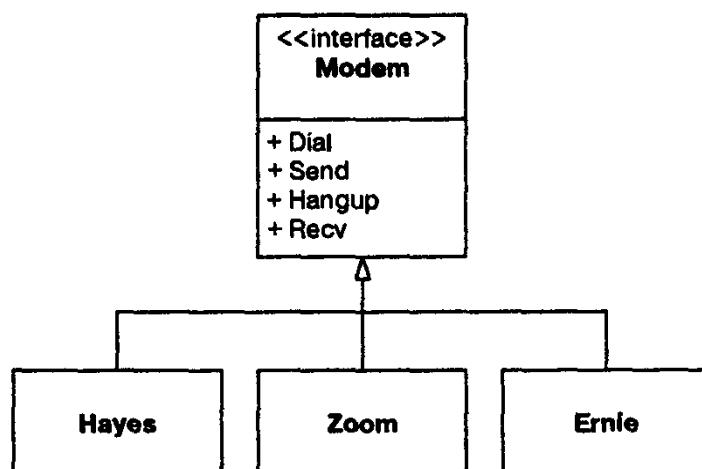


Рис. 28.1. Иерархия Modem

Каким образом можно сконфигурировать эти модемы для работы с UNIX, не помещая метод ConfigureForUnix в интерфейс Modem? Можно воспользоваться

<sup>1</sup>[GOF95], с. 331.

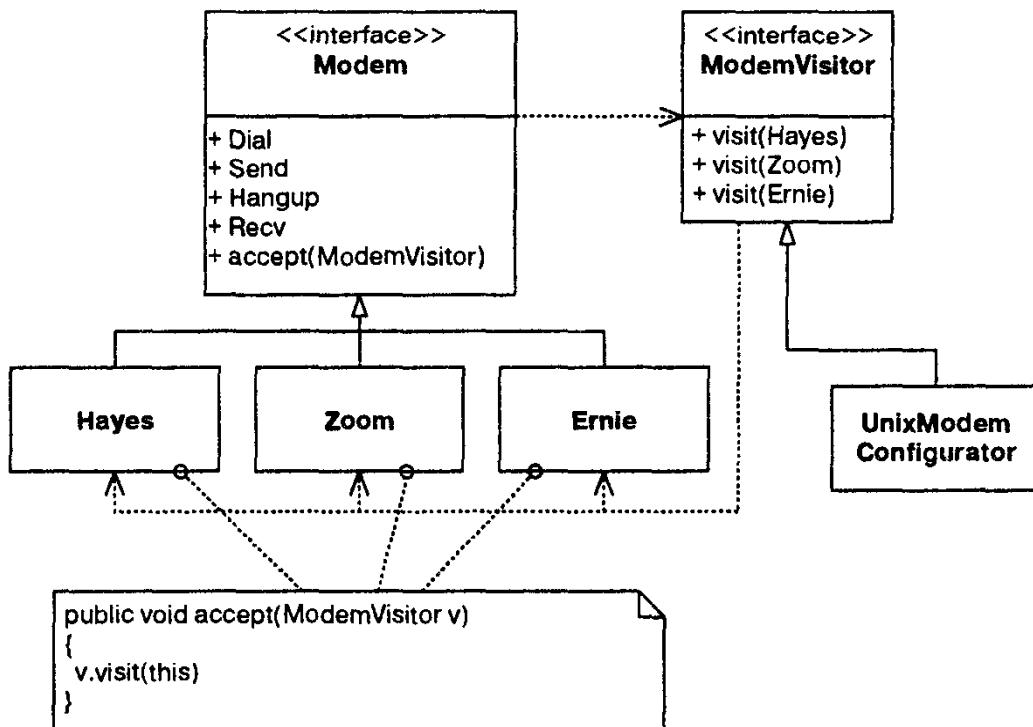


Рис. 28.2. Шаблон Visitor

ваться методикой под названием *дуальная отсылка (dual dispatch)*, которая составляет основу шаблона **Visitor**.

На рис. 28.2 показана структура шаблона **Visitor**, а листинги 28.1–28.6 отображают соответствующий код на языке Java. Листинг 28.7 демонстрирует тестовый код, который удостоверяет работу шаблона **Visitor** и в то же время демонстрирует методы повторного применения.

#### Листинг 28.1. Modem.java

```

public interface Modem
{
 public void dial(String pno);
 public void hangup();
 public void send(char c);
 public char recv();
 public void accept(ModemVisitor v);
}

```

#### Листинг 28.2. ModemVisitor.java

```

public interface ModemVisitor
{
 public void visit(HayesModem modem);
 public void visit(ZoomModem modem);
 public void visit(ErnieModem modem);
}

```

---

**Листинг 28.3. HayesModem.java**

---

```
public class HayesModem implements Modem
{
 public void dial(String pno){}
 public void hangup(){}
 public void send(char c){}
 public char recv() {return 0;}
 public void accept(ModemVisitor v) {v.visit(this);}

 String configurationString = null;
}
```

---

---

**Листинг 28.4. ZoomModem.java**

---

```
public class ZoomModem implements Modem
{
 public void dial(String pno){}
 public void hangup(){}
 public void send(char c){}
 public char recv() {return 0;}
 public void accept(ModemVisitor v) {v.visit(this);}

 int configurationValue = 0;
}
```

---

---

**Листинг 28.5. ErnieModem.java**

---

```
public class ErnieModem implements Modem
{
 public void dial(String pno){}
 public void hangup(){}
 public void send(char c){}
 public char recv() {return 0;}
 public void accept(ModemVisitor v) {v.visit(this);}

 String internalPattern = null;
}
```

---

---

**Листинг 28.6. UnixModemConfigurator.java**

---

```
public class UnixModemConfigurator implements ModemVisitor
{
 public void visit(HayesModem m)
 {
 m.configurationString = "&sl=4&D=3";
 }

 public void visit(ZoomModem m)
}
```

```
{
 m.configurationValue = 42;
}

public void visit(ErnieModem m)
{
 m.internalPattern = "C is too slow";
}
}
```

---

#### Листинг 28.7. TestModemVisitor.java

```
import junit.framework.*;
public class TestModemVisitor extends TestCase
{
 public TestModemVisitor(String name)
 {
 super(name);
 }

 private UnixModemConfigurator v;
 private HayesModem h;
 private ZoomModem z;
 private ErnieModem e;

 public void setUp()
 {
 v = new UnixModemConfigurator();
 h = new HayesModem();
 z = new ZoomModem();
 e = new ErnieModem();
 }

 public void testHayesForUnix()
 {
 h.accept(v);
 assertEquals("&sl=4&D=3", h.configurationString);
 }

 public void testZoomForUnix()
 {
 z.accept(v);
 assertEquals(42, z.configurationValue);
 }

 public void testErnieForUnix()
 {
 e.accept(v);
 assertEquals("C is too slow", e.internalPattern);
 }
```

Обратите внимание, что в иерархии `Visitor` имеется метод для каждого производного модуля посещаемой иерархии (`Modem`). В данном случае идет речь о повороте на 90° — от производных модулей к методам.

Данный тестовый код показывает, что при конфигурировании модема для среды UNIX программист создает экземпляр класса `UnixModemConfigurator` и передает его функции `accept` из `Modem`. Соответствующий производный модуль `Modem` затем вызывает `visit (this)` для `ModemVisitor`, базового класса `UnixModemConfigurator`. Если данным производным модулем является `Hayes`, тогда `visit (this)` вызывает `public void visit (Hayes)`. Это приведет к развертыванию функции `public void visit (Hayes)` в `UnixModemConfigurator`, которая затем сконфигурирует модем `Hayes` для Unix.

После построения этой структуры и при добавлении новых функций по конфигурированию операционной системы, к `ModemVisitor` добавляются новые производные модули, причем иерархия `Modem` при этом не изменяется. Итак, шаблон `Visitor` заменяет производные модули `ModemVisitor` на методы иерархии `Modem`.

Этот процесс и называется дуальной отсылкой (dual dispatch), поскольку выполняются две полиморфные отсылки. Первая связана с функцией `accept`. В результате отсылки определяется тип объекта, вызываемого функцией `accept`. Вторая отсылка представлена методом `visit`, который определяется для выполнения определенной функции. Эти две отсылки позволяют довольно быстро реализовать `Visitor`.

## Представление шаблона `Visitor` в виде матрицы

В результате двух отсылок шаблона `Visitor` образуется матрица функций. В примере, связанном с модемом, в строках матрицы представлены различные типы модемов. В столбцах определяются различные типы операционных систем. Каждая ячейка матрицы заполнена функцией, описывающей процесс инициализации определенного модема для некоторой операционной системы.

## Шаблон Acyclic `Visitor`

Заметим, что базовый класс посещенной (visited) иерархии (`Modem`) зависит от базового класса иерархии `Visitor` (`ModemVisitor`). Также отметим, что базовый класс иерархии `Visitor` имеет функцию для каждого производного модуля посещенной иерархии. Поэтому существует цикл зависимостей, связывающих вместе все производные посещенные модули (все `Modems`). Вследствие этого довольно сложно компилировать структуру `visitor` путем приращений или добавлять к посещенной иерархии новые производные модули.

Шаблон *Visitor* отлично функционирует в программах, где обновляемой иерархии нечасто требуются новые производные модули. Если *Hayes*, *Zoom* и *Ernie* являются единственными производными модулями *Modem*, что обычно желательно, или новые производные модули *Modem* практически не взаимодействуют, применение *Visitor* вполне уместно.

С другой стороны, если иерархия обладает изменчивым характером, что требует создания большого количества новых производных модулей, базовый класс *Visitor* (т.е. *Modemvisitor*) будет обновляться и повторно компилироваться наряду со всеми производными модулями всякий раз, когда к посещенной иерархии добавляется новый производный модуль. При работе в C++ складывается худшая ситуация. Целая посещенная иерархия должна проходить повторную компиляцию и развертываться заново при добавлении любого нового производного модуля.

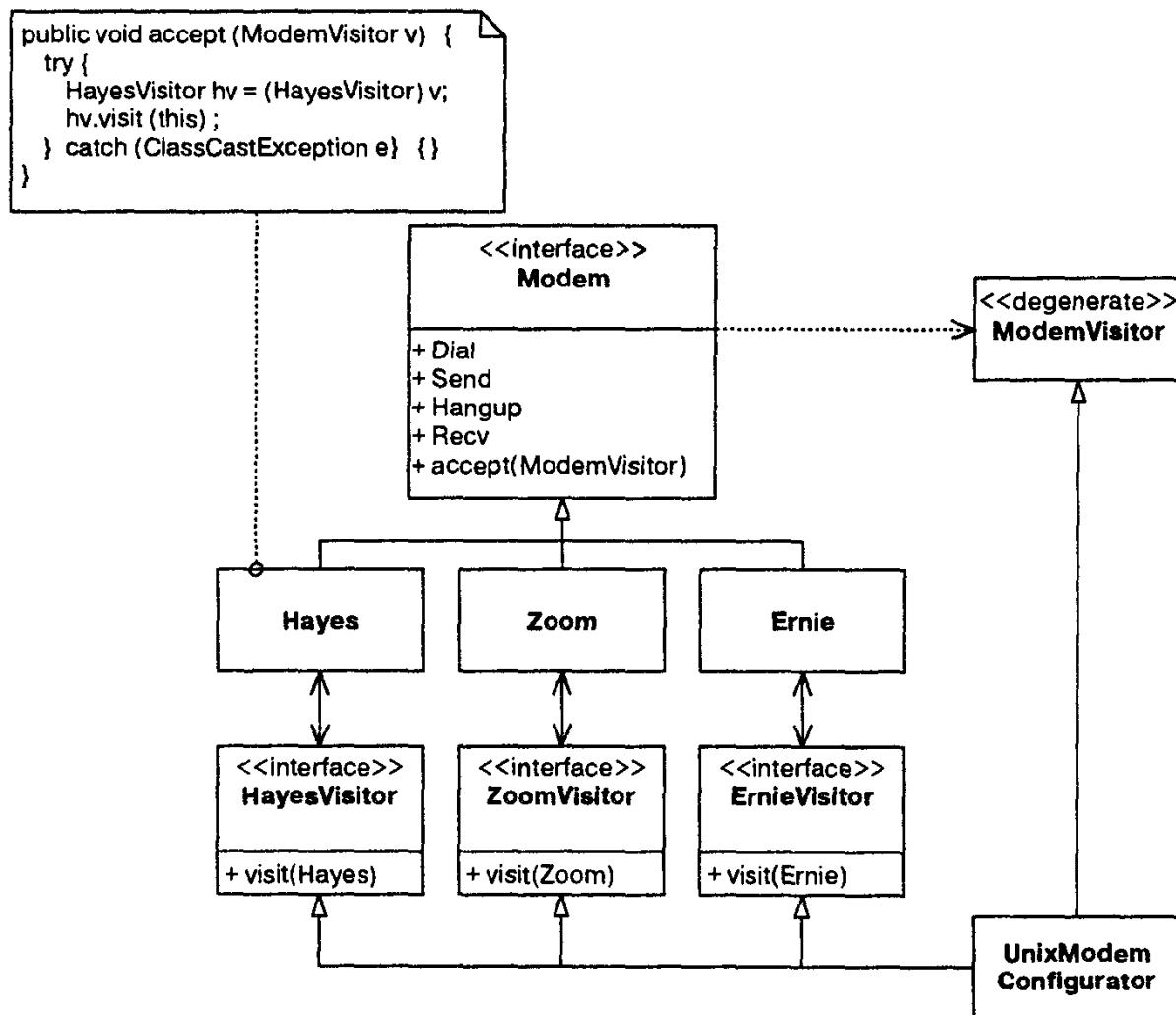


Рис. 28.3. Шаблон Acyclic Visitor

Для решения этой проблемы применяется вариация шаблона, известная под названием *Acyclic Visitor*<sup>2</sup> (рис. 28.3.) Эта вариация прерывает цикл зави-

<sup>2</sup>[PLOPD3], с. 93.

симости, что приводит к вырождению базового класса `Visitor` (`Modemvisitor`)<sup>3</sup>. Поскольку в этом методе довольно мало классов, базовый класс становится независимым от производных модулей посещенной иерархии.

Производные модули `Visitor` также порождаются интерфейсами `Visitor`. Для каждого производного модуля посещенной иерархии предусмотрено по одному интерфейсу `Visitor`. А это означает поворот на 180° от производных классов к интерфейсам. Функции `accept` в производных посещенных модулях направляют базовый класс<sup>4</sup> `Visitor` к соответствующему интерфейсу `Visitor`. Если все проходит успешно, метод вызывает соответствующую функцию `visit`. Фрагменты соответствующего кода приведены в листингах 28.8–28.16.

---

#### Листинг 28.8. Modem.java

---

```
public interface Modem
{
 public void dial(String pno);
 public void hangup();
 public void send(char c);
 public char recv();
 public void accept(ModemVisitor v);
}
```

---



---

#### Листинг 28.9. ModemVisitor.java

---

```
public interface ModemVisitor
{
}
```

---



---

#### Листинг 28.10. ErnieModemVisitor.java

---

```
public interface ErnieModemVisitor
{
 public void visit(ErnieModem m);
}
```

---



---

#### Листинг 28.11. HayesModemVisitor.java

---

```
public interface HayesModemVisitor
{
 public void visit(HayesModem m);
}
```

---

<sup>3</sup>Вырожденный класс вообще не содержит методов. При работе в C++ он является чисто виртуальным деструктором. При работе в Java подобные классы называют "маркерами интерфейсов".

<sup>4</sup>При работе в C++ используется `dynamic_cast`.

---

**Листинг 28.12. ZoomModemVisitor.java**

---

```
public interface ZoomModemVisitor
{
 public void visit(ZoomModem m);
}
```

---

---

**Листинг 28.13. ErnieModem.java**

---

```
public class ErnieModem implements Modem
{
 public void dial(String pno){}
 public void hangup(){}
 public void send(char c){}
 public char recv() {return 0;}
 public void accept(ModemVisitor v)
 {
 try
 {
 ErnieModemVisitor ev = (ErnieModemVisitor)v;
 ev.visit(this);
 }
 catch (ClassCastException e)
 {
 }
 }
 String internalPattern = null;
}
```

---

---

**Листинг 28.14. HayesModem.java**

---

```
public class HayesModem implements Modem
{
 public void dial(String pno){}
 public void hangup(){}
 public void send(char c){}
 public char recv() {return 0;}
 public void accept(ModemVisitor v)
 {
 try
 {
 HayesModemVisitor hv = (HayesModemVisitor)v;
 hv.visit(this);
 }
 catch (ClassCastException e)
 {
 }
 }
}
```

---

```
 String configurationString = null;
}
```

---

**Листинг 28.15. ZoomModem.java**

---

```
public class ZoomModem implements Modem
{
 public void dial(String pno){}
 public void hangup(){}
 public void send(char c){}
 public char recv() {return 0;}
 public void accept(ModemVisitor v)
 {
 try
 {
 ZoomModemVisitor zv = (ZoomModemVisitor)v;
 zv.visit(this);
 }
 catch(ClassCastException e)
 {
 }
 }
 int configurationValue = 0;
}
```

---

---

**Листинг 28.16. TestModemVisitor.java**

---

```
import junit.framework.*;
public class TestModemVisitor extends TestCase
{
 public TestModemVisitor(String name)
 {
 super(name);
 }

 private UnixModemConfigurator v;
 private HayesModem h;
 private ZoomModem z;
 private ErnieModem e;

 public void setUp()
 {
 v = new UnixModemConfigurator ();
 h = new HayesModem();
 z = new ZoomModem();
 e = new ErnieModem();
 }

 public void testHayesForUnix()
```

```

{
 h.accept(v);
 assertEquals("&s1=4&D=3", h.configurationString);
}

public void testZoomForUnix()
{
 z.accept(v);
 assertEquals(42, z.configurationValue);
}

public void testErnieForUnix()
{
 e.accept(v);
 assertEquals("C is too slow", e.internalPattern);
}
}

```

---

Таким образом, прерывается цикл зависимости и облегчается добавление производных посещенных модулей, а также выполнение компиляций путем приращений. К сожалению, все это значительно усложняет решение. Хуже того, время, затрачиваемое на изменение направленности, зависит от посещенной иерархии, и с трудом поддается оцениванию.

Кроме того, для систем, функционирующих в режиме реального времени, большой и трудно прогнозируемый объем времени, затрачиваемый на изменение направления, не позволяет применить шаблон **Acyclic Visitor**. При работе с другими системами недостатком является сложность данного шаблона. Но для систем с переменной посещенной иерархией, для которых большое значение имеет компиляция с приращением, подобный шаблон вполне уместен.

## **Шаблон Acyclic Visitor подобен разреженной матрице**

Подобно тому, как шаблон **Visitor** создает матрицу функций, с типом **visited** в строках и исполняемой функцией — в столбцах, шаблон **Acyclic Visitor** формирует *разреженную* матрицу. Классы **Visitor** вовсе не обязательно реализуют функции **visit** для каждого производного модуля **visited**. Например, если модемы **Ernie** нельзя конфигурировать для работы с **unix**, **UnixModemConfigurator** не реализует интерфейс **ErnieVisitor**. Поэтому шаблон **Acyclic Visitor** позволяет игнорировать определенные комбинации производных модулей и функций. Иногда это является преимуществом.

## **Применение Visitor в генераторах отчетов**

Обычно шаблон **Visitor** применяется для просмотра больших структур данных и генерирования отчетов. В этом случае объекты структур данных могут не

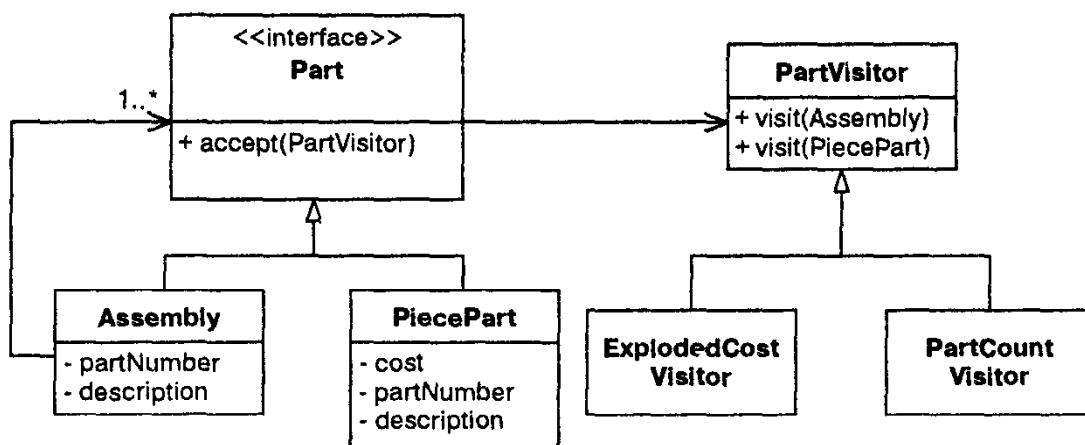


Рис. 28.4. Структура, генерирующая отчет о перечне материалов

располагать кодом, генерирующими отчеты. Для включения новых отчетов добавляются новые шаблоны *Visitor*, а код в структурах данных не изменяется. Это значит, что отчеты могут помещаться в отдельных компонентах и развертываться по отдельности только теми заказчиками, кто в них нуждается.

Рассмотрим простую структуру данных, представляющую список материалов (рис. 28.4). Имеется неограниченное количество отчетов, которые генерируются этой структурой данных. Например, можно генерировать отчет об общей стоимости всего агрегата, или же генерируется отчет, перечисляющий все составные части агрегата.

Каждый из этих отчетов может генерироваться методами класса *Part*. Например, к классу *Part* можно добавить методы `getExplodedCost` и `getPieceCount`. Эти методы можно реализовать в каждом производном модуле из *Part*, чтобы выполнялась соответствующая отчетность. К сожалению, отсюда следует, что каждый новый отчет, получаемый заказчиками, должен изменять иерархию *Part*.

Согласно принципу персональной ответственности (SRP, Single Responsibility Principle), следует отделять код, изменяемый по различным причинам. Иерархия *Part* может при необходимости изменять новые виды частей. Но для получения новых видов отчетов иерархия не может изменяться. Поэтому желательно отделять отчеты от иерархии *Part*. Структура шаблона *Visitor*, показанная на рис. 28.4, иллюстрирует этот процесс.

Каждый новый отчет может записываться как новый *Visitor*. Запишем функцию `accept` из *Assembly* для как *Visitor*, а также ее для всех включенных экземпляров *Part*. Таким образом, совершается обход всего дерева. Для каждого узла на этом дереве для отчета вызывается соответствующая функция `visit`. Отчет аккумулирует необходимые статистические данные. Затем к отчету можно обращаться за интересующими данными и представлять их пользователю.

Данная структура позволяет создавать неограниченное количество отчетов, не оказывая при этом никакого влияния на иерархию *Part*. Более того, каждый

отчет можно компилировать и распределять независимо от всех остальных. Листинги 28.17–28.23 показывают, как реализуется все сказанное при работе в Java.

---

**Листинг 28.17. Part.java**

---

```
public interface Part
{
 public String getPartNumber();
 public String getDescription();
 public void accept(PartVisitor v);
}
```

---

---

**Листинг 28.18. Assembly.java**

---

```
import java.util.*;

public class Assembly implements Part
{
 public Assembly(String partNumber, String description)
 {
 itsPartNumber = partNumber;
 itsDescription = description;
 }

 public void accept(PartVisitor v)
 {
 v.visit(this);
 Iterator i = getParts ();
 while (i.hasNext())
 {
 Part p = (Part)i.next();
 p.accept(v);
 }
 }

 public void add(Part part)
 {
 itsParts.add(part);
 }

 public Iterator getParts()
 {
 return itsParts.iterator();
 }

 public String getPartNumber()
 {
 return itsPartNumber;
```

```
public String getDescription()
{
 return itsDescription;
}

private List itsParts = new LinkedList();
private String itsPartNumber;
private String itsDescription;
}
```

---

**Листинг 28.19. PiecePart.java**

---

```
public class PiecePart implements Part
{
 public PiecePart(String partNumber, String description, double cost)
 {
 itsPartNumber = partNumber;
 itsDescription = description;
 itsCost = cost;
 }

 public void accept(PartVisitor v)
 {
 v.visit(this);
 }

 public String getPartNumber()
 {
 return itsPartNumber;
 }

 public String getDescription()
 {
 return itsDescription;
 }

 public double getCost()
 {
 return itsCost;
 }

 private String itsPartNumber;
 private String itsDescription;
 private double itsCost;
}
```

**Листинг 28.20. PartVisitor.java**

```
public interface PartVisitor
{
 public void visit(PiecePart pp);
 public void visit(Assembly a);
}
```

---

**Листинг 28.21. ExplodedCostVisitor.java**

```
public class ExplodedCostVisitor implements PartVisitor
{
 private double cost = 0;
 public double cost () {return cost;}
 public void visit(PiecePart p)
 {cost += p.getCost();}

 public void visit(Assembly a) {}
}
```

---

**Листинг 28.22. PartCountVisitor.java**

```
import java.util.*;
public class PartCountVisitor implements PartVisitor
{
 public void visit(PiecePart p)
 {
 itsPieceCount++;
 String partNumber = p.getPartNumber();
 int partNumberCount = 0;
 if (itsPieceMap.containsKey(partNumber))
 {
 Integer carrier = (Integer)itsPieceMap.get(partNumber);
 partNumberCount = carrier.intValue();
 }
 partNumberCount++;
 itsPieceMap.put(partNumber, new Integer(partNumberCount));
 }

 public void visit(Assembly a)
 {}

 public int getPieceCount() {return itsPieceCount;}
 public int getPartNumberCount() {return itsPieceMap.size();}
 public int getCountPorPart(String partNumber)
 {
 int partNumberCount = 0;
```

```
if (itsPieceMap.containsKey(partNumber))
{
 Integer carrier = (Integer)itsPieceMap.get(partNumber);
 partNumberCount = carrier.intValue();
}
return partNumberCount;
}

private int itsPieceCount = 0;
private HashMap itsPieceMap = new HashMap();
}
```

---

**Листинг 28.23. TestBOMReport.java**

---

```
import junit.framework.*;
import java.util.*;
public class TestBOMReport extends TestCase
{
 public TestBOMReport(String name)
 {
 super(name);
 }

 private PiecePart p1;
 private PiecePart p2;
 private Assembly a;
 public void setUp()
 {
 p1 = new PiecePart("997624", "MyPart", 3.20);
 p2 = new PiecePart("7734", "Hell", 666);
 a = new Assembly("5879", "MyAssembly");
 }

 public void testCreatePart()
 {
 assertEquals("997624", p1.getPartNumber());
 assertEquals("MyPart", p1.getDescription());
 assertEquals(3.20, p1.getCost(), .01);
 }

 public void testCreateAssembly()
 {
 assertEquals("5879", a.getPartNumber());
 assertEquals("MyAssembly", a.getDescription());
 }

 public void testAssembly()
 {
 a.add(p1);
 a.add(p2);
 }
}
```

```
Iterator i = a.getParts();
PiecePart p = (PiecePart)i.next();
assertEquals(p, p1);
p = (PiecePart)i.next();
assertEquals(p, p2);
assert(i.hasNext() == false);
}

public void testAssemblyOfAssemblies()
{
 Assembly subAssembly = new Assembly("1324", "SubAssembly");
 subAssembly.add(p1);
 a.add(subAssembly);
 Iterator i = a.getParts();
 assertEquals(subAssembly, i.next());
}

private boolean p1Found = false;
private boolean p2Found = false;
private boolean aFound = false;

public void testVisitorCoverage()
{
 a.add(p1);
 a.add(p2);
 a.accept(new PartVisitor(){
 public void visit(PiecePart p)
 {
 if (p == p1)
 p1Found = true;
 else if (p == p2)
 p2Found = true;
 }

 public void visit(Assembly assy)
 {
 if (assy == a)
 aFound = true;
 }
 });
 assert(p1Found);
 assert(p2Found);
 assert(aFound);
}

private Assembly cellphone;
void setUpReportDatabase()
{
 cellphone = new Assembly("CP-7734", "Cell Phone");
 PiecePart display = new PiecePart("DS-1428", "LCD Display", 14.37)
```

```
PiecePart speaker = new PiecePart("SP-92", "Speaker", 3.50);
PiecePart microphone = new PiecePart("MC-28", "Microphone", 5.30);
PiecePart cellRadio = new PiecePart("CR-56", "Cell Radio", 30);
PiecePart frontCover = new PiecePart("FC-77", "Front Cover", 1.4);
PiecePart backCover = new PiecePart("RC-77", "RearCover", 1.2);
Assembly keypad = new Assembly("KP-62", "Keypad");
Assembly button = new Assembly("B52", "Button");
PiecePart buttonCover = new PiecePart("CV-15", "Cover", .5);
PiecePart buttonContact = new PiecePart("CN-2", "Contact", 1.2);
button.add(buttonCover);
button.add(buttonContact);
for (int i=0; i<15; i++)
 keypad.add(button);
cellphone.add(display);
cellphone.add(speaker);
cellphone.add(microphone);
cellphone.add(cellRadio);
cellphone.add(frontCover);
cellphone.add(backCover);
cellphone.add(keypad);
}

public void testExplodedCost()
{
 setUpReportDatabase();
 ExplodedCostVisitor v = new ExplodedCostVisitor();
 cellphone.accept(v);
 assertEquals(81.27, v.cost0,.001);
}

public void testPartCount()
{
 setUpReportDatabase();
 PartCountVisitor v = new PartCountVisitor();
 cellphone.accept(v);
 assertEquals(36, v.getPieceCount());
 assertEquals(8, v.getPartNumberCount());
 assertEquals("DS-1428", 1, v.getCountForPart("DS-1428"));
 assertEquals("SP-92", 1, v.getCountForPart("SP-92"));
 assertEquals("MC-28", 1, v.getCountForPart("MC-28"));
 assertEquals("CR-56", 1, v.getCountForPart("CR-56"));
 assertEquals("RC-77", 1, v.getCountForPart("RC-77"));
 assertEquals("CV-15", 15, v.getCountForPart("CV-15"));
 assertEquals("CN-2", 15, v.getCountForPart("CN-2"));
 assertEquals("Bob", 0, v.getCountForPart("Bob"));
}
```

## Другие применения шаблона Visitor

В общем случае шаблон *Visitor* может использоваться в любом приложении, где структура данных должна интерпретироваться разными методами. Часто компиляторы создают промежуточные структуры данных, представляющие синтаксически корректный код источника. Эти структуры данных затем применяются для генерирования скомпилированного кода. Можно представить шаблон *Visitor*, конвертирующий промежуточную структуру данных в список ссылок или даже в UML-диаграмму.

Во многих приложениях применяется конфигурация структур данных. Иногда можно представить другие подсистемы приложения, инициализирующие себя на основе данных о конфигурировании путем использования собственных шаблонов *Visitor*.

При использовании шаблонов *Visitor* применяемая структура данных никогда не зависит от того, к чему она применяется. Можно создавать новые шаблоны *Visitor*, вносить изменения в уже существующие, но все эти конструкции могут разворачиваться в целях установки узлов (*sites*) без повторной компиляции или повторного развертывания существующих структур данных. В этом и состоит преимущество шаблона *Visitor*.

## Шаблон Decorator<sup>5</sup>

Шаблон *Visitor* позволяет добавлять методы к существующим иерархиям, не изменяя последние. Другой шаблон, выполняющий аналогичную задачу, называется *Decorator*.

Вернемся к иерархии *Modem* на рис. 28.1. Представим, что приложение применяется большим количеством пользователей. Каждый пользователь может запрашивать систему о доступе с помощью модема к другому компьютеру. Некоторые пользователи пожелают слышать звонки модемов. Другие предпочтут их не слышать.

Все это можно реализовать, запрашивая мнение пользователя в любом расположении кода, где поддерживается модемная связь. Если пользователь хочет слышать звуковое сопровождение, установим высокое значение громкости динамика. В противном случае отключим его.

```
...
Modem m = user.getModem();
if (user.wantsLoudDial())
 m.setVolume(11); // это значение превышает 10, не так ли?
m.dial(...);
...
```

<sup>5</sup>[GOF95].

Этот фрагмент кода дублировался сотни раз, пока приложение в течение 80 недель использовалось для обработки изображений, а также длились довольно трудные сеансы отладки. Благодаря этому удалось избежать многих неточностей.

Другая опция позволяет установить флаг в объекте `modem` и воспользоваться методом `dial` для проверки и установки соответствующего уровня громкости.

```
...
public class HayesModem implements Modem
{
 private boolean wantsLoudDial = false;
 public void dial(...)
 {
 if (wantsLoudDial)
 {
 setVolume(11);
 }
 ...
 }
 ...
}
```

Этот вариант значительно лучше, но по-прежнему для любого производного модуля `Modem` выполняется дублирование. Авторы новых производных модулей `Modem` должны помнить о необходимости репликации этого кода. Поскольку речь идет о памяти программистов, это занятие довольно рискованное.

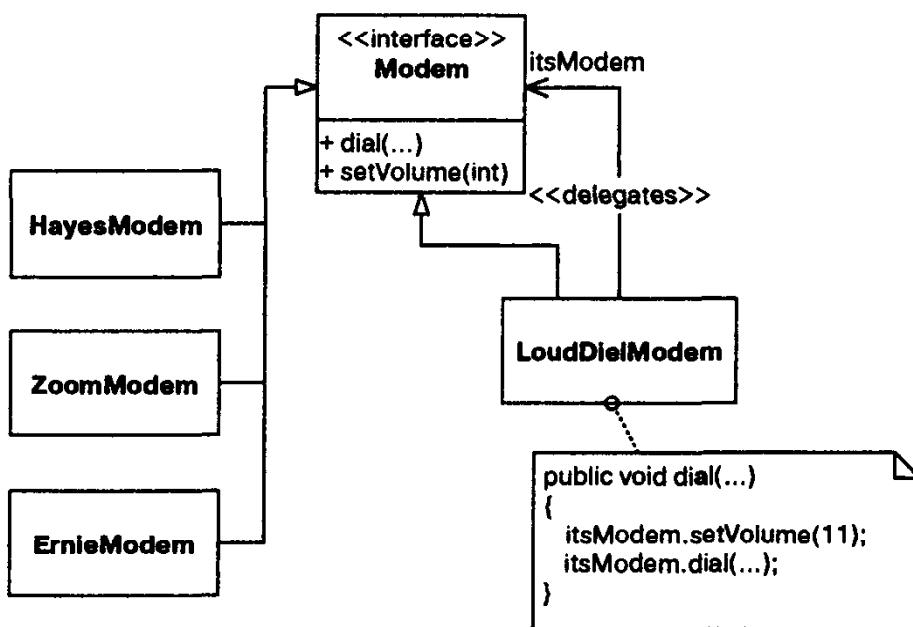
В данном случае можно воспользоваться шаблоном `Template Method`<sup>6</sup> путем замены `Modem`, применяемого в интерфейсе, на класс, сохраняя значение переменной `wantsLoudDial` и тестируя ее в функции `dial` до вызова функции `dialForReal`.

```
public abstract class Modem
{
 private boolean wantsLoudDial = false;
 public void dial(...)
 {
 if (wantsLoudDial)
 {
 setVolume(11);
 }
 dialForReal (...)
 }

 public abstract void dialForReal(...);
}
```

Это значительно лучше, но на класс `Modem` оказывают влияние прихоти пользователя? Зачем ему сведения о “громком” наборе номера? А зачем нужно изме-

<sup>6</sup>См. “Шаблон `Template Method`” в гл. 14.



**Рис. 28.5.** Шаблон Decorator – LoudDialModem

нение всякий раз, когда пользователь получает запросы типа завершения регистрации перед прекращением сеанса связи?

Снова вступает в силу принцип общего закрытия (CCP, Common-Closure Principle). Следует разделить моменты, изменение которых связано с различными причинами. Также можно обратиться к принципу персональной ответственности (SRP, Single-Responsibility Principle), поскольку подключение с озвучиванием этого процесса не связано с внутренними функциями Modem и поэтому не является частью Modem.

Шаблон *Decorator* решает эту задачу, создавая совершенно новый класс под названием *LoudDialModem*, который является производным модулем *Modem* и выполняет делегирование ко включенному экземпляру *Modem*. Функция *dial* перехватывается и перед делегированием устанавливается высокий уровень громкости. На рис. 28.5 показана эта структура.

Теперь решение о звуковом сопровождении подключения может выполняться в одном месте. А именно в том фрагменте кода, где пользователь размещает свои предпочтения, если он нуждается в озвучивании подключения. Можно создать `LoudDialModem`, передать ему функции модема, тогда пользователь ничего не заметит. Метод `dial` сначала установит высокий уровень громкости, а потом передаст эти сведения пользовательскому модему. Затем `LoudDialModem` может выполнять функции пользовательского модема без каких-либо изменений в системе, на которую оказывает влияние. Листинги 28.24–28.27 включают описанный код.

### Листинг 28.24. Modem.java

## public interface Modem

```
public void dial(String pno);
public void setSpeakerVolume(int volume);
public String getPhoneNumber();
public int getSpeakerVolume();
}
```

---

**Листинг 28.25. HayesModem.java**

```
public class HayesModem implements Modem
{
 public void dial(String pno)
 {
 itsPhoneNumber = pno;
 }

 public void setSpeakerVolume(int volume)
 {
 itsSpeakerVolume = volume;
 }

 public String getPhoneNumber()
 {
 return itsPhoneNumber;
 }

 public int getSpeakerVolume()
 {
 return itsSpeakerVolume();
 }

 private String itsPhoneNumber;
 private int itsSpeakerVolume;
}
```

---

**Листинг 28.26. LoudDialModem.java**

```
public class LoudDialModem implements Modem
{
 public LoudDialModem(Modem m)
 {
 itsModem = m;
 }

 public void dial(String pno)
 {
 itsModem.setSpeakerVolume(10);
 itsModem.dial(pno);
 }
}
```

```
public void setSpeakerVolume(int volume)
{
 itsModem.setSpeakerVolume(volume);
}

public String getPhoneNumber()
{
 return itsModem.getPhoneNumber();
}

public int getSpeakerVolume()
{
 return itsModem.getSpeakerVolume();
}
private Modem itsModem;
}
```

---

---

**Листинг 28.27. ModemDecoratorTest.java**

```
import junit.framework.*;
public class ModemDecoratorTest extends TestCase
{
 public ModemDecoratorTest(String name)
 {
 super(name);
 }

 public void testCreateHayes()
 {
 Modem m = new HayesModem();
 assertEquals(null, m.getPhoneNumber());
 m.dial("5551212");
 assertEquals("5551212", m.getPhoneNumber());
 assertEquals(0, m.getSpeakerVolume());
 m.setSpeakerVolume(10);
 assertEquals(10, m.getSpeakerVolume());
 }

 public void testLoudDialModem()
 {
 Modem m = new HayesModem();
 Modem d = new LoudDialModem(m);
 assertEquals(null, d.getPhoneNumber());
 assertEquals(0, d.getSpeakerVolume());
 d.dial("5551212");
 assertEquals("5551212", d.getPhoneNumber());
 assertEquals(10, d.getSpeakerVolume());
 }
}
```

## Множественные декораторы

Иногда в одной иерархии может быть два и больше декораторов (decorator). Например, если необходимо декорировать иерархию Modem с помощью LogoutExitModem, который пересыпает строку 'exit' всякий раз, когда вызывается метод Hangup. Второй декоратор полностью дублирует делегированный код, написанный для LoudDialModem. Этот дубликат кода можно устранить, создавая новый класс под названием ModemDecorator, который поддерживает весь делегированный код. Затем действительные декораторы могут просто формироваться из ModemDecorator и отменять лишь те методы, которые необходимо. На рис. 28.6 и в листингах 28.28 и 28.29 показана описанная структура.

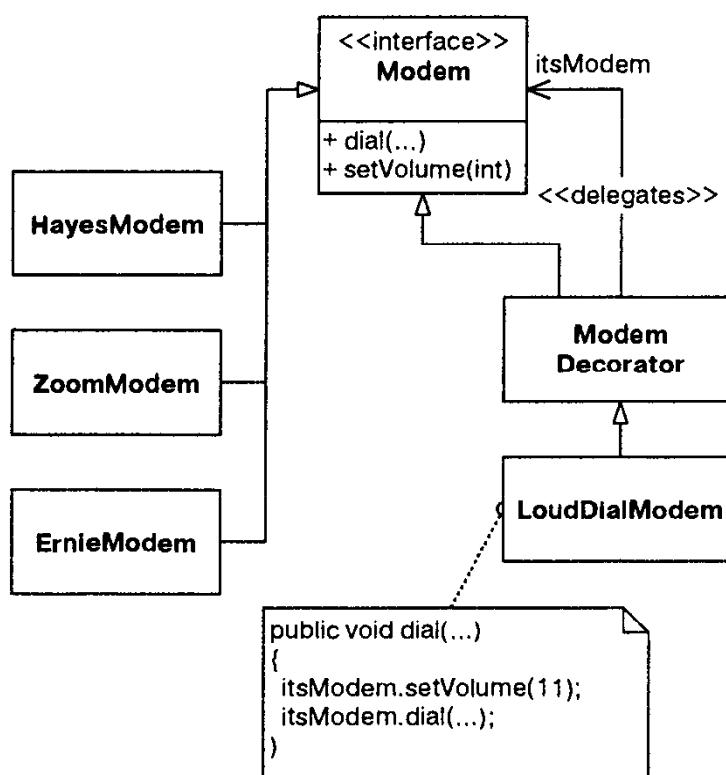


Рис. 28.6. ModemDecorator

---

### Листинг 28.28. ModemDecorator.java

```

public class ModemDecorator implements Modem
{
 public ModemDecorator(Modem m)
 {
 itsModem = m;
 }

 public void dial(String pno)
 {
 itsModem.dial(pno);
 }
}

```

```

public void setSpeakerVolume(int volume)
{
 itsModem.setSpeakerVolume(volume);
}

public String getPhoneNumber()
{
 return itsModem.getPhoneNumber();
}

public int getSpeakerVolume()
{
 return itsModem.getSpeakerVolume();
}

protected Modem getModem()
{
 return itsModem;
}
private Modem itsModem;
}

```

---

**Листинг 28.29.**

```

public class LoudDialModem extends ModemDecorator
{
 public LoudDialModem(Modem m)
 {
 super(m);
 }

 public void dial(String pno)
 {
 getModem().setSpeakerVolume(10);
 getModem().dial(pno);
 }
}

```

---

**Шаблон Extension Object**

Рассмотрим другую возможность по добавлению к иерархии функциональных свойств без изменения самой иерархии. Эта возможность состоит в применении шаблона *Extension Object*<sup>7</sup>. Этот шаблон не только обладает более сложными характеристиками по сравнению с другими шаблонами, но и более универсален и гибок. Каждый объект в иерархии поддерживает список специальных объектов расширения. Также каждый объект поддерживает метод, позволяющий обна-

<sup>7</sup>[PLOPD3], с. 79.

ружить объект расширения по названию. Объект расширения (extension object) поддерживает методы манипуляции исходным объектом иерархии.

Рассмотрим опять систему, представляющую список материалов. Необходимо разработать для каждого объекта возможность по созданию в этой иерархии XML-представления для данной системы. Можно поместить в эту иерархию методы `toXML`, но при этом нарушится принцип SRP. Вполне возможно, чтобы содержимое (stuff) BOM и XML располагались в одном классе. С помощью Visitor можно создать XML, но при этом не произойдет отделение XML-кода для каждого типа объекта BOM. При обращении к `Visitor` весь объем генерируемого XML-кода для каждого BOM-класса будет находиться в том же объекте `Visitor`. А если нужно разделять генерацию XML-кода для каждого BOM-объекта в его собственном классе?

Шаблон `Extension Object` поддерживает необходимые для этого возможности. Код, представленный в листингах 28.30–28.41, показывает иерархию BOM-объектов с двумя различными типами объектов расширения. Один вид объекта расширения конвертирует BOM-объекты в XML-код. Другой тип объекта расширения конвертирует BOM-объекты в строки CSV (значения, разделенные запятыми, comma-separated value). Для получения доступа к первому типу объекта расширения следует обратиться к `getExtension("XML")`, а для второго — к `getExtension("CSV")`. Эта структура показана на рис. 28.7 и использует

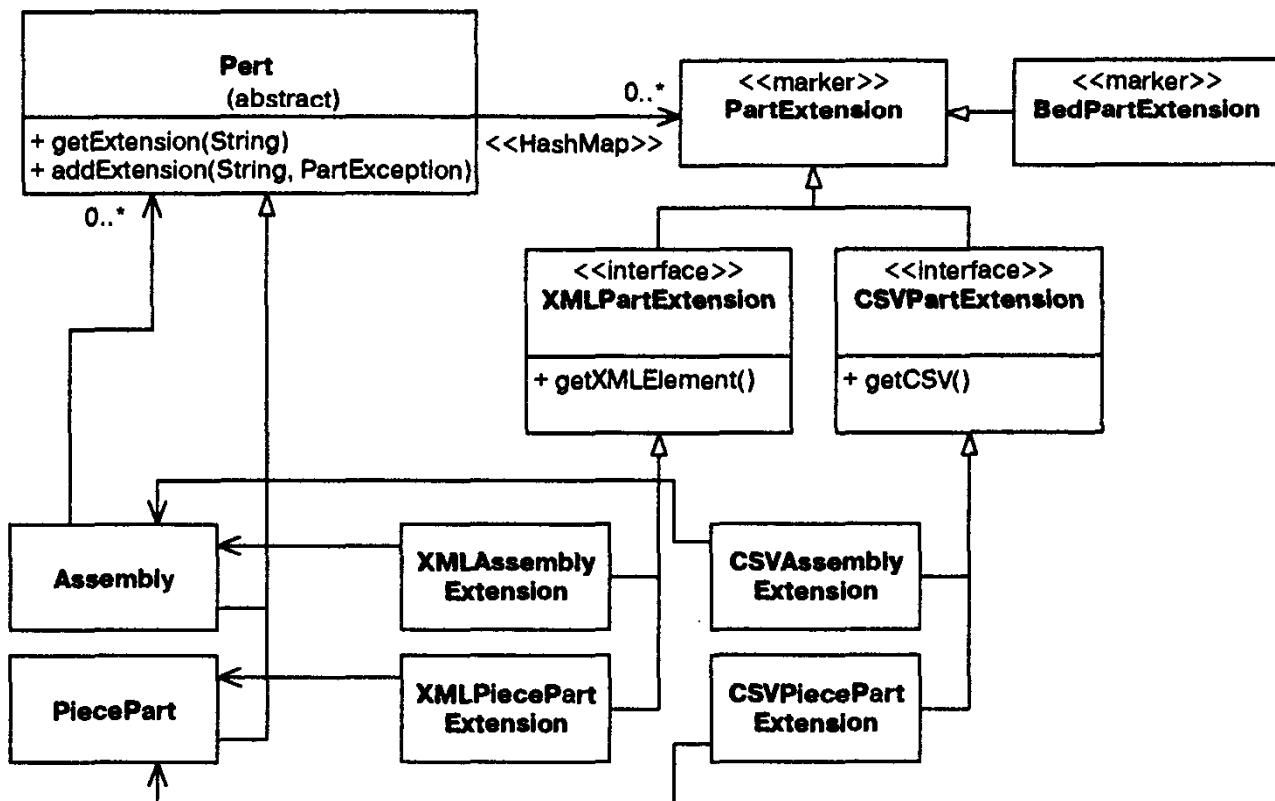


Рис. 28.7. Шаблон Object Extension

завершенный код. Стереотип “маркер” (“marker”) обозначает интерфейс маркера (т.е. интерфейс, где отсутствуют методы).

Важно понимать, что этот код нельзя оформить в виде листингов 28.30–28.41 без некоторой подготовки. Код изменяется от одного тестового испытания к другому. Первый файл исходного кода (листинг 28.30) отображает все тестовые испытания. Они создавались в том порядке, который здесь и отражен. Каждое тестовое испытание записывалось до того, как формировался соответствующий код. Код создавался только после успешного выполнения тестового испытания. И код не является более сложным, чем *выполненное* тестовое испытание. Поэтому код создавался постепенно, на рабочей базе. Автор пытается с помощью эволюционного подхода сформировать шаблон Extension Object.

---

#### Листинг 28.30. TestBOMXML.java

---

```
import junit.framework.*;
import java.util.*;
import org.jdom.*;

public class TestBOMXML extends TestCase
{
 public TestBOMXML(String name)
 {
 super(name);
 }

 private PiecePart p1;
 private PiecePart p2;
 private Assembly a;

 public void setUp()
 {
 p1 = new PiecePart("997624", "MyPart", 3.20);
 p2 = new PiecePart("7734", "Hell", 666);
 a = new Assembly("5879", "MyAssembly");
 }

 public void testCreatePart()
 {
 assertEquals("997624", p1.getPartNumber());
 assertEquals ("MyPart ", p1.getDescription());
 assertEquals(3.20, p1.getCost(), .01);
 }

 public void testCreateAssembly()
 {
 assertEquals("5879", a.getPartNumber());
 assertEquals("MyAssembly", a.getDescription());
 }
}
```

```
public void testAssembly()
{
 a.add(p1);
 a.add(p2);
 Iterator i = a.getParts();
 PiecePart p = (PiecePart)i.next ();
 assertEquals(p, p1);
 p = (PiecePart)i.next();
 assertEquals(p, p2);
 assert(i.hasNext() == false);
}

public void testAssemblyOfAssemblies()
{
 Assembly subAssembly = new Assembly("1324", "SubAssembly");
 subAssembly.add(p1);
 a.add(subAssembly);

 Iterator i = a.getParts();
 assertEquals(subAssembly, i.next());
}

public void testPiecePart1XML()
{
 PartExtension e = p1.getExtension("XML");
 XMLPartExtension xe = (XMLPartExtension)e;
 Element xml = xe.getXMLElement();
 assertEquals("PiecePart", xml.getName());
 assertEquals("997624", xml.getChild("PartNumber").getTextTrim());
 assertEquals("MyPart", xml.getChild("Description").getTextTrim());
 assertEquals(3.2,
 Double.parseDouble(xml.getChild("Cost").getTextTrim()), .01);
}

public void tesCPiecePart2XML()
{
 PartExtension e = p2.getExtension("XML");
 XMLPartExtension xe = (XMLPartExtension)e;
 Element xml = xe.getXMLElement();
 assertEquals("PiecePart", xml.getName());
 assertEquals("7734", xml.getChild("PartNumber").getTextTrim());
 assertEquals("Hell", xml.getChild("Description").getTextTrim());
 assertEquals(666,
 Double.parseDouble(xml.getChild("Cost").getTextTrim()), .01);
}

public void testSimpleAssemblyXML()
{
 PartExtension e = a.getExtension("XML");
 XMLPartExtension xe = (XMLPartExtension)e;
```

```
Element xml = xe.getXMLElement();
assertEquals("Assembly", xml.getName());
assertEquals("5879", xml.getChild("PartNumber").getTextTrim());
assertEquals("MyAssembly",
 xml.getChild("Description").getTextTrim());
Element parts = xml.getChild("Parts");
List partList = parts.getChildren();
assertEquals(0, partList.size());
}

public void testAssemblyWithPartsXML()
{
 a.add(p1);
 a.add(p2);
 PartExtension e = a.getExtension("XML");
 XMLPartExtension xe = (XMLPartExtension)e;
 Element xml = xe.getXMLElement();
 assertEquals("Assembly", xml.getName());
 assertEquals("5879", xml.getChild("PartNumber").getTextTrim());
 assertEquals("MyAssembly",
 xml.getChild("Description").getTextTrim());

 Element parts = xml.getChild("Parts");
 List partList = parts.getChildren();
 assertEquals(2, partList.size());

 Iterator i = partList.iterator ();
 Element partElement = (Element)i.next();
 assertEquals("PiecePart", partElement.getName());
 assertEquals("997624",
 partElement.getChild("PartNumber").getTextTrim());

 partElement = (Element)i.next();
 assertEquals("PiecePart", partElement.getName());
 assertEquals("7734",
 partElement.getChild("PartNumber").getTextTrim());
}

public void testPiecePart1toCSV()
{
 PartExtension e = pl.getExtension("CSV");
 CSVPartExtension ce = (CSVPartExtension)e;
 String csv = ce.getCSV();
 assertEquals("PiecePart,997624,MyPart,3.2", csv);
}

public void testPiecePart2toCSV()
{
 PartExtension e = p2.getExtension("CSV");
 CSVPartExtension ce = (CSVPartExtension)e;
```

```
String csv = ce.getCSV();
assertEquals("PiecePart,7734,Hell,666.0", csv);
}

public void testSimpleAssemblyCSV()
{
 partExtension e = agetExtension("CSV");
 CSVPartExtension ce = (CSVPartExtension)e;
 String csv = ce.getCSV();
 assertEquals("Assembly,5879,MyAssembly", csv);
}

public void testAssemblyWithPartsCSV()
{
 a.add(p1);
 a.add(p2);
 PartExtension e = a.getExternalStorage("CSV");
 CSVPartExtension ce = (CSVPartExtension)e;
 String csv = ce.getCSV();
 assertEquals("Assembly,5879,MyAssembly," +
 "(PiecePart,997624,MyPart,3.2)," +
 "(PiecePart,7734,Hell,666.0)", csv);
}

public void testBadExtension()
{
 PartExtension pe =
 p1.getExternalStorage("ThisStringDoesn'tMatchAnyException");
 assert(pe instanceof BadPartExtension);
}
```

---

### Листинг 28.31. Part.java

---

```
import java.util.*;

public abstract class Part
{
 HashMap itsExtensions = new HashMap();

 public abstract String getPartNumber();
 public abstract String getDescription();

 public void addExtension(String extensionType,
 PartExtension extension)
 {
 itsExtensions.put(extensionType, extension);
 }

 public PartExtension getExtension(String extensionType)
```

```
{
 PartExtension pe =
 (PartExtension) itsExtensions.get(extensionType);
 if (pe == null)
 pe = new BadPartExtension();
 return pe;
}
}
```

---

**Листинг 28.32. PartExtension.java**

```
public interface PartExtension
{
}
```

---

**Листинг 28.33. PiecePart.java**

```
public class PiecePart extends Part
{
 public PiecePart(String partNumber, String description, double cost)
 {
 itsPartNumber = partNumber;
 itsDescription = description;
 itsCost = cost;
 addExtension("CSV", new CSVPiecePartExtension(this));
 addExtension("XML", new XMLPiecePartExtension(this));
 }

 public String getPartNumber()
 {
 return itsPartNumber;
 }

 public String getDescription()
 {
 return itsDescription;
 }

 public double getCost()
 {
 return itsCost;
 }

 private String itsPartNumber;
 private String itsDescription;
 private double itsCost;
}
```

---

---

**Листинг 28.34. Assembly.java**

```
import java.util.*;

public class Assembly extends Part
{
 public Assembly(String partNumber, String description)
 {
 itsPartNumber = partNumber;
 itsDescription = description;
 addExtension("CSV", new CSVAssemblyExtension(this));
 addExtension("XML", new XMLAssemblyExtension(this));
 }

 public void add(Part part)
 {
 itsParts.add(part);
 }

 public Iterator getParts()
 {
 return itsParts.iterator();
 }

 public String getPartNumber()
 {
 return itsPartNumber;
 }

 public String getDescription()
 {
 return itsDescription;
 }

 private List itsParts = new LinkedList();
 private String itsPartNumber;
 private String itsDescription;
}
```

---

---

**Листинг 28.35. XMLPartExtension.java**

```
import org.jdom.*;

public interface XMLPartExtension extends PartExtension
{
 public Element getXMLElement();
}
```

---

---

**Листинг 28.36. XMLPiecePartException.java**

---

```
import org.jdom.*;

public class XMLPiecePartExtension implements XMLPartExtension
{
 public XMLPiecePartExtension(PiecePart part)
 {
 itsPiecePart = part;
 }

 public Element getXMLElement()
 {
 Element e = new Element("PiecePart");
 e.addContent(
 new Element("PartNumber").setText(
 itsPiecePart.getPartNumber()));
 e.addContent(
 new Element("Description").setText(
 itsPiecePart.getDescription()));
 e.addContent(
 new Element("Cost").setText(
 Double.toString(itsPiecePart.getCost())));
 return e;
 }

 private PiecePart itsPiecePart = null;
}
```

---

---

**Листинг 28.37. XMLAssemblyExtension.java**

---

```
import org.jdom.*;
import java.util.*;

public class XMLAssemblyExtension implements XMLPartExtension
{
 public XMLAssemblyExtension(Assembly assembly)
 {
 itsAssembly = assembly;
 }

 public Element getXMLElement()
 {
 Element e = new Element("Assembly");
 e.addContent(new Element("PartNumber")
 .setText(itsAssembly.getPart Number()));
 e.addContent(new Element("Description")
 .setText(itsAssembly.getDes cription()));
 Element parts = new Element("Parts");
```

```
e.addContent(parts);
Iterator i = itsAssembly.getParts ();
while (i.hasNext())
{
 Part p = (Part) i.next();
 PartExtension pe = p.getExtension("XML");
 XMLPartExtension xpe = (XMLPartExtension)pe;
 parts.addContent(xpe.getXMLElement());
}
return e;
}

private Assembly itsAssembly = null;
}
```

---

**Листинг 28.38. CSVPartExtension.java**

---

```
public interface CSVPartExtension extends PartExtension
{
 public String getCSV();
}
```

---

---

**Листинг 28.39. CSVPiecePartExtension.java**

---

```
public class CSVPiecePartExtension implements CSVPartExtension
{
 private PiecePart itsPiecePart = null;
 public CSVPiecePartExtension(PiecePart part)
 {
 itsPiecePart = part;
 }

 public String getCSV()
 {
 StringBuffer b = new StringBuffer("PiecePart,");
 b.append(itsPiecePart.getPartNumber());
 b.append ",";
 b.append(itsPiecePart.getDescription());
 b.append ",";
 b.append(itsPiecePart.getCost());
 return b.toString();
 }
}
```

---

---

**Листинг 28.40. CSVAssemblyExtension.java**

---

```

import Java.util.Iterator;

public class CSVAssemblyExtension implements CSVPartExtension
{
 private Assembly itsAssembly = null;

 public CSVAssemblyExtension(Assembly assy)
 {
 itsAssembly = assy;
 }

 public String getCSV()
 {
 StringBuffer b = new StringBuffer("Assembly,");
 b.append(itsAssembly.getPartNumber());
 b.append(",");
 b.append(itsAssembly.getDescription());
 Iterator i = itsAssembly.getParts();
 while (i.hasNext())
 {
 Part p = (Part) i.next();
 CSVPartExtension ce = (CSVPartExtension)p.getExtension("CSV");
 b.append(",{");
 b.append(ce.getCSV ());
 b.append("}");
 }
 return b.toString();
 }
}

```

---



---

**Листинг 28.41. BadPartExtension.java**

---

```

public class BadPartExtension implements PartExtension
{
}

```

---

Заметим, что объекты расширения загружаются в каждый ВОМ-объект с помощью конструктора объекта. Это значит, что с некоторой натяжкой, но объекты ВОМ по-прежнему зависят от классов XML и CSV. Если нарушится эта слабая зависимость, можно создать объект *Factory*<sup>8</sup>, который создает объекты ВОМ и загружает их расширения.

Тот факт, что объекты расширения могут загружаться в объект, создает дополнительные возможности по повышению гибкости. Определенные объекты расширения можно вставлять либо удалять из объектов в зависимости от состоя-

<sup>8</sup>См. шаблон *Factory* в гл 21.

ния системы. Довольно легко увлечься этой гибкостью. Для большинства случаев, возможно, вы ограничитеесь имеющимися результатами. Исходная реализация `PiecePart.getExtention (String extensionType)` имеет следующий вид:

```
public PartExtension getExtension(String extensionType)
{
 if (extensionType.equals("XML"))
 return new XMLPiecePartExtension(this);
 else if (extensionType.equals("CSV"))
 return new XMLAssemblyExtension(this);

 return new BadPartExtension();
}
```

Автора не впечатляет подобный код, поскольку он виртуально идентичен коду в `Assembly.getExtension`. Решение `HashMap` в `Part` позволяет избегать дублирования и является более простым вариантом. Каждый, кто ознакомится с ним, получит точное представление о том, как получать доступ к объектам расширения.

## Резюме

Семейство шаблонов `Visitor` поддерживает много способов модификации поведения иерархии классов без внесения изменений в эти классы. Таким образом соблюдается принцип OCP. Также поддерживаются механизмы по сегрегации различных типов функциональных возможностей, что позволяет избежать разбиения классов на части с помощью различных функций. Таким образом соблюдается принцип CCP. Ясно, что принципы SRP, LSP и DIP также применимы к структуре семейства `Visitor`.

Шаблоны `Visitor` весьма привлекательны для специалистов. Применяйте их тогда, когда они действительно необходимы. Часто случается так, что решение, полученное с помощью `Visitor`, можно реализовать и более простыми методами.

## Напоминание

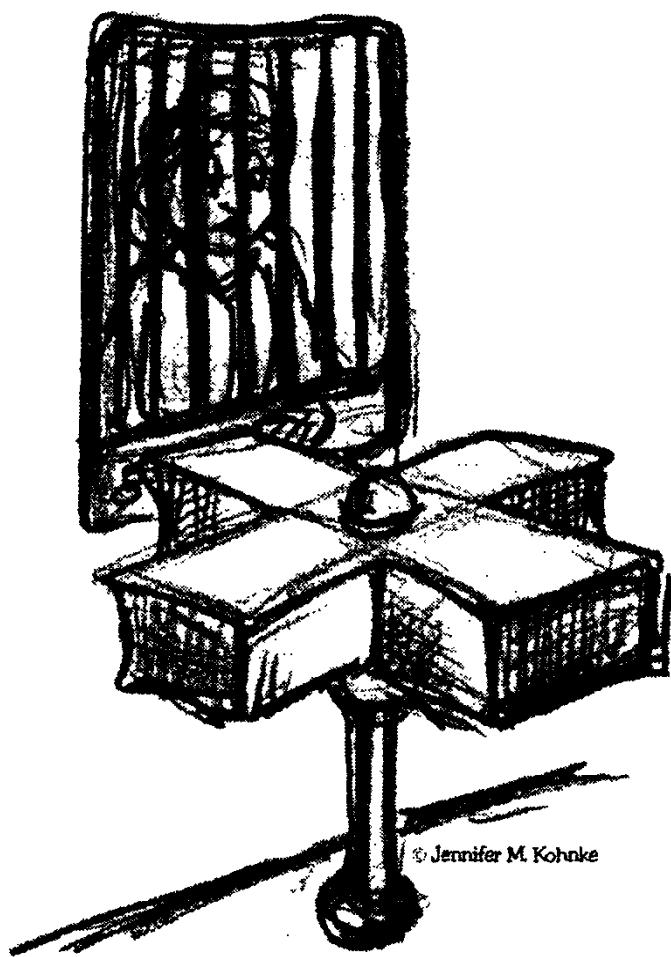
После ознакомления с материалом этой главы можно вернуться назад, к главе 9, и решить проблему упорядочения форм.

## Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
2. Martin R. C. и др. *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley, 1998.

# 29

## Шаблон State



Если для изменения состояния нет возможностей, их нет и для его сохранения.

---

Эдмунд Берк

Автомат с конечным числом состояний (Finite state automata) представляет собой наиболее полезную абстракцию в арсенале программных средств. Таким образом, поддерживается простой и элегантный способ исследования и уточнения поведения сложной системы. Также поддерживается универсальная стратегия по реализации, которую удобно представлять и легко обновлять. Этот подход применяется на всех уровнях системы, от контроля на высоком уровне GUI<sup>1</sup> до коммуникационных протоколов, находящихся на самом нижнем уровне. Данный подход иллюстрирует почти универсальную методику применения.

## Автомат с конечным числом состояний

Простым примером машины с конечным числом состояний (FSM, finite state machine) является турникет в метро. Этот механизм контролирует доступ пассажиров к поездам метро. На рис. 29.1 показан начальный этап функционирования FSM, управляющей турникетом метро. Эта диаграмма известна как *диаграмма перехода между состояниями* (state transition diagram, STD)<sup>2</sup>.

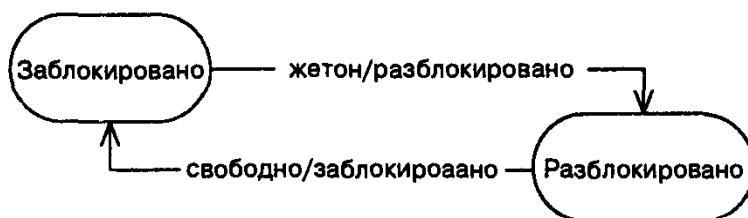


Рис. 29.1. Простая машина FSM, моделирующая функционирование турникета

Диаграмма STD состоит, по крайней мере, из четырех частей. В виде кругов представлены *состояния* (states). Состояния соединяются стрелками под названием *переходы* (transitions). Переходы отмечены названиями *события* (event) и *действия* (action). Диаграмма STD, изображенная на рис. 29.1, читается следующим образом.

- Если машина находится в состоянии *Locked* и происходит событие *coin*, тогда она переходит в состояние *Unlocked* и вызывают действие *unlock*.
- Если машина находится в состоянии *Unlocked* и происходит событие *pass*, тогда она переходит в состояние *Locked* и вызывают действие *lock*.

Этими двумя предложениями полностью описывается диаграмма, изображенная на рис. 29.1. Каждое предложение связано с одной стрелкой перехода и че-

<sup>1</sup>См. “Архитектура taskmaster” в гл 30.

<sup>2</sup>См. “Состояния и внутренние переходы”, “Переходы между состояниями” и “Вложенные состояния” в приложении Б.

тырьма элементами: начальное состояние, событие, вызывающее переход, заключительное состояние и выполняемое действие. Конечно, эти переходные предложения можно свести в простую таблицу под названием *таблица переходных состояний* (STT, state transition table). Подобная таблица может иметь следующий вид.

|          |      |          |        |
|----------|------|----------|--------|
| Locked   | coin | Unlocked | unlock |
| Unlocked | Pass | Locked   | lock   |

Каким образом функционирует эта машина? Предположим, что FSM начинает “свою жизнь” с состояния **Locked**. Пассажир подходит к турникету и опускает в него монету. Тогда программа получает событие **coin**. Первый переход в STT состоит в том, что в состоянии **Locked** есть событие **coin**, затем реализуется переход в состояние **Unlocked** и вызов действия **unlock**. Тогда программное обеспечение изменяет свое состояние на **Unlocked** и вызывает функцию **unlock**. Затем пассажир проходит через открытый турникет, что свидетельствует о проверке программой события **pass**. Поскольку FSM находится теперь в состоянии **Unlocked**, реализуется второе переходное состояние, что вынуждает машину вернуться назад, к состоянию **Locked**, и приводит к вызову функции **lock**.

Ясно, что диаграммы STD и STT представляют собой простые и элегантные описания поведения данной машины. Но речь идет также об универсальных инструментах по разработке. Одним из достоинств этих инструментов является легкость обнаружения необычных и плохо управляемых условий. Рассмотрим, например, каждое состояние на рис. 29.1 и применим оба известных события. Заметим, что для обработки события **coin** в состоянии **Unlocked** переход отсутствует, также нет перехода для обработки события **pass** в состоянии **Locked**.

Эти недоработки иллюстрируют довольно серьезные логические просчеты и часто приводят к программистским ошибкам. Программисты обычно уделяют внимание нормальному ходу событий и недостаточно учитывают наличие аномальных ситуаций. Диаграммы STD или STT позволяют программисту легко проверить, охвачено ли в процессе разработки любое событие в любом состоянии.

Машину FSM можно дополнить необходимыми переходами. Новая версия показана на рис. 29.2. Если пассажир опустит в автомат несколько жетонов, машина останется в состоянии **Unlocked** и появится сообщение “благодарю”, что должно вызвать у пассажира желание продолжать “денежный дождь”<sup>3</sup>. Если пассажир сможет пройти через заграждение (возможно, с помощью увесистого орудия труда), FSM останется в состоянии **Locked**, но будет подавать звуковой сигнал тревоги.



Рис. 29.2. Машина FSM, моделирующая турникет, которая включает аномальные события

## Методики реализации

### Вложенные операторы Switch/Case

Для реализации машины FSM предусмотрено много различных стратегий. Первая и наиболее “прямолинейная” состоит в использовании вложенных операторов `switch/case`. Листинг 29.1 показывает одну из таких реализаций.

---

#### Листинг 29.1. Turnstile.java (вложенная реализация Switch/Case)

---

```

package com.objectmentor.PPP.Patterns.State.turnstile;

public class Turnstile
{
 // Состояния
 public static final int LOCKED = 0;
 public static final int UNLOCKED = 1;

 // События
 public static final int COIN = 0;
 public static final int PASS = 1;

 /*частная переменная*/ int state = LOCKED;

 private TurnstileController turnstileController;

 public Turnstile(TurnstileController action)
 {
 turnstileController = action;
 }

 public void event(int event)
 {
 switch (state)
 {
 case LOCKED:
 switch (event)
 {
 case COIN:
 state = UNLOCKED;
 turnstileController.unlock();
 }
 }
 }
}

```

```
 break;
 case PASS:
 turnstileController.alarm();
 break;
 }
 break;
case UNLOCKED:
 switch (event)
 {
 case COIN:
 turnstileController.thankyou();
 break;
 case PASS:
 state = LOCKED;
 turnstileController.lock();
 break;
 }
 break;
}
```

Вложенный оператор `switch/case` подразделяет код на четыре взаимоисключающие эксклюзивных зоны, каждая из которых соответствует одному из переходов в диаграмме STD. Каждая зона изменяет состояние должным образом и затем вызывает соответствующее действие. Таким образом, зона для `Locked` и `Coin` изменяет состояние на `Unlocked` и вызывает событие `unlock`.

Этот код имеет несколько интересных аспектов, не имеющих ничего общего со вложенным оператором `switch/case`.

### Листинг 29.2. TurnstileController.java

```
package com.objectmentor.PPP.Patterns.State.turnstile;

public interface TurnstileController
{
 public void lock();
 public void unlock();
 public void thankyou();
 public void alarm();
}
```

### Листинг 29.3. TestTurnstile.java

```
package com.objectmentor.PPP.Patterns.Stateturnstile;

import junit.framework.*;
import junit.swingui.TestRunner;
```

```
public class TestTurnstile extends TestCase
{
 public static void main(String[] args)
 {
 TestRunner.main(new String[]{"TestTurnstile"});
 }

 public TestTurnstile(String name)
 {
 super(name);
 }

 private Turnstile t;
 private boolean lockCalled = false;
 private boolean unlockCalled = false;
 private boolean thankyouCalled = false;
 private boolean alarmCalled = false;

 public void setUp()
 {
 TurnstileController controllerSpoof = new TurnstileController()
 {
 public void lock() {lockCalled = true;}
 public void unlock() {unlockCalled = true;}
 public void thankyou() {thankyouCalled = true;}
 public void alarm() {alarmCalled = true;}
 };

 t = new Turnstile(controllerSpoof);
 }

 public void testInitialConditions()
 {
 assertEquals(Turnstile.LOCKED, t.state);
 }

 public void testCoinInLockedState()
 {
 t.state = Turnstile.LOCKED;
 t.event(Turnstile.COIN);
 assertEquals(Turnstile.UNLOCKED, t.state);
 assert(unlockCalled);
 }

 public void testCoinInUnlockedState()
 {
 t.state = Turnstile.UNLOCKED;
 t.event(Turnstile.COIN);
 assertEquals(Turnstile.UNLOCKED, t.state)
 assert(thankyouCalled);
 }
}
```

```

}

public void testPassInLockedState()
{
 t.state = Turnstile.LOCKED;
 t.event(Turnstile.PASS);
 assertEquals(Turnstile.LOCKED, t.state);
 assert(alarmCalled);
}

public void testPassInUnlockedState()
{
 t.state = Turnstile.UNLOCKED;
 t.event(Turnstile.PASS);
 assertEquals(Turnstile.LOCKED, t.state);
 assert(lockCalled);
}
}

```

---

## Переменная состояния области доступа пакета

Заметим, что четыре тестовые функции имеют названия `testCoinInLockedState`, `testCoinInUnlockedState`, `testPassInLockedState` и `testPassInUnlockedState`. Эти функции по отдельности тестируют четыре перехода для машины FSM. Они выполняют это путем перевода переменной `state` из `Turnstile` в состояние, проверку которого необходимо выполнить, а затем вызывается проверяемое событие. Чтобы протестировать доступ к переменной `state`, эти функции не должны быть частными. Поэтому создана область доступа пакетов, и внесен комментарий, указывающий на то, что переменная должна иметь частный характер.

Согласно принципам объектно-ориентированного программирования, все переменные экземпляры класса должны быть частными. Автор игнорирует это правило, тем самым нарушая инкапсуляцию `Turnstile`.

Можно ли так поступать?

Не замечая ошибки, автор предпочитает сохранять переменную `state` частной. Но тогда отвергается тестовый код, позволяющий обработку этой величины. В области доступа пакетов можно создать соответствующие методы `setState` и `getState`, но это кажется нецелесообразным. Автор не пытается выявить переменную `state` для любого класса, отличного от `TestTurnstile`, поэтому следует создать методы установки (`setter`) и получения (`getter`), тогда в области доступа пакетов можно получать и устанавливать эту переменную, не так ли?

Одним из существенных недостатков Java является отсутствие понятия, подобного `friend` из языка C++. Если бы Java располагал оператором `friend`, можно было бы сохранить за переменной `state` приватный статус, а также объ-

явить `TestTurnstile` “другом” `Turnstile`. Поскольку сделать этого нельзя, ограничимся указанием для `state` пакетного диапазона.

## Тестирование действий

Рассмотрим интерфейс `TurnstileController`, код которого приводится в листинге 29.2. Он спроектирован таким образом, что класс `TestTurnstile` может гарантировать классу `Turnstile` вызов методов для корректного выполнения действий, соблюдая при этом нужный порядок. Без этого интерфейса значительно труднее гарантировать верное функционирование машины состояний.

Этот подход служит хорошим примером обратного влияния тестирования на разработку. Если бы машина состояний создавалась без предварительного тестирования, вряд ли бы появился интерфейс `TurnstileController`. Интерфейс `TurnstileController` прекрасно отделяет логику машины с конечным числом состояний от тех действий, которые необходимо выполнить. Другая машина FSM, используя отличную от этой логику, может применять `TurnstileController` без какого-либо воздействия.

## Преимущества и недостатки реализации `Switch/Case`

При обращении к простым машинам состояния, вложенная реализация `switch/case` элегантна и эффективна. Все состояния и события можно просмотреть с помощью одной-двух страниц кода. Но при обращении к более крупным машинам FSM ситуация существенно меняется. Если машина состояний охватывает более дюжины состояний и событий, код “разрастается” до многих страниц. Поддержка длинных вложенных операторов `switch/case` представляет собой довольно трудную задачу, и ошибки в данном случае неизбежны.

Другой особенностью вложенных операторов `switch/case` является отсутствие удачного разделения между логикой машины с конечным числом состояний и кодом по реализации соответствующих действий. Это разделение представлено в листинге 29.1, поскольку действия реализованы в производном модуле `TurnstileController`. Но в большинстве машин FSM, использующих вложенные операторы `switch/case`, реализация действий скрыта среди операторов `case`. Это также показано в листинге 29.1.

## Интерпретация таблиц переходов

В процессе реализации машины FSM обычно используется методика, основанная на создании таблиц данных, которые описывают переходы. Подобная таблица интерпретируется механизмом, обрабатывающим события. Механизм находит переход, соответствующий данному событию, вызывает соответствующее действие и изменяет состояние.

Листинг 29.4 демонстрирует код, создающий таблицу переходов, а листинг 29.5 отображает механизм переходов. Оба листинга содержат фрагменты кода из полной реализации в листинге 29.12, который приведен в конце этой главы.

---

#### Листинг 29.4. Формирование таблицы переходов для турникета

---

```
public Turnstile(TurnstileController action)
{
 turnstileController = action
 addTransition(LOCKED, COIN, UNLOCKED, unlock());
 addTransition(LOCKED, PASS, LOCKED, alarm());
 addTransition(UNLOCKED, COIN, UNLOCKED, thankyou());
 addTransition(UNLOCKED, PASS, LOCKED, lock());
}
```

---



---

#### Листинг 29.5. Механизм перехода

---

```
public void event(int event)
{
 for (int i = 0; i < transitions.size (); i++)
 {
 Transition transition = (Transition) transitions.elementAt(i);
 if (state == transition.currentState && event == transition.event)
 {
 state = transition.newState;
 transition.action.execute();
 }
 }
}
```

---

### Преимущества и недостатки, связанные с интерпретацией таблиц переходов

Одно из важных преимуществ состоит в том, что код, формирующий таблицу переходов, можно просматривать подобно “канонической” таблице переходов состояний. Довольно легко представить четыре строки `addTransition`. Логика механизма состояния основана на том, что все собрано в одном месте и не связано с реализацией действий.

Поддержку машины с конечным числом состояний, аналогичную рассмотренной выше, можно легко сравнить со вложенной реализацией `switch/case`. Новый переход добавляется к конструктору `Turnstile` посредством новой строки `addTransition`.

Другим положительным аспектом этого подхода является тот факт, что таблицу можно легко изменять в процессе выполнения программы. Таким образом можно модифицировать логику машины состояний. Механизмы, подобные описанным, применяются для быстрого “латания” сложных машин с конечным числом состояний.

Еще одним преимуществом данного подхода является возможность создания нескольких таблиц, каждая из которых представляет иную логику машины FSM. Эти таблицы можно выделить во время выполнения программы на основе начальных условий.

Для оценивания подхода применяется методика, основанная на хронометраже времени просмотра. Изучение таблицы переходов занимает довольно много времени. Другим важным фактором является объем кода, предназначенного для поддержки таблицы. Если внимательно изучить листинг 29.12, можно заметить довольно много небольших функций поддержки, которые позволяют упростить запись выражений в таблицу переходов состояния, реализованную в листинге 29.4.

## Шаблон State<sup>4</sup>

Другая методика по реализации машины с конечным числом состояний представлена шаблоном State. Этот шаблон комбинирует эффективность вложенного оператора `switch/case` с гибкостью таблицы интерпретации переходов.

На рис. 29.3 представлена структура такого решения. Класс `Turnstile` располагает общедоступными методами для событий и защищенными методами для действий. Он имеет ссылку на интерфейс под названием `TurnstileState`. Два производных модуля из `TurnstileState` представляют два состояния машины FSM.

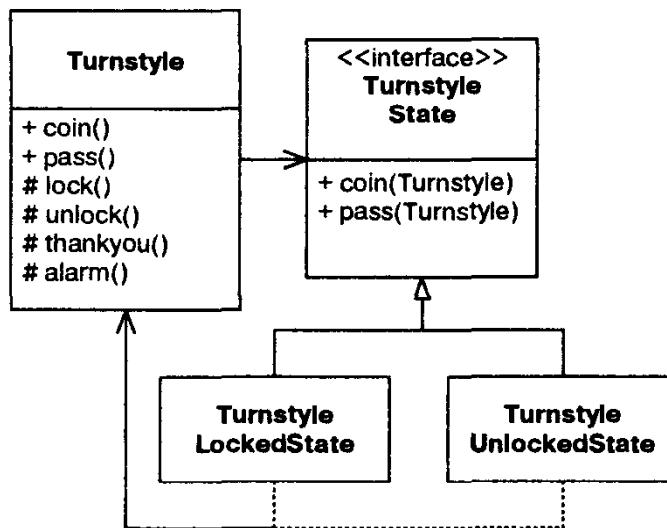


Рис. 29.3. Шаблон State, моделирующий работу турнкета в метро

Если вызывается один или два метода события из `Turnstile`, это событие передается объекту `TurnstileState`. Методы из `TurnstileLockedState` реализуют соответствующие действия для состояния `LOCKED`. Методы из `TurnstileUnlockedState` реализуют соответствующие действия для состояния

<sup>4</sup>[GOF95]

UNLOCKED. Чтобы изменить состояние машины FSM, ссылка в объекте `Turnstile` присваивается экземпляру одного из этих производных модулей.

Листинг 29.6 демонстрирует интерфейс `TurnstileState` и два производных модуля. Машина состояний легко просматривается в четырех методах этих производных модулей. К примеру, метод `coin` из `LockedTurnstileState` указывает объекту `Turnstile` на изменение состояния и перевода его в `UNLOCKED`, а затем вызывает функцию действия `unlock` из `Turnstile`.

---

#### Листинг 29.6. `TurnstileState.java`

---

```
interface TurnstileState
{
 void coin(Turnstile t);
 void pass(Turnstile t);
}

class LockedTurnstileState implements TurnstileState
{
 public void coin(Turnstile t)
 {
 t.setUnlocked();
 t.unlock();
 }

 public void pass(Turnstile t)
 {
 t.alarm();
 }
}

class UnlockedTurnstileState implements TurnstileState
{
 public void coin(Turnstile t)
 {
 t.thankyou();
 }

 public void pass(Turnstile t)
 {
 t.setLocked();
 t.lock();
 }
}
```

---

Код класса `Turnstile` показан в листинге 29.7. Обратите внимание на статические переменные, содержащие производные модули `TurnstileState`. Эти классы не включают переменные, поэтому нет необходимости в более чем одном экземпляре. Наличие в переменных производных экземпляров производных моду-

лей `TurnstileState` приводит к тому, что не нужно создавать новый экземпляр всякий раз, когда изменяется состояние. Поскольку эти переменные статичны, не нужно создавать новые экземпляры производных модулей в том событии, где следует пользоваться более чем одним экземпляром `Turnstile`.

---

**Листинг 29.7. Turnstile.java**

---

```
public class Turnstile
{
 private static TurnstileState lockedState =
 new LockedTurnstileState();
 private static TurnstileState unlockedState =
 new UnlockedTurnstileState();

 private TurnstileController turnstileController;
 private TurnstileState state = lockedState;

 public Turnstile(TurnstileController action)
 {
 turnstileController = action;
 }

 public void coin()
 {
 state.coin(this);
 }

 public void pass()
 {
 state.pass(this);
 }

 public void setLocked()
 {
 state = lockedState;
 }

 public void setUnlocked()
 {
 state = unlockedState;
 }

 public boolean isLocked()
 {
 return state == lockedState;
 }

 public boolean isUnlocked()
 {
 return state == unlockedState;
 }
}
```

```

}

void thankyou()
{
 turnstileController.thankyou();
}

void alarm()
{
 turnstileController.alarm();
}

void lock()
{
 turnstileController.lock();
}

void unlock()
{
 turnstileController.unlock();
}
}

```

---

## **Сравнение шаблона State с шаблоном Strategy**

Диаграмма, изображенная на рис. 29.3, является прямой реминисценцией шаблона *Strategy*<sup>5</sup>. Оба шаблона располагают контекстным классом; оба выполняют делегирование к полиморфному базовому классу, имеющему несколько производных модулей. Отличие состоит в том (рис. 29.4), что в шаблоне *State* производные модули содержат обратные ссылки к контекстному классу. Основная функция производных модулей состоит в выделении и вызове методов контекстного класса с помощью этой ссылки. В шаблоне *Strategy* не существует подобного ограничения или намерения. Производным модулям *Strategy* не нужно содержать ссылку к контексту, а также они не требуют вызова методов для контекста. Поэтому все экземпляры шаблона *State* также являются экземплярами шаблона *Strategy*, но не все экземпляры шаблона *Strategy* выполняют эти же функции в шаблоне *State*.

## **Преимущества и недостатки шаблона State**

Шаблон *State* поддерживает строгое разделение между действиями и логикой механизма состояния. Действия реализуются в классе *Context*, а логика распределяется с помощью производных модулей класса *State*. Поэтому довольно легко изменять одно, не оказывая влияния на другое. Например, несложно

---

<sup>5</sup> См. описание шаблона *Strategy*.

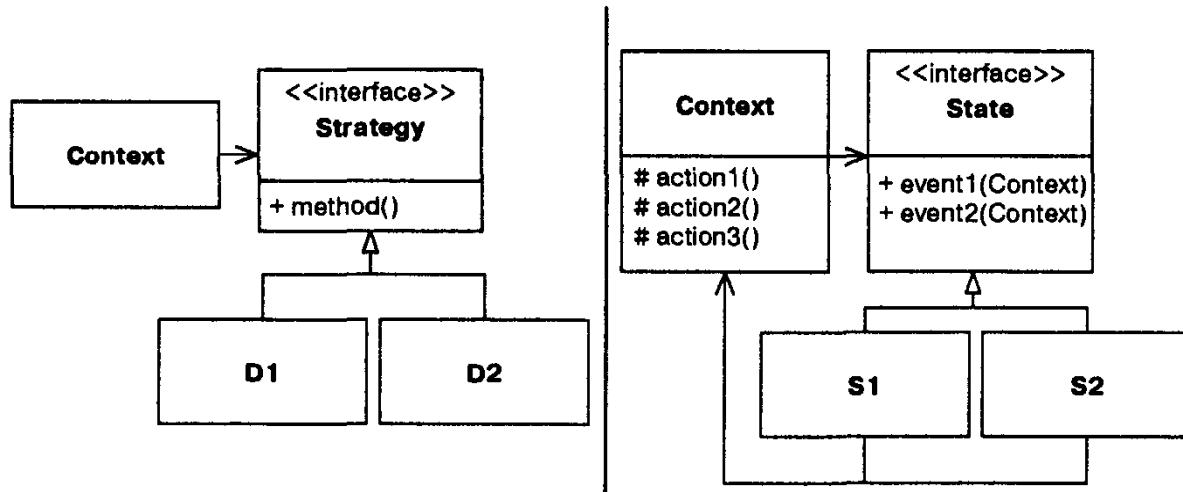


Рис. 29.4. Шаблоны State и Strategy

повторно использовать действия класса **Context**, применяя различную логику состояния.

Для этого просто используются различные производные модули класса **State**. В качестве альтернативы можно создавать подклассы **Context**, обновляющие или заменяющие действия без изменения логики производных модулей **State**.

Другим преимуществом этой методики является ее высокая эффективность. Возможно, рассматриваемая методика столь же эффективна, как реализация вложенных операторов **switch/case**. Т.е. можно утверждать, что подход, основанный на применении таблиц, весьма гибок, а подход, использующий вложенные операторы **switch/case**, более чем эффективен.

Но, во-первых, написание производных модулей **State** является довольно скучным занятием. При создании механизма, насчитывающего 20 состояний, можно и со счета сбиться. Во-вторых, логика распределена. Отсутствует какое-либо место, где можно просмотреть всю конструкцию. Поэтому поддержка весьма затруднена. Это напоминает непрозрачность подхода, связанного с вложением **switch/case**.

## Компилятор машины состояний (SMC, State-Machine Compiler)

Поскольку довольно утомительно записывать производные модули состояния, а также удобно логику механизма состояния собрать в одном месте, автор создал компилятор, транслирующий таблицу переходов состояния в классы, необходимые для реализации шаблона **State**. Этим компилятором можно воспользоваться бесплатно, загрузив его с Web-узла <http://www.objectmentor.com>. Ввод данных в компилятор показан в листинге 29.8. Можно воспользоваться следующим синтаксисом:

```
currentState
{
 event newState action
 ...
}
```

Первые четыре строки описывают название машины состояний, имя контекстного класса, начальное состояние и название исключения, которое возникает в случае недопустимого события.

---

#### Листинг 29.8. Turnstile.sm

---

```
FSMName Turnstile
Context TurnstileActions
Initial Locked
Exception FSMError
{
 Locked
 {
 coin Unlocked unlock
 pass Locked alarm
 }
 Unlocked
 {
 coin Unlocked thankyou
 pass Locked lock
 }
}
```

---

Чтобы применить компилятор, необходимо сформировать класс, объявляющий функции действия. Название этого класса указано в строке `Context`. В данном случае он называется `TurnstileActions` (листинг 29.9).

---

#### Листинг 29.9. TurnstileActions.java

---

```
public abstract class TurnstileActions
{
 public void lock(){}
 public void unlock(){}
 public void thankyou(){}
 public void alarm() {}
}
```

---

Компилятор генерирует класс, который является производным для контекста. Название сгенерированного класса указано на строке `FSMName`. Автор назвал его `Turnstile`.

В `TurnstileActions` реализованы функции действий. Но автор склоняется к мысли о необходимости создания другого класса, который является производным для сгенерированного класса и реализует здесь функции действия. Этот класс показан в листинге 29.10.

---

**Листинг 29.10. TurnstileFSM.java**

---

```
public class TurnstileFSM extends Turnstile
{
 private TurnstileController controller;
 public TurnstileFSM(TurnstileController controller)
 {
 this.controller = controller;
 }

 public void lock()
 {
 controller.lock();
 }

 public void unlock()
 {
 controller.unlock();
 }

 public void thankyou()
 {
 controller.thankyou();
 }

 public void alarm()
 {
 controller.alarm();
 }
}
```

---

Это все, что следует записать. Компилятор SMC сгенерирует оставшуюся часть. Результирующие структуры показаны на рис. 29.5. Автор назвал их *Three-Level Finite State Machine*<sup>6</sup>.

Три уровня обеспечивают максимальную гибкость при сравнительно низких затратах. Можно создать большое число машин с конечным числом состояний, просто формируя производные модули из *TurnstileActions*. Также можно реализовать эти действия многими способами, создавая производные модули на основе *Turnstile*.

Заметим, что сгенерированный код полностью изолирован от записанного на ми кода. Вам не приходится обновлять сгенерированный код. Можно даже не изучать его. Можно уделить ему столько же внимания, сколько уделяется двоичному коду.

Сгенерированный код, а также другой вспомогательный код для этого примера можно найти в листингах 29.13–29.15.

---

<sup>6</sup>[PLoPD1]

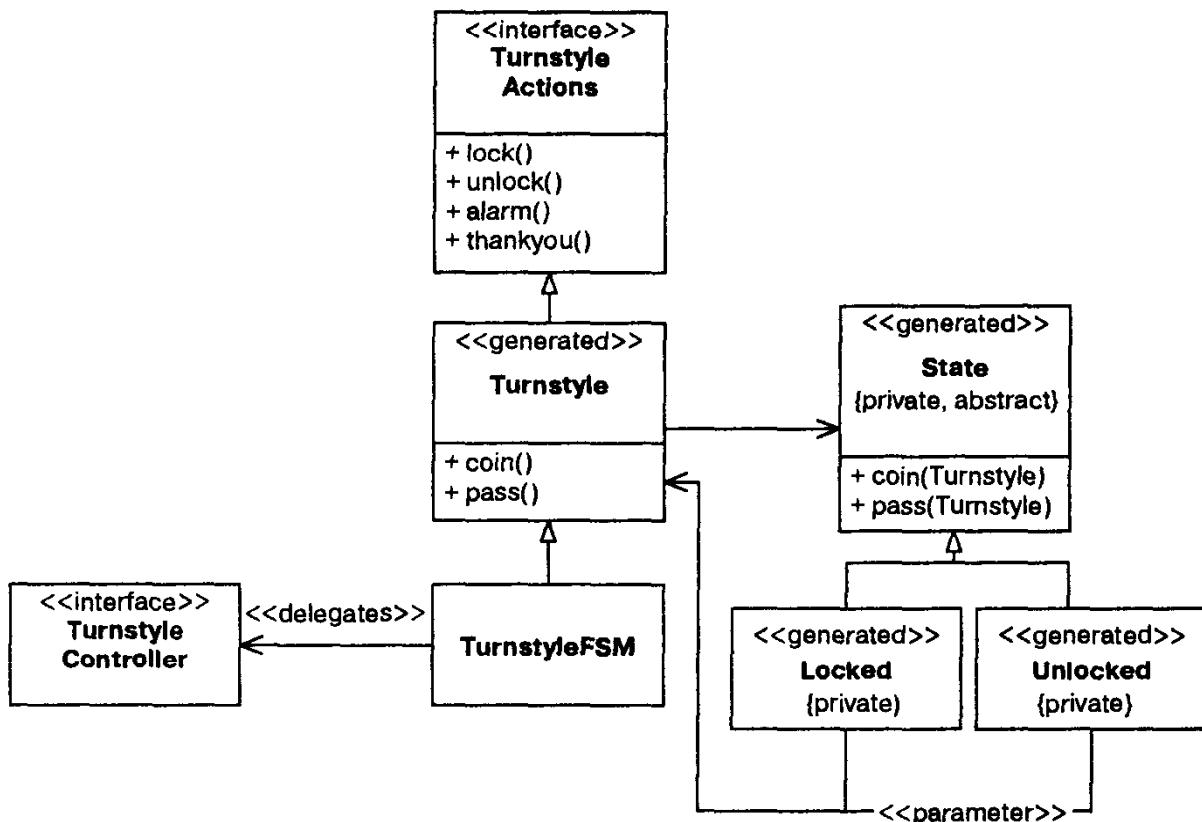


Рис. 29.5. Трехуровневая машина FSM

## Подход SMC и шаблон State

Описание машины с конечным числом состояний содержит все в одном месте и облегчает дальнейшую поддержку. Логика машины с конечным числом состояний строго изолирована от реализации действий, позволяя вносить изменения лишь в некоторые из них, оставляя неизменными другие. Решение эффективно, элегантно и нуждается в минимальном кодировании.

Но в данном случае следует обращаться к компилятору SMC. Необходимо ознакомиться с тем, как используется другой инструмент. В данном случае инструмент сравнительно прост в инсталляции и в применении. (Обратитесь к листингу 29.16 и предыдущим параграфам.) И все это совершенно бесплатно!

## Область применения машин состояний

Машины состояний (и SMC) применяются в нескольких различных классах приложений.

## Политики высокоуровневых приложений для GUI

Графическая революция, начавшаяся в 1980 году, привела, наряду с решением других задач, к созданию *лишенных состояния* (*stateless*) интерфейсов для пользовательского применения. В то время в интерфейсах компьютеров доминировали текстовые подходы, использующие иерархические меню. Можно было легко потерять данные в структуре меню, не учитывая *состояния* экрана. Интерфейс GUI помог решить эту проблему, минимизировав количество изменений состояний, через которые проходят экранные преобразования. В современных GUI большое внимание уделяется сохранению на экране общих свойств, а также недопущению такой ситуации, когда пользователь запутается в скрытых состояниях.

Но код, реализующий “лишенные состояния” (“*stateless*”) GUI, обладает строго управляемым состоянием. В таком GUI код должен обрабатывать пункты меню и кнопки, которые теряют подсветку, учитывать, какие подокна должны появиться, какие клавиши табуляции следует активизировать, уточнять место расположения фокуса и т.д. Все эти решения относятся к состоянию данного интерфейса.

Долгое время автор полагал, что управление этими факторами является довольно запутанным процессом, если не организовать их в единую контрольную структуру. В качестве такой структуры лучше всего подходит машина FSM. Теперь автор при создании почти всех GUI применяет машину FSM, сгенерированную компилятором SMC (или его предшественниками).

Рассмотрим машину состояния, код которой приведен в листинге 29.11. Эта машина контролирует GUI для регистрационной части приложения. Получая событие, связанное с запуском, машина выводит экран регистрации. Если пользователь скроет клавишу ввода, машина проверит пароль. Если пароль успешно введен, выполняется переход в состояние `loggedIn` и запускается процесс пользователя (не показан). Если пароль введен неверно, на экране отображается сообщение об этом. Если пользователь игнорирует это сообщение, скрывается кнопка `OK`; если же нет — скрывается кнопка `Cancel`. Если неверный пароль введен три раза подряд (событие `thirdBadPassword`), механизм блокирует экран до тех пор, пока не будет введен пароль администратора.

---

### Листинг 29.11. login.sm

---

```
Initial init
{
 init
 {
 start logginin displayLoginScreen
 }

 logginIn
```

```

{
 enter checkingPassword checkPassword
 cancel init clearScreen
}

checkingPassword
{
 passwordGood loggedIn startUserProcess
 passwordBad notifyingPasswordBad displayBadPasswordScreen
 thirdBadPassword screenLocked displayLockScreen
}

notifyingPasswordBad
{
 OK checkingPassword displayLoginScreen
 cancel init clearScreen
}

screenLocked
{
 enter checkingAdminPassword checkAdminPassword
}

checkingAdminPassword
{
 passwordGood init clearScreen
 passwordBad screenLocked displayLockScreen
}
}

```

---

Здесь в машине состояний отражена высокоуровневая политика приложения. Высокоуровневая политика сосредоточена в одном месте и довольно легко поддерживается. Это значительно упрощает запись оставшейся части кода в системе, поскольку данный код не сливаются с кодом политики.

Ясно, что этот подход может применяться для интерфейсов, отличных от GUI. Конечно, используются и более простые подходы, как для текстовых интерфейсов, так и для интерфейсов компьютер-компьютер. Но GUI обычно усложняется, возрастает и потребность в подобном подходе.

## **Контроллеры, обеспечивающие взаимодействие GUI**

Допустим, что пользователю нужно рисовать на экране прямоугольники. Для этого следует выполнить следующие действия: щелкнуть на пиктограмме прямоугольника из окна палитры. Затем мышь размещается в предполагаемом окне, в одной из вершин прямоугольника. С помощью кнопки мыши выполняется перетаскивание по направлению к предполагаемой второй вершине. Во время перетаскивания на экране отображается анимированное изображение потенциального

прямоугольника. Пользователь манипулирует прямоугольником, получая желаемые очертания и удерживая кнопку мыши в нажатом состоянии в процессе перетаскивания. Затем программа прекращает анимацию и рисует фиксированный прямоугольник на экране.

Конечно, в любой момент пользователь может прервать этот процесс, щелкнув на другой пиктограмме палитры. Если пользователь перетаскивает мышь за пределы предполагаемого окна, анимация исчезает. Если мышь возвращается в окно, анимация появляется вновь.

Наконец, после завершения создания прямоугольника пользователь может нарисовать другой, просто щелкнув кнопкой мыши и выполнив перетаскивание в окне. Нет необходимости щелкать на пиктограмме прямоугольника в окне палитры.

Описанный процесс и представляет собой машину с конечным числом состояний. Диаграмма переходов состояния показана на рис. 29.6. Сплошной круг со стрелкой обозначает начальное положение машины состояния<sup>7</sup>. Сплошной круг с нарисованным открытым кругом отображает заключительное состояние машины.

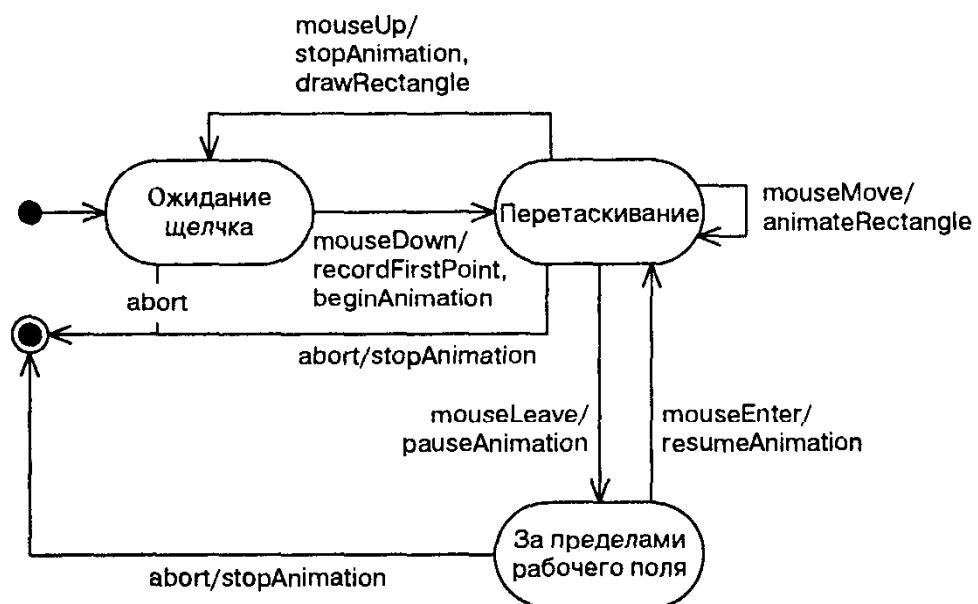


Рис. 29.6. Машина состояний по формированию прямоугольников

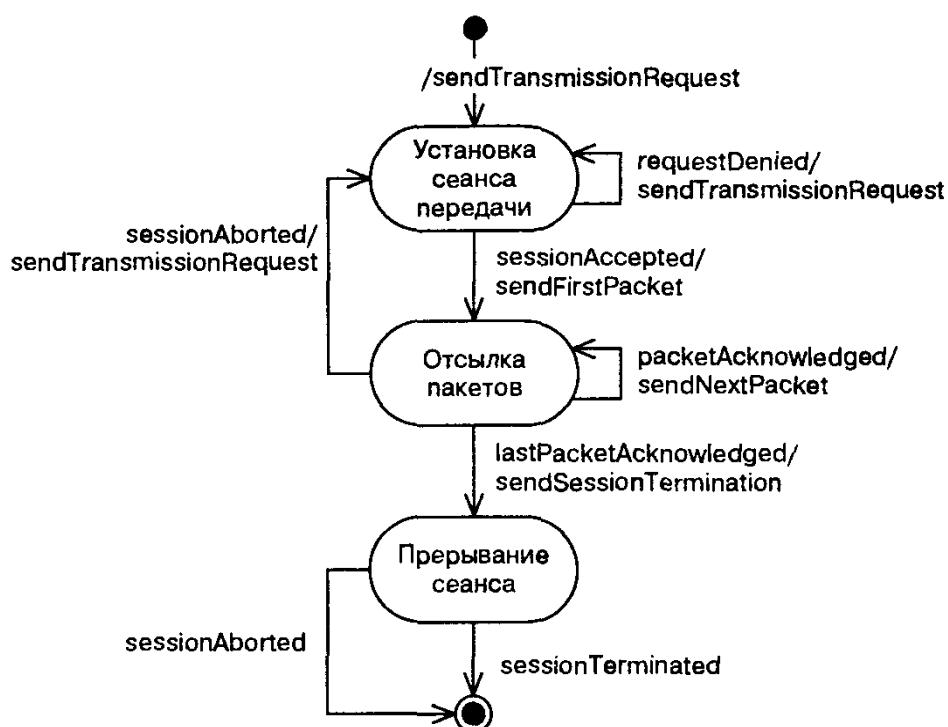
Взаимодействия с GUI обычно характерны для процессов, связанных с машинами с конечным числом состояний. Ими управляет пользователь, вводя события. Эти события приводят к изменению состояния данного взаимодействия.

<sup>7</sup>См. “Состояния и внутренние переходы” в приложении Б.

## Распределенная обработка

Распределенная обработка представляет другую ситуацию, при которой состояние системы изменяется на основе введенных событий. Предположим, что с одного узла сети на другой передается большой массив информации. Также предположим, что время отклика сети очень долго, поэтому следует максимально урезать передаваемый блок информации и пересылать его в виде группы небольших пакетов.

МашинаСостояний, описывающая этот сценарий, показана на рис. 29.7. Она запускается при запросе сеанса передачи, описывает процесс пересылки каждого пакета и ожидания подтверждения, а завершается после прекращения сеанса.



**Рис. 29.7.** Пересылка крупных блоков с помощью большого количества пакетов

## Резюме

Машины с конечным числом состояний используются недостаточно широко. Существует много сценариев, где применение этой машины привело бы к упрощению кода, формированию более гибкого и точного кодирования. Применение шаблона State и простых инструментов по генерированию кода на основе таблиц переходов состояний может оказать существенную помощь на всех этапах обработки кода.

## Листинги

### Применение табличной интерпретации: turnstile.java

Этот листинг демонстрирует, каким образом машину с конечным числом состояний можно реализовать путем интерпретации вектора структур переходов данных. Этот код полностью совместим с TurnstileController из листинга 29.2 и TurnstileTest из листинга 29.3.

---

#### Листинг 29.12. Turnstile.java: применение табличной интерпретации

---

```
import Java.util.Vector;

public class Turnstile
{
 // Состояния
 public static final int LOCKED = 0;
 public static final int UNLOCKED = 1;

 // События
 public static final int COIN = 0;
 public static final int PASS = 1;

 /*private*/ int state = LOCKED;
 private TurnstileController turnstileController;
 private Vector transitions = new Vector ();

 private interface Action
 {
 void execute();
 }

 private class Transition
 {
 public Transition(int currentState, int event,
 int newState, Action action)
 {
 this.currentState = currentState;
 this.event = event;
 this.newState = newState;
 this.action = action;
 }

 int currentState;
 int event;
 int newState;
 Action action;
 }

 public Turnstile(TurnstileController action)
```

```
turnstileController = action;
addTransition(LOCKED, COIN, UNLOCKED, unlock());
addTransition(LOCKED, PASS, LOCKED, alarm());
addTransition(UNLOCKED, COIN, UNLOCKED, thankyou());
addTransition(UNLOCKED, PASS, LOCKED, lock());
}

private void addTransition(int currentState, int event,
 int newState, Action action)
{
 transitions.add(
 new Transition(currentState, event, newState, action));
}

private Action lock()
{
 return new Action(){public void execute(){doLock();}};
}

private Action thankyou()
{
 return new Action(){public void execute(){doThankyou();}};
}

private Action alarm()
{
 return new Action(){public void execute(){doAlarm();}};
}

private Action unlock()
{
 return new Action(){public void execute(){doUnlock();}};
}

private void doUnlock()
{
 turnstileController.unlock();
}

private void doLock()
{
 turnstileController.lock();
}

private void doAlarm()
{
 turnstileController.alarm();
}

private void doThankyou()
```

```
{
 turnstileController.thankyou();
}

public void event(int event)
{
 for (int i = 0; i < transitions.size(); i++)
 {
 Transition transition = (Transition) transitions.elementAt(i);
 if (state == transition.currentState &&
 event == transition.event)
 {
 state = transition.newState;
 transition.action.execute();
 }
 }
}
```

## Генерация с помощью SMC и другие вспомогательные файлы: turnstile.java

Листинги 29.13–29.16 завершают код для примера SMC, связанного с турникетом. Turnstile.java генерируется SMC. Генератор формирует избыточную информацию, но сам код весьма удачен.

### Листинг 29.13. Turnstile.java (сгенерирован SMC)

```

//
// FSM: Turnstile
// Context: TurnstileActions
// Exception: FSMError
// Version:
// Generated: Thursday 09/06/2001 at 12:23:59 CDT

//
// class Turnstile
// Это класс Finite State Machine
//
public class Turnstile extends TurnstileActions
{
 private State itsState;
 private static String itsVersion = "";

 // переменные экземпляров для каждого состояния
 private static Locked itsLockedState;
 private static Unlocked itsUnlockedState;
```

```
// конструктор
public Turnstile()
{
 itsLockedState = new Locked();
 itsUnlockedState = new Unlocked();

 itsState = itsLockedState;

 // Entry функции для: Locked
}

// функции-предки

public String getVersion()
{
 return itsVersion;
}

public String getCurrentStateName()
{
 return itsState.stateName();
}

// функции событий - перед текущим State

public void pass() throws FSMError
{
 itsState.pass();
}

public void coin() throws FSMError
{
 itsState.coin();
}

//

// private class State

// Это базовый класс State

//

private abstract class State
{
 public abstract String stateName();

 // заданные по умолчанию функции событий

 public void pass() throws FSMError
 {
 throw new FSMError("pass", itsState.stateName());
 }
}
```

## Глава 29. Шаблон State

---

```
public void coin() throws FSMError
{
 throw new FSMError("coin", itsState.stateNa
}
}

//

//

// class Locked

// обрабатывает Locked State и его события

//

private class Locked extends State
{
 public String stateName()
 { return "Locked"; }

 //

 // отвечает событию coin
 //

 public void coin()
 {
 unlock();

 // изменить состояние
 itsState = itsUnlockedState;
 }

 //

 // отвечает событию pass
 //

 public void pass()
 {
 alarm();
 // изменяет состояние
 itsState = itsLockedState;
 }
}

//

//

// класс Unlocked

// обрабатывает Unlocked State и его события

//

private class Unlocked extends State
{
 public String stateName()
 { return "Unlocked"; }

 //
 // соответствует событию pass
}
```

```

//

public void pass()

{

 lock();

 // изменяет состояние

 itsState = itsLockedState;

}

//

// соответствует событию coin

//

public void coin()

{

 thankyou();

 // изменяет состояние

 itsState = itsUnlockedState;

}

}
}

```

---

Исключение `FSMError` вызывается компилятором SMC в случае появления недопустимого события. Пример, связанный с турникетом, столь прост, что недопустимое событие отсутствует, поэтому исключение лишено смысла. Но при обращении к большим машинам состояний появляются события, которые не должны происходить при определенных состояниях. Эти переходы не упоминаются в вводных данных SMC. Поэтому, если подобное событие когда-либо произойдет, код генерирует исключение.

---

#### Листинг 29.14. `FSMError.java`

---

```

public class FSMError extends Exception

{

 public FSMError(String event, String state)

 {

 super("Invalid event:" + event + " in state:" + state);

 }
}

```

---

Тестовый код для SMC, генерирующий механизм состояния, подобен во всех остальных тестовых программ, приведенных в этой главе. Причем отличия весьма незначительны.

---

#### Листинг 29.15. `SMCTurnstileTest.java`

---

```

import junit.framework.*;

import junit.swingui.TestRunner;

public class SMCTurnstileTest extends TestCase

```

## Глава 29. Шаблон State

---

```
{
 public static void main(String[] args)
 {
 TestRunner.main(new String[]{"SMCTurnstileTest"});
 }

 public SMCTurnstileTest(String name)
 {
 super(name);
 }

 private TurnstileFSM t;
 private boolean lockCalled = false;
 private boolean unlockCalled = false;
 private boolean thankyouCalled = false;
 private boolean alarmCalled = false;

 public void setUp()
 {
 TurnstileController controllerSpoof =
 new TurnstileController()
 {
 public void lock() {lockCalled = true;}
 public void unlock() {unlockCalled = true;}
 public void thankyou() {thankyouCalled = true;}
 public void alarm() {alarmCalled = true;}
 };

 t = new TurnstileFSM(controllerSpoof);
 }

 public void testInitialConditions()
 {
 assertEquals("Locked", t.getCurrentStateName());
 }

 public void testCoinInLockedState() throws Exception
 {
 t.coin();
 assertEquals("Unlocked", t.getCurrentStateName());
 assert(unlockCalled);
 }

 public void testCoinInUnlockedState() throws Exception
 {
 t.coin(); // находится в состоянии Unlocked
 t.coin();
 assertEquals("Unlocked", t.getCurrentStateName());
 assert(thankyouCalled);
 }
}
```

```

public void testPassInLockedState() throws Exception
{
 t.pass();
 assertEquals("Locked", t.getCurrentStateName());
 assert(alarmCalled);
}

public void testPassInUnlockedState() throws Exception
{
 t.coin(); // unlock
 t.pass();
 assertEquals("Locked", t.getCurrentStateName());
 assert(lockCalled);
}
}

```

---

Класс `TurnstileController` идентичен другим классам, приведенным в этой главе. Можно просмотреть эти классы в листинге 29.2.

Файл `ant` для генерирования кода `Turnstile.java` показан в листинге 29.16. Обратите внимание, что он очень простой. Конечно, если необходимо воспользоваться DOS-окном, можно ввести следующую команду:

`Java smc.Smc -f TurnstileFSM.sm`

---

#### Листинг 29.16. build.xml

---

```

<project name="SMCTurnstile" default="TestSMCTurnstile" basedir=".">
 <property environment="env" />

 <path id="classpath">
 <pathelement path="${env.CLASSPATH}"/>
 </path>

 <target name="TurnstileFSM">
 <java classname="smc.Smc">
 <arg value="-f TurnstileFSM.sm"/>
 <classpath refid="classpath" />
 </java>
 </target>
</project>

```

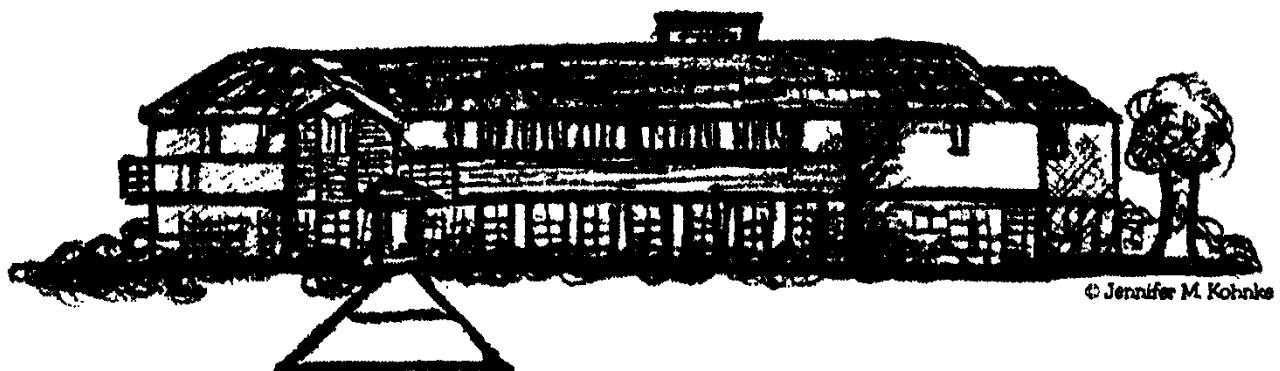
---

## Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley. 1995.
2. Coplien and Schmidt. *Pattern Languages of Program Design*. Reading. MA: Addison-Wesley, 1995.

# 30

## Схема ETS



*Роберт Мартин и Джеймс Ньюкирк*

В данной главе описывается важный программный проект, который разрабатывался с марта 1993 до конца 1997 года. ПО создавалось по заказу Службы по тестированию образовательного уровня (Educational Testing Service, ETS), и авторы главы работали над этой разработкой совместно с программистами из фирмы Object Mentor, Inc.

В главе уделяется пристальное внимание как методическому, так и менеджерскому компонентам, имеющим важное значение для создания повторно используемой схемы. Формирование подобной схемы необходимо для успешного завершения проекта, а процесс разработки и предыстория создания этой схемы поучительны.

Ни один программный проект не разрабатывается в идеальной среде. Чтобы правильно представлять технические аспекты разработки, важно верно учитывать

требования среды. Перед тем как погрузиться в аспекты программного инжиниринга этого проекта, рассмотрим условия, влияющие на рабочий процесс, а также среду разработки.

## Введение

### Обзор проекта

Чтобы в США или Канаде получить лицензию для работы в качестве архитектора, необходимо сдать соответствующий экзамен. После успешной сдачи экзамена государственная лицензионная комиссия выдает лицензию для ведения архитектурной практики. Экзаменационные испытания разработаны Службой по тестированию образовательного уровня (Educational Testing Service, ETS) при непосредственном участии Национального совета по регистрации архитектурных работ (National Council of Architectural Registration Boards, NCARB)).

Тестирование состоит из девяти этапов и проходит в течение нескольких дней. На протяжении трех этапов, относящихся к проверке графических представлений, кандидатам предлагается сконструировать разработки в среде, подобной CAD, рисуя и располагая соответствующим образом объекты. Например, могут быть предложены следующие задания:

- дизайн этажа для строения определенного вида;
- дизайн крыши для существующего строения;
- размещение предложенного строения на участке земли и дизайн окружающего пространства, системы дорог и дорожек для обслуживания строения.

В прошлом кандидаты создавали свои проекты с помощью карандаша и бумаги. Затем эти документы передавались жюри для просмотра.

В 1989 году NCARB поручила ETS сделать заключение о применимости автоматизированной системы для оценки графической части проектов. В 1992 году как ETS, так и NCARB пришли к единому мнению о возможности подобной системы. Более того, именно объектно-ориентированный подход позволял системе соответствовать постоянно меняющимся требованиям. Итак, они обратились к фирме Object Mentor, Inc. (OMI) с предложением заняться этим проектом.

В марте 1993 года с OMI был заключен контракт о частичном создании подобной системы тестирования. Годом позже, после успешного выполнения заказанных работ, с OMI заключен второй контракт по завершению разработки всей системы.

### Структура программы

Структура, выбранная ETS, имеет довольно элегантный вид. Экзамен по графике разбивается на 15 различных проблемных тем, называемых миниатюрами.

Каждая миниатюра служит для тестирования определенной области знаний. Допустим, одна миниатюра тестирует представление кандидата о дизайне потолка, другая — посвящена дизайну этажа.

Каждая миниатюра затем разбивается на два раздела. Раздел, посвященный подаче материала, представляет графический интерфейс пользователя, куда кандидат вручную вносит свое решение проблемы. В разделе оценивания просматривается и оценивается решение, сформированное в предыдущем разделе. Доставка материала производится там, где удобно кандидату. Затем решения передаются на центральный пульт для оценки решений.

## **Сценарии**

Несмотря на то, что имеется только 15 миниатюр, на основе каждой может разрабатываться большое число возможных сценариев. Сценарий уточняет природу предложенного кандидату задания. Например, миниатюра, содержащая план этажа, может иметь один сценарий, предлагающий кандидату создать соответствующий дизайн для библиотеки, в то время как другой сценарий относится к дизайну бакалейной лавки. Поэтому программы миниатюр составлены в обобщенной манере. На этапах доставки материала и оценивания учитывается задача конкретного сценария.

## **Платформа**

Программа формирования экзаменационных тем, а также оценивания результатов экзамена реализована в среде Windows 3.1 (позже была обновлена до уровня Win95/NT). Самы программы написаны на языке C++ с помощью методик объектного ориентирования.

## **Первый контракт**

В марте 1993 года с ОМІ был заключен контракт на разработку этапов доставки и оценивания для всех миниатюр: "Building Design" (Проект здания). Подобное решение мотивировалось рекомендациями Буча (Booch) по разработке элементов, отличающихся высоким уровнем риска.

## **Проект здания**

Проект здания обеспечивает тестирование способности кандидата создать дизайн этажа для относительно простого двухэтажного строения. Кандидату предлагается для дизайна некоторое строение, определяется перечень требований и ограничений. Затем кандидат применял наработки предложенной программы для планировки в своем проекте.

Затем программа оценивания проверяла решение с учетом многих факторов, по которым и определялся уровень знаний кандидата. Речь идет о следующих аспектах.

- Удовлетворяет ли строение запросам клиента?

- Соблюдается ли код строения?
- Смог ли кандидат продемонстрировать логику дизайна?
- Правильно ли расположено строение и его комнаты?

## Начало работы

На раннем этапе над проектом работали только мы вдвоем (Мартин и Ньюкирк). Согласно контракту с ETS, предстояло создать для проекта здания (Building Design) программы формирования тем и оценивания. Также в связи с проектом здания авторам хотелось разработать повторно используемую схему.

В 1997 году уже существовало 15 функциональных программ по формированию экзаменационного материала и оцениванию миниатюр. На это ушло четыре года. Предполагалось, что повторно используемая схема позволит авторам приблизиться к цели. Подобная схема также может оказать неоценимую помощь в совершенствовании содержания и повышении качества миниатюр. В конце концов, схожие возможности в различных миниатюрах желательно обрабатывать аналогичными методами.

Итак, в марте 1993 года авторы приступили к созданию двух компонентов проекта здания, а также схемы, которую можно повторно применять для оставшихся четырнадцати миниатюр.

## Успех

В сентябре 1993 года завершилась работа над первыми версиями программ формирования экзаменационного материала и оценивания результатов, и эти программы демонстрировались представителям NCARB и ETS. Демонстрация прошла успешно, а пробные испытания намечалось провести в январе 1994 года.

Как обычно случается при работе над большинством проектов, после того, как пользователи увидели реально функционирующую программу, они пришли к выводу, что их требования не соответствуют реальным запросам. В течение 1993 года еженедельно заказчикам в ETS направлялись промежуточные версии миниатюр, и непосредственно до сентябрьской демонстрации проекта в разработку постоянно вносились изменения и усовершенствования.

После демонстрации количество изменений и усовершенствований резко возросло. Два автора постоянно вносили изменения в проект, проводили тестирование и готовили программы к “полевым испытаниям”.

Изменения в классификации проекта здания нарастали подобно “снежному кому” и после испытания, что значительно увеличило загрузку авторов даже по сравнению с первым кварталом 1994 года.

В декабре 1993 года начались переговоры по заключению контракта по обработке оставшейся части миниатюр. Эти переговоры заняли три месяца. В марте 1994 года между ETS и OMI был заключен контракт на создание схемы и 10 дополн-

нительных миниатюр. Инженеры компании ETS обрабатывают на основе нашей схемы оставшиеся пять миниатюр.

## Схема

Еще в 1993 года, в самом начале работы над проектом здания, один из авторов (Ньюкирк) в течение недели сотрудничал с одним из инженеров компании ETS, подготавливая его для работы над предстоящим проектом. Цель работы состояла в демонстрации того, каким образом повторно применяемая схема на языке C++, а именно код, состоящий не менее чем из 60 000 строк, может заново использоваться для формирования других миниатюр. Но дело продвигалось тугу. В конце недели стало понятно, что единственная возможность получения повторно используемой схемы состоит в вырезании и вставке частей кода источника в новые миниатюры. Ясно, что эта возможность была не самой оптимальной.

Оглядываясь назад, можно выделить две причины неудач при создании рабочей схемы. Во-первых, в фокусе внимания оказался именно проект здания (Building Design), за исключением всех остальных миниатюр. Во-вторых, в течение многих месяцев основные усилия разработчиков были направлены на устранение требований и соблюдение графика работ. Эти два момента вместе привели к тому, что пристально рассматривались понятия, связанные с Building Design, но схема оставалась вне круга решаемых задач.

В некотором смысле, авторы при заключении контракта переоценили возможности объектно-ориентированной технологии. Предполагалось, что при использовании C++ и тщательном выполнении объектно-ориентированной разработки создание повторно используемой схемы не составит особых трудностей. В этом и состояла ошибка. Пришлось открывать истину — создание повторно используемых схем представляет собой *трудный путь*.

В марте 1994 года, после подписания нового контракта, к проекту подключились еще два инженера и началась разработка новых миниатюр. Авторы продолжали верить в то, что схема необходима и нисколько не поколебались в убеждении, что то, над чем приходилось работать, уводит в сторону от процесса создания подобной схемы. Ясно, что надо было изменять стратегию.

## Команда образца 1994 года

- Роберт Мартин, архитектор и ведущий дизайнер, более 20 лет стажа.
- Джеймс Ньюкирк, дизайнер и руководитель проекта, более 15 лет стажа.
- Бхама Рао, дизайнер и программист, более 12 лет стажа.
- Уильям Митчелл, дизайнер и программист, более 15 лет стажа.

## Срок завершения работ

Срок завершения работ устанавливался таким образом, чтобы тестирование можно было использовать во время экзаменов в 1997 году. Кандидаты проходили тестирование в феврале, а работы оценивались в мае. Это требование носило абсолютный характер.

## Стратегия

При составлении расписания и учете гарантий по качеству и содержательности программы принята новая стратегия, направленная на создание схемы. Сохранены части исходной схемы, состоящей из 60 тысяч строк, но основная часть подверглась изменениям.

## Альтернатива, от которой пришлось отказаться

Одна из возможностей состояла в повторной разработке схемы и полном завершении этого процесса еще до начала работы над миниатюрами. Многие идентифицировали этот процесс с подходом, используемым в архитектуре. Но эта возможность была отвергнута, поскольку большие объемы кодировок схемы невозможно проверить в рамках функционирующих миниатюр. А использовать предположения не хотелось.

Ребекка Вифс-Брок (Rebecca Wirfs-Brock) как-то сказала: “Чтобы в полной мере убедиться, что для данного домена создана правильная архитектура, необходимо сформировать, по меньшей мере, три приложения, которые выполняют роль схемы (а затем отбросить их)<sup>1</sup>. После неудачной попытки создать схему, разработка ее проходила параллельно с созданием нескольких новых миниатюр. Можно было сравнивать аналогичные особенности миниатюр и воплощать их в обобщенной и повторно применяемой форме.

Началась работа над четырьмя миниатюрами в параллельном режиме. Как выяснилось в процессе работы, определенные части оказались аналогичными. Затем они рефакторизовались в более общую форму и применялись во всех трех миниатюрах. Итак, было положено доброе начало при повторном использовании схемы, по крайней мере, в трех миниатюрах.

Аналогичным образом обрабатывались и рефакторизовались части проекта здания (Building Design). Поскольку рассмотренные части успешно функционировали во всех трех миниатюрах, их поместили в схему.

Ниже приводится перечень добавленных в схему общих возможностей.

- Структура окон UI, экранов сообщений, процедура рисования окон, палитры кнопок и т.д.

---

<sup>1</sup>[ВООСН-ОС], с. 275.

- Создание, перемещение, настройка, идентификация и удаление графических элементов.
- Изменение масштаба и прокрутка.
- Создание простых эскизных элементов типа линий, кругов и ломаных линий.
- Время, фиксирующее представление миниатюры, и автоматическое прерывание обработки.
- Сохранение и восстановление файлов решения, включая исправление ошибок.
- Математические модели многих геометрических элементов: прямой, луча, сегмента, точки, прямоугольника, круга, дуги, треугольника, многоугольника и т.д. Эти модели включают такие методы, как Intersection (Нахождение пересечения), Area (Вычисление площади), IsPointIn, IsPointOn и т.д.
- Оценивание и взвешивание отдельных вычислительных свойств.

За восемь месяцев схема увеличилась до 60 000 строк кода на языке C++, что составляет приблизительно годичный объем труда одного программиста. Но эта схема может применяться повторно в четырех различных миниатюрах.

## Результаты

### Игнорирование одной возможности

Что же теперь делать с прежним проектом здания? По мере формирования схемы и успешного повторного ее использования в новых миниатюрах, теряется интерес к этому проекту. Нежелательно применять эту разработку во всех оставшихся миниатюрах, а для поддержки и дальнейшего развития следует использовать другие средства. Несмотря на то, что на разработку проекта здания затрачено более года рабочего времени одного программиста, авторы пришли к решению полностью отказаться от старой версии. Они приступили к повторной разработке и реализации подобного проекта в процессе работы.

### Большой период первоначальной разработки

Негативным результатом выбранной стратегии, связанной со схемой, являлся относительно длительный период разработки первых миниатюр. На разработку программ поставки первых четырех миниатюр потребовалось приблизительно четыре года работы одного программиста.

### Эффективность повторного использования

На момент завершения работ над начальными миниатюрами схема включала приблизительно 60 000 строк кода на языке C++, а программы поставок миниатюр имели небольшие размеры. Каждая программа содержала около 4 000 строк кода

базового кода (т.е. кода, аналогичного для каждой миниатюры). Также в каждой программе было в среднем из 6 000 строк, специфичных для приложения. Самая малая миниатюра содержала немногим более 500 строк кода, относящихся именно к данному приложению, а самая большая — около 12 000 таких строк. Конечно, это просто замечательно, что в среднем пять шестых кода каждой миниатюры взято из схемы. И только одна десятая кода в пределах этих программ уникальна.

## **Производительность труда при разработке программ**

После выполнения первых четырех миниатюр, значительно сократилось время, затрачиваемое на разработку. Поставки семи крупных программ (включая и новый вариант проекта здания) были завершены за 18 человеко-месяцев. Объемы программных кодов этих новых миниатюр остались приблизительно такими же, как кодировки первых четырех миниатюр.

Более того, программа проекта здания, на написание первоначального варианта которой затрачен год работы одного программиста, с помощью схемы заново создана буквально за 2,5 человека-месяцев, т.е. мы достигли почти шестикратного повышения производительности.

С другой стороны, на создание каждой из первых пяти миниатюр, включая проект здания, потребовалось по одному человеко-году. На разработку каждой из последующих миниатюр уже требовалось около 2,6 человеко-месяцев — производительность работы выросла на 400%.

## **График еженедельных версий**

С самого начала работы над проектом и в процессе его реальной разработки, версии для ETS создавались еженедельно. Служба ETS проводила тестирование и оценивание этих версий, затем разработчикам направлялся список изменений. Авторы изучали предложения и разрабатывали совместно с ETS еженедельный график внесения поправок. Сложные изменения или же изменения, не отличающиеся особой важностью, часто откладывались на более поздний период, а основное внимание уделялось более приоритетным поправкам. Таким образом ETS контролировала работу над проектом и выполнение графика.

## **Надежность и гибкость проекта**

Одним из наиболее привлекательных аспектов проекта являлся тот факт, что архитектура проекта и схема существенно перерабатывались с учетом потока изменяющихся требований. По мере продвижения разработки редкая неделя проходила без обработки длинного перечня внесенных изменений и фиксации неточностей. Некоторые изменения возникали при исправлении ошибок, но большая часть была вызвана появлением дополнительных требований. И наряду со всеми обнов-

лениями, важными и неважными замечаниями, внесениями исправлений в разгар работ над проектом, “разработка программного обеспечения шла успешно”<sup>2</sup>.

## Окончательный результат

К февралю 1997 года кандидаты на получение лицензии для работы в качестве архитекторов во время регистрационных экзаменов начали работать с предложенными им программами. В мае 1997 года получены первые оценки. Система хорошо зарекомендовала себя и успешно применяется до сих пор. В настоящее время каждый кандидат на работу в качестве архитектора в Северной Америке проходит экзамены с помощью этого программного обеспечения.

## Разработка схемы

### Общие требования к вычислительным приложениям

Предположим, что необходимо провести тестирование определенных знаний или умений. Организация ETS разработала для программ NCARB довольно доскональную схему. Для иллюстрации этой схемы рассмотрим простой воображаемый пример — тестирование на предмет знания основных математических понятий.

Во время тестирования студентам предлагается 100 математических заданий, которые охватывают большое количество тем — от простого сложения и вычитания до сложных вариантов, связанных с цепочным умножением и делением. Удовлетворительный результат (“прохождение”) означает, что студент проявил хорошо осведомленность и имеет достаточный уровень математической подготовки. Неудовлетворительный результат (“провал”) означает, что студент не овладел этими знаниями и умениями. В некоторых случаях наблюдается некоторая неопределенность, чему и соответствует оценка “сомнительно”.

Но предследуется другая цель. Следует оценить некий общий уровень подготовленности студента. Основные математические сведения разбиваются на подтемы, а затем знания студента оцениваются по каждой из этих подтем.

Например, предположим, что студент неверно определил результат умножения. Возможно, он полагает, что  $7 \times 8$  равно 42. Тогда результаты для большей части операций умножения и деления ошибочны. Естественно студент заслуживает при тестировании неудовлетворительной оценки. С другой стороны, предположим, что студент *абсолютно все*, кроме приведенной операции, выполнил верно. Студент правильно находил промежуточные ответы в цепочных произведениях и верно структурировал задания при нахождении частных. Итак, студент допустил *единственную* ошибку:  $7 \times 8 = 42$ . вполне возможно, что после некоторых наводящих вопросов этот студент получит удовлетворительную оценку (“пройдет” тестирование).

<sup>2</sup>Pete Brittingham, NCARB Project Manager, ETS.

Итак, каким образом следует структурировать оценивание теста, чтобы определить те области основных математических знаний, где студент удовлетворительно “прошел” оценивание, от тех областей, где ему не удалось этого сделать? Рассмотрим диаграмму на рис. 30.1. На этой диаграмме прямоугольниками отмечены экспертные области, входящие в тестирование. Линии отображают иерархическую зависимость. Таким образом, знание основных математических понятий зависит от базовых терминов и фактов, которые, в свою очередь, требуют владения четырьмя основными арифметическими операциями. А эти операции основаны на сложении, которое подчиняется законам коммутативности и ассоциативности.

Ответвляющиеся прямоугольники получили название “возможностей”. Возможности представляют разделы знаний, которые можно оценивать путем указания значений — приемлемо (A), неприемлемо (U) либо “сомнительно” (I). Таким образом, располагая 100 заданиями и соответствующим количеством студенческих ответов, можно каждую возможность применить к заданию и определить оценку. В случае рассмотрения возможности “Слагу” (“Прохождение”) изучается каждое задание на сложение и сравнивается со студенческим ответом. Если студент верно выполнил все задания по сложению, возможности “Слагу” сопоставляется результат ’A’. Если же студент неверно выполнил задание, следует уточнить, является ли ошибка достаточно серьезной. Необходимо рассмотреть различные варианты подобных ошибок, чтобы определить, что привело студента к ошибочному решению. После определения, с высокой степенью вероятности, сути сделанной ошибки, соответствующим образом изменяется оценка по данному свойству. В заключение, оценка, возвращаемая данным свойством, имеет статистический характер. Она основывается на количестве неверных ответов, к которым приводят ошибки при выполнении действий.

Если, например, студент сделал ошибки в половине заданий на сложение, и большинство этих ошибок связаны с неверным подходом к решению задач, для возможности “Слагу” наверняка возвращается значение ’U’. Если, с другой стороны, только четверть ошибок связаны с ошибками, относящимися к возможности “Слагу”, можно возвратить значение ’I’.

И наконец, все возможности оцениваются изложенным выше способом. При тестировании каждая возможность оценивается отдельно. Диапазон оценок для различных возможностей составляет анализ представлений студента по основам математических знаний.

Следующий этап состоит в комбинировании заключительного результата подобного анализа. Чтобы получить этот результат, оценки возможностей объединяются с учетом иерархии, а также весов и матриц. Обратите внимание, что на рис. 30.1 между уровнями иерархии находятся пиктограммы матриц. Матрицы ассоциируются с фактором взвешивания оценки по каждой возможности, а затем для оценивания этого уровня иерархии устанавливается определенная система соотношений. Например, матрица, расположенная под узлом Addition (Сложение),

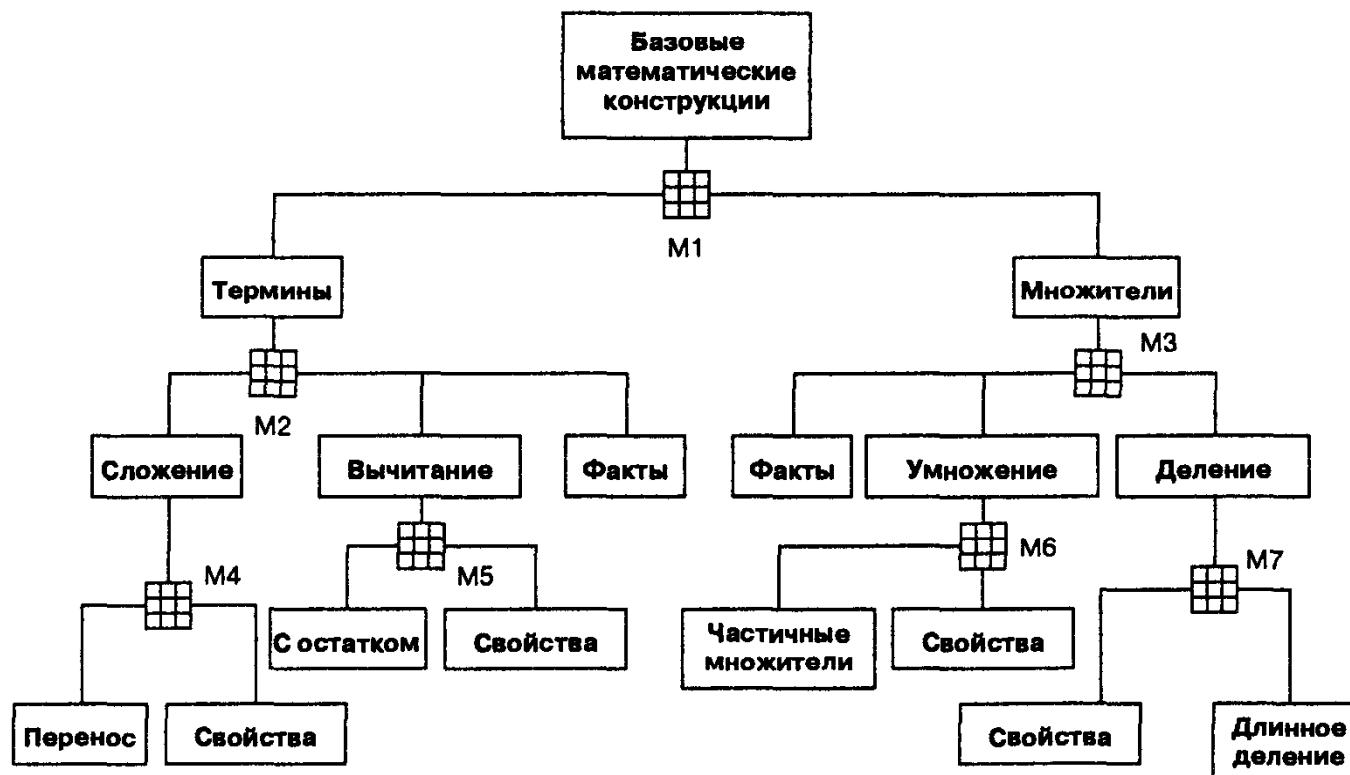


Рис. 30.1. Иерархия возможностей в применении к основным математическим знаниям

|  |  | 0 | 1 | 2 | 3 |   |
|--|--|---|---|---|---|---|
|  |  | 0 | A | I | U | U |
|  |  | 1 | A |   | U |   |
|  |  | 2 | I | U |   |   |
|  |  | 3 | U |   |   |   |

Входы:  
перенос X 2  
свойства X 1

Выход:  
Сложение

Рис. 30.2. Матрица сложения

устанавливает веса, применяемые для оценки возможностей Сагту и Properties, а затем описывает соответствие, генерирующее общую оценку для Addition.

На рис. 30.2 показан вид одной из этих матриц. Вводные данные, определенные возможностью Сагту, имеют большее значение, чем вводные данные, поступившие после рассмотрения возможности Properties, поэтому вес первых данных удваивается. Взвешенные оценки затем складываются, а результат применяется к матрице.

Например, оценка 'I' сформирована благодаря возможности Сагту, а оценка 'A' — Properties. Оценки 'U' отсутствуют, поэтому воспользуемся крайним левым столбцом матрицы. Коэффициентом взвешивания для 'I' является 2, поэтому применяем третью строку этой матрицы, формируя результат для 'I.' Заметим, что матрица имеет "дыры". Они символизируют невозможные условия. При на-

личии текущих коэффициентов взвешивания не существует комбинаций оценок, приводящих к пустым ячейкам этой матрицы.

Подобная схема коэффициентов взвешивания и матриц повторяется на каждом уровне иерархии до тех пор, пока не получим заключительную оценку. Поэтому заключительная оценка является как объединение и повторное объединение оценок для различных возможностей. Структура иерархии нуждается в точной настройке специалистами-психологами из ETS.

## Разработка схемы оценивания

На рис. 30.3 показана статическая структура схемы оценивания. Структуру можно разбить на два основных раздела. Три класса, расположенные справа и выделенные более четким шрифтом, не входят в схему. Они представляют классы, которые следует записывать для каждого определенного оценивающего приложения. Оставшаяся часть классов на рис. 30.3 относится к общим для всех оценивающих приложениям классам схемы.

Самым важным классом в схеме оценивания является **Evaluator**. Этот класс представляет собой абстрактный класс, представляющий как узлы “листьев”, так и матичные узлы дерева оценивания. Функция **Evaluate(ostream&)** вызывается в том случае, если необходимо получить оценку узла на дереве оценивания.

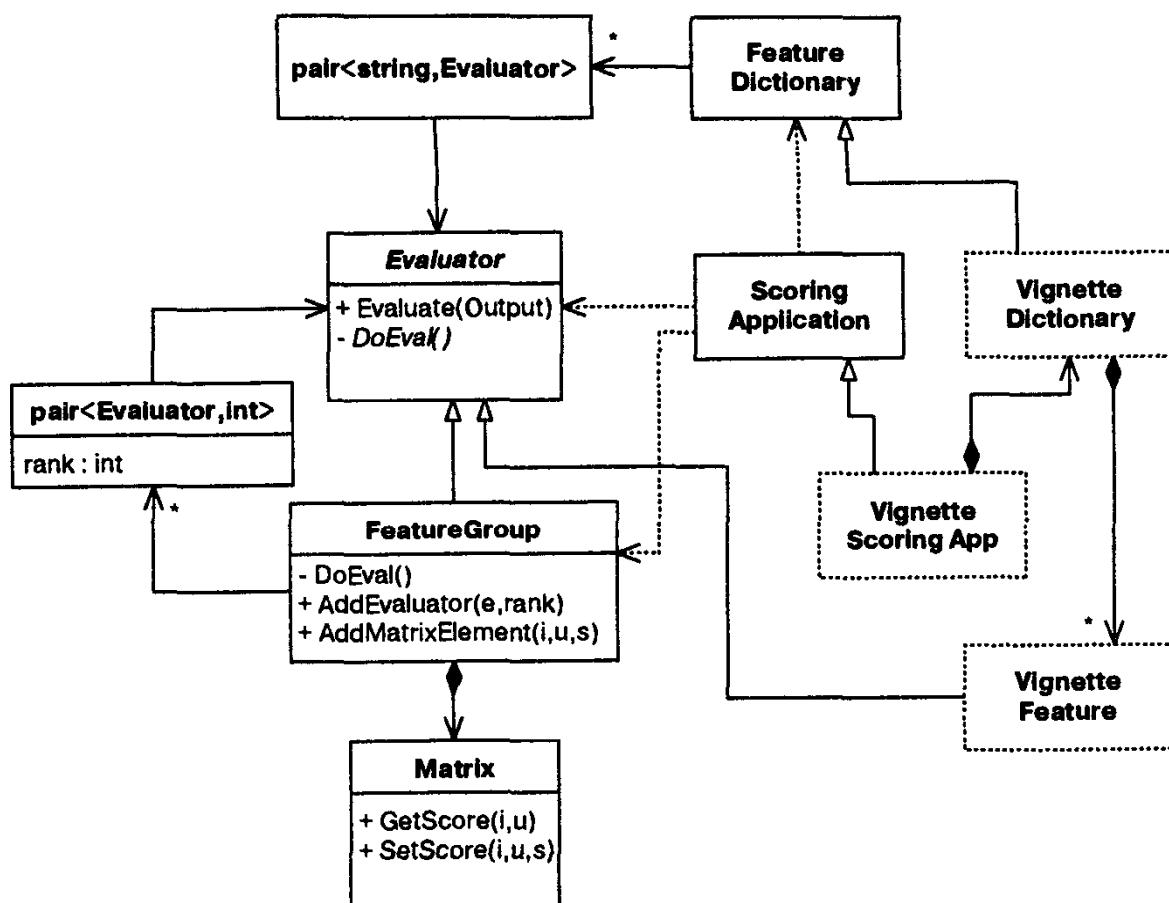


Рис. 30.3. Схема оценивания

Эта функция применяет шаблон Template Method<sup>3</sup>, что позволяет поддерживать стандартный метод регистрации оценок, выводимых устройством вывода.

---

#### Листинг 30.1. Модуль Evaluator

---

```
class Evaluator
{
public:
 enum Score {A, I, U, F, X};
 Evaluator();
 virtual ~Evaluator();

 Score Evaluate(ostream& scoreOutput);
 void SetName(const String& theName) {itsName = theName;}
 const String& GetName() {return itsName;}

private:
 virtual Score DoEval() = 0;

 String itsName;
};
```

---

Обратите внимание на листинги 30.1 и 30.2. Функция `Evaluate()` вызывает частную, чисто виртуальную функцию под названием `DoEval()`. Выполнение этой функции прекращается в случае реального оценивания узла на дереве оценивания. Она возвращает оценку и разрешает `Evaluate()` выводить ее в стандартной форме.

---

#### Листинг 30.2. Метод Evaluator::Evaluate

---

```
Evaluator::Score Evaluator::Evaluate(ostream& o)
{
 static char scoreName[] = {'A', 'I', 'U', 'F', 'X'};
 o << itsName << " : ";
 score = DoEval();
 o << scoreName [score] << endl;
 return score;
}
```

---

Узлы “листьев” на дереве оценивания представлены с помощью класса `VignetteFeature` (рис. 30.3). Действительно, в каждом оценивающем приложении существуют дюжины подобных классов. И каждый раз `DoEval()` отменяется для вычисления оценки некоторой собственной возможности.

Узлы матрицы дерева оценивания представлены с помощью класса `FeatureGroup` (рис. 30.3). Листинг 30.3 демонстрирует, какой вид имеет этот класс. В процессе создания объекта `FeatureGroup` участвуют две функции. Первая функция — `AddEvaluator`, а вторая — `AddMatrixElement`.

---

<sup>3</sup>[GOF95]

---

**Листинг 30.3. FeatureGroup**

---

```
class FeatureGroup : public Evaluator
{
public:
 FeatureGroup(const RWCString& name);
 virtual ~FeatureGroup();

 void AddEvaluator(Evaluator* e, int rank);

 void AddMatrixElement(int i, int u, Score s);

private:
 Evaluator::Score DoEval();
 Matrix itsMatrix;
 vector<pair<Evaluator*,int> > itsEvaluators;
};
```

---

Функция `AddEvaluator` позволяет добавлять дочерние узлы к `FeatureGroup`. Например, возвращаясь к рис. 30.1, можно заключить, что узлом `Addition` должен быть `FeatureGroup`, а `AddEvaluator` вызывается дважды для загрузки в него узлов `Carry` и `Properties`. Функция `AddEvaluator` позволяет указывать разряд определенного оценщика. Разряд представляет собой множитель, применяемый к поступающей от модуля оценке. Поэтому, если `AddEvaluator` вызывается для добавления `Carry` к `Addition FeatureGroup`, следует указать разряд 2, поскольку вес возможности `Carry` вдвое превышает вес возможности `Properties`.

Функция `AddMatrixElement` добавляет к матрице ячейку. Ее следует вызывать для каждой заполняемой ячейки. Например, при создании матрицы на рис. 30.2 используется последовательность вызовов из листинга 30.4.

---

**Листинг 30.4. Генерирование матрицы сложения**

---

```
addition.AddMatrixElement(0,0,Evaluator::A);
addition.AddMatrixElement(0,1,Evaluator::I);
addition.AddMatrixElement(0,2,Evaluator::U);
addition.AddMatrixElement(0,3,Evaluator::U);
addition.AddMatrixElement(1,0,Evaluator::A);
addition.AddMatrixElement(1,2,Evaluator::U);
addition.AddMatrixElement(2,0,Evaluator::I);
addition.AddMatrixElement(2,1,Evaluator::U);
addition.AddMatrixElement(3,0,Evaluator::U);
```

---

Функция `DoEval` просто выполняет итерацию, используя список оценщиков. При этом соответствующие оценки умножаются на разряд, и произведение добавляется к соответствующим аккумуляторам оценок `I` и `U`. После однократного

заполнения эти аккумуляторы применяются в качестве матричных индексов при получении заключительной оценки. (листинг 30.5.)

---

#### Листинг 30.5. Метод FeatureGroup::DoEval

---

```
Evaluator::Score FeatureGroup::DoEval()
{
 int sumU, sumI;
 sumU = sumI = 0;
 Evaluator::Score s, rtnScore;
 Vector<Pair<Evaluator*, int> >::iterator ei;
 ei = itsEvaluators.begin()

 for(; ei != itsEvaluators.end(); ei++)
 {
 Evaluator* e = (*ei).first;
 int rank = (*ei).second;

 s = e.Evaluate(outputStream);

 switch(s)
 {
 case I:
 sumI += rank;
 break;
 case U:
 sumU += rank;
 break;
 }
 } // для ei
 rtnScore = itsMatrix.GetScore(sumI, sumU);
 return rtnScore;
}
```

---

Остается один вопрос: как формируется дерево оценивания? Очевидно, что психологи из службы ETS должны иметь возможность изменять топологию и коэффициенты взвешивания дерева без внесения реальных изменений в приложение. Поэтому дерево оценивания формируется с помощью класса Vignette-ScoringApp (рис. 30.3).

Каждое приложение по оцениванию имеет для этого класса собственную реализацию. Благодаря этому классу формируется производный модуль для класса FeatureDictionary. Этот класс включает отображение строк к указателям Evaluator.

При запуске приложения по оцениванию контроль переходит к схеме оценивания. В классе ScoringApplication вызывается метод, благодаря которому создается соответствующий производный модуль для FeatureDictionary. Затем просматривается специальный текстовый файл, описывающий топологию

дерева оценивания и его коэффициенты взвешивания. Этот текстовый файл идентифицирует возможности с помощью специальных названий. Эти названия ранее ассоциировались с соответствующими указателями `Evaluator` в `FeatureDictionary`.

Поэтому в своей простейшей форме приложение оценивания является ничем иным, как набором возможностей и методом, конструирующими `FeatureDictionary`. Формирование и оценивание дерева оценок выполняется с помощью схемы и является общим для всех приложений, выполняющих оценивание.

## Использование шаблона Template Method

Одна из миниатюр тестирует способности кандидатов создавать план этажа для некоторого строения, например, библиотеки или полицейского отделения. При работе с этой миниатурой кандидат рисует комнаты, коридоры, двери, окна, проемы в стенах, лестницы, лифты и т.д. Программа конвертирует рисунок в структуру данных, которую может интерпретировать программа по оценке работы. Модель объекта имеет вид, показанный на рис. 30.4.

Объекты в этой структуре данных имеют минимальные функциональные возможности. Их трудно причислить к полиморфным объектам, они представляют собой простые носители данных. Речь идет о модели с чисто представительскими свойствами.

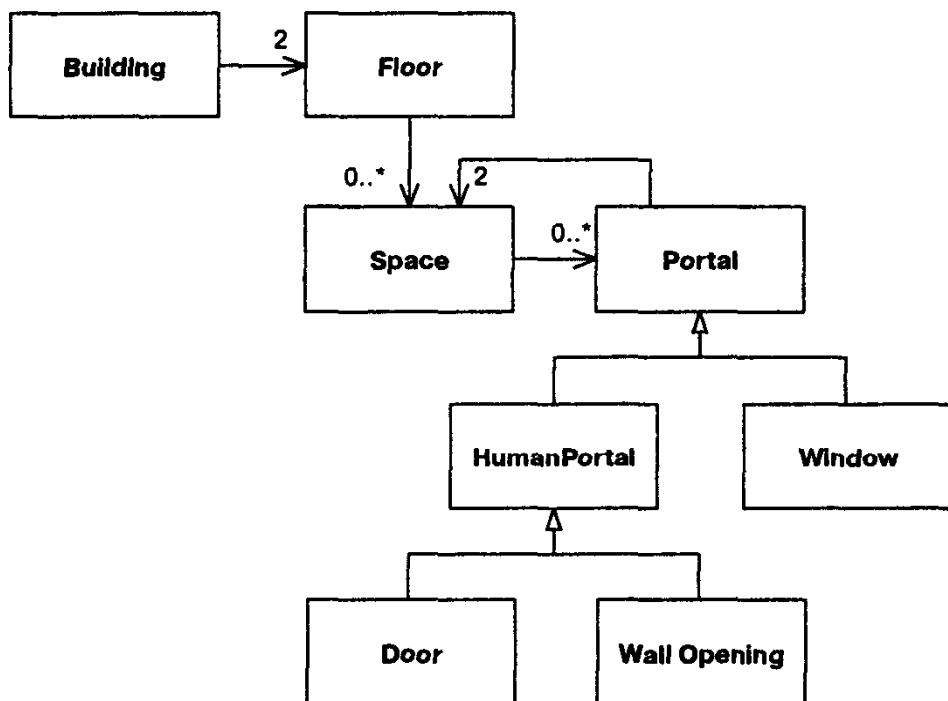


Рис. 30.4. Структура данных, применяемых в процессе планирования этажа

В данном случае рассматривается двухэтажное строение. На каждом этаже находится много помещений, в каждом из которых есть много проходов, ведущих в другие помещения. Причем проходами могут быть и окна либо проемы.

Оценка работы выполняется путем тестирования предложенного решения с учетом набора некоторых *свойств*. Этими качествами служат, например, следующие факторы.

- Нарисовал ли студент все необходимые помещения?
- Характеризуется ли каждое помещение соответствующими относительными размерами?
- Есть ли вход в каждое помещение?
- Указаны ли окна для помещений, имеющих наружные стены?
- Видны ли горы из окна офиса президента?
- Удобен ли доступ к “черному ходу” из кухни?
- Удобно ли сообщение между закусочной и кухней?
- Легко ли перемещаться по коридорам и получать доступ в каждую из комнат?

Психологи из службы ETS высказали пожелание об удобстве изменения формы матрицы оценивания. Они хотели бы изменять коэффициенты взвешивания, перегруппировывать возможности в различные подиерархии и т.д. Также высказывалось пожелание об удалении ненужных возможностей наряду с добавлением новых. Большая часть этих манипуляций связана с изменением единственной конфигурации текстового файла.

Исходя из интересов исполнителя, остановимся на подсчете включенных в матрицу возможностей. Итак, для каждой возможности сформируем классы. Каждый из этих классов *Feature* располагает методом *Evaluate*, при помощи которого просматривается представленная на рис. 30.4 структура данных, а также вычисляется оценка. Это значит, что дюжины и дюжины классов *Feature* обрабатывают одну и ту же структуру данных. Имеем впечатляющее дублирование кода.

## Однократная запись цикла

Чтобы работать с дублированием кода, попробуем применить шаблон *Template Method*. Впервые этим шаблоном авторы воспользовались в 1993 и 1994 годах, еще задолго до того, как узнали о работе с шаблонами. Принцип действия авторов получил название “однократной записи цикла” (листинги 30.6 и 30.7). Приводим реальные модули на языке C++ из этой программы.

---

**Листинг 30.6. solspcft.h**

---

```
/* $Header: /Space/src_repository/ets/grande/vgfeat/
solspcft.h,v 1.2 1994/04/11 17:02:02 rmartin Exp $ */

#ifndef FEATURES SOLUTION_SPACE_FEATURE_H
#define FEATURES SOLUTION_SPACE_FEATURE_H

#include "scoring/eval.h"

template <class T> class Query;

class SolutionSpace;
//-----
// Название
// SolutionSpaceFeature
//
// Описание
// Этот класс является базовым, который
// поддерживает цикл, который обрабатывает
// набор пространств решений и затем находит
// все соответствующие пространства решений.
// Чисто-виртуальные функции поддерживаются
// при обнаружении пространства решений.
//

class SolutionSpaceFeature : public Evaluator
{
public:
 SolutionSpaceFeature(Query<SolutionSpace*>&);
 virtual ~SolutionSpaceFeature();
 virtual Evaluator::Score DoEval();
 virtual void NewSolutionSpace(const SolutionSpace&) = 0;
 virtual Evaluator::Score GetScore () = 0;

private:
 SolutionSpaceFeature(const SolutionSpaceFeature&);
 SolutionSpaceFeature& operator= (const SolutionSpaceFeature&);
 Query<SolutionSpace*>& itsSolutionSpaceQuery;
};

#endif
```

---



---

**Листинг 30.7. solspcft.cpp**

---

```
/* $Header: /Space/src_repository/ets/grande/vgfeat/
solspcft.cpp,v 1.2 1994/04/11 17:02:00 rmartin Exp $ */

#include "componen/set.h"
```

```
#include "vgsolut/solspc.h"
#include "componen/query.h"
#include "vgsolut/scfilter.h"
#include "vgfeat/solspcft.h"

extern ScoringFilter* GscoreFilter;

SolutionSpaceFeature::SolutionSpaceFeature(Query<SolutionSpace*>& q)
: itsSolutionSpaceQuery(q) {}

SolutionSpaceFeature::~SolutionSpaceFeature() {}

Evaluator::Score SolutionSpaceFeature::DoEval()
{
 Set<SolutionSpace*>& theSet = GscoreFilter->GetSolutionSpaces();
 SelectiveIterator<SolutionSpace*>ai(theSet, itsSolutionSpaceQuery);
 for (; ai; ai++)
 {
 SolutionSpace& as = **ai;
 NewSolutionSpace(as);
 }
 return GetScore();
}
```

Этот код был разработан в 1994 году, поэтому он имеет немного “странный” вид для тех, кто использует STL. И все же, если не обращать внимания на дополнительные итераторы, здесь представлен классический шаблон Template Method. Функция DoEval с помощью цикла охватывает все объекты SolutionSpace. Затем вызывается чисто виртуальная функция NewSolutionSpace. Производные модули SolutionSpaceFeature реализуют NewSolutionSpace и оценивают каждое пространство с помощью определенного критерия оценивания.

Производные модули SolutionSpaceFeature включают те возможности, которые отвечают за включение соответствующих пространств в решение, за наличие подходящих площадей и соблюдение пропорций, за правильное размещение лифтов и т.д.

Примечательным является размещение в одном месте цикла, охватывающего структуру данных. Все свойства по оцениванию являются наследниками этого цикла, а не реализуют его заново.

Некоторые возможности служат для измерения характеристик порталов, связанных с помещениями. Поэтому шаблон используется заново, создается класс PortalFeature, который является производным от SolutionSpaceFeature. Реализация NewSolutionSpace в пределах PortalFeature представляет собой цикл по всем проходам аргумента SolutionSpace и вызывает чисто виртуальную функцию NewPortal(const Portal&). (рис. 30.5.)

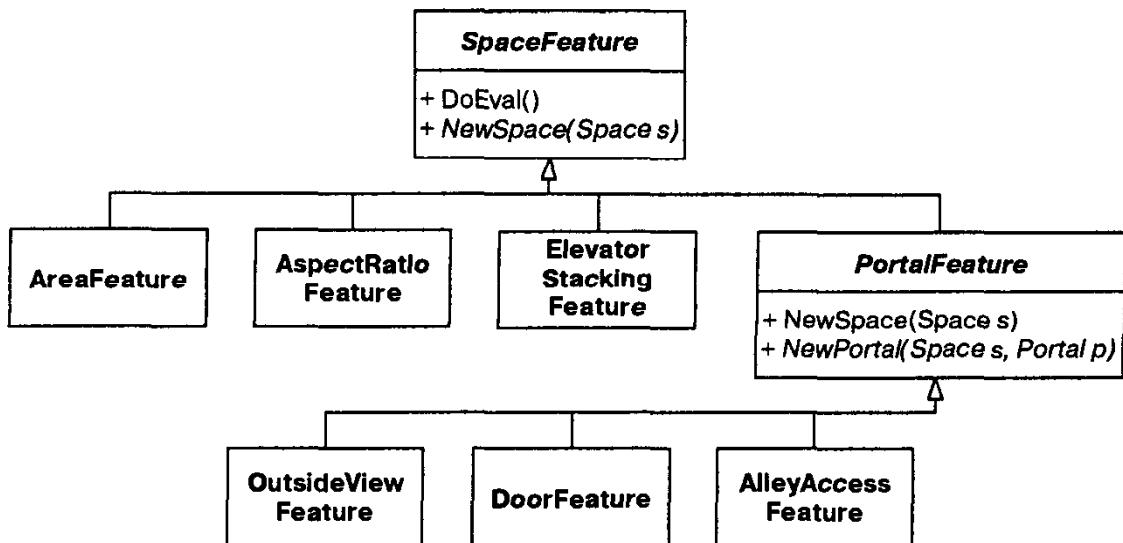


Рис. 30.5. Структура Template Method для свойств оценивания

Данная структура позволяет создавать много различных возможностей по оцениванию, каждая из которых отличается структурой данных по планировке этажа. Причем обычно неизвестно, какой вид примет структура данных при планировке этажа. Если изменятся подробности структуры данных по планировке этажа (например, вместо собственных итераторов будут использоваться STL), необходимо будет вносить изменения только в два класса.

Почему шаблон *Template Method* был предпочтен *Strategy*<sup>4</sup>? Рассмотрим, насколько свободнее будет соединение при использовании *Strategy*! (рис. 30.6.)

При работе с шаблоном *Template Method* в случае, если вносятся изменения в алгоритмы, обрабатывающие структуру данных, необходимо вносить изменения в *SpaceFeature* и *PortalFeature*. По всей вероятности, в данном случае необходимо рекомпилировать *все* свойства. Но при использовании шаблона *Strategy* изменения ограничиваются двумя классами *Driver*. Практически не возникает необходимости в обязательной повторной компиляции свойств.

Так почему же для применения выбран шаблон *Template Method*? Этот шаблон проще. Дело также в том, что структура данных обычно изменяется редко. А также повторная компиляция всех свойств занимает всего несколько минут.

Итак, несмотря на то, что наследственные зависимости шаблона *Template Method* приводят к более компактно связанной разработке, а также шаблон *Strategy* лучше согласуется с принципом DIP, чем шаблон *Template Method*, все эти преимущества *не стоят двух дополнительных классов*, которые возникают при реализации шаблона *Strategy*.

<sup>4</sup>Ясно, что авторы не думали об этом относительно данных терминов. На момент принятия решения названия шаблонов еще не были придуманы.

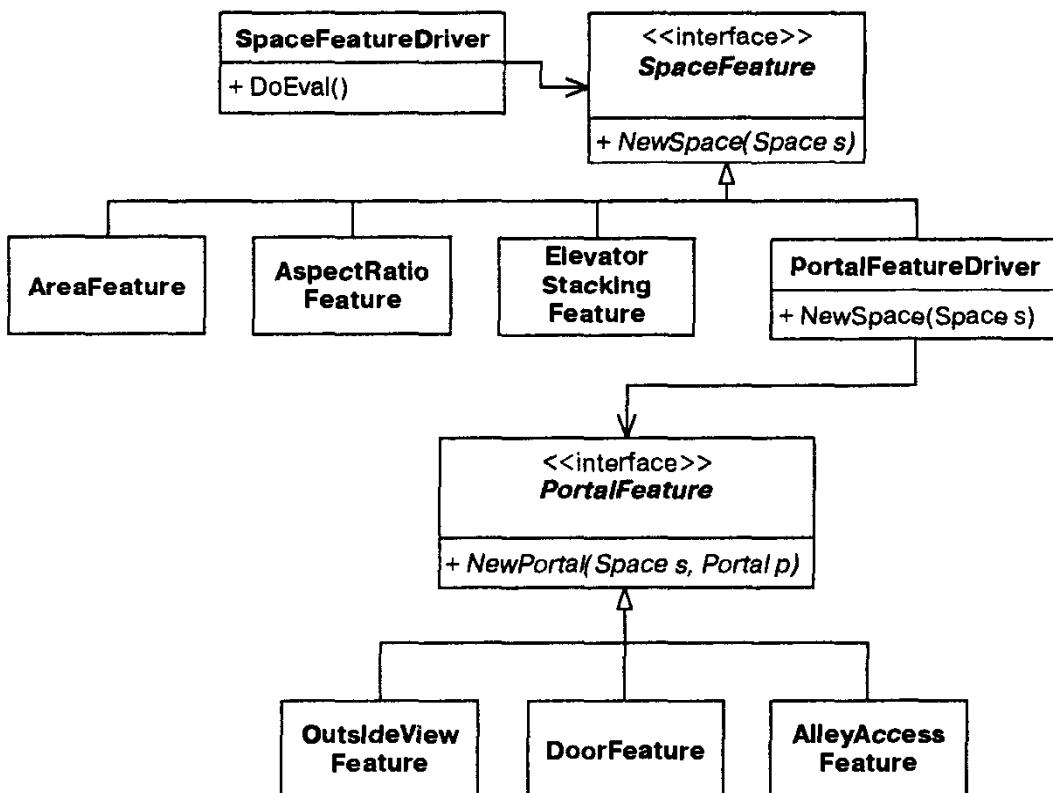


Рис. 30.6. Структура по оцениванию планировки этажа с помощью шаблона Strategy

## Общие требования к разрабатываемым приложениям

Различные программы, составляющие систему, немного перекрываются. Например, структура экрана одинакова для всех миниатюр. В левой части экрана находится окно, не содержащее ничего, кроме столбца кнопок. Это окно называется окном команд (“Command Window”). Кнопки в окне команд действуют как элементы управления приложением. Они отмечены метками **Place Item**, **Erase**, **Move/Adjust**, **Zoom** и **Done**. При щелчках на этих кнопках приложение будет выполнять требуемые действия.

Справа от окна команд находится окно задач (**Task Window**). Это довольно большая по площади область, где можно выполнять прокрутку и изменять масштаб. Именно сюда пользователь может вносить свое решение. Команды, вызываемые в окне команд, обычно применяются для обновления содержимого окна задач. Конечно, большинство команд, вызываемых с помощью окна команд, существенным образом используются в пределах окна задач.

Например, при размещении комнаты в плане этажа пользователь щелкает на кнопке **Place Item** в **Command Window**. После этого появляется меню, содержащее возможные варианты комнат. Пользователь выбирает вид комнаты, которую необходимо разместить на этаже. Затем пользователь перемещает мышь в окне **Task Window** и щелкает на то место, куда следует поместить эту комнату. В зависимости от миниатюры, закрепляется левый верхний угол изображения комнаты,

на которую щелкнул пользователь. Нижний левый угол изображения “растягиваемой” комнаты следует за движениями мыши в Task Window до тех пор, пока пользователь не щелкнет вторично, закрепляя в этом положении нижний левый угол этого изображения.

Выполняемые действия подобны, но не идентичны для каждой миниатюры. В некоторых миниатюрах не используются изображения комнат, а применяются контурные линии, линии свойств или потолки. Несмотря на имеющиеся отличия, общая парадигма оперирования в пределах миниатюр довольно сходна.

Благодаря этому сходству можно заново применять одни и те же приемы. Необходимо создать объектно-ориентированную схему, отражающую большое число сходных черт, а также удобным образом выделяющую отличия. И тогда успех обеспечен.

## Разработка схемы поставки версий

Итак, код схемы ETS увеличился в объеме до 75 000 строк кода. Ясно, что затруднительно отобразить все подробности данной схемы. Поэтому для изучения были выбраны два наиболее иллюстративных элемента схемы: модель события и архитектура мастера по решению задач (taskmaster).

### Модель событий

Каждое предпринятое пользователем действие приводит к генерированию событий. Если пользователь щелкает на кнопке, генерируется событие, связанное с названием кнопки. Если пользователь выделяет пункт меню, генерируется событие, название которого и отображает этот пункт. Важной проблемой при формировании схемы является упорядочивание этих событий.

Эта проблема возникает вследствие того, что с помощью схемы можно обрабатывать большую часть событий, и все же каждая отдельная миниатюра при обработке с помощью схемы определенного события может иметь свои особенности. Итак, необходимо найти возможность, согласно которой иногда в миниатюрах можно аннулировать обработку событий.

Проблема усложнялась тем, что перечень элементов не являлся законченным. В каждой миниатуре из Command Window выбирается определенный набор кнопок, наряду с пунктами меню. Поэтому в схеме необходимо упорядочивать события, общие для всех миниатюр, и в то же время каждой миниатуре позволено отменять заданную по умолчанию обработку; также необходимо, чтобы каждая миниатюра в определенном порядке располагала собственные, специфические для данной миниатюры события. А это задание не из легких.

В качестве примера рассмотрим рис. 30.7. На диаграмме<sup>5</sup> показана небольшая часть машины с конечным числом состояний, которая применяется для упоря-

---

<sup>5</sup>Система обозначений для диаграмм состояния полностью приведена в приложении Б.

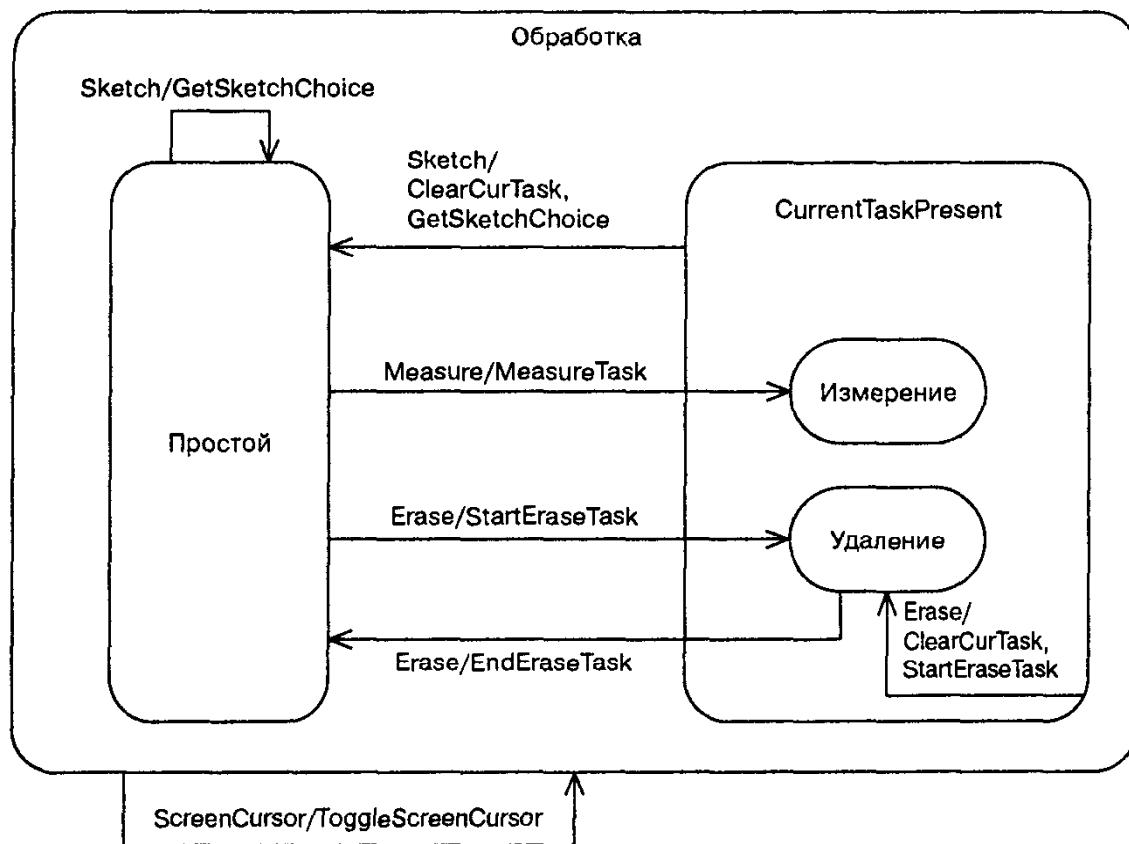


Рис. 30.7. Процессор событий Command Window

дочивания событий, происходящих в Command Window некоторой миниатюры. Каждая миниатюра имеет собственную специальную версию этой машины с конечным числом состояний.

На рис. 30.7 показано, каким образом взаимодействуют три различных типа событий. Сначала рассмотрим простейший случай, событие ScreenCursor. Это событие генерируется, если пользователь щелкает на кнопке Change Cursor. Если пользователь щелкает на кнопке, курсор в Task Window переключается между стрелкой и полноэкранным изображением перекрестия. Поэтому, несмотря на измененное состояние курсора, состояние процессора события не изменится.

Если пользователь пожелает удалить нарисованный им объект, он щелкает на кнопке Erase. Затем производится щелчок на пункте или пунктах в Task Window, которые необходимо удалить. Наконец, пользователь опять щелкает на кнопке Erase для подтверждения удаления. Механизм состояния на рис. 30.7 показывает, как работает процессор событий Command Window. Первое событие Erase приводит к переходу из состояния Idle в состояние Erasing и запускает Erase Task. В следующем разделе более подробно рассматриваются Tasks. В настоящий момент достаточно знать, что Erase Task связан со всеми событиями, происходящими в Task Window.

Обратите внимание, что даже в состоянии Erasing событие ScreenCursor функционирует должным образом и не накладывается на операцию erase. Заме-

тим, что вне состояния *Erasing* есть две возможности. Если происходит другое событие *Erase*, тогда *Erase Task* завершается, подтверждая устранение, а механизм состояния переходит назад, в состояние простоя (*idle*). Это и является обычным вариантом завершения операции *erase*.

Кроме того, если в процессе удаления щелкнуть на кнопке *Command Window*, которая начинает выполнение другой задачи (например, на кнопке *Sketch*), тогда прекращается выполнение *Erase Task*, и удаление отменяется.

На рис. 30.7 показано, как это правило применяется совместно с событием *Sketch*, но есть также не показанные здесь иные события, которые взаимодействуют аналогичным образом. Если пользователь выбирает кнопку *Sketch*, независимо от того, находится система в состоянии *Erasing* либо в состоянии *Idle*, система перейдет в состояние *Idle* и произойдет вызов функции *GetSketchChoices*. Эта функция представит меню *Sketch* (набросок), включающий список операторов, с которыми может работать пользователь. Одной из этих операций является измерение (*measure*).

Если пользователь выбирает из меню *Sketch* пункт *measure*, происходит событие *Measure* и запускается выполнение задачи измерения (*Measure Task*). В процессе измерения пользователь может щелкать на две точки в *Task Window*. Эти две точки отмечены крошечными перекрестиями (tiny little cross hairs), и расстояние между ними указывается в небольшом окне сообщения, в нижней части экрана. Затем пользователь может щелкать еще на две точки, затем еще и еще и т.д. При выполнении задачи измерения отсутствует возможность обычного выхода. Вместо этого пользователь должен щелкать на кнопке, которая приводит к запуску другого задания, например, *Erase* или *Sketch*.

## Разработка модели события

На рис. 30.8 показана статическая модель классов, которые реализуются процессором событий *Command Window*. Находящаяся справа иерархия представляет *CommandWindow*, представленная слева иерархия описывает машину с конечным числом состояний, транслирующую события в действия.

Классы *CommandWindow*, *StandardCommandWindow* и *StandardFSM* являются схематическими классами. Оставшаяся часть специфична для данной миниатюры. *CommandWindow* поддерживает реализации стандартных действий, таких как *MeasureTask* и *EraseTask*.

События принимаются *VignetteCommandWindow*. Они передаются машине с конечным числом состояний, которая транслирует их в действия. Затем эти действия передаются в иерархию *CommandWindow*, которая их реализует.

Класс *CommandWindow* поддерживает реализации для таких стандартных действий, как *MeasureTask* и *EraseTask*. “Стандартное действие” представляет собой действие, общее для всех миниатюр. *StandardCommandWindow* поддерживается для упорядочения стандартных событий для машины с конечным

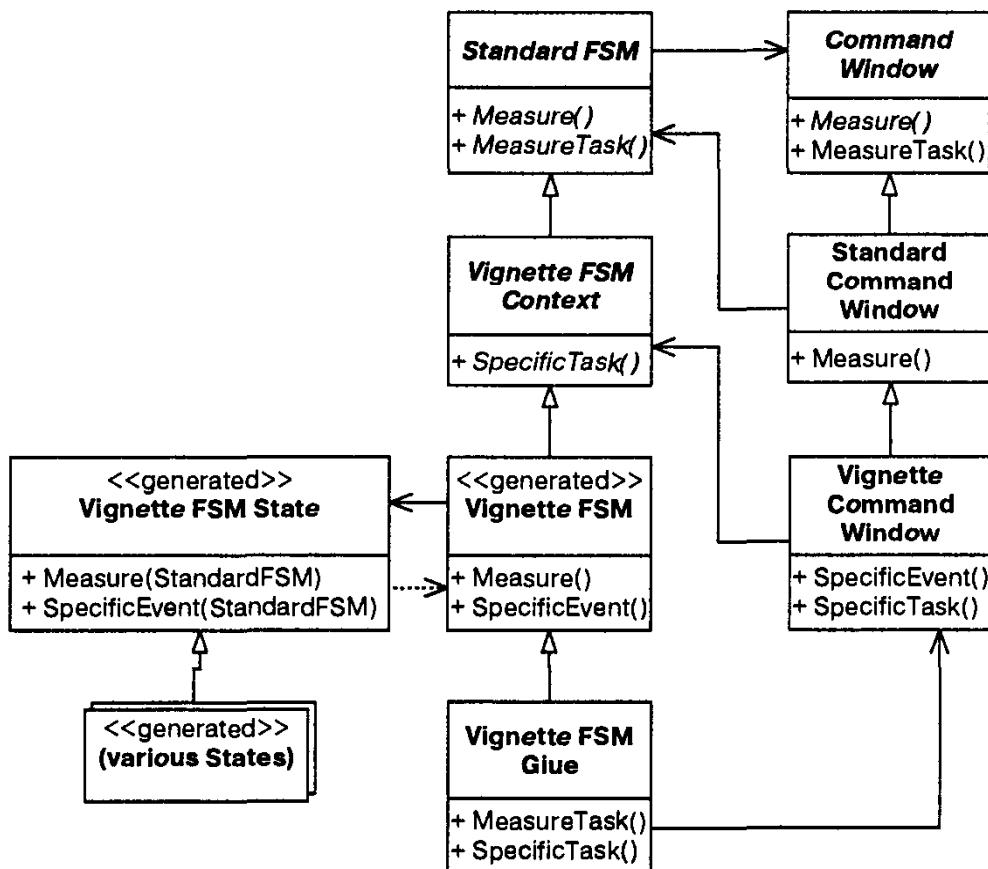


Рис. 30.8. Статическая модель процессора событий Command Window

числом состояний. **VignetteCommandWindow** является специфичной для ми- ниатюры и поддерживается как при реализации определенных действий, так и при упорядочении некоторых событий. Также позволяет отменить стандартные ре- ализации и процедуры упорядочения.

Таким образом, схема поддерживает заданные по умолчанию реализации и упорядочения для всех общих заданий. И ни одна из этих реализаций или упоря- дочений не может отменяться миниатюрой.

### Отслеживание стандартного события

На рис. 30.9 показано, каким образом стандартное событие для машины с ко- нечным числом состояний упорядочивается и транслируется в стандартное дей- ствие. Message 1 является событием **Measure**. Оно поступает из GUI и передается в **VignetteCommandWindow**. Заданное по умолчанию упорядочение для этого события поддерживается **StandardCommandWindow**, поэтому в сообщении 1.1 событие пересыпается к **StandardFSM**.

Класс **StandardFSM** является схематическим классом, который поддержи- вает интерфейс для всех входящих стандартных событий и всех выходящих стан- дартных событий. Ни одна из этих функций не реализуется на этом уровне.

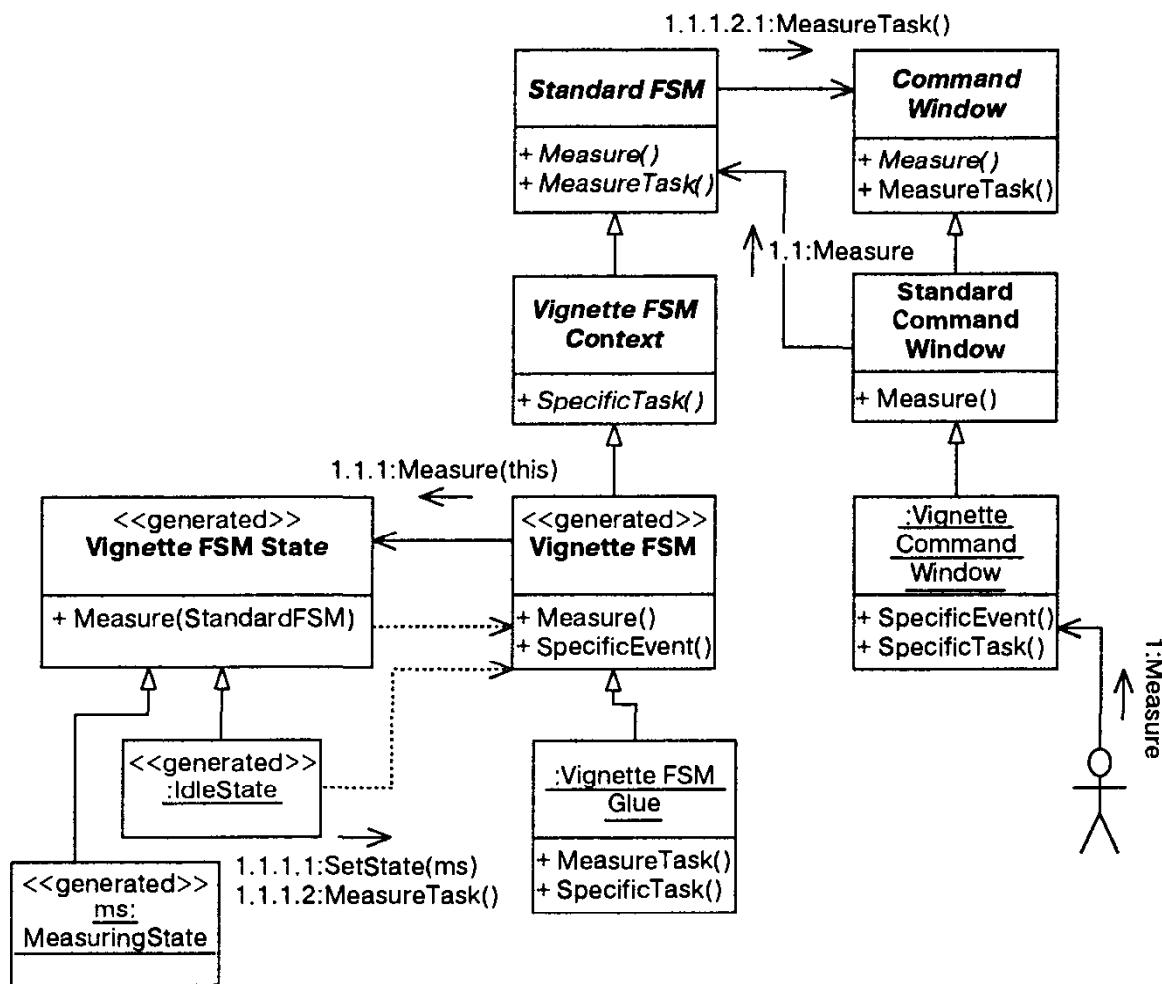


Рис. 30.9. Обработка события Measure

*VignetteFSMContext* добавляет для событий и действий, присущих именно данной миниатюре, интерфейсы, но не реализации.

Реальная работа по трансляции события к действию проводится в классах *VignetteFSM* и *VignetteFSMState*. *VignetteFSM* содержит реализации для всех функций событий. Итак, сообщение 1.1: *Measure* развертывается вниз на этом уровне. *VignetteFSM* в качестве отклика пересыпает 1.1.1: *Measure (this)* к объекту *VignetteFSMState*.

Класс *VignetteFSMState* является абстрактным классом. Для каждого состояния машины с конечным числом состояний существуют производные модули этого класса. При создании рис. 30.9 было принято, что текущим состоянием FSM является *Idle* (рис. 30.7). Поэтому сообщение 1.1.1: *Measure (this)* разворачивается к объекту *IdleState*. Этот объект откликается на пересылку двух сообщений назад, к *VignetteFSM*. Первым сообщением является 1.1.1.1: *SetState (ms)*, оно изменяет состояние FSM на состояние *Measuring*. Вторым сообщением является 1.1.1.2: *MeasureTask()* и представляет собой действие, требуемое событием *Measure* в состоянии *Idle*.

Сообщение *MeasureTask* в конечном итоге реализуется в классе *VignetteFSMGlue*, который распознает действие в виде стандартного действия, объ-

явленного в `CommandWindow`, и таким образом направляет его в сообщение 1.1.1.2.1: `MeasureTask`, тем самым, замыкая круг.

Механизмом по конвертированию событий в действия является шаблон `State`. Этот шаблон широко используется во многих разделах схемы, что и будет показано в следующем разделе. “Общий” стереотип, который появляется в классах шаблона `State`, указывает, что эти классы автоматически генерируются SMC.

## Отслеживание события, специфичного для миниатюры

На рис. 30.10 показано, что произойдет при наличии события, специфического для миниатюры. Опять сообщение 1: `SpecificEvent` воспринимается `VignetteCommandWindow`. Но поскольку упорядочивание специфических событий реализуется на этом уровне, `VignetteCommandWindow` пересыпает сообщение 1.1: `SpecificEvent`. Более того, сообщение пересыпается к классу `VignetteFSMContext`, где впервые объявлен метод `SpecificEvent`.

Аналогично, событие разворачивается для `VignetteFSM`, которое в сообщении 1.1.1: `SpecificEv` согласуется с текущим состоянием объекта для генерирования определенного действия. Как и раньше, объект состояния откликается двумя сообщениями, 1.1.1.1: `SetState` и 1.1.1.2: `SpecificTask`.

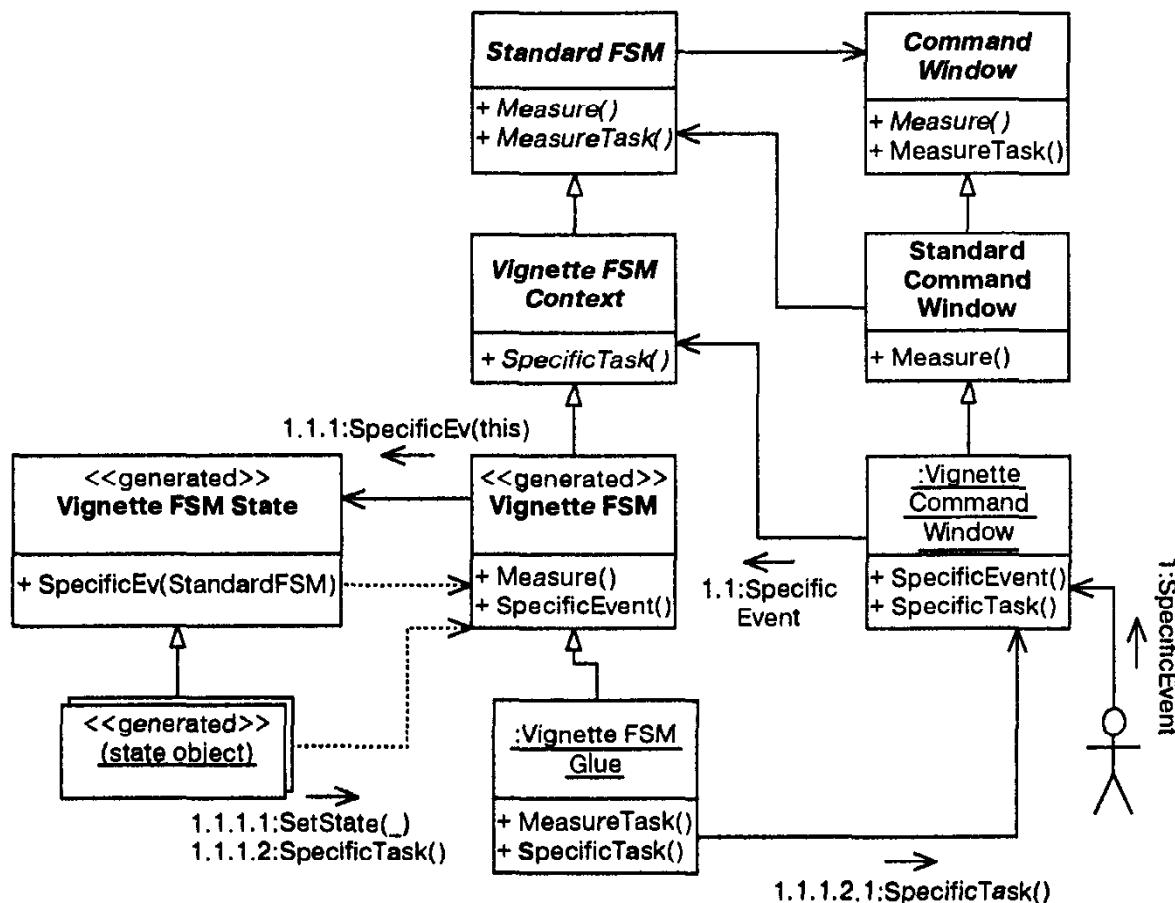


Рис. 30.10. Обработка специфического события

Сообщение разворачивается вниз, к `VignetteFSMGlue`. Но в то же время оно распознается как действие, специфичное для миниатюры, и поэтому направляется непосредственно к `VignetteCommandWindow`, где реализуются специфические действия.

## Генерирование и повторное использование машины состояний Command Window

На этом этапе трудно не удивиться обилию классов, которые применяются для упорядочения событий и действий. Следует обратить внимание, что, несмотря на множество классов, имеется лишь несколько объектов. Конечно, созданные экземпляры объектов являются просто `VignetteCommandWindow`, `VignetteFSMGlue`, а также различными объектами состояния, которые являются тривиальными и генерируются автоматически.

Несмотря на то, что поток объектов кажется сложным, в действительности они довольно просты. Окно проверяет событие, передает его машине FSM для трансляции к действию и опять получает доступ к действию из машины FSM. Оставшаяся часть сложных действий связана с отделением стандартных действий, известных как схема, от специфических действий, относящихся к миниатюре.

Другим фактором, оказавшим влияние на принятие решения о разбиении классов, является использование SMC для автоматического генерирования классов машины с конечным числом состояний. Рассмотрим следующее описание и вернемся к рис. 30.7:

```
Idle
{
 Measure Measuring MeasureTask
 Erase Erasing StartEraseTask
 Sketch Idle GetSketchChoice
}
```

Обратите внимание, что этот простой текст описывает все переходы, которые могут иметь место, если механизм состояния находится в состоянии `Idle`. Три строки в скобках идентифицируют событие, вызывающее переход, целевое состояние перехода и действие, выполняемое при переходе.

Компилятор SMC<sup>6</sup> получает доступ к тексту в этой форме и генерирует классы, обозначенные как “генерированные”. Данный код, генерируемый SMC, не требует ни редактирования, ни проверки.

Использование SMC для генерирования машины состояний значительно упрощает процесс создания этой части процессора события. Разработчик записывает `VignetteCommandWindow` вместе с соответствующими реализациями для определенных событий и действий. Также разработчик должен записать `VignetteFSMGlue`.

---

<sup>6</sup>SMC, State Machine Compiler, свободно распространяемый продукт, который доступен на Web-узле <http://www.objectmentor.com>.

teFSMContext, который просто объявляет интерфейсы для определенных событий и действий. Затем разработчик должен записывать класс VignetteFSMGlue, который просто отсылает действия назад к VignetteCommandWindow. Ни одна из этих задач обычно не вызывает затруднений.

Перед разработчиком стоят еще и другие задачи. Он должен создать описание машины с конечным числом состояний для SMC. Эта машина состояний в действительности довольно сложна. Диаграмма, приведенная на рис. 30.7, не отражает всей сложности процесса. В реальной миниатуре используется несколько дюжин событий, каждое из которых имеет свое собственное поведение.

К счастью, большинство миниатюр подобны друг другу. Поэтому в качестве модели можно использовать описание стандартного механизма состояния, а для каждой миниатюры вносить относительно небольшие модификации. Таким образом, каждая миниатюра имеет собственное описание машины FSM.

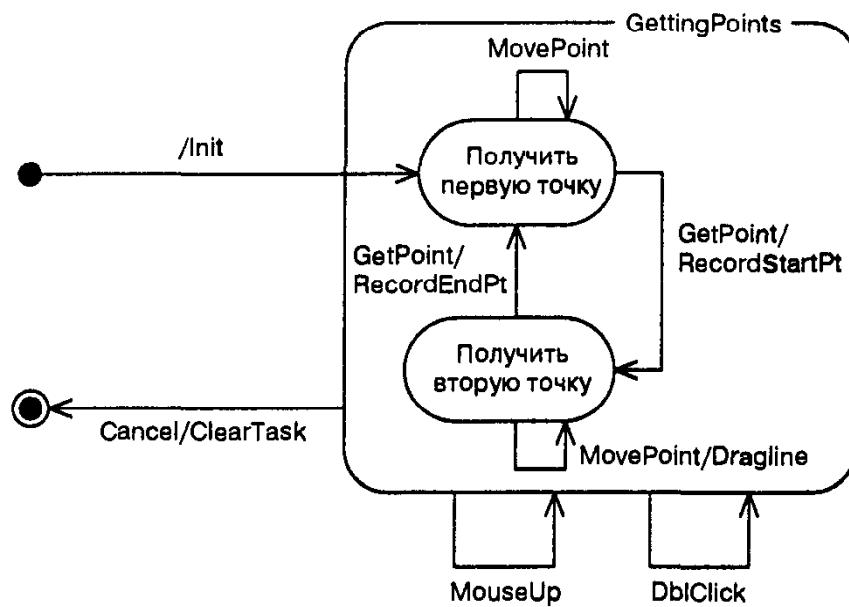
Этот подход не является вполне удовлетворительным, поскольку файлы, описывающие машину FSM, очень похожи. Конечно, бывают ситуации, когда в механизм генерирования состояния необходимо вносить изменения. Это значит, что в каждый из файлов, описывающих FSM, вносятся идентичные или почти идентичные изменения. Но этот путь довольно утомителен, а также может способствовать появлению ошибок.

Вполне возможно изобрести несколько иную схему отделения генерирующей части описания машины FSM от определенной части. Но потом придется убедиться в том, что этот подход не стоит приложенных усилий. Это вовсе не то решение, которое избирают вторично.

## Архитектура мастера задач (Taskmaster)

У нас уже была возможность увидеть, как события конвертируются в действия, а также каким образом преобразование зависит от сравнительно сложной машины с конечным числом состояний. Теперь нам придется изучить, как обрабатываются действия. Забегая вперед, отметим, что мы приедем к выводу: каждое действие управляет машиной с конечным числом состояний, причем это обстоятельство отнюдь не радует разработчиков.

Рассмотрим MeasureTask, обсуждаемый в предыдущем разделе. Пользователь вызывает это задание, если необходимо измерить расстояние между двумя точками. После вызова пользователь щелкает на точку в TaskWindow. В этой точке отображается небольшое “перекрестье”. Если пользователь переместит мышь вкруговую, появляется растягиваемая линия, которая соединяет точку, где выполнен щелчок, с текущим положением мыши. Более того, текущая длина этой линии отображается в отдельном окне сообщения. Если пользователь щелкает вторично, появляется другое “перекрестье”, растягиваемая линия исчезает, и в окне сообще-



**Рис. 30.11.** Машина с конечным числом состояний, моделирующая задачу об измерениях

ния появляется заключительное значение расстояния между двумя точками. Если пользователь щелкает еще раз, процесс начинается снова.

После того как процессор событий выберет `MeasureTask`, как показано на рис. 30.9, `CommandWindow` создает реальный объект `MeasureTask`, а затем запускается машина с конечным числом состояний, показанная на рис. 30.11.

`MeasureTask` начинает существование, вызывая функцию `init` и затем принимая состояние `GetFirstPoint`. GUI-события, происходящие в `TaskWindow`, направляются к заданию, которое выполняется в настоящее время. Поэтому, если пользователь перемещает мышь в `TaskWindow`, текущая задача получает сообщение `MovePoint`. Заметим, что в состоянии `GetFirstPoint` ничего не происходит (как и ожидалось).

Если пользователь наконец щелкает в `Task Window`, происходит событие `GotPoint`. Это приводит к переходу к состоянию `GetSecondPoint` и вызову действия `RecordStartPt`. Это действие приведет к появлению “перекрестья”, а также к запоминанию расположения, где отправной точкой был щелчок мыши.

Событие `MovePoint` в состоянии `GetSecondPoint` приводит к вызову действия `Dragline`. Этим действием устанавливается режим XOR<sup>7</sup> и рисуется линия, начиная с оставшейся в памяти отправной точки до текущего положения мыши. Также вычисляется расстояние между этими двумя точками и отображается в окне сообщений.

<sup>7</sup>XOR-режим представляет собой такой режим, где можно разместить GUI. В этом случае существенно упрощается задача перетаскивания растягиваемых линий или форм поверх существующих форм на экране. Если вам пока непонятно, что же будет происходить, не беспокойтесь.

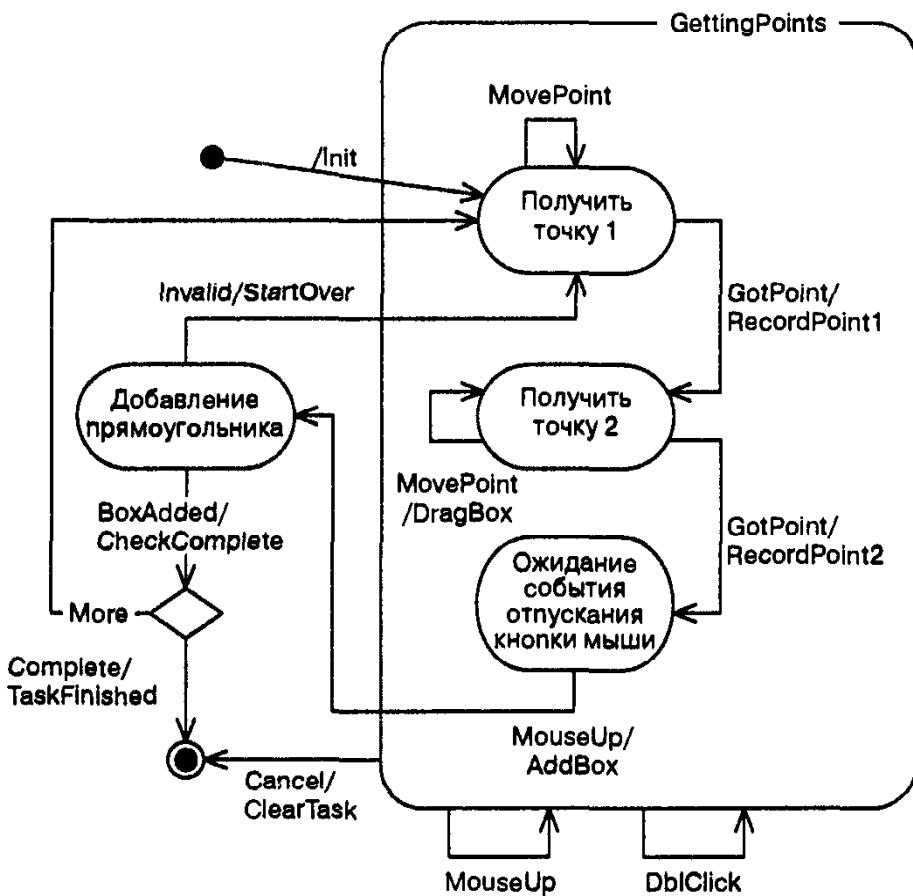


Рис. 30.12. Двухточечный прямоугольник

Чаще всего события MovePoint происходят, если мышь перемещается поверх Task Window, поэтому движение линии и ее длина, отображаемая в окне сообщения, будет постоянно обновляться до тех пор, пока движется мышь.

Если пользователь щелкнет вторично, выполнится переход назад, к состоянию GetFirstPoint, и вызовется действие RecordEndPt. Это действие отключает режим xor, удаляет линию между первой точкой и текущим положением мыши, рисует “перекрестие” в точке щелчка и отображает расстояние между точкой щелчка в окне сообщения и начальной точкой.

Эта последовательность событий повторяется столько, сколько пожелает пользователь. Она прерывается, если задание отменяется с помощью CommandWindow, возможно, в качестве ответа на щелчок пользователя на командной кнопке.

На рис. 30.12 показано немногого более сложное задание — рисование “двуточечного” прямоугольника. Двухточечный прямоугольник представляет собой прямоугольник, который рисуется на экране с помощью двух щелчков. Первый щелчок закрепляет один угол. Затем за мышью следует растягиваемый прямоугольник. Если пользователь щелкает вторично, прямоугольник становится постоянным.

Как и раньше, задание начинается в состоянии GetPoint1, после вызова Init. В этом состоянии игнорируется движение мыши. Если щелкнуть мышью,

осуществляется переход к `GetPoint2` и вызов действия `RecordPoint1`. Это действие записывает точку щелчка в качестве отправной точки.

В состоянии `GetPoint2` движение мыши приводит к вызову действия `DragBox`. Эта функция устанавливает режим `XOR` и рисует растяжимый прямоугольник из отправной точки до текущего положения мыши.

Если происходит второй щелчок мышью, осуществляется переход к состоянию `WaitingForUp` и вызов `RecordPoint2`. Эта функция просто записывает заключительную точку для этого прямоугольника. Режим `XOR` не отменяется, удаляется растяжимый прямоугольник или рисуется реальный прямоугольник, поскольку нет уверенности в достоверности прямоугольника.

На этом этапе мышь остается нажатой, и пользователь не пропустить кнопку мыши. Но с этим торопиться не следует, в противном случае некоторое другое задание приведет к событию `mouse-up` и все запутает. При ожидании игнорируется любое движение мыши, причем прямоугольник закреплен в точке, где последний раз щелкали мышью.

После активизации мыши выполняется переход в состояние `AddingBox` и вызов функции `AddBox`. Эта функция проверяет, правильно ли скомпонован прямоугольник (`box`). Существует много причин, по которым прямоугольник может быть неправильно скомпонован. Он может быть “вырожденным” (т.е. начальная и конечная точки оказываются одинаковыми), или же он может конфликтовать с другими частями рисунка. Каждая миниатюра имеет свои особенности и может отклонить пользовательский рисунок.

Если прямоугольник (`box`) будет нарисован неверно, генерируется событие `Invalid` и машина состояния переходит назад, к `GetPoint1`, и в то же время вызывается функция `StartOver`. Если же прямоугольник (`box`) выполнен верно, генерируется событие `BoxAdded`. Тогда вызывается функция `CheckComplete`. Это — вторая функция, специфичная для миниатюры. Она определяет, следует ли пользователю продолжать рисование других прямоугольников, или же задание завершается.

Без преувеличения, схема содержит очень много подобных заданий. Каждое подобное задание представлено с помощью производного модуля класса `Task`. (Рис. 30.13.) Каждое задание имеет в своих рамках машину с конечным числом состояний, и сложность этих заданий постепенно возрастает. Поэтому не представляется возможным показать эти задания здесь. Аналогично, каждая из этих машин состояния генерируется SMC.

На рис. 30.13 показана архитектура Taskmaster. Эта архитектура связывает `CommandWindow` с `Taskwindow`, формирует задачи, выбранные пользователем, и контролирует их выполнение.

На диаграмме показаны две задачи, машины FSM для которых изображены на рис. 30.11 и 30.12. Заметим, что применяется шаблон `State` и классы, генерируемые в каждой из этих задач. Все эти классы, вплоть до `MeasureTaskImplemen-`

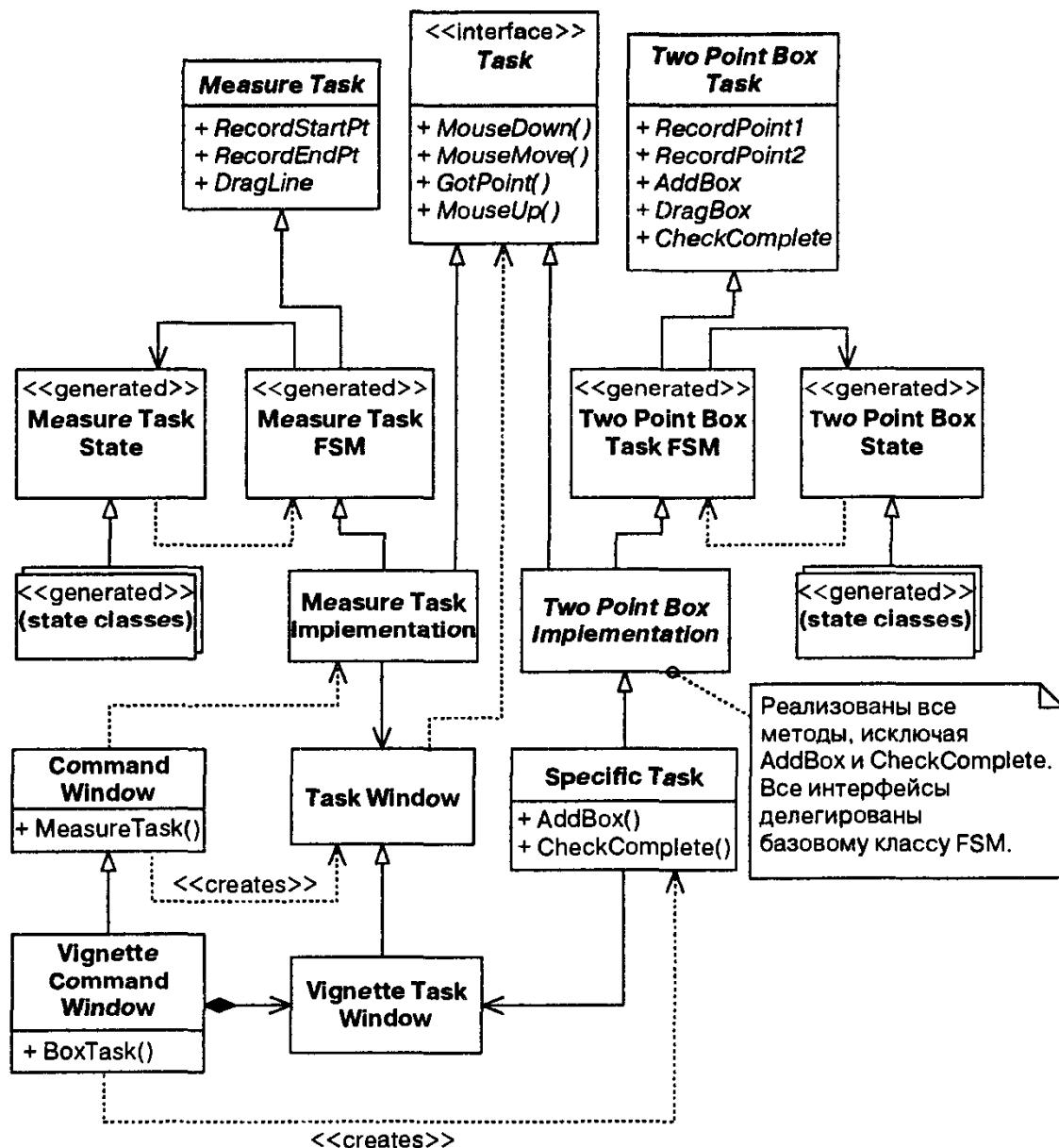


Рис. 30.13. Архитектура Taskmaster

`plementation` и `TwoPointBoxImplementation`, входят в схему. Конечно, единственными классами, которые должен записать разработчик, являются `VignetteTaskWindow` и специфические производные модули для классов задач.

Классы `MeasureTaskImplementation` и `TwoPointBoxImplementation` представляют большое количество включенных в схему разнообразных задач. Заметим, что эти классы являются абстрактными. Существует несколько функций, среди которых `AddBox` и `CheckComplete`, которые не реализуются. Каждая миниатюра реализует эти функции в случае необходимости.

Таким образом, задания, включенные в схему, управляют множеством взаимодействий во всех миниатюрах. Пожелает ли разработчик нарисовать прямоугольник (draw a box) или объект, относящийся к прямоугольнику (box), тогда разработчик может обратиться к новому заданию из `TwoPointBoxImplementation`.

Если же необходимо с помощью единственного щелчка просто разместить на сцене некоторые объекты, можно отменить `SinglePointPlacementTask`. Если же возникает необходимость выполнить рисунок на основе ломаной линии, можно отменить `PolylineTask`. Эти задачи контролируют взаимодействия, выполняют весь объем перетаскивания и позволяют инженеру проверять достоверность имеющихся и формировать новые объекты.

## Резюме

Конечно, можно было бы дополнительно рассмотреть элементы схемы, относящиеся к вычислительной геометрии, или же части схемы, отвечающие за хранение и просмотр решений, направляемых в файлы и получаемых из них. Также можно было бы обсудить структуру параметрических файлов, которые разрешают каждому миниатюрному приложению реализовать большое количество вариаций в одной и той же миниатуре. К сожалению, объем книги не позволяет реализовать эти замыслы.

Однако авторы полагают, что рассмотренные аспекты схемы являются наиболее показательными. Стратегии, примененные в этой схеме, можно реализовать и в других схемах, получая таким образом собственные повторно используемые схемы.

## Литература

1. Booch G. *Object-Oriented Design with Applications*. Redwood City. CA: Benjamin Cummings, 1991.
2. Booch G. *Object Solutions*. Menlo Park. CA: Addison-Wesley, 1996.
3. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

# **ЧАСТЬ V**

## **Приложения**

# A

## UML-запись I: CGI-пример

В ходе анализа и проектирования ПО выполняется процесс, который требует внедрения некой формы записи. До настоящего времени предпринималось множество попыток разработки подобной формы записи. И некоторые из них были вполне удачными, в результате чего появились блок-схемы, диаграммы потоков данных, диаграммы связи между сущностями, а также некоторые другие формы записи.

Приход эры объектно-ориентированного программирования повлек за собой “взрывообразное” возникновение самых различных форм записи. Появились буквально десятки форм записи, применяемых для представления результатов объектно-ориентированного анализа и проектирования.

Ниже приводится краткий перечень наиболее распространенных форм записи:

- Booch94<sup>1</sup>;
- ОМТ (Object Modeling Technique, техника моделирования объектов), описанная Рамбахом (Rumbaugh) и другими авторами<sup>2</sup>;
- RDD (Responsibility Driven Design, проектирование, управляемое системой ответственостей), описанная Верф-Броком и другими авторами<sup>3</sup>;
- запись Кода-Йордона (Coad/Yourdon), описанная Питером Кодом (Peter Coad) и Эдом Йордоном (Ed Yourdon)<sup>4</sup>.

Среди перечисленных форм записи наиболее важными являются Booch 94 и ОМТ. Запись Booch 94 славится строгим форматом фраз, описывающих этап проектирования, в то время как ОМТ лучше подходит для записи результатов.

В данном случае имеет место дилемма, которая представляет некоторый интерес. В 80-е и 90-е годы прошлого века одним из преимуществ объектно-ориентированного подхода считалось то, что результаты анализа и проектирования могут представляться одной и той же формой записи. Вполне возможно, что

---

<sup>1</sup>[BOOCH94]

<sup>2</sup>[RUMBAUGH91]

<sup>3</sup>[WIRFS90]

<sup>4</sup>[GOAD91A]

это была всего лишь “защитная реакция”, вызванная строгим разделением между записями структурированного анализа и проектирования. Попытки преодолеть раскол между структурированным анализом и проектированием были заранее обречены на провал.

С появлением объектно-ориентированных форм записи оказалось, что одна и та же форма записи может использоваться в процессе анализа и проектирования. Но еще через десять лет оказалось, что аналитики и разработчики проектов выбрали свои любимые формы записи. Специалисты в области анализа склонялись в сторону ОМТ, а разработчики проектов предпочли запись Booch 94. В результате оказалось, что одной формы записи недостаточно на “все случаи жизни”. Запись, предназначенная для фиксации результатов анализа, не может применяться в процессе проектирования и наоборот.

Запись UML имеет простую форму, но ей присуща высокая степень универсализма. Отдельные формы этой записи могут применяться для фиксации результатов, в то время как другие формы — в процессе проектирования. Поэтому специалисты в области анализа и проектирования могут пользоваться UML-записью.

В настоящей главе рассматриваются две точки зрения относительно UML-записи. Сначала описывается ее применение в процессе анализа, а затем переход к этапу проектирования. Данное описание будет производиться в форме небольшого практического занятия.

Обратите внимание на то, что порядок рассмотрения, предусматривающий на первом месте анализ, а на втором — проектирование, является искусственным и не может рекомендоваться для обязательного применения. Тем более что ни в одном из практических занятий, рассмотренных в книге, не различаются эти две стадии разработки проекта. Таким образом, лишь упрощается рассмотрение методов использования UML-записи на различных уровнях абстракции. В реальных проектах все уровни абстракции задействуются одновременно, а не последовательно.

## **Система записи на курсы: описание проблемы**

Предположим, что мы имеем дело с компанией, которая организует профессиональные учебные курсы по объектно-ориентированному анализу и проектированию. В этом случае потребуется система, позволяющая отслеживать изучаемые в данный момент времени курсы, а также обеспечивающая учет студентов, посещающих эти курсы. Подробное описание этой системы приводится ниже.

### **Система учета посещаемости курсов**

Пользователи должны получить доступ к меню, где перечислены все доступные учебные курсы. В этом меню они могут выбрать понравившиеся им курсы. После завершения выбора отображается всплывающее окно, в котором пользователь может ввести следующую информацию.

- Имя (Name).
- Телефонный номер (Phone number).
- Номер факса (Fax number).
- Адрес электронной почты (E-mail address).

Также должен быть предусмотрен способ внесения платежей за курсы. В частности, доступен один из следующих способов.

- Чек (Check).
- Счет-фактура (Purchase order).
- Кредитная карта (Credit card).

Если пользователь выбрал платеж по чеку, отображается форма, в которой следует указать номер чека.

В случае, когда пользователь выбрал метод платежа по кредитной карте, отображается форма, в которой нужно указать номер кредитной карты, срок ее действия, а также имя в том виде, в котором оно написано в кредитной карте. Если же пользователь выбрал метод платежа с помощью счет-фактуры, в отобразившейся на экране форме нужно указать номер счета-фактуры (Р0#), название компании, а также имя и фамилию человека, ответственного за выписку счета-фактуры.

Как только будет внесена вся указанная выше информация, пользователь может щелкнуть на кнопке Submit (Отправить). При этом появляется другой экран, на котором отображается вся введенная пользователем ранее информация. Здесь также отображаются указания напечатать копию экрана, подписать печатную копию, а затем отослать ее по факсу в центр, выполняющий запись на курсы.

Можно также отослать этот документ по электронной почте составителю перечня курсов, а копию отослать пользователю.

Система "осведомлена" о максимально допускаемом количестве студентов в каждом классе, поэтому в случае превышения лимита появляется пометка "Sold Out" (Продано).

Составитель перечня курсов может отсылать электронные сообщения всем студентам, зачисленным на тот или иной курс, путем заполнения специальной формы и выбора соответствующего курса. Благодаря этой форме составитель может ввести требуемое сообщение, а затем щелкнуть на кнопке, в результате чего оно будет отослано всем студентам, зачисленным на выбранный курс.

Составитель перечня курсов может также выводить на экран специальную форму, в которой отображается статус всех студентов, зачисленных на проводящиеся в настоящий момент курсы. Эта информация позволяет проверить посещаемость курсов студентами, а также факт оплаты за обучение.

## Идентификация действующих лиц и примеров использования

Одна из задач, выдвигаемых в процессе анализа требований, заключается в идентификации действующих лиц и примеров использования. Обратите внимание на тот факт, что в реальной системе не всегда четко выделены первоочередные задачи. Но в нашем случае подобное выделение будет целесообразным. Хотя на практике часто бывает так, что завершение всегда важнее начала.

## Исполнители

Зачастую роль исполнителей играют пользователи системы. Иногда в этом качестве выступают другие системы. В рассматриваемом примере в качестве исполнителей выступают пользователи-люди.

### Сотрудник курсов

Этот исполнитель зачисляет студента на курс. Он общается со студентами в процессе выбора подходящего для него курса, а также вносит сведения, относящиеся к студенту и к выбранному методу платежа.

### Составитель перечня курсов

Этот исполнитель получает уведомления по электронной почте о случае каждого зачисления на курсы. Он также отсылает электронные сообщения студентам и принимает отчеты о зачислении и платежах.

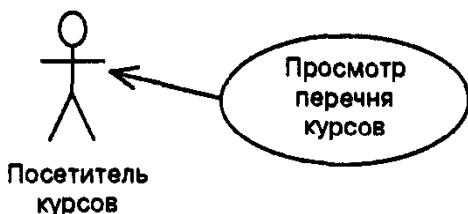
### Студент

Этот исполнитель получает подтверждение по электронной почте о зачислении, а также электронное уведомление от составителя перечня курсов. Затем студент посещает ранее выбранные им курсы.

## Варианты использования

После определения действующих лиц следует указать взаимодействия между ними в системе. Подобные спецификации называются “вариантами использования” (use case). Варианты использования описывают взаимодействия между исполнителем и системой с “точки зрения действующего лица”. При этом не описываются внутренние детали, связанные с функционированием системы, а также пользовательский интерфейс.

### Вариант использования #1: просмотр меню курсов

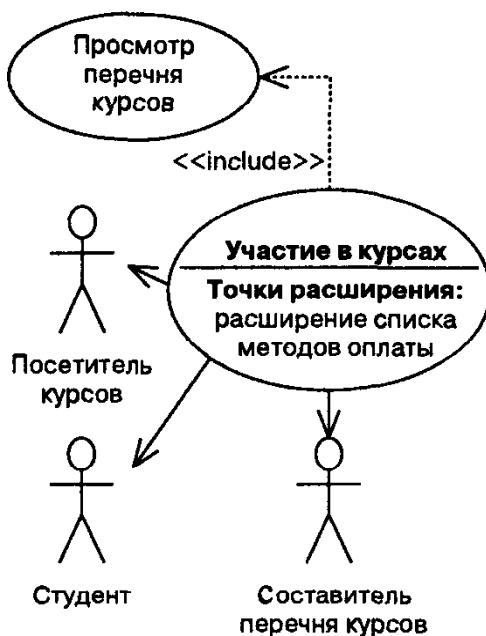


Составитель перечня курсов запрашивает список доступных курсов, воспользовавшись каталогом курсов. В результате система отображает полный перечень курсов. Сюда включены позиции, описывающие название, место и время проведения, а также стоимость курсов. Здесь же отображается предельно допустимое количество студентов для данного курса, а также статус курса (открыт или закрыт для записи).

### **Запись варианта использования**

На указанной выше диаграмме приведены исполнитель и вариант использования, включенные в диаграмму вариантов использования. Исполнитель представлен небольшой фигурой человечка, а вариант использования — эллипсом. Изображенная здесь пара символов иллюстрирует направление потока данных.

### **Вариант использования #2: запись на курсы**



Сначала составитель перечня курсов просматривает меню, содержащее список доступных курсов (вариант использования #1). Затем он выбирает требуемые курсы. Система отображает запрос на ввод имени, телефонного номера, номера факса и электронного адреса студента. Также предлагается указать предпочтительный способ платежа.

### **Точка расширения: указание способа платежа**

Сотрудник курсов заполняет форму абитуриента. Студент и составитель перечня курсов по электронной почте отсылают подтверждение зачисления на курсы. Сотрудник курсов просматривает подтверждение зачисления и просит его распечатать. Затем сотрудник подписывает документ и отсылает его по факсу, набрав заранее определенный номер.

## Расширение и применение вариантов использования

Вариант использования #2 представляет собой точку расширения. Это означает, что из этой точки могут исходить другие варианты использования. Ниже показаны соответствующие примеры: #2.1, #2.2 и #2.3. Соответствующие им описания включены в предыдущий вариант использования в точке расширения. Они указывают необязательные данные, которые требуется вводить в зависимости от выбранного способа платежа.

Вариант использования #2 также имеет взаимосвязь “включения” с вариантом использования #1. Это означает, что описание варианта использования #1 включено в соответствующем месте варианта использования #2.

Обратите внимание на разницу между расширением и включением. Если один вариант использования включает другой, включающий вариант использования ссылается на включенный вариант использования. Если же один вариант использования расширяет другой, ни один из них не ссылается на другой. Вместо этого расширяющий вариант использования выбирается на основе содержимого и включается в подходящей точке расширяемого варианта использования.

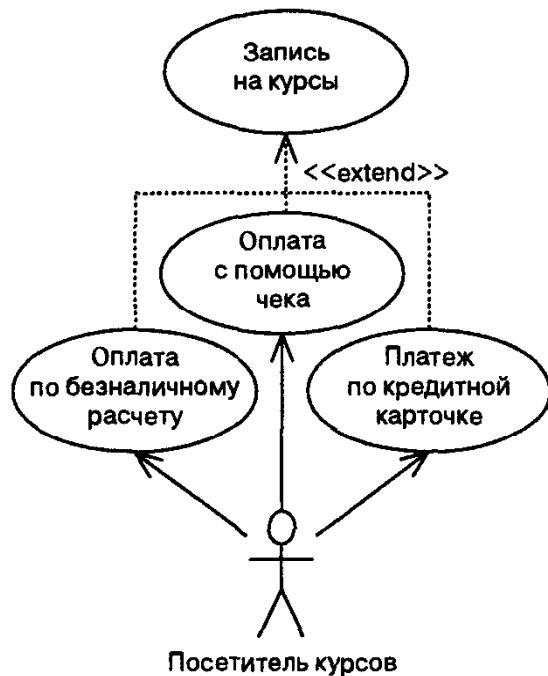
Взаимосвязь «включения» используется в том случае, если требуется гарантировать эффективность структуры вариантов использования путем свертывания повторяющихся операций в меньшие по размеру варианты использования, которые будут разделяться другими вариантами использования. Преследуемая в данном случае цель заключается в управлении изменениями и в устраниении избыточности. Путем перемещения общих частей вариантов использования в единичный включенный вариант использования достигается экономия за счет того, что в случае необходимости изменений будет модифицироваться только включенный вариант использования.

Взаимосвязь «расширения» используется в том случае, если внутри варианта использования имеется несколько альтернатив. В этом случае отделяется неизменная часть варианта использования от изменяемых. Неизменная часть становится расширяемым вариантом использования, а изменяемые части — расширяющими вариантами использования. И снова целью является эффективное управление изменениями. В рассматриваемом примере, в случае добавления новой возможности платежа, создаются новые расширяющие варианты использования, но существующие варианты использования при этом изменять не нужно.

### Запись взаимосвязи «включения»

Диаграмма, описывающая вариант использования #2, демонстрирует связь варианта использования “Запись на курсы” с вариантом использования “Просмотр меню курсов”. Сама взаимосвязь указана пунктирной линией, которая завершается стрелкой.

## Вариант использования #2.1: оплата по счету-фактуре



Сотруднику курсов предлагается в счете-фактуре с номером Р0# указать название компании, а также имя и телефонный номер ответственного лица.

## Вариант использования #2.2: оплата по чеку

Сотруднику курсов предлагается указать номер чека.

## Вариант использования #2.3: платеж с помощью кредитной карты

Сотруднику курсов предлагается указать номер кредитной карты, срок ее действия, а также имя, которое написано на кредитной карте.

## Вариант использования #3: отсылка электронных сообщений студентам

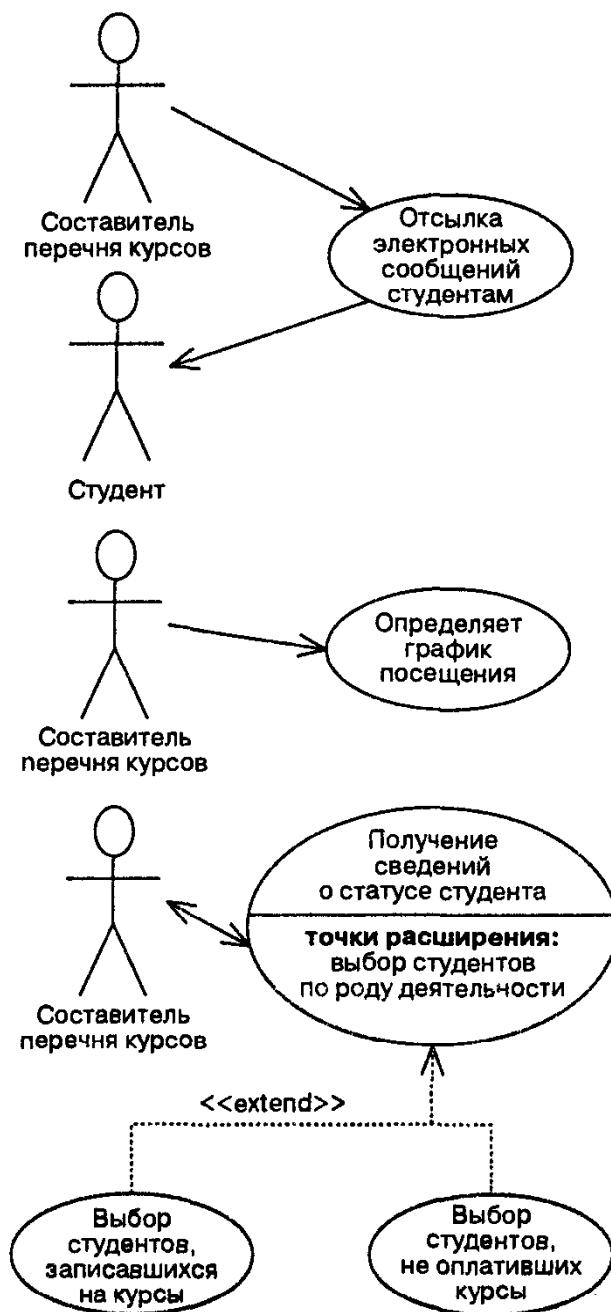
Составитель перечня курсов выбирает нужный курс, а затем вводит текст сообщения. Система отсылает сообщение по электронным адресам, соответствующим всем студентам данного курса.

## Вариант использования #4: проверка уровня посещаемости

Составитель перечня курсов выбирает курс, а также студента, который посещает этот курс. Система отображает сведения о студенте, о его посещаемости, а также о том, заплатил ли он за курсы. Составитель перечня курсов может изменять статус посещаемости или платежей.

## Вариант использования #5: получение сведений о статусе студента

Составитель перечня курсов выбирает студентов в соответствии с присвоенными им приоритетами.



### **Точка расширения: “выбор студентов по приоритетам”**

Система отображает статус, свидетельствующий о посещаемости и платежах, а затем генерирует единственный отчет.

### **Вариант использования #5.1: выбор студентов, посещающих курс**

Система отображает перечень всех курсов. Составитель перечня курсов выбирает один из них. Система выбирает всех студентов данного курса.

### **Вариант использования #5.2: отбор студентов, не оплативших обучение**

После указания составителя перечня курсов система перечисляет всех студентов, которые записаны на курсы, а статус платежа свидетельствует, кто из них не оплатил свое обучение.

## Варианты использования: повторение

Рассмотренные ранее варианты использования описывали поведение системы, соответствующее ожиданиям пользователей. Обратите внимание на то, что в данном случае мы не рассматривали детали реализации пользовательского интерфейса. Даже не упоминались пиктограммы, элементы меню, кнопки или прокручиваемые списки. На самом деле даже в оригинальной спецификации пользовательский интерфейс описан подробнее, чем в рассматриваемых примерах использования. Это было сделано преднамеренно. Ставилась цель до предела “облегчить” примеры использования, упростив тем самым их поддержку. После завершения разработки данных примеров использования возможны самые разнообразные реализации.

## Системные граничные диаграммы

Полный набор вариантов использования приводится в резюме, показанном на системной граничной диаграмме (рис. А.1). На этой диаграмме показаны все примеры использования системы, заключенные в прямоугольник, который представляет границы системы. Действующие лица находятся вне системы и связаны с примерами использования с помощью ассоциаций, которые отображают потоки данных.

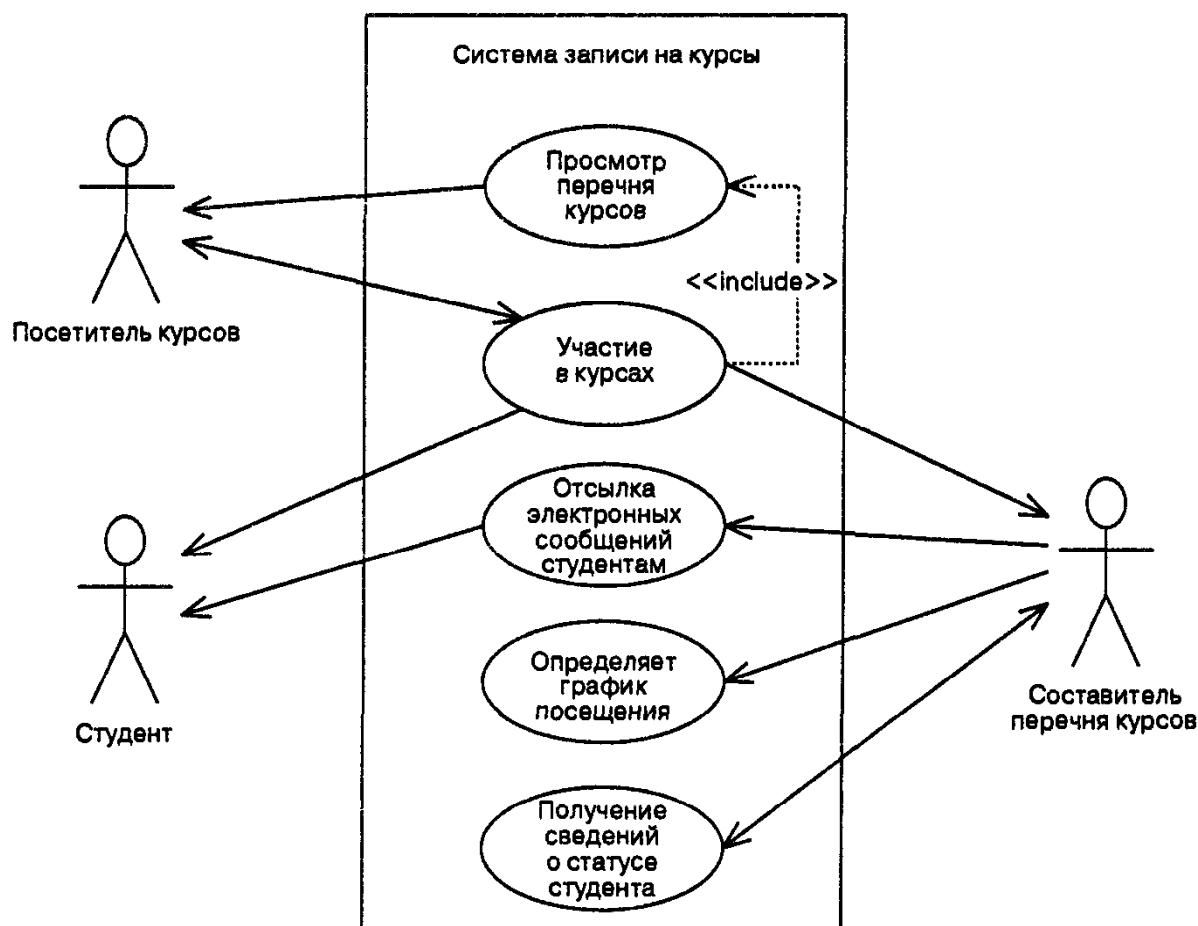


Рис. А.1. Системная граничная диаграмма

## Для чего необходимы диаграммы?

Диаграммы вариантов использования, включая системные граничные диаграммы, не относятся к категории диаграмм программных структур. С их помощью невозможно получить сведения о распределении в системе создаваемых программных элементов. Эти диаграммы используются для общения между людьми, особенно между аналитиками и другими участниками разработки проекта. С их помощью можно охарактеризовать функциональные свойства системы с применением терминов, относящихся к различным типам пользователей системы.

Более того, подобные диаграммы могут оказаться достаточно полезными в целях представления описания системы различным категориям пользователей. Пользователи из каждой отдельной категории преимущественно интересуются своими собственными вариантами использования. Связь между исполнителями и вариантами использования позволяет сфокусировать внимание каждого отдельного пользователя на применяемых им примерах использования. В больших системах могут оказаться полезными системные граничные диаграммы в соответствии с типами исполнителей.

И наконец, учитывайте первый закон Мартина, имеющий отношение к составлению документов.

*Не пишите документацию до тех пор, пока не возникнет крайняя потребность в этом.*

Описанные диаграммы могут быть полезными, но достаточно часто они не нужны. Не рассматривайте их в качестве абсолютно необходимых и важных документов. Рисуйте их в том случае, если это диктуется производственной необходимостью.

## Модель предметной области

Модель предметной области представляет собой набор диаграмм, которые оказывают помощь при определении терминов, применяемых в вариантах использования. Эти диаграммы отображают ключевые объекты, а также взаимосвязи между ними. Не следует полагать, что эта модель эквивалентна модели разрабатываемой программы. Важен тот факт, что аналитик и разработчик проектов могут использовать модель предметной области в качестве описательного инструмента, который призван облегчить специалистам фиксацию сведений о принимаемых решениях, а также взаимодействие с другими специалистами. Объекты в модели предметной области необязательно соответствуют процессу объектно-ориентиро-

ванной разработки ПО, поскольку не всегда подобное соответствие дает какие-либо преимущества<sup>5</sup>.

В записи Booch 94 и ОМТ диаграммы предметной области практически не отличаются от диаграмм, которые представляют структуру и проект ПО. В худшем случае диаграммы модели предметной области рассматриваются в высокоуровневом проектировании документов, поэтому они используются для определения высокоуровневой структуры самого ПО.

В целях избежания подобного вида ошибок можно воспользоваться преимуществами, обеспечиваемыми UML, а также применить специальный вид сущности, именуемой “типов” в моделях предметной области. “Тип” представляет роль, которую может играть объект. “Тип” может представлять операции и атрибуты, а также связи между другими “типов” сущностей. Однако “типов” не может представлять класс или объект в смысле проектирования. Он не представляет элемент ПО, поэтому не устанавливается непосредственное соответствие с кем-либо. Он представляет концептуальную сущность, используемую при описании той или иной проблемы.

## Каталог курсов

Пример первой абстракции предметной области может представлять каталог курсов. Эта абстракция представляет перечень всех предлагаемых курсов. Она включена в состав модели предметной области (рис. А.2). Здесь сущность Course Catalog указывает две других сущности, которые представляют абстракции предметной области: сущности CourseCatalog и Course. Сущность CourseCatalog предлагает многие сущности Course.

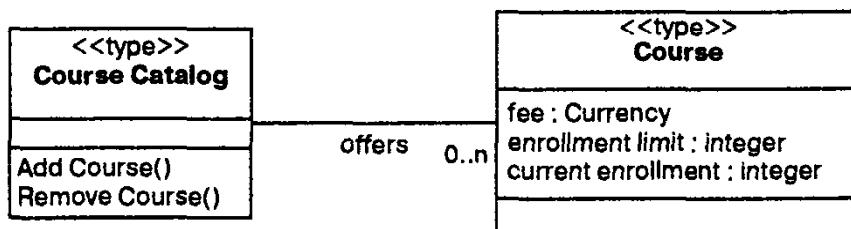
## Запись в модели предметной области

Запись, продемонстрированная на рис. А.2, указывает две абстракции предметной области в виде UML-классов, которым свойственен “типов”. (Дополнительные

<sup>5</sup>[JACOBSON], с. 133 — “Не следует полагать, что наилучшие (наиболее стабильные) системы создаются только на основе объектов, которые соответствуют сущностям из реального мира...”

Там же на с. 167 — “При использовании [других] методов, эта [доменная] модель также формирует базу для фактической реализации; таким образом устанавливается прямое соответствие между объектами и классами во время реализации. Однако этого нельзя сказать в случае с OOSE, [...]. Наш опыт разработки не может привести к однозначному решению. Поэтому лучше разработать модель анализа, которая является более устойчивой и лучше поддерживается в плане будущих изменений по сравнению с моделью предметной области, которая является базовой для проектирования и реализации”.

[BOOCHE96], с. 108 — “... в незрелых проектах наблюдается тенденция к использованию модели предметной области на основе результатов проведенного анализа, которая может применяться для кодирования [...]. Таким образом пропускается стадия дальнейшего проектирования. Здоровые проекты характеризуются тем, что выполняется некоторая дополнительная работа, включая такие аспекты, как конкуренция, массовое производство, безопасность, распределение, а также некоторые другие вопросы, в результате чего завершающая модель проекта может немного отличаться”.



**Рис. А.2.** Модель предметной области для сущности Course Catalog

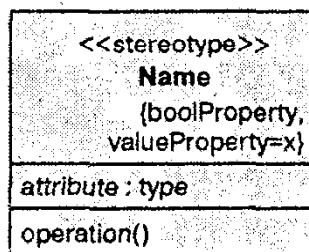
сведения можно найти в примечании “Обзор UML: семантика и запись классов”.) Отсюда можно сделать вывод, что классы представляют собой концептуальные элементы предметной области проблемы, которые непосредственно не связаны с программными классами. Обратите внимание, что сущность CourseCatalog включает две операции: AddCourse и RemoveCourse. В “типе” операции соответствуют *ответственостям*. Таким образом, CourseCatalog включает ответственность, которая позволяет добавлять и удалять курсы. И снова речь идет о концепциях, а не о спецификациях функций-членов внутри реальных классов. Эти объекты используются для взаимодействия с пользователями, а не в целях определения программной структуры.

Аналогичным образом, атрибуты, соответствующие сущности Course, представляют собой концепции. Они указывают на то, что сущность Course может быть ответственна за хранение сведений об оплате, о предельной заполняемости курсов, а также о текущем наборе курсов.

#### Обзор UML: семантика и запись классов

В UML-записи класс представляется в виде прямоугольника, разделенного на три меньших по размеру прямоугольника. В первом маленьком прямоугольнике указывается название класса, во втором определяются атрибуты класса, а в третьем — операции класса.

Информация, которая содержится в прямоугольнике, определяющем имя, может изменяться с помощью стереотипа и свойств. Стереотип отображается над именем и заключен в угловые кавычки («»). Свойства отображаются в нижней части, справа от имени, и заключены в фигурные скобки.



Стереотипы являются именами, которые ссылаются на “вид” представленного UML-класса. В UML под классом понимается именованная сущность, которая снабжена атрибутами и операциями. Стереотип, заданный по умолчанию, является “классом реализации”, причем в UML класс непосредственно связан с записью класса в программах, написанных на языках C++, Java, Smalltalk или Eifel. Атрибуты соответствуют переменным-членам, а операции соответствуют функциям-членам.

Однако при указании стереотипа «тип» класс UML не соответствует программной сущности во всех случаях. Скорее он соответствует концептуальной сущности, которая существует в предметной области проблемы. Атрибуты представляют информацию, которая имеет логическую связь с данной концептуальной сущностью, а операции представляют ответственности для данной концептуальной сущности.

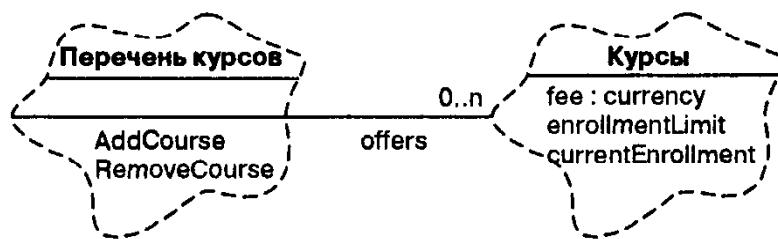
Существуют также и другие предопределенные стереотипы, которые будут рассмотрены в следующих разделах главы. Пользователь может создавать свои собственные стереотипы. Однако стереотип — это нечто большее, чем обычный комментарий. Он указывает способ, с помощью которого могут интерпретироваться все элементы в классе UML. Поэтому, если создается новый стереотип, его следует хорошо определить.

Свойства — это прежде всего структурированные комментарии. Свойства указываются в виде списка, разделенного запятыми и заключенного в фигурные скобки. Каждое свойство представляет собой пару имя = значение. Причем если знак равенства пропущен, предполагается, что свойство является булевым и получает значение “истина”. В противном случае типом значения будет строка.

Существует множество предварительно определенных свойств, которые будут рассмотрены далее в главе. Любой пользователь может добавлять собственные свойства. Например, можно создавать свойства, имеющие следующий вид: {author=Robert C. Martin, date='12/6/97, SPR=4033}.

## **Концепция, реализация, а также применение пиктограмм “облаков”**

Автор этой книги приложил много сил и затратил уйму времени, пока постиг различие между классом на концептуальном уровне (“тип”) и классом в процессе проектирования на уровне реализации. На самом деле понять это было важно, поскольку концептуальные диаграммы могут ошибочно восприниматься в качестве спецификаций для структуры и архитектуры ПО. Помните о том, что концептуальные диаграммы предназначены облегчать общение между участниками общего дела, вследствие чего они не содержат каких-либо технических аспектов, связанных с описанием структуры программы.



**Рис. А.3.** Модель предметной области для каталога курсов, включающая пиктограммы “облака”

В результате использования стереотипов описанные типы диаграмм могут быть пересмотрены. Классы UML для моделей предметной области очень похожи на классы UML, применяемые в процессе проектирования и реализации. К тому же UML разрешает подставлять различные пиктограммы для разных стереотипов. Для увеличения степени различий между описанными типами диаграмм следует воспользоваться пиктограммой “облака”, обеспечивающей представление “типа” классов, которые относятся к этой пиктограмме. В результате изменяется модель предметной области для сущности *Course Catalog*, показанной на рис. А.2 таким образом, что она приобретает вид, указанный на рис. А.3.

### Завершение модели предметной области

До настоящего времени модель предметной области предназначалась для представления каталога курсов, включающего все предлагаемые курсы. Но именно здесь коренятся некоторые проблемы. Что следует понимать под курсом? Один и тот же курс может читаться много раз, а также в различных местах, причем его могут вести различные инструкторы. Очевидно, что в этом случае требуются две различных сущности. Сначала обратимся к сущности *Course*. Эта сущность представляет сам курс, но не содержит сведений о датах, местоположениях или инструкторах. Поэтому следует воспользоваться второй сущностью, *Session*. Она представляет дату, место проведения, а также инструктора того или иного курса (рис. А.4).

### Запись

Линии, соединяющие сущности, называются ассоциациями. Все ассоциации, изображенные на рис. А.4, являются именованными, хотя это и не обязательно. Обратите внимание, что имена являются глаголами. Маленький черный прямоугольник, указанный возле имени, указывает на сказуемое в предложении, образованное двумя сущностями и ассоциацией. В качестве подобных имен могут использоваться фразы “*Course Catalog offers many Courses*” (Каталог курсов предлагает много курсов), “*Session Schedule schedules many Sessions*” (График сеансов планирует много сеансов) и “*Many students are enrolled in a Session*” (Многие студенты придут на сеанс).

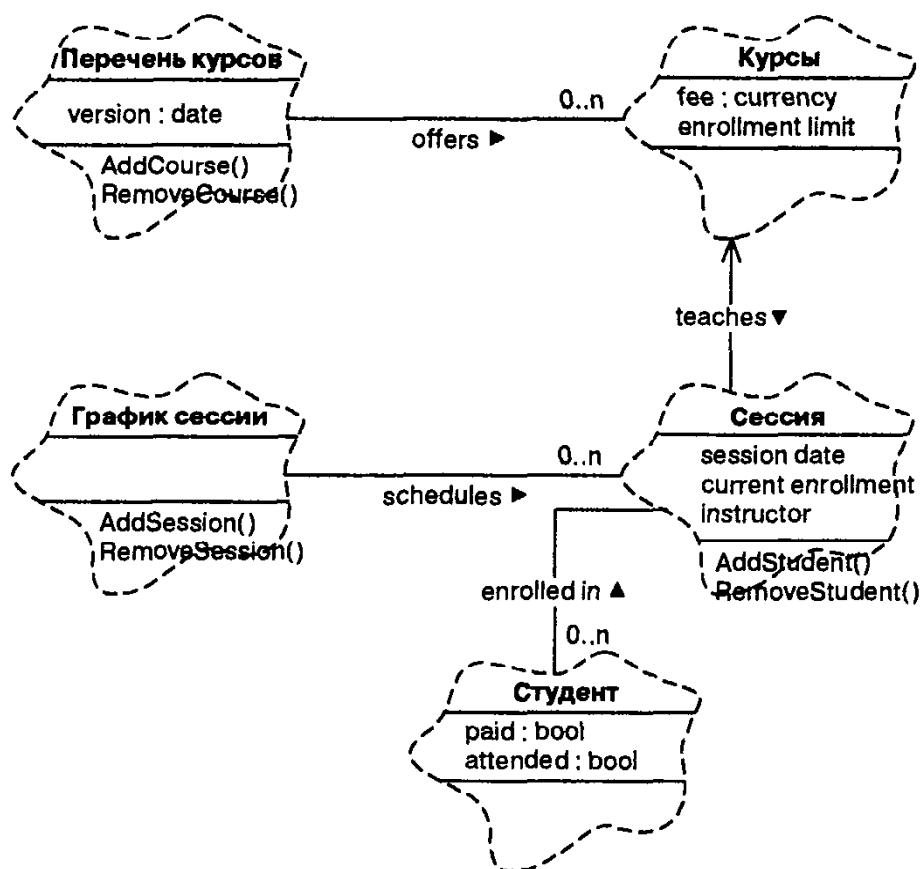


Рис. А.4. Курсы и сеансы

В данном случае использовалось слово “many” из указанных выше предложений, где пиктограмма “0..\*” представляет соответствующую взаимосвязь. Эта пиктограмма представляет категорию различных пиктограмм **многообразия**, которые могут помещаться на конце ассоциации. Они указывают количество существ, которые формируют ассоциацию. Заданное по умолчанию многообразие – “1”. (Обратите внимание на примечание “Многообразия”.)

Предполагается, что ассоциации являются двунаправленными до тех пор, пока не указана стрелка. При установке двунаправленной ассоциации две сущности “знают” друг о друге. Например, понятно, что сущность CourseCatalog должна “знать” о соответствующих ей Courses, и вполне целесообразно, чтобы каждая сущность Course могла “знать” о сущности CourseCatalog. То же самое можно сказать о сущностях SessionSchedule и Session.

Если указана стрелка, “знания” ограничиваются указанным направлением. В результате Sessions “знает” о Courses, но Courses ничего не “знает” о Sessions.

### **Многообразия**

Ассоциации могут “украшаться” пиктограммами многообразий. Ниже приведен соответствующий перечень:

- 0..\* нуль ко многим;
- \* нуль ко многим;
- 1..\* один ко многим;
- 0..1 нуль или один;
- 6 точно шесть;
- 3..6 между тремя и шестью;
- 3..5 три или пять.

Между точками могут указываться любые положительные числа, которые также могут разделяться запятыми.

## Итерации вариантов использования

Из указанной на рисунке диаграммы следуют два вывода. Во-первых, варианты использования используют некорректный язык во многих ситуациях. Там, где говорится о каталогах курсов и самих курсов, на самом деле подразумеваются графики сеансов и сами сеансы. Во-вторых, достаточно много вариантов использования остались “за бортом”. Требуется поддержка сущностей CourseCatalog и SessionSchedule. Сущность Courses нуждается в добавлении и удалении из CourseCatalog, а Sessions должна добавляться и удаляться из SessionSchedule.

Таким образом, путем создания модели предметной области достигается лучшее понимание сути проблемы. Подобное лучшее понимание позволяет улучшать и совершенствовать примеры использования. Итерация между двумя перечисленными сущностями является естественной и необходимой.

Если вы прослеживаете практическое занятие вплоть до его завершения, можно продемонстрировать указанные выше изменения. Но в интересах эффективного представления записи уместно пропустить итерацию, связанную с вариантами использования.

## Архитектура

Теперь пришло время заняться проектированием программ. Архитектура представляет программные структуры, формирующие каркас приложения. Классы и взаимосвязи, относящиеся к архитектурной карте, очень тесно связаны с кодом.

## Выбор программной платформы

Прежде всего необходимо изучить программную платформу, на которой будет выполняться разработанное приложение. В распоряжении пользователя имеются следующие варианты для выбора.

1. CGI-приложение, предназначенное для выполнения в Web. Доступ к формам записи на курсы, а также к некоторым другим формам, осуществляется с помощью Web-браузера. Данные могут находиться на Web-сервере, а CGI-сценарии должны вызываться Web-браузером, в результате чего обеспечивается доступ к данным, а также их обработка.
2. Приложение базы данных. Можно приобрести реляционную базу данных, а затем использовать пакет форм и 4GL для написания приложения.
3. Визуальные компоненты. Можно приобрести пакет визуального программирования. Интерфейс, предназначенный для взаимодействия компьютера с пользователем, может быть создан с помощью инструментов визуального конструирования. Эти инструменты могут вызывать программные функции, которые требуются для организации хранения, выборки, а также обработки данных.

Также существуют и другие возможности. Можно писать полнофункциональные программы на языке С, не используя библиотеку или поддержку со стороны инструментальных средств, отличных от компилятора. Но подобную практику трудно назвать рациональной. Ведь есть соответствующие вполне функциональные инструменты, которые следует использовать при разработке программ.

В рамках рассматриваемого примера предположим, что разрабатывается Web-приложение. Это имеет смысл, если сотрудники курсов постоянно перемещаются по разным странам, а услуги по набору на курсы предоставляются через Internet.

## Web-архитектура

На этом этапе следует определиться с общей архитектурой Web-приложений. Сколько нужно разработать Web-страниц и какие CGI-программы они будут вызывать? На рис. А.5 показано, каким образом можно начинать описывать эти спецификации.

## Запись

На рис. А.5 представлена компонентная диаграмма. Пиктограммы указывают физические программные компоненты. Можно воспользоваться стереотипом в целях указания типа компонентов. На диаграмме показано, что меню Session Menu (Меню сеансов) отображается в виде Web-страницы (код HTML), которая гене-

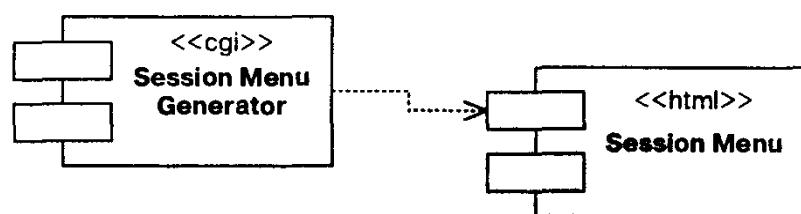


Рис. А.5. Архитектура меню сеансов

рирована CGI-программой, именуемой Session Menu Generator (Генератор меню сеансов). Пунктирная стрелка между двумя компонентами представляет собой взаимосвязь между зависимостями. Эти взаимосвязи определяют, что компоненты “знают” о существовании других компонентов. В рассматриваемом случае программа Session Menu Generator создает Web-страницу Session Menu, в результате чего она “знает” об этой странице. С другой стороны, Web-страница Session Menu ничего не “знает” о своем генераторе.

### Заказные пиктограммы

На рис. А.5 изображены два различных типа компонентов. Для того чтобы их внешний вид отличался, расширим набор UML путем включения двух новых пиктограмм — для CGI-программ и HTML-страниц, соответственно. На рис. А.6 показаны компоненты, вызываемые в варианте использования #2, “Запись на курсы”. Web-страницы указаны в виде страниц, обозначенных буквой ‘W’. CGI-программы указаны в кружочках и помечены буквами “CGI”.

### Поток компонентов

На рис. А.6 представлены две новые Web-страницы, а также новая CGI-программа. Было принято решение, что запуск приложения начинается с демонстра-

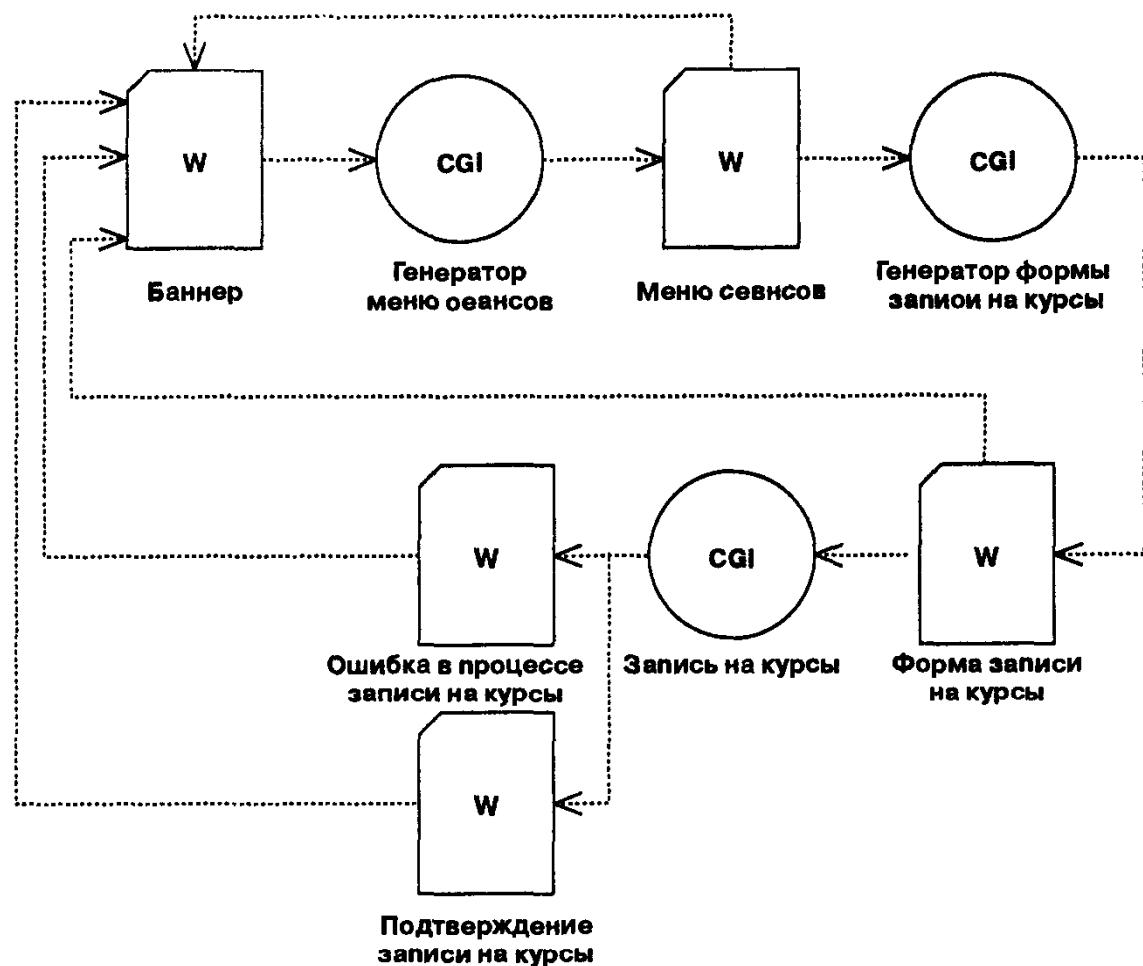


Рис. А.6. Компоненты приложения, реализующего запись на курсы

ции баннерной страницы. Вероятно, эта страница будет включать ссылки на различные типы операций, выполняемых пользователями. Баннерная страница вызывает Session Menu Generator, в результате чего генерируется страница Session Menu. Страница Session Menu может содержать ссылки или кнопки, которые позволяют пользователю записываться на курсы. Страница Session Menu вызывает CGI-программу Enrollment Form Generator, в результате чего создается форма, используемая в процессе записи на курсы. Как только пользователь заполняет форму, вызывается CGI-программа Enroll. Эта программа проверяет правильность, а также заполняет форму информацией. Если указанные в форме данные некорректны, генерируется страница Enrollment Error (Ошибка при записи на курсы); в противном случае генерируется страница подтверждения, а также отсылаются необходимые сообщения по электронной почте. (И снова обратитесь к варианту использования #2.)

## Увеличение степени гибкости

Внимательный читатель, наверное, уже заметил, что рассматриваемой компонентной модели не хватает гибкости. Большинство Web-страниц генерируется CGI-программами. Это означает, что HTML-текст, входящий в состав Web-страниц, должен входить в CGI-программы. В результате изменение Web-страниц влечет за собой модификацию и перестройку CGI-программ. Может случиться так, что проще будет создать множество сгенерированных Web-страниц, воспользовавшись “услугами” хорошего HTML-редактора.

Исходя из этих соображений, CGI-программы должны быть “осведомлены” о шаблоне генерируемых Web-страниц. Шаблон помечается с помощью специальных флагов, которые будут заменены HTML-кодом, генерируемым CGI-программами. Это означает, что CGI-программы совместно используют некоторые данные. Они считают файлы HTML-шаблона и добавляют к нему свои биты HTML-кода<sup>6</sup>.

На рис. A.7 показана результирующая компонентная диаграмма. Обратите внимание на добавленную пиктограмму WT. Она представляет HTML-шаблон, который является текстовым файлом в формате HTML. А последний содержит специальные пометки, используемые CGI-программами в качестве точек включения генерируемого HTML-кода. Обратите внимание на направление взаимосвязи зависимостей между CGI-программами и HTML-шаблонами. Помните о взаимо-

<sup>6</sup>Эта глава была написана задолго до появления языка XSLT. В настоящее время описанная проблема решается путем написания CGI-сценария (или сервлета), который генерирует XML-код, а затем вызывает сценарий XSLT в целях трансляции XML в HTML. С другой стороны, генерирование HTML-кода с помощью XSLT не позволяет разрабатывать Web-страницы с помощью WYSIWYG-редактора. Иногда мне кажется, что схема с применением шаблона, кратко описанная в этой главе, является наилучшей во многих отношениях.

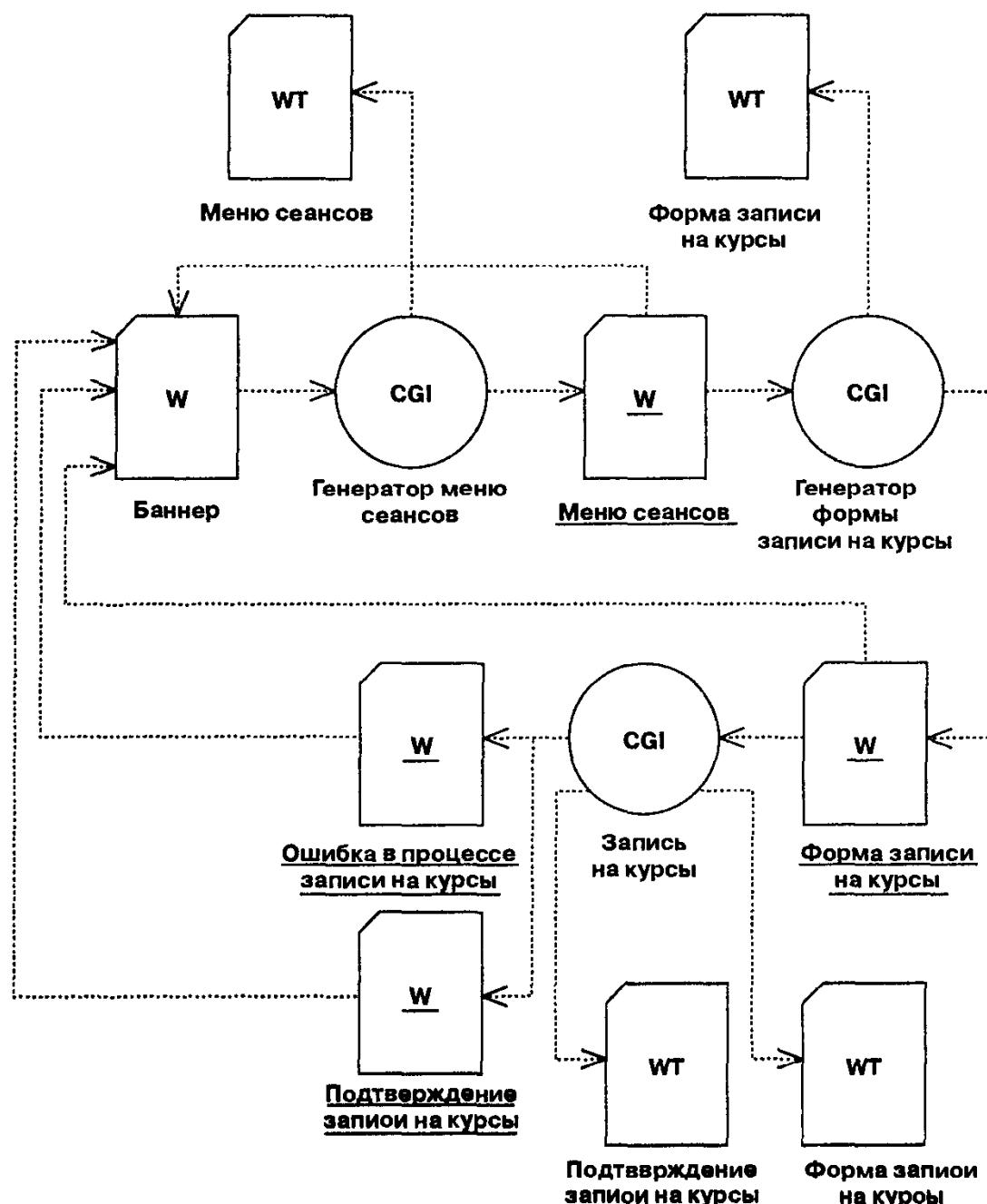


Рис. А.7. Добавление HTML-шаблонов на компонентную диаграмму приложения, касающегося набора на курсы

связи зависимостей вместо потоков данных. CGI-программы “знают” о HTML-шаблонах.

### Дихотомия “спецификация/экземпляр”

На рис. А.7 подчеркнуты названия сгенерированных Web-страниц. Это сделано только потому, что эти страницы существуют только во время выполнения. Они представляют собой экземпляры HTML-шаблонов. В UML действует соглашение, в соответствии с которым подчеркиваются экземпляры. Экземпляр представляет собой программный элемент, который был сгенерирован на основе некоторого рода спецификации (исходный документ). Представьте себе, что элементы, чьи

имена не являются подчеркнутыми, представляют элементы, которые должны создаваться вручную и выступать в качестве спецификаций. Элементы с подчеркнутыми названиями — следствие некоторого процесса, генерирующего их на основе указанных спецификаций.

## Использование HTML-шаблонов

Благодаря HTML-шаблонам архитектура приложения приобретает большую степень гибкости. Каким образом работает эта методика? Как CGI-программы помогают сгенерированный ими вывод в подходящих местах внутри HTML-шаблонов?

Рассмотрим размещение специального HTML-дескриптора в файлах HTML-шаблона. Подобный дескриптор отмечает позицию в сгенерированном HTML-файле, в которой CGI-программа будет включать свой вывод. Однако сгенерированные Web-страницы могут включать несколько секций, каждая из которых имеет свою точку вставки, где CGI-программа может включать HTML-код. Поэтому каждый HTML-шаблон может иметь более одной точки вставки, а CGI-программа должна быть способна указать, каким образом ее вывод направляется данным дескриптором.

Дескриптор может выглядеть так: <имя вставки>, где “имя” — произвольная строка, которая идентифицирует точку вставки для CGI-программы. Такой дескриптор, как <заголовок вставки>, позволяет CGI-программе указывать ‘заголовок’ имени, а затем полностью заменяться сгенерированным выводом.

Ясно, что каждый дескриптор заменяется строкой символов, поэтому каждый из них представляет некий вид именованного потока символов. Если бы в CGI-программу был включен некий фрагмент кода на языке C++, он бы имел следующий вид:

```
HTMLTemplate myPage("myPage.html");
myPage.insert("header",
 "<h1> this is a header </h1>\n");
cout << myPage.Generate();
```

Этот код отсылает HTML-инструкции оператору cout, который был сгенерирован на основе шаблона myPage.html, в котором дескриптор <заголовок вставки> был заменен строкой "<h1> this is a header </h1>\n".

На рис. A.8 показан возможный проект класса HTMLTemplate. Этот класс хранит имя файла шаблона в качестве атрибута. Сюда также включены методы, которые разрешают включать строки замены в специальных именованных точках вставки. Экземпляры HTMLTemplate могут включать отображение, связывающее имя точки вставки со строкой замены.

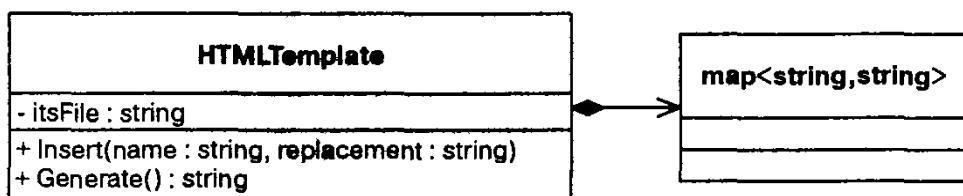


Рис. А.8. Проект HTMLTemplate

## Запись

Итак, получилась первая настоящая диаграмма классов. На ней изображены два класса, связанные с помощью *композиционной* взаимосвязи. Пиктограмма, используемая для записи классов HTMLTemplate и map<строка, строка>, не является для нас новой. Она уже рассматривалась в примечании “Обзор UML: семантика и запись классов”. Синтаксис, используемый для записи атрибутов и операций, будет описан в примечании “Атрибуты и операции”.

### Атрибуты и операции

Атрибуты и операции могут сопровождаться следующими инкапсулирующими спецификаторами:

- + общедоступный;
- частный;
- # защищенный.

Тип атрибута может указываться в виде идентификатора, который следует за атрибутом и отделен от него двоеточием (например, count : int).

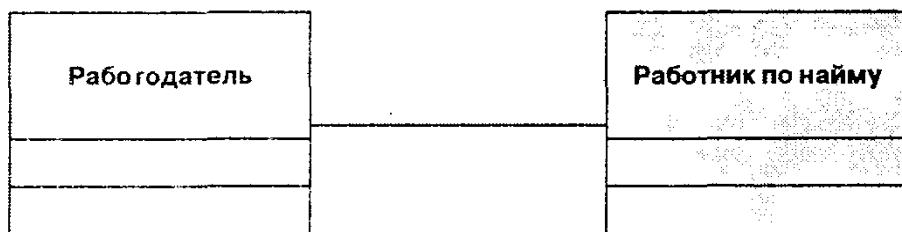
Аналогично, типы аргументов функции указываются с помощью формата записи, предусматривающего использование двоеточия (например, SetName (name : string)).

Возвращаемый тип операции может указываться с помощью идентификатора, который следует за именем и списком аргументов операции, причем тоже отделен от них двоеточием (например, Distance (from : Point): float).

Стрелка, указанная в ассоциации, объединяющей два класса на рис. А.8, указывает на то, что класс HTMLTemplate “знает” о классе map<строка, строка>, но класс map ничего не “знает” о классе HTMLTemplate. Черный ромб, находящийся возле ассоциации класса HTMLTemplate, идентифицирует ее как специальный случай ассоциации, именуемой композицией. (Обратите внимание на примечание “Ассоциация, агрегирование и композиция”.) Он указывает на то, что от класса HTMLTemplate зависит продолжительность существования класса map.

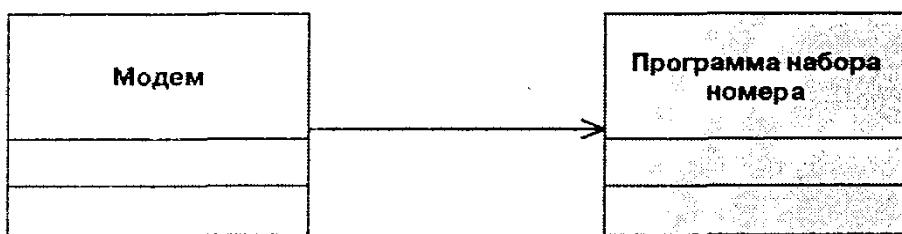
## Ассоциация, агрегирование и композиция

### Ассоциация



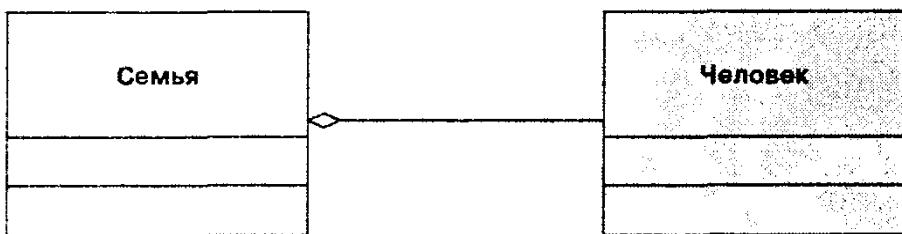
Ассоциация — это взаимосвязь между двумя классами, которая позволяет экземплярам классов, созданных на их базе, отсылать сообщения другим экземплярам классов (т.е. должны существовать ссылки между объектами, с которыми связаны классы). Это моделируется линией, которая соединяет два класса. Ассоциации часто реализуются в виде экземпляров переменных в одном классе, который указывает другой класс или ссылается на него.

### Направленная ассоциация



Перемещение по ассоциации ограничено стрелками. При наличии подобной стрелки ассоциация может быть “пройдена” только в том направлении, которое ей указано. Это означает, что класс, на который указывает стрелка, ничего не “знает” о связанном с ним классом.

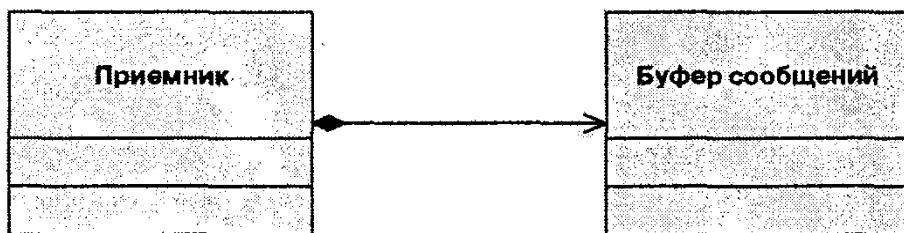
### Агрегирование



Агрегирование является специальной формой ассоциации. Это взаимоотношение обозначается с помощью белого ромба, указанного возле класса-агрегата. Агрегирование влечет взаимосвязь “целое/часть”. Класс, находящийся рядом с белым ромбом, является “целым”, а другой класс — “частью”. Взаимосвязь “целое/часть” является чисто коннотативной; отсутствует семантическое отличие от ассоциации<sup>7</sup>.

<sup>7</sup>Обратите внимание на следующее исключение. Не разрешаются рефлексивные (циклические) агрегирующие взаимосвязи. Поэтому экземпляры не могут принимать участие в циклах агрегирования. Если это правило не выполняется, все экземпляры в цикле могут быть частью самих себя.

## Композиция



Композиция является специальной формой агрегирования. Этот вид взаимосвязи обозначается черным ромбом. Она приводит к тому, что категория “целое” становится ответственной за продолжительность существования “части”. Эта ответственность не предполагает операций по созданию или удалению. Скорее, в результате ее применения “целое” должно рассматриваться таким образом, будто из него удаляется “часть”. Этого можно добиться путем непосредственного удаления “части” или передачи “части” другой сущности, которая определяет для нее ответственность.

## Слой интерфейса базы данных

Любая CGI-программа должна получать доступ к данным, которые представляют описания курсов, классов, студентов и т.д. Назовем все эти объекты учебной базой данных. До сих пор не определена форма этой базы данных. Она может выступать в виде реляционной базы данных или набора табличных файлов. Нельзя, чтобы архитектура приложения зависела от формы хранения данных. Весьма желательно, чтобы само приложение не изменялось в случае модификации базы данных. Поэтому приложение “отгораживается” от базы данных с помощью уровня интерфейса базы данных (DIL – Database Interface Layer).

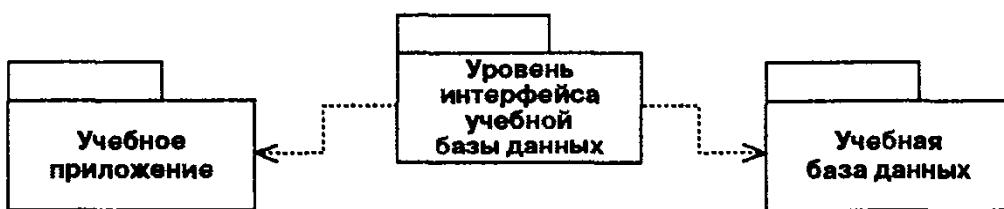


Рис. А.9. Характеристики зависимости уровня интерфейса базы данных

В целях достижения максимальной эффективности уровень DIL должен снабжаться специальными характеристиками зависимости, показанными на рис. А.9. Уровень DIL зависит от приложения и от базы данных. Ни приложение, ни база данных ничего не “знают” друг о друге. В результате можно изменять базу данных без обязательной модификации самого приложения. Можно также изме-

Поэтому часть может содержать целое. Учтите, что это правило не исключает возможность участия классов в циклах агрегирования, а лишь ограничивает экземпляры классов.

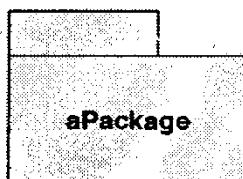
нять приложение, не затрагивая саму базу данных. Кроме того, можно полностью заменить формат базы данных или ее механизма, не затрагивая само приложение.

## Запись

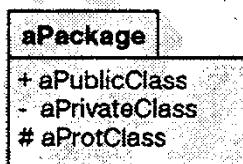
На рис. А.9 показан специальный вид диаграммы классов, называемой “диаграммой пакетов”. Пиктограммы обозначают пакеты. Их внешний вид напоминает папки файлов. Подобно файловой папке, пакет является контейнером. (Обратите внимание на примечание “Пакеты и подсистемы”.) Пакеты, показанные на рис. А.9, включают такие программные компоненты, как классы, HTML-файлы, основные файлы CGI-программ и т.д. Пунктирная стрелка, которая соединяет пакеты, представляет взаимосвязь *зависимостей*. Стрелка указывает на цель зависимости. Зависимость между пакетами приводит к тому, что зависимый пакет не может использоваться без пакета, от которого он зависит.

### Пакеты и подсистемы

Пакеты изображаются в виде больших прямоугольников с маленькими прямоугольными “вкладками” в верхнем левом углу большого прямоугольника. Обычно имя пакета отображается в большом прямоугольнике.

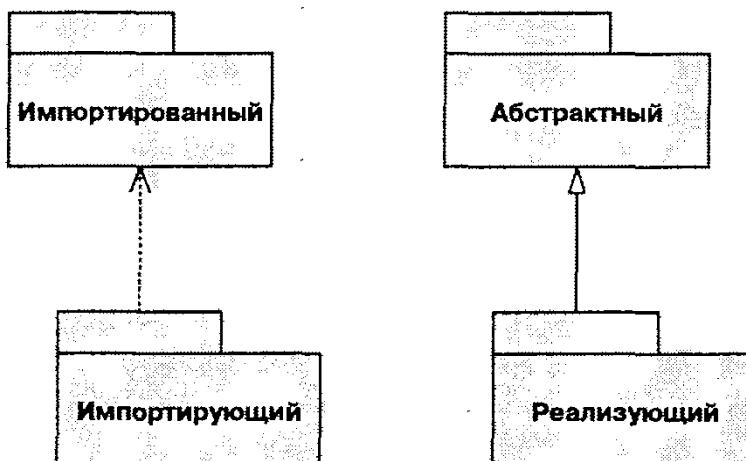


Названия пакетов могут указываться на “вкладках”, а само содержимое — помещаться в большом прямоугольнике. В качестве содержимого могут указываться классы, файлы или другие пакеты. Каждый из этих объектов может предваряться одной из пиктограмм инкапсуляции (-, +, #), которые обозначают, что они являются частными, общедоступными или защищенными внутри пакета.



Пакеты могут подключаться с помощью одной из двух различных взаимосвязей. Пунктирная стрелка зависимости называется *зависимостью импорта*. Она представляет взаимосвязь зависимостей со стереотипом «импорт». Этот стереотип применяется по умолчанию в том случае, если зависимость используется вместе с пакетом. Начало стрелки прикреплено к импортирующему пакету, а конец указывает на импортированный пакет. Зависимость импорта приводит к тому, что импортирующий пакет может “просматривать” любой из общедоступных элементов импортированного пакета. Это означает, что элементы импортирующего пакета могут использовать любой общедоступный элемент импортированного пакета.

Пакеты также могут подключаться с помощью взаимосвязей генерализации. В этом случае основание треугольной стрелки связано с общим или абстрактным пакетом, а другой конец взаимосвязи относится к реализующему пакету. Реализующий пакет отображается любым общедоступным или защищенным элементом абстрактного пакета.



Определено несколько стереотипов пакета. По умолчанию задается стереотип «пакет», который обозначает контейнер, не включающий какие-либо специальные ограничения. Этот контейнер может включать любые данные, которые могут моделироваться средствами UML. Как правило, он используется в целях обозначения физического *повторно выпускаемого* модуля. Подобные пакеты могут отслеживаться системами управления конфигурацией и контроля версий. Они могут представляться подкаталогами файловой системы или системой модулей языка (например, пакетами Java или JAR-файлами). Пакеты обеспечивают разбиение системы, в результате чего улучшаются возможности по разработке и повышается степень повторного использования.

Стереотип «подсистема» в случае наличия пакетов намечает логический элемент, который в дополнение к описанию содержащихся в модели элементов также указывает их поведение. Подсистема может снабжаться набором операций. Эти операции должны поддерживаться примерами использования или взаимодействием внутри пакетов. Подсистемы представляют *поведенческое* разбиение системы или приложения.



Существует два вида пакетов. Разбиение, которое способствует улучшению способности к разработке и повторному использованию, прямо противоположно разбиению, которое основано на поведении системы. Первое разбиение часто используется разработчиками программ в форме модулей управления конфигурацией и контроля версий. Второе разбиение часто применяется аналитиками

в целях описания системы на интуитивном уровне, а также для выполнения анализа воздействий в случае изменения или добавления свойств.

## Интерфейс базы данных

Классы, образующие пакет *Training Application*, должны получать доступ к базе данных. На практике подобный доступ реализуется с помощью набора интерфейсов, которые включены в состав пакета *Training Application* (рис. А.10). Эти интерфейсы представляют типы в модели предметной области (см. рис. А.4). Интерфейсы реализуются с помощью классов, образующих DIL-пакет. Эти классы могут применяться другими классами из пакета *Training Application* в целях получения доступа к данным, хранящимся в базе данных. Обратите внимание, что направление зависимостей, показанных на рис. А.10, соответствует направлению взаимосвязей импорта между пакетами, представленными на рис. А.9.

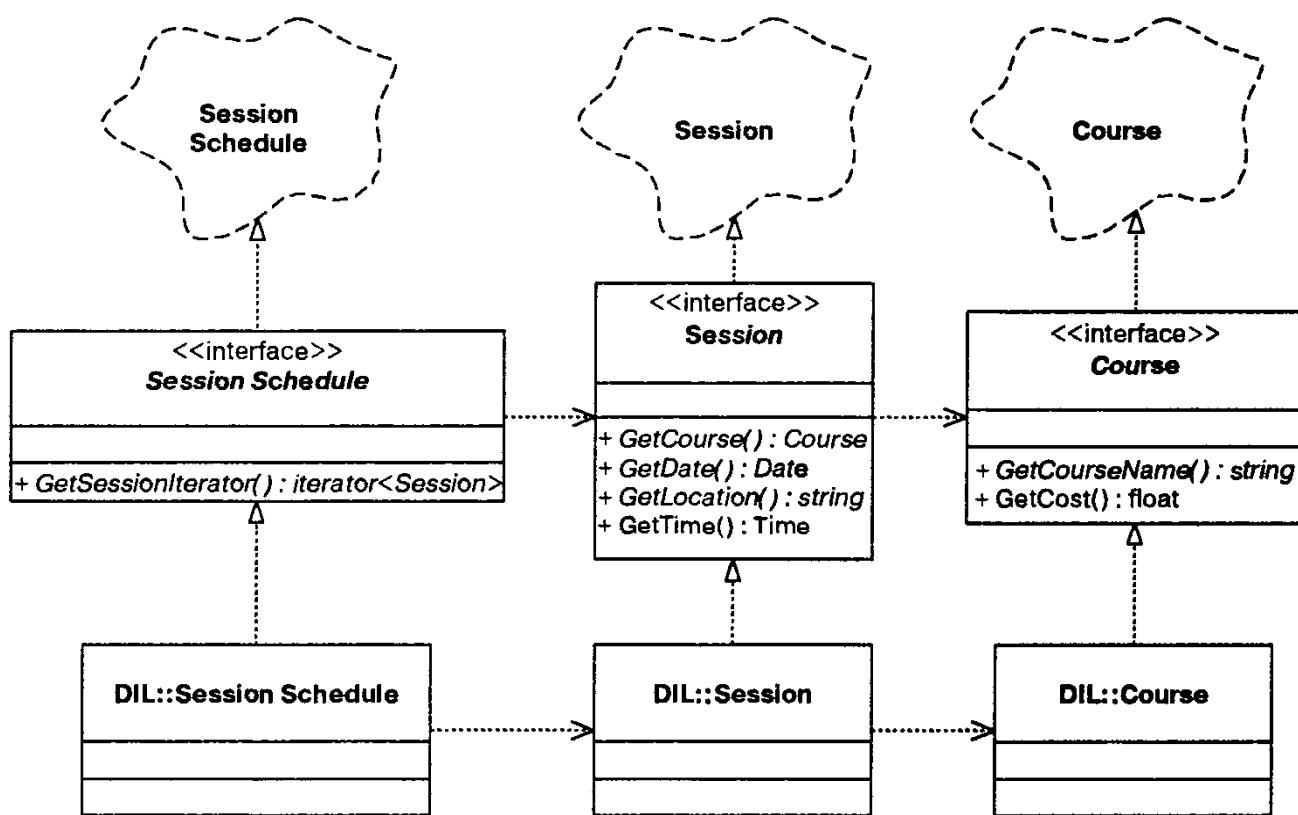


Рис. А.10. Классы интерфейса базы данных для пакета *Training Application*

## Запись

Если при записи названия класса используется курсивный стиль, следовательно, класс является абстрактным<sup>8</sup>. Поскольку интерфейсы являются целиком абстрактными классами, в их названиях рекомендуется использовать курсивный

<sup>8</sup>Абстрактный класс включает, как минимум, один чистый (или абстрактный) метод.

стиль. При записи операций также применяется курсивный стиль, что означает их абстрактный характер. Классы, имеющие отношение к DIL, также связываются в интерфейсах с помощью взаимосвязей *реализации*, которые рисуются в виде пунктирных линий, причем обратная сторона треугольных стрелок указывает на интерфейсы. В Java в этом случае используются взаимосвязи “реализации”. В C++ применяется наследование.

Интерфейсы являются физическими структурами. В таких языках программирования, как C++ и Java, им соответствуют файлы исходного кода. С другой стороны, типы не являются физическими структурами, поэтому им невозможно сопоставить эквивалент в виде файла исходного кода. На рис. А.10 иллюстрируется взаимосвязь между интерфейсами и представляемыми ими типами. В данном случае не идет речь о физической взаимосвязи или о каком-либо соответствующем исходном коде. Здесь подразумевается взаимосвязь, которая отображает соответствие между физическими сущностями проекта и моделью предметной области. Это соответствие нечасто является столь “прозрачным”, как изображено на этом рисунке.

Взаимосвязи *импорта* между пакетами (рис. А.9) показаны на рис. А.10 с применением двойного двоеточия. Класс `DIL::Session`, называемый `Session`, существует внутри пакета `DIL` и отображается (может быть импортирован) в пакете `TrainingApplication`. Получается, что два класса с именем `Session` доступны, если они находятся в различных пакетах.

## Генератор меню сеансов

Вернемся к рис. А.7, где показана первая CGI-программа, именуемая `SessionMenuGenerator`. Этой программе соответствует первый вариант использования, описанный в начале главы. Каким же образом будет выглядеть проект этой CGI-программы?

Понятно, что он должен быть HTML-представлением графика сеансов. Поэтому необходим шаблон `HTMLTemplate`, который позволяет передавать фактические данные из графика сеансов в меню сеансов. Эта программа также использует интерфейс `SessionSchedule` для получения доступа к экземплярам `Session` и `Course` базы данных в целях выборки названий курсов, времени и места проведения, а также стоимости. На рис. А.11 показана последовательная диаграмма, которая описывает данный процесс.

Объект `SessionMenuGenerator` создается модулем `main` и контролирует все приложение. Он создает объект `HTMLTemplate`, передавая ему имя файла шаблона. Он также выбирает объект `iterator<Session>`<sup>9</sup> из интерфейса `SessionSchedule`. При этом осуществляются итерации для каждого `Session`

<sup>9</sup>Эта глава была написана до того, как библиотека STL получила широкое распространение. Поэтому автор использовал собственную контейнерную библиотеку, для которой применяются шаблоны, образованные итераторами.

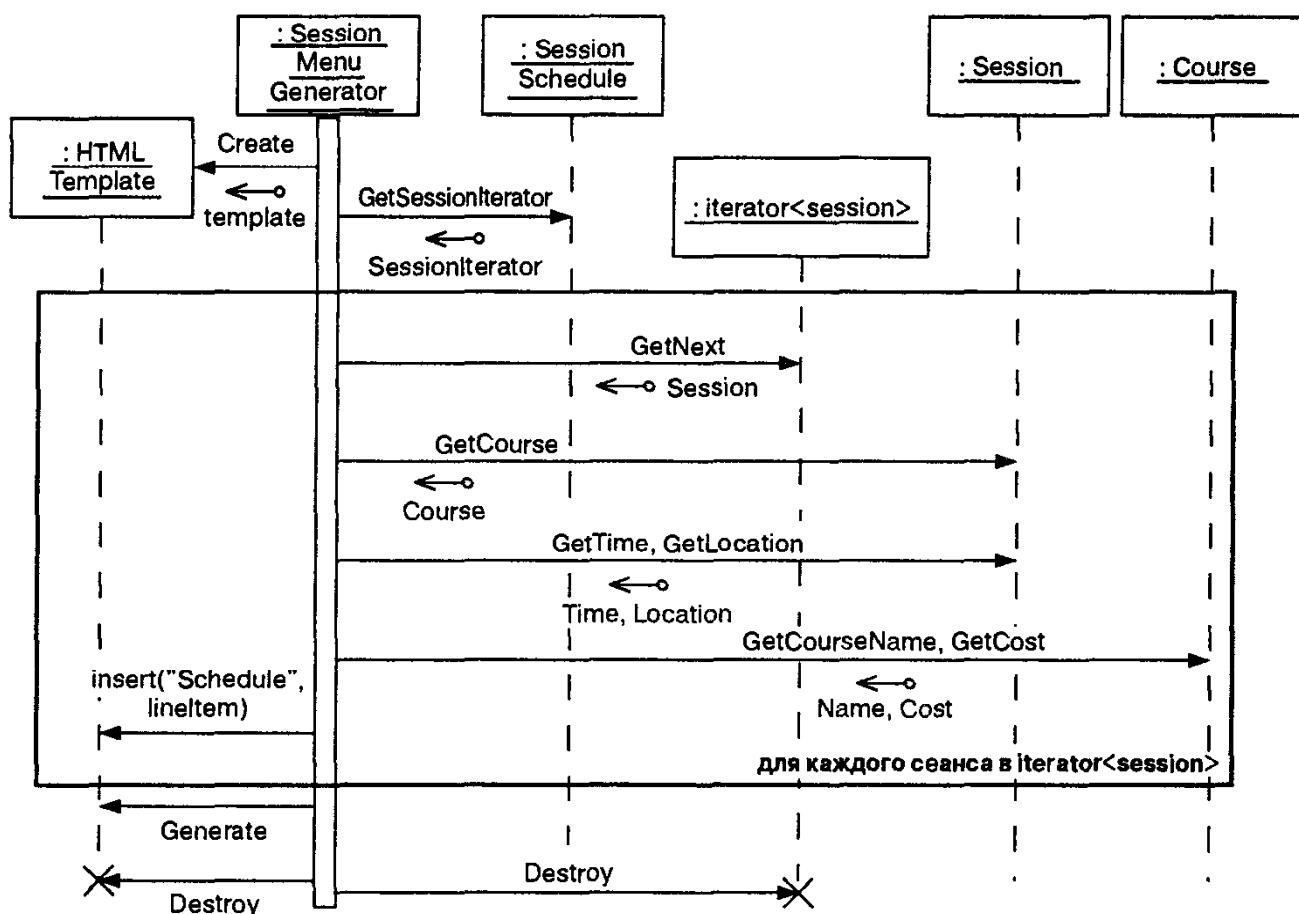


Рис. A.11. Последовательная диаграмма SessionMenuGenerator

в iterator<Session>, запрашивая соответствующий Session для каждого Course. Выбираются сведения о времени и месте проведения курсов из объекта Session, а также название и стоимость курсов из объекта Course. Последнее действие, выполняемое в цикле, заключается в создании элемента линии на основании всей перечисленной информации, а затем его включение в объект HTMLTemplate в точке вставки Schedule. После завершения цикла программа SessionMenuGenerator вызывает метод Generate из класса HTMLTemplate, который затем уничтожается.

## Запись

Названия, указанные в прямоугольниках последовательной диаграммы на рис. А.11, являются подчеркнутыми. Это означает, что в данном случае представляются не классы, а объекты. Имена объектов образуются из двух элементов, разделенных двоеточием. Перед двоеточием находится *простое имя* объекта. После двоеточия указывается имя класса или интерфейса, который реализуется объектом. На рис. А.11 все простые имена пропускаются, поэтому указанные там имена начинаются двоеточием.

Пунктирные линии, “подвешенные” к объектам, называются “*линиями жизни*”. Эти линии представляют продолжительность существования объектов. Все

объекты на рис. А.11, исключая `HTMLTemplate` и `iterator<Session>`, отображают “линии жизни”, которые начинаются вверху и заканчиваются внизу. В соответствии с принятым соглашением, это означает, что данные объекты существуют до начала выполнения сценария и прекращают свое существование после завершения сценария. С другой стороны, класс `HTMLTemplate` явным образом создается и разрушается программой `SessionMenuGenerator`. Подобное обстоятельство иллюстрируется с помощью стрелки, которая указывает на `HTMLTemplate`, который создает класс. Символ ‘X’, указывающий продолжительность существования, находится в нижней части рисунка. Программа `SessionMenuGenerator` также уничтожает итератор `iterator<Session>`; однако не всегда понятно, какой объект создал его раньше. Возможно, что создателем является производный класс `SessionSchedule`. Поэтому на рис. А.11 неявно показан процесс создания объекта `Iterator<Session>`, позиция, символизирующая начало “линий жизни” объекта, показывает время его создания (момент, когда сообщение `GetSessionIterator` отсылается объекту `SessionSchedule`).

Стрелки возле “линий жизни” представляют сообщения. Время идет в направлении “сверху-вниз”, поэтому диаграмма показывает последовательность сообщений, которые передаются между объектами. Эти сообщения именуются с помощью меток, примыкающих к стрелкам. Короткие стрелки в кружках на конце называются *маркерами данных*. Они представляют элементы данных, которые передаются в контексте сообщения. Если они указывают направление передачи сообщения, определяются параметры сообщения. Если их направление противоположно направлению передачи сообщения, они представляют собой возвращаемые сообщением значения.

Длинный тонкий прямоугольник на “линии жизни” `SessionMenuGenerator` называется *активацией*. Активация определяет продолжительность вызова метода или функции. В рассматриваемом случае сообщение, запустившее метод, не отображается. Другие “линии жизни”, показанные на рис. А.11, не включают активацию, поскольку методы являются очень короткими и не генерируют другие сообщения.

Жирный прямоугольник, который окружает некоторые сообщения (рис. А.11), определяет цикл. Критерий завершения цикла указывается в нижней части этого прямоугольника. В рассматриваемом случае заключенные в прямоугольник сообщения повторяются до тех пор, пока не будут проверены все объекты `Session` внутри `iterator<Session>`.

Обратите внимание, что двум стрелкам соответствуют несколько сообщений. Благодаря этому приему сокращается количество стрелок. Сообщения отсылаются в упомянутом порядке, а возвращаемые значения принимаются в том же порядке.

## Абстрактные классы и интерфейсы в последовательных диаграммах

Читатели, наверное, уже заметили, что некоторые из объектов на рис. A.11 созданы на основе экземпляров интерфейсов. Например, объект `SessionSchedule` является одним из классов интерфейсов базы данных. Даже может показаться, что в этом случае нарушается принцип, согласно которому экземпляры объектов не могут создаваться на основе абстрактных классов или интерфейсов.

Имя класса объекта в последовательной диаграмме необязательно должно совпадать с именем фактического типа объекта. Достаточно, чтобы объект просто соответствовал интерфейсу именованного класса. В статических языках программирования, таких как C++, Java или Eifel, объект должен принадлежать либо именованному классу в последовательной диаграмме, либо классу, который является производным от класса или интерфейса в последовательной диаграмме. В динамических языках программирования, таких как Smalltalk или Objective-C, достаточно, чтобы объект соответствовал именованному интерфейсу на последовательной диаграмме<sup>10</sup>.

Таким образом, объект `SessionSchedule`, изображенный на рис. A.11, ссылается на объект, чей класс реализуется или наследуется на основе интерфейса `SessionSchedule`.

### Статическая модель Session Menu Generator

Показанная на рис. A.11 динамическая модель влечет статическую модель, приведенную на рис. A.12. Обратите внимание на то, что в качестве взаимосвязей используются зависимости или стереотипизированные ассоциации. Это связано с тем, что ни один из классов, показанных на рисунке, не хранит экземпляры переменных, которые ссылаются на другие классы. Все эти взаимосвязи являются временными, поскольку они не делятся дальше, чем активизируется прямоугольник на “линии жизни” `SessionMenuGenerator` (рис. A.11).

Взаимосвязь между `SessionMenuGenerator` и `SessionSchedule` имеет специальное значение. Обратите внимание на то, что `SessionSchedule` отображается вместе со свойством `{singleton}`. Это означает, что только один объект `SessionSchedule` может существовать внутри приложения, причем доступ к нему осуществляется в глобальной области доступа. (Также обратите внимание на описание шаблона `Singleton` в главе 16.)

<sup>10</sup>Если вам пока не совсем понятен излагаемый материал, не беспокойтесь. В динамических языках программирования (Smalltalk и Objective-C) можно произвольно отсылать любые сообщения выбранным объектам. При этом компилятор не проверяет, было ли принято сообщение объектом. Если во время выполнения программы было отослано сообщение объекту, который не распознал его, возникает ошибка времени выполнения. Исходя из этого, становится возможным существование двух совершенно различных и несвязанных объектов, которые принимают одни и те же сообщения. Говорят, что подобные объекты соответствуют одному и тому же интерфейсу.

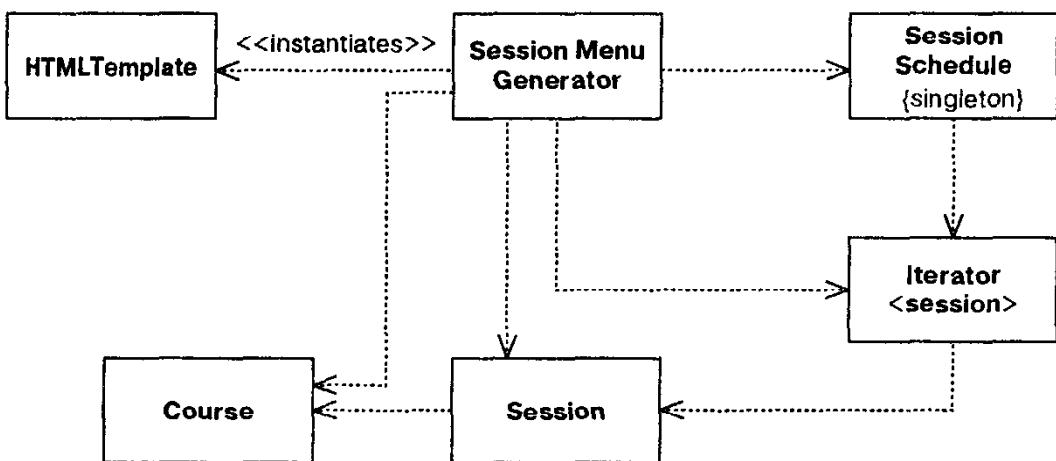


Рис. А.12. Статическая модель приложения Session Menu Generator

Зависимость между `SessionMenuGenerator` и `HTMLTemplate` переносится стереотипом “создает”. Это означает, что `SessionMenuGenerator` создает экземпляры `HTMLTemplate`.

Ассоциации, образованные стереотипом “параметр”, показывают, что объекты осуществляют взаимный поиск с помощью аргументов методов или возврата значений.

### **Каким образом объект `SessionMenuGenerator` получает контроль?**

Прямоугольник активации “линии жизни” `SessionMenuGenerator`, изображенный на рис. А.11, не показывает, каким образом был запущен объект. Возможно, что на некотором высшем уровне сущности, например, `main()` вызывает метод `SessionMenuGenerator`. Этот метод может называться `Run()`. Все это достаточно интересно, поскольку появляется возможность создавать много других CGI-программ, которые будут запускаться на выполнение с помощью метода `main()`. Возможно, в данном случае идет речь о базовом классе или интерфейсе `CGIProgram`, который определяет метод `Run()`, и из которого, возможно, наследуется `SessionMenuGenerator`.

### **Передача результатов пользовательского ввода в CGI-программу**

Класс `CGIProgram` позволяет решить и некоторые другие вопросы. Все CGI-программы обычно вызываются броузером после того, как пользователь заполнит поля некой формы, отображаемой на экране. Данные, введенные пользователем, передаются функции `main()` в CGI-программе через стандартный поток ввода. Таким образом функция `main()` может передавать ссылку на поток стандартного ввода объекту `CGIProgram`, в результате чего данные становятся доступными производным классам.

В какой же форме передаются данные от броузера CGI-программе? В виде набора “имя-значение”. Каждое поле в форме, которое заполняется пользователем, имеет свое имя. Следовательно, нужно “научить” производные классы

CGIProgram запрашивать значение конкретного поля, основываясь на его имени. Например:

```
string course = GetValue("course");
```

Здесь `main()` создает класс `CGIProgram` и снабжает его необходимыми данными путем передачи стандартного потока ввода конструктору. Затем функция `main()` вызывает метод `Run()` для класса `CGIProgram`, разрешая его “открытие”. Производный класс `CGIProgram` вызывает метод `GetValue(строка)` с целью получить доступ к данным в форме.

В данном случае имеет место одна дилемма. Если функция `main()` будет обобщенной, придется создать соответствующие производные классы `CGIProgram`, количество которых может быть достаточно большим. Каким же образом можно избежать “размножения” функции `main()`?

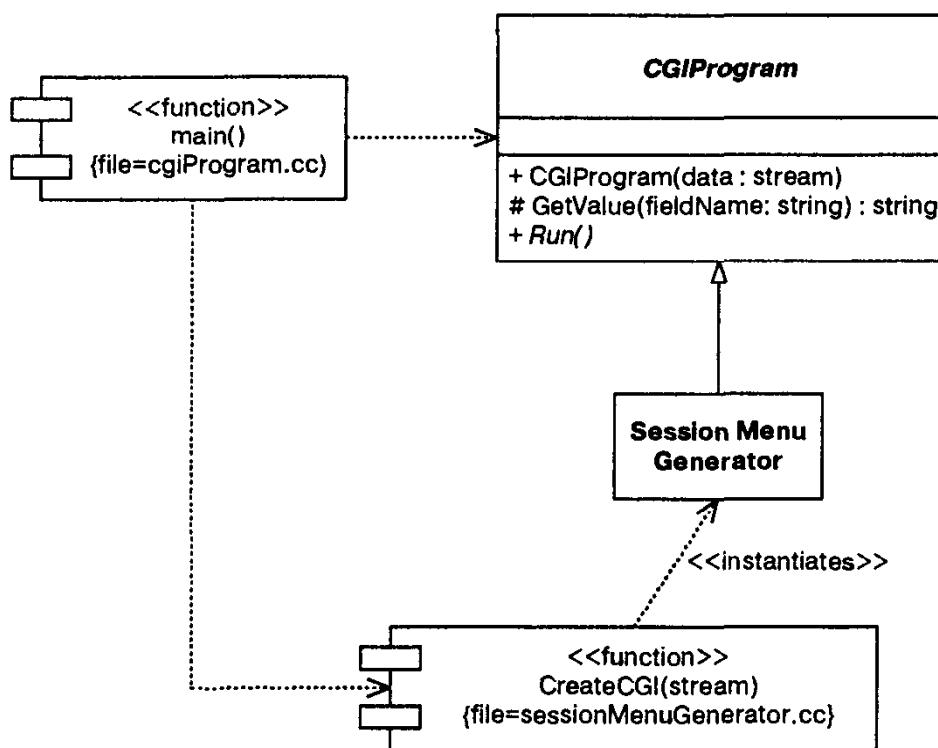


Рис. А.13. Архитектура CGI-программ

Эта проблема решается с помощью временно связанного полиморфизма. Для этого функция `main()` реализуется в файле реализации для класса `CGIProgram` (`cgiProgram.cc`). Внутри функции `main()` объявляется глобальная функция, `CreateCGI`. Однако данная функция не реализована в полном смысле этого слова. Скорее, она будет реализована в файле реализации производного класса `CGIProgram(SessionMenuGenerator.cc)` (рис. А.13).

Авторам CGI-программ больше не требуется создавать программу `main()`. Для каждого производного класса `CGIProgram` им следует поддерживать реализацию глобальной функции `CreateCGI`. Эта функция возвращает производный

класс обратно `main()`, причем этот класс может обрабатываться произвольным образом.

На рис. А.13 показано использование компонентов вместе со стереотипом “функция”, который представляет свободные глобальные функции. Здесь также показано применение свойств, с помощью которых отображаются файлы реализации, использованные функцией. Обратите внимание, что функция `CreateCGI` комментируется с помощью свойства `{file=sessionMenuGenerator.cc}`.

## Резюме

Достаточно подробно была рассмотрена запись UML в контексте простого примера. Были продемонстрированы различные соглашения по записи, используемые на различных фазах разработки ПО. Приведен пример анализа проблемы на основе примеров использования и типов, с помощью которых формировалась модель предметной области приложения. Было показано, каким образом классы, объекты и компоненты могут связываться в статических и динамических диаграммах в целях описания архитектуры и приемов разработки программ. Для каждой из упомянутых концепций приводятся примеры применения записи UML, а также демонстрируются приемы использования этой записи.

На этом изучение UML и проектирования программ не завершается. В приложении Б рассматривается другой пример, позволяющий завершить изучение UML, а также различных моментов, связанных с анализом и проектированием программ.

## Литература

1. Booch Gr. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings: 1994.
2. Rumbaugh и др. *Object Oriented Modeling and Design*. Prentice Hall: 1991.
3. Wirfs-Brock R. и др. *Designing Object-Oriented Software*. Prentice Hall: 1990.
4. Coad P., Yourdon E. *Object Oriented Analysis*. Yourdon Press: 1991.
5. Jacobson I. *Object Oriented Sofnware Engineering a Use Case Driven Approach*. Addison-Wesley, 1992.
6. Cockburn A. *Structuring Use Cases with Goals*, <http://members.aol.com/acockburn/papers/usecass.htm>.
7. Kennedy E. *Object Practitioner's Guide*, <http://www.zoo.6.uk/~z0001039/PracGuides>, 1997.
8. Booch G. *Object Solutions*. Addison-Wesley, 1995.
9. Gamma и др. *Design Patterns*. Addison-Wesley, 1995.

# Б

## UML-запись II: STATMUX

В этой главе продолжается изучение UML-записи с детальным обсуждением ее отдельных аспектов. В качестве контекста для данного исследования мы рассмотрим проблему статистического мультиплексора.

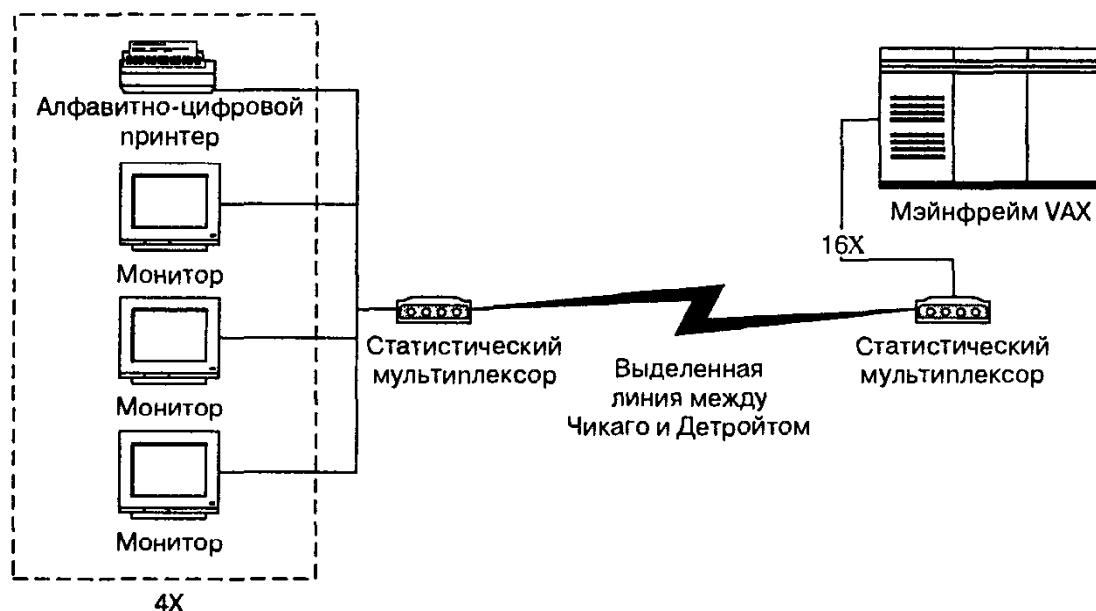
### Определение статистического мультиплексора

Статистический мультиплексор — это устройство, которое позволяет передавать несколько потоков последовательных данных через одну линию телекоммуникаций. Рассмотрим, например, устройство, в которое вмонтирован один модем со скоростью передачи данных 56 Кбит/с и которое имеет 16 последовательных портов. Если два устройства такого типа соединить посредством одной телефонной линии, символы, посылаемые в порт 1 на одном устройстве, будут выходить из порта 1 на другом устройстве. Такое устройство может поддерживать 16 полнодуплексных одновременно происходящих сеансов связи через один модем.

На рис. Б.1 представлено стандартное приложение для такого устройства 80-х годов прошлого века. Допустим, что в Чикаго находится совокупность ASCII-терминалов и принтеров, которые нужно подключить к компьютеру VAX в Детройте. В нашем распоряжении имеется выделенная линия со скоростью передачи данных 56 Кбит/с, соединяющая эти две географические точки между собой. Статистический мультиплексор создает между этими точками 16 виртуальных последовательных каналов.

Понятно, что если все 16 каналов функционируют одновременно, они распределяют между собой пропускную способность 56 Кбит/с, достигая при этом фактической скорости передачи битов несколько ниже, чем 3500 бит/с из расчета на одно устройство. Однако рабочий режим большинства терминалов значительно ниже 100 %. В действительности, рабочий цикл большинства приложений, как правило, составляет менее 10 %. Таким образом, несмотря на то, что линия используется совместно, согласно статистике, каждый пользователь фактически работает со скоростью передачи данных 56 Кбит/с.

Проблемный вопрос, который мы собираемся исследовать в этой главе, заключается в программном обеспечении, которое лежит в основе статистического



**Рис. Б.1.** Стандартное приложение статистического мультиплексора

мультиплексора. Это программное обеспечение управляет модемом и аппаратно реализованными последовательными портами. Оно также определяет протокол мультиплексирования, применяемый для совместного использования линии связи между всеми последовательными портами.

## Программная среда

На рис. Б.2 представлена блочная диаграмма<sup>1</sup>, которая отображает расположение программного обеспечения в статистическом мультиплексоре. Программное обеспечение находится в модуле между 16 последовательными портами и модемом.

Каждый последовательный порт генерирует два сигнала прерывания, направляемые главному процессору. Один из этих сигналов генерируется сразу по готовности порта к отправке символа, а другой — по факту получения символа. Аналогичные сигналы прерывания генерируются также модемом. Следовательно, в систему поступают 34 сигнала прерывания. Приоритетность сигналов прерывания, поступающих из модема, выше, чем приоритетность сигналов, поступающих из последовательного порта. В результате обеспечивается передача данных модемом с максимально возможной скоростью 56 Кбит/с, даже если другие последовательные порты периодически “тормозят”.

Кроме того, есть еще и таймер, который генерирует сигнал прерывания каждую миллисекунду. Этот таймер позволяет программному обеспечению планировать события в нужном порядке.

<sup>1</sup>Блочные диаграммы — одна из форм языка GML (galactic modeling language, язык галактического моделирования) Кена Бека (Kent Beck). Диаграммы GML состоят из линий, прямоугольников, окружностей, овалов, а также других форм, необходимых для получения сечений.

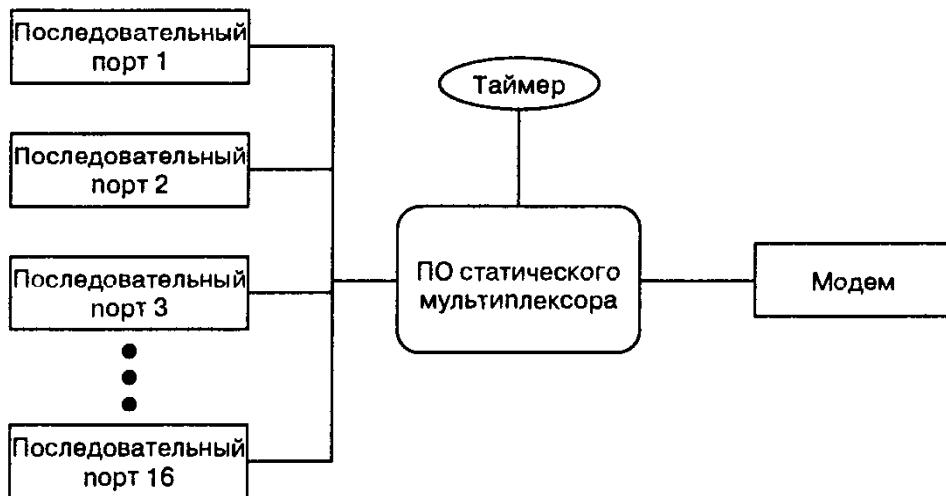


Рис. Б.2. Блочная диаграмма, отображающая строение статического мультиплексора

## Ограничения режима реального времени

Методом несложного подсчета можно продемонстрировать проблему, с которой сталкивается система. В любой заданный момент времени 34 источника сигналов прерывания могут потребовать возможность генерировать 5600 прерываний в секунду плюс дополнительная тысяча сигналов прерывания в секунду от таймера, что в сумме составляет 191 400 сигналов прерывания в секунду. Таким образом, программное обеспечение имеет возможность потратить не более 5,2 мс на обслуживание каждого такого сигнала. Это очень жесткие временные рамки, и поэтому возникает необходимость в достаточно быстром процессоре, который позволит исключить возможность потери символов.

Эту проблему еще в больше мере усугубляет тот факт, что системе, помимо обслуживания сигналов прерывания, предстоит выполнить значительный объем другой работы. Система также должна управлять протоколом связи через modem, собирать входящие символы от последовательных портов и отделять те символы, которые необходимо отправить в серийныйпорт. На выполнение всех этих задач требуется определенное время, которое необходимо выделить в промежутках между обслуживанием отдельных символов прерывания.

К счастью, максимальная постоянная способность системы составляет всего лишь 11 200 символов в секунду (т.е. имеется в виду количество символов, которые можно одновременно послать и получить через modem). Это означает, что в среднем между отдельными символами<sup>2</sup> мы располагаем временем почти 90 мс.

Поскольку имеющиеся в нашем распоряжении стандартные службы обработки прерываний (ISR, Interrupt Service Routines) не могут функционировать более 5,2 мс, мы располагаем 94% вычислительной мощности процессора, которую

<sup>2</sup>“Целая вечность”...

можно дополнительно распределять между прерываниями. Это означает, что нам не нужно беспокоиться о сохранении эффективности вне служб ISR.

## Служба обработки прерываний входа

Службы такого рода должны быть написаны на языке ассемблера. Первостепенная цель использования ISR-служб заключается в том, чтобы получить символ от аппаратного обеспечения и сохранить его с возможностью его дальнейшей обработки с помощью программного обеспечения, не принадлежащего к разряду ISR-служб. В этих целях, как правило, используется кольцевой буфер.

На рис. Б.3 представлена классовая диаграмма, которая демонстрирует структуру служб ISR, обрабатывающих результаты ввода и относящихся к ним кольцевых буферов. Для того чтобы описать уникальные явления, имеющие отношение к использованию служб обработки прерываний, мы задействовали в нашем исследовании несколько искусственно созданных новых стереотипов и свойств.

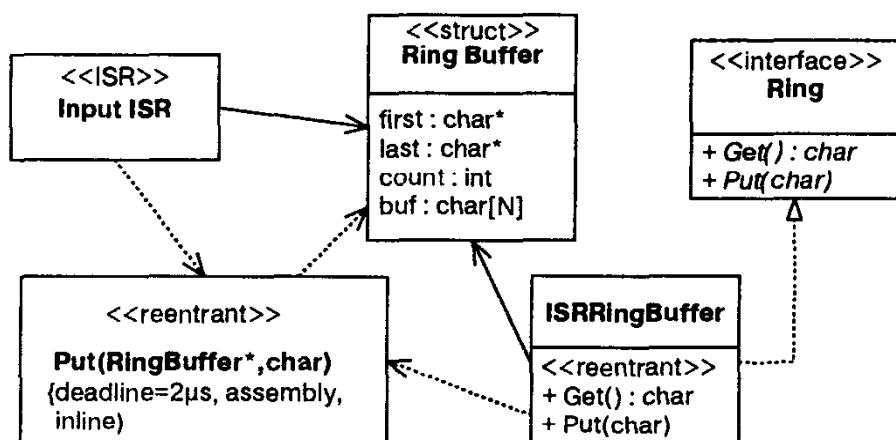


Рис. Б.3. Служба, обрабатывающая результаты ввода

Первым на рисунке представлен класс служб по обработке прерываний входа. Этот класс имеет стереотип “ISR”, который указывает, что данный класс относится к классу служб обработки прерываний. Классы такого рода пишутся на языке ассемблера и имеют всего лишь один метод. Этот метод не имеет имени и вызывается только при возникновении прерывания.

Класс `InputISR` связан с относящимся к нему кольцевым буфером `RingBuffer`. Стереотип “struct”, относящийся к классу `RingBuffer`, указывает, что данный класс не имеет методов. Стереотип “struct” – это исключительно структура данных. Данный стереотип был искусственно создан из тех соображений, чтобы доступ к нему можно было получить с помощью функций языка ассемблера, которые при других обстоятельствах не имеют доступа к методам класса.

Функция `Put(RingBuffer*, char)` изображена в пиктограмме класса и обозначается стереотипом “reentrant”. Этот стереотип, используемый таким об-

разом, представляет свободную функцию, которая защищена от прерываний<sup>3</sup>. Эта функция позволяет добавлять символы в кольцевой буфер. Она вызывается с помощью класса `InputISR` по факту получения символа.

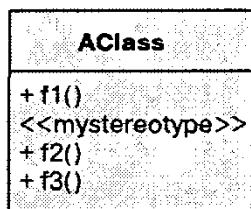
Свойства функции свидетельствуют о том, что она имеет предельный срок режима реального времени 2 мс, что она должна быть написана на языке ассемблера и что при каждом ее вызове должно применяться внутритекстовое кодирование. Два последних из упомянутых свойств направлены на обеспечение первого свойства.

Класс `ISRRingBuffer` – это регулярный класс, методы которого выполняются за пределами службы обработки прерываний. Этот класс применяет структуру `RingBuffer`, а также поддерживает для нее фасад класса. Все методы, присущие данному классу, соответствуют стереотипу “reentrant” и, следовательно, защищены от прерываний.

Класс `ISRRingBuffer` реализует интерфейс `Ring`. С помощью этого интерфейса клиенты вне служб обработки прерываний могут получить доступ к символам, хранящимся в кольцевых буферах. Этот класс находится на границе интерфейса, разделяющего прерывания и остальную часть системы.

## Стереотипы в окнах списка

Когда в ячейках списка классов появляются стереотипы, они имеют особое значение. Элементы, которые находятся под тем или иным стереотипом, всегда соответствуют ему.



В данном примере функция `f1()` не имеет явно выраженного стереотипа. А функции `f2()` и `f3()` соответствуют стереотипу «`mystereotype`».

Не существует каких-либо ограничений относительно количества стереотипов, которые могут появляться в списковом окне такого рода. Каждый новый стереотип подменяет предыдущий. Все элементы, изображенные между двумя стереотипами, соответствуют стереотипу над ними.

Пустой стереотип «» может использоваться в середине списка и указывает на то, что последующие элементы не имеют явно выраженного стереотипа.

<sup>3</sup>Реентерабельность – это сложный вопрос, рассмотрение которого выходит за рамки этой главы. Читателю лучше обратиться к специальным книгам, посвященным конкурентному программированию, а также программированию в режиме реального времени, например, книге Дуга Ли (Doug Lea) *Concurrent Programming in Java*. Addison-Wesley, 1997.

## Поведение кольцевого буфера

Кольцевые буфера можно описать с помощью “простой” машины состояний, изображенной на рис. Б.4. Такая машина состояний демонстрирует, что произойдет, если для объектов класса `ISRRingBuffer` вызвать методы `Get()` и `Put()`.

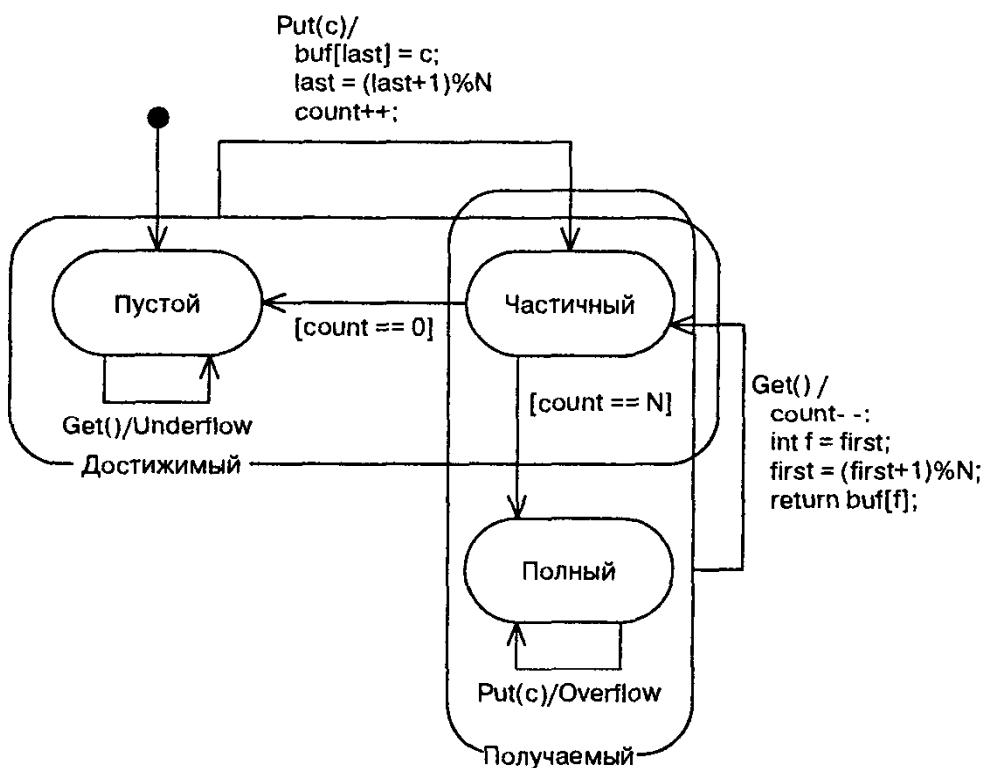


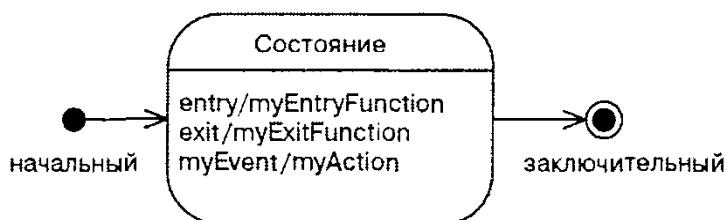
Рис. Б.4. Диаграмма состояния кольцевого буфера

На диаграмме показаны три состояния, в которых может находиться кольцевой буфер. В начале диаграммы представлено состояние `Empty`. В состоянии `Empty` в кольцевом буфере отсутствуют символы. В этом состоянии вызов метода `Get()` повлечет за собой очистку буфера. В данной диаграмме не отображено, что происходит во время очистки или переполнения буфера. Эти вопросы будут рассмотрены несколько позже. Два состояния `Empty` и `Partial` представляют собой подсостояния суперсостояния `Puttable`. Суперсостояние `Puttable` представляет собой те состояния, в которых метод `Put()` выполняется, не вызывая переполнения буфера. Каждый раз при вызове метода `Put()` из одного из состояний `Puttable`, входящий символ сохраняется в буфере, при этом соответственно устанавливаются значение счетчика и индексы. Состояния `Partial` и `Full` — это состояния, принадлежащие суперсостоянию `Gettable`. Суперсостояние `Gettable` представляет собой те состояния, в которых функция `Get()` выполняется, не вызывая переполнения буфера. При вызове в этих состояниях функции `Get()`, в кольцевом буфере удаляется следующий символ, который затем возвращается вызывающей функции. Соответственно устанавливаются значение счетчика и индексы. В состоянии `Full` вызов функции `Put()` приводит к возникновению `Overflow`.

Описанные выше два перехода из состояния `Partial` регулируются *режимом защиты*. Машина выполняет переход из состояния `Partial` в состояние `Empty` каждый раз, когда переменная счетчика достигает нулевого значения. Точно так же машина переходит из состояния `Partial` в состояние `Full` каждый раз, когда переменная счетчика достигает значения размера буфера ( $N$ ).

## Состояния и внутренние переходы

В UML-записи состояние представлено прямоугольником с закругленными углами. Этот прямоугольник может состоять из двух ячеек.



В верхней ячейке просто указывается имя состояния. Если в ячейке имя не определено, в таком случае состояние является анонимным. Все анонимные состояния отличаются друг от друга.

В нижней ячейке перечисляются внутренние переходы определенного состояния. Внутренние переходы обозначаются как `eventName/action`. Имя `eventName` – это имя события, которое может произойти во время пребывания машины в данном состоянии. Машина реагирует на событие, оставаясь в этом состоянии и выполняя заданное действие.

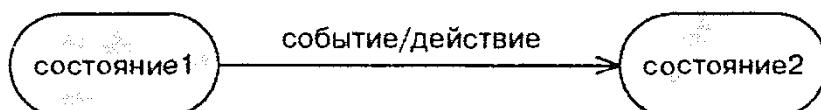
Существует два специальных события, которые могут использоваться во внутренних переходах. Эти события изображены в пиктограмме вверху. Событие входа происходит в момент входа в состояние. Событие выхода происходит в момент выхода из состояния (даже в случае, если машина немедленно возвращается в это состояние).

Действие `action` может представлять собой имя другой машины с конечным числом состояний, которая имеет как начальное, так и конечное состояние. Или же действие может быть представлено в форме процедурного выражения, написанного на каком-либо компьютерном языке или с помощью псевдокода. В такой процедуре могут использоваться операции и переменные объекта (если такие имеются), которые содержат в себе состояние машины. Или же действие может быть представлено в форме `^object.message (arg1, arg2, ...)`, в результате чего именованному объекту отсылается именованное сообщение. Существуют две специальные пиктограммы *псевдосостояния*, которые изображены на предыдущей диаграмме. Слева от диаграммы изображена черная окружность, которая представляет собой начальное псевдосостояние. Справа

изображен “бычий глаз”, который представляет конечное псевдосостояние. При первоначальном вызове машины с конечным числом состояний выполняется переход из начального псевдосостояния в состояние, с которым было установлено соединение. Таким образом, для выхода из начального псевдосостояния требуется выполнить только один переход. Когда происходит событие, вызывающее переход в конечное псевдосостояние, машина состояний завершает свою работу, прекращая обработку событий.

## Переходы между состояниями

Машина с конечным числом состояний — это набор состояний, связанных между собой переходами. Переходы представлены в виде стрелок, которые соединяют одно состояние с другим. Переход обозначен ярлыком с именем события, которое вызывает данный переход.



На данном рисунке представлены два состояния, связанные между собой одним переходом. Переход выполняется только тогда, когда машина находится в состоянии `state1`, и при этом происходит соответствующее событие. После начала выполнения перехода осуществляется выход из состояния `state1` наряду с соответствующим действием.

Событие перехода может быть квалифицировано определением условия режима защиты. Переход выполняется только в том случае, если произойдет соответствующее событие *и* если условие режима защиты имеет значение `true`. Условия режима защиты представлены булевыми выражениями, которые записываются в квадратных скобках после имени события (например, `myEvent [myGuardCondition]`).

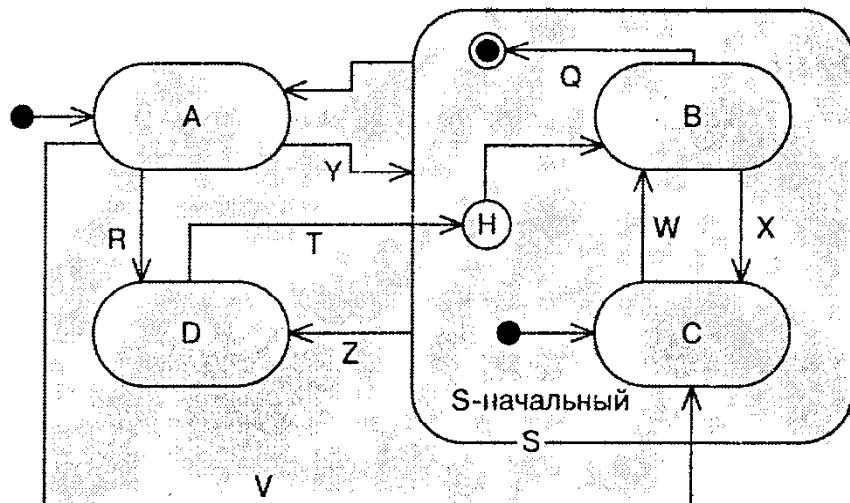
Действия переходов абсолютно аналогичны действиям внутренних переходов в рамках состояний. (См. “Состояния и внутренние переходы”.)

## Вложенные состояния

Если одна пиктограмма состояния заключает в себе одну или несколько других пиктограмм, такие вложенные состояния называются *подсостояниями* внешнего *суперсостояния*.

В приведенной выше диаграмме состояния В и С — это подсостояния суперсостояния S. Машина состояний начинается в состоянии А, которое отображено с помощью начального псевдосостояния. Выполнение перехода V активизирует

ет подсостояние С в рамках суперсостояния S. В результате выполняется вызов функций входа в оба состояния С и S.



Если во время пребывания машины в состоянии А выполняется переход Y, машина входит в суперсостояние S. Переход в суперсостояние должен привести к активизации одного из принадлежащих ему подсостояний. Если переход завершается на границе суперсостояния, как в случае перехода Y, в таком случае выполняется переход из начального псевдосостояния в рамках данного суперсостояния. Таким образом, переход Y вызывает переход из начального S-псевдосостояния в подсостояние С.

После начала выполнения перехода Y осуществляется вход как в суперсостояние S, так и в подсостояние С. В этот момент происходит вызов любых действий входа в состояния S и С. Вызов действий входа в суперсостояние предшествует вызову действий входа в подсостояние. Это может спровоцировать переходы W и X, в результате которых машина выполнит перемещение между подсостояниями В и С. При этом действия выхода и входа выполняются в обычном порядке, но поскольку выход из суперсостояния S не выполняется, отсутствует и вызов функций выхода из этого состояния.

В итоге выполняется переход Z. Обратите внимание, что переход Z покидает границу суперсостояния. Это означает, что независимо от того, активизировано или нет подсостояние В или С, переход Z вызывает перемещение машины в состояние D. Такое решение эквивалентно двум отдельным переходам: одному — из состояния С в состояние D, а другому — из состояния В в состояние D. Оба такие перехода обозначены ярлыком Z. При выполнении перехода Z осуществляется соответствующее действие выхода из подсостояния, а затем выполняется действие выхода из суперсостояния S. После этого происходит вход в состояние D и выполняется действие выхода из него.

Обратите внимание, что переход Q завершается на конечном псевдосостоянии. Если происходит переход Q, завершается суперсостояние S. В результате выполняется непомеченный переход из S в A. Завершение суперсостояния также

приводит к восстановлению хронологии. Это явление описывается далее в настоящем приложении.

Переход Т завершается на специальной пиктограмме в рамках суперсостояния S. Эта пиктограмма называется *историческим* маркером. При выполнении перехода Т снова активизируется то подсостояние, которое последним активизировалось в рамках суперсостояния S. Таким образом, если переход Z произошел во время пребывания С в активном состоянии, он повлечет за собой повторную активизацию С.

Если переход Т выполняется тогда, когда исторический маркер находится в неактивном состоянии, в таком случае он повлечет за собой непомеченный переход из исторического маркера в подсостояние В. Это — стандартное событие, соответствующее ситуации отсутствия хронологии. Исторический маркер находится в неактивном состоянии, если вход в S никогда не происходил или если только что произошел выход из состояния S выполнением перехода Q.

Следовательно, в результате последовательности событий Y-Z-T машина останется в подсостоянии С, в то время в результате последовательности событий R-T и Y-W-Q-R-T она останется в подсостоянии В.

## Служба обработки прерываний выхода

Обработка прерываний выхода в значительной степени аналогична обработке прерываний входа. Однако существует несколько отличий. Части системы, которые не подвержены прерываниям, загружают выходной кольцевой буфер символами, предназначенными для дальнейшей отправки. Последовательный порт генерирует сигнал прерывания в любой момент его готовности к принятию следующего символа. Служба обработки прерываний захватывает следующий символ из кольцевого буфера и отсылает его в последовательный порт.

Если в момент перехода последовательного порта в состояние готовности в кольцевом буфере отсутствуют символы, в таком случае необходимость в службе обработки прерываний отсутствует. Последовательный порт уже подал сигнал о своей готовности принять новый символ, и он не подаст такой сигнал снова до тех пор, пока он не получит предназначенный для отправки символ и не завершит процесс его передачи. Таким образом поток прерываний прекращается. Это означает, что нам необходимо воспользоваться определенной стратегией, чтобы возобновить прерывания выхода при поступлении новых символов. На рис. Б.5 показана структура службы обработки прерываний выхода. На рисунке отчетливо прослеживается схожесть структуры такой службы со структурой службы обработки прерываний входа, которая была представлена на рис. Б.3. Однако обратите внимание на зависимость между "вызовами", отсылаемыми объектом `ISRRingBuffer` в адрес объекта `OutputISR`. Это свидетельствует о том, что

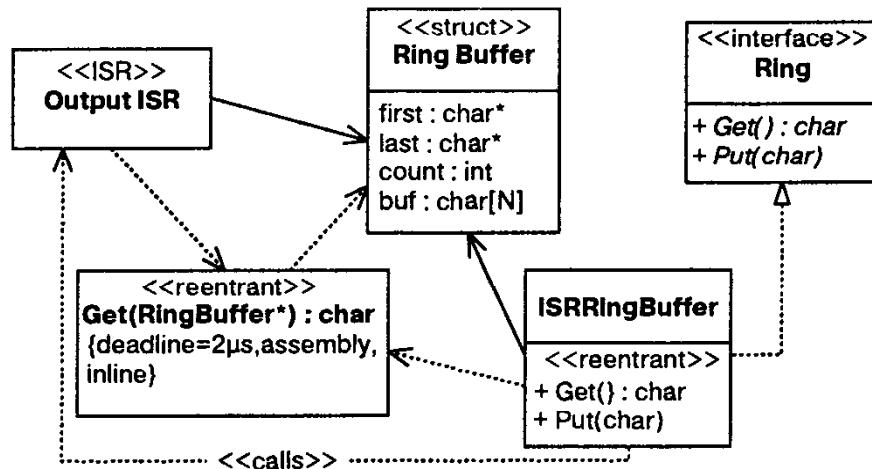


Рис. Б.5. Служба обработки прерываний выхода

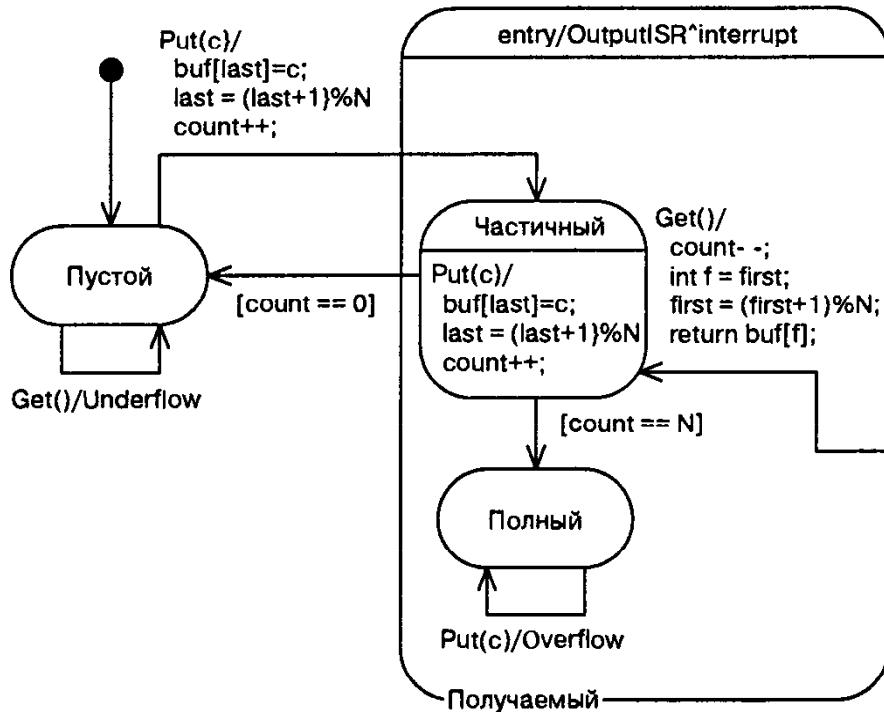


Рис. Б.6. Машина состояний, отображающая структуру службы по обработке прерываний выхода

объект **ISRRingBuffer** может заставить объект **OutputISR** выполняться так, как будто было получено прерывание.

На рис. Б.6 показаны необходимые видоизменения машины конечного числа состояний кольцевых буферов выхода. Сравните этот рисунок с рис. Б.4. Обратите внимание, что на данном рисунке отсутствует суперсостояние **Puttable** и отображены два перехода **Put**. Первый переход **Put** обеспечивает перемещение из состояния **Empty** в состояние **Partial**. В результате выполняется действие входа в суперсостояние **Gettable** и генерируется искусственное прерыва-

ние OutputISR<sup>4</sup>. Второй переход Put носит внутренний характер и происходит в рамках состояния Partial.

## Протокол связи

Два статических мультиплексора обмениваются между собой информацией через свои модемы. Каждый мультиплексор отсылает другому пакеты через линию телекоммуникаций. Следует отметить, что такая линия несовершенна, и в процессе коммуникации могут возникнуть ошибки и прерывания связи. При передаче символы могут затеряться или исказиться, а электрические разряды или другого вида электромагнитная интерференция могут привести к возникновению ложных символов. Следовательно, между двумя модемами необходимо разместить протокол связи. Такой протокол должен обеспечивать полную и достоверную передачу пакетов, а также возможность повторной передачи пакетов в случае их искажения или потери.

На рис. Б.7 представлена *диаграмма действий*. Диаграмма отображает протокол связи, который будет использоваться статистическими мультиплексорами в рассматриваемом нами примере. Этот протокол представляет собой относительно прямолинейный протокол раздвижного окна с конвейерной обработкой, позволяющий совмещать передачу прямых и обратных пакетов<sup>5</sup>.

Протокол начинает выполняться в начальном псевдосостоянии, в котором выполняется инициализация нескольких переменных и создаются три независимых потока. Переменные будут рассмотрены несколько позже в разделах, которые посвящены рассмотрению потоков, зависящих от таких переменных. “Временной поток” используется для повторной передачи пакетов, если в течение установленного временного периода не поступило соответствующее подтверждение о получении пакета. Этот поток также используется для обеспечения своевременной отправки подтверждений, свидетельствующих об успешном поступлении пакетов. Отсылающий поток используется для отправки пакетов, стоящих в очереди для последующей передачи. Принимающий поток используется для получения, подтверждения и обработки пакетов. Рассмотрим каждый из этих потоков поочереди.

### Отсылающий поток

Переменная s содержит серийный номер, который отмечается на ближайшем исходящем пакете. Каждый пакет получает серийный номер в диапазоне 0 . . . N.

<sup>4</sup>Механизм, применяемый для генерирования искусственного прерывания подобного рода, сильно зависит от платформы. На некоторых компьютерах возможно просто вызвать ISR, аналогично функции. Другие компьютеры требуют более детально разработанных протоколов, искусственно вызывающих ISR.

<sup>5</sup>Для получения дополнительных сведений об этом коммуникационном протоколе обратитесь к *Computer Networks*. 2-е издание. Tanenbaum, Prentice Hall, 1988, раздел 4.4.

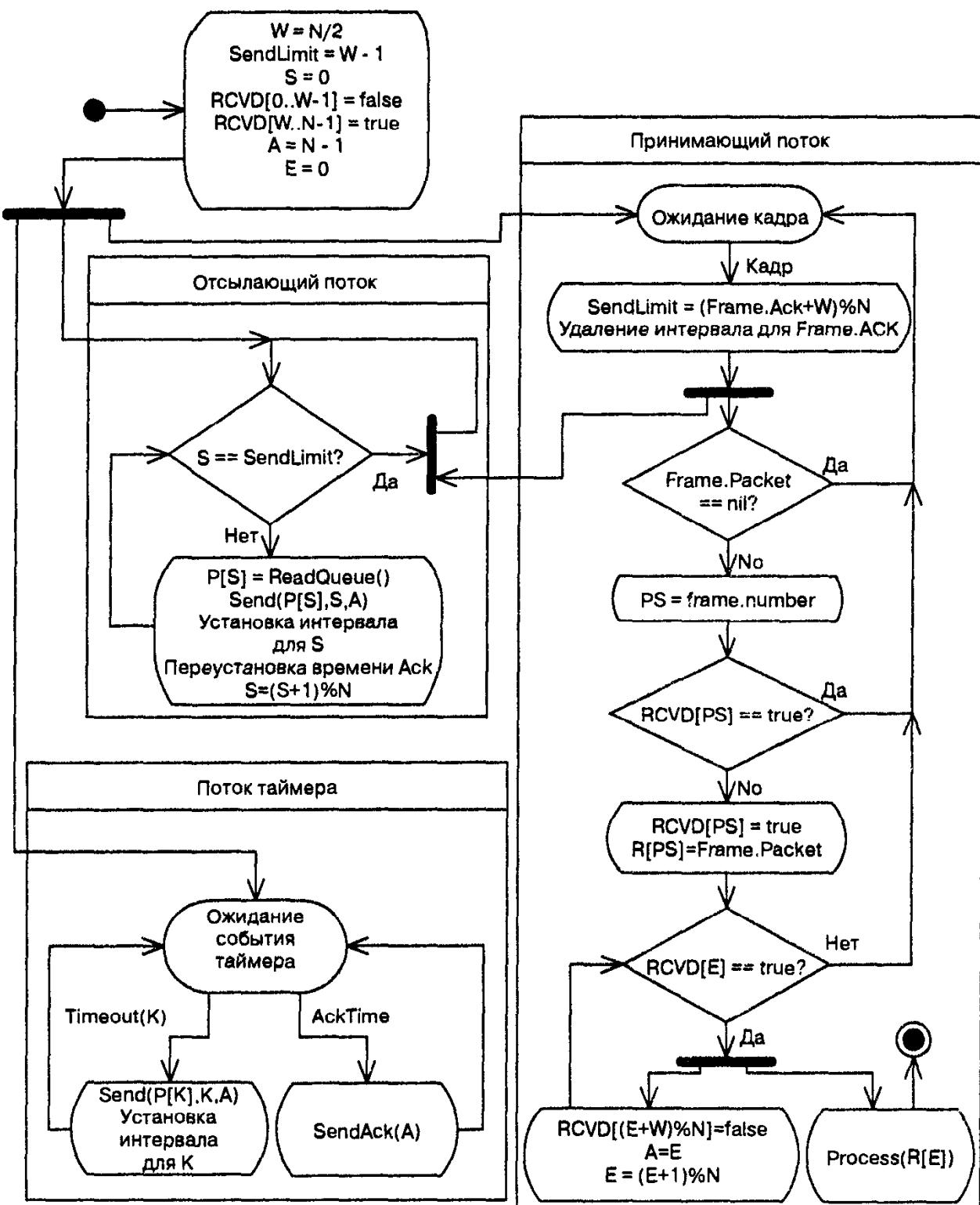


Рис. Б.7. Диаграмма активности протокола связи

Отсылающий поток продолжает отправлять пакеты, не ожидая поступления на них соответствующих подтверждений, до тех пор, пока число пакетов, подтверждение для которых отсутствует, не достигнет числа  $W$ . Число  $W$  установлено в виде соотношения  $N/2$ . Это означает, что в любой заданный момент времени задействовано не больше половины серийных номеров. Переменная  $SendLimit$ , как правило, представляет серийный номер пакета, который находится за пре-

делами окна ( $w$ ), и поэтому это наименьший серийный номер, который нельзя отправить.

По мере того как отсылающий поток продолжает отсылать пакеты, переменная  $s$  приращивается по модулю  $N$ . Когда  $s$  достигает значения  $SendingLimit$ , поток блокируется до тех пор, пока значение  $SendingLimit$  не будет изменено принимающим потоком. Если переменная  $s$  еще не достигла значения  $SendingLimit$ , в таком случае из очереди выталкивается новый пакет с помощью функции `ReadQueue()`. Пакет помещается в позицию  $s$  массива  $p$ . Пакет находится в этом массиве на тот случай, если понадобится его повторная передача. Затем пакет отсылается вместе со своим серийным номером ( $s$ ) и ответным подтверждением ( $A$ ).  $A$  — это серийный номер последнего полученного пакета. Эта переменная обновляется приемным потоком.

Как только пакет отправлен, для него запускается время ожидания. Если время ожидания истекает до того, как на пакет поступит соответствующее подтверждение, временной поток делает вывод о факте потери пакета либо подтверждения, и повторяет передачу пакета.

В этот момент выполняется также установка таймера  $Ack$ , контролирующего своевременную отправку подтверждений. Когда время на таймере  $Ack$  истекает, временной поток делает вывод, что с момента отправки последнего подтверждения истек слишком большой период времени, и подтверждает получение последнего поступившего пакета.

## Принимающий поток

Этот поток начинается с инициализации нескольких переменных.  $Rcvd$  — это массив булевых флагов, обозначенных серийными номерами. По факту получения пакетов им присваиваются в массиве  $rcvd$  значения `true`. По факту обработки пакета серийный номер, равный значению  $w$ , обозначается как `false`.  $E$  — это серийный номер пакета, поступление которого ожидается и который будет обработан следующим. Значение  $E$  всегда представлено как  $A+1$  на модуль  $N$ . Остальным значениям массива  $rcvd$  в процессе инициализации присваиваются значения `true`. Это значит, что эти серийные номера находятся за пределами разрешенного окна. В случае получения пакетов с такими серийными номерами они рассматривались бы как резервные копии и были бы удалены.

Принимающий поток ожидает поступления фрейма. Фрейм может представлять собой пакет или простое подтверждение. В любом случае такой фрейм содержит подтверждение о последнем успешно полученном пакете. Значение  $SendingLimit$  обновляется, и инициируется отправной поток. Обратите внимание, что значение  $SendingLimit$  устанавливается на значение  $w$ , следующее сразу после последнего подтвержденного фрейма. Таким образом, отправителю разрешается использовать только половину пространства серийного номера, начиная с последнего подтвержденного пакета, при этом отправитель и получатель

оговаривают, какая половина пространства серийного номера считается в текущий момент действительной.

Если фрейм содержит пакет, мы получаем серийный номер этого пакета и проверяем массив `rcvd` с целью выяснить, был ли уже получен пакет с таким же серийным номером. Если это так, поступивший пакет уничтожается как резервный. В противном случае массив `rcvd` обновляется и в него вносятся данные о получении пакета, при этом полученный пакет сохраняется в массиве `R`.

Несмотря на то, что отправка пакетов происходит в порядке серийных номеров, их получение может происходить неупорядоченно. Этот факт объясняется уже тем, что пакеты могут теряться и передаваться повторно. Таким образом, даже если мы только что получили пакет с номером `PS`, это может быть не тот пакет, который мы на самом деле ожидали (`E`). Если это на самом деле так, мы просто ожидаем получения пакета `E`. Однако, если выполняется условие `PS == E`, для обработки пакета порождается отдельный поток. При этом разрешенное окно серийных номеров перемещается путем установки ячейки `E+w` массива `rcvd` равной `false`. В конечном счете переменная `A` устанавливается на значение `E`. Это означает, что `E` – это последний успешно полученный серийный номер, после чего значение `E` приращивается.

## Временной поток

Таймер просто ожидает события, сигнализирующего об истечении установленного на нем времени. Существует два вида событий, которые могут произойти. Событие `Timeout(K)` указывает на то, что пакет был отослан отправным потоком, но на него не поступило подтверждение. Таким образом, поток таймера повторно передает пакет `x` и переустанавливает значение на его таймере.

Событие `AckTime` генерируется периодическим таймером с повторным запуском. Этот таймер отправляет событие `AckTime` каждые `X` миллисекунд. Но таймер можно повторно запустить так, чтобы время его работы превышало `X` миллисекунд. Поток отсылки повторно запускает этот таймер каждый раз при отсылке пакета. Такое решение в данном случае уместно, поскольку каждый пакет несет с собой ответное подтверждение. Если в течении `X` мс пакет не был отослан, происходит событие `AckTime`, и поток таймера отсылает фрейм подтверждения.

## Вот это да!

Вам, наверное, было нелегко разобраться в рассматриваемом вопросе. Диаграмма сама по себе может отображать намерение ее составителя, но дополнительные примечания, несомненно, существенно помогут в процессе ее интерпретации.

Каким образом можно определить, что диаграмма корректна? Никак! Я ничуть не удивлюсь, если это озадачит некоторых читателей. Диаграммы, как правило, можно протестировать точно так же, как тестируется код. Поэтому нам придется

подождать до тех пор, пока с помощью кода не будет установлена корректность данного алгоритма.

Эти две проблемы ставят под вопрос полезность таких диаграмм. Они могут эффективно применяться в целях обучения, но такую диаграмму нельзя считать в достаточной степени точной, чтобы рассматривать ее в качестве единственной спецификации проекта. Необходимы также текст, код и тесты.

## Диаграммы действий

Диаграммы действий — это “гибрид” диаграмм перехода состояний, блок-схем и сетей Петри. Их применение особо эффективно при описании основанных на событиях алгоритмов с несколькими потоками.

Диаграммы действий принадлежат к классу диаграмм состояний. Такая диаграмма — это стандартное графическое изображение состояний, связанных между собой переходами. Отличительной чертой диаграммы действий являются специальные виды состояний и переходов.

### Состояние действия



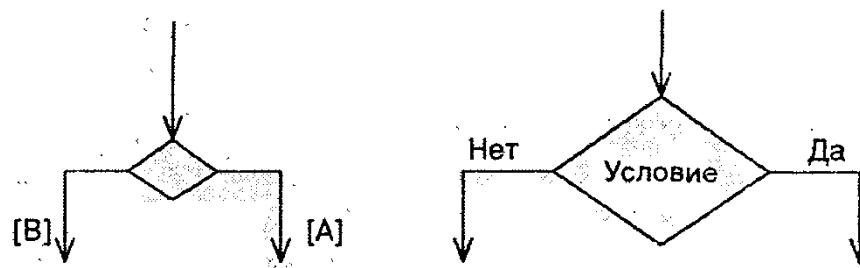
*Состояние действия* изображается в виде прямоугольника с плоской верхней и нижней гранью и закругленными боковыми гранями. Эта пиктограмма отличается от обычной пиктограммы состояния тем, что такая пиктограмма имеет острые углы, в то время как углы пиктограммы состояния закруглены. Внутри состояния действия находится одно или несколько процедурных утверждений, которые отображают действия входа в это состояние. (Такая структура напоминает структуру скна процесса в блок-схеме.)

При входе в состояние действия немедленно выполняются соответствующие действия входа в это состояние. После выполнения этих действий выполняется выход из состояния действия. Для исходящего перехода не нужен ярлык события, так как “событие” — это, по сути, выполнение всех действий входа. Однако, выход из состояния может выполняться с помощью нескольких исходящих переходов, каждому из которых присуще взаимно исключающее условие режима защиты. Совокупность условий режима защиты всегда должна равняться значению `true` (т.е. в состоянии действия невозможно “застрять”).

### Решения

Тот факт, что состоянию действия присуще большое количество исходящих защищенных переходов, означает, что выбор данного состояния может рассмат-

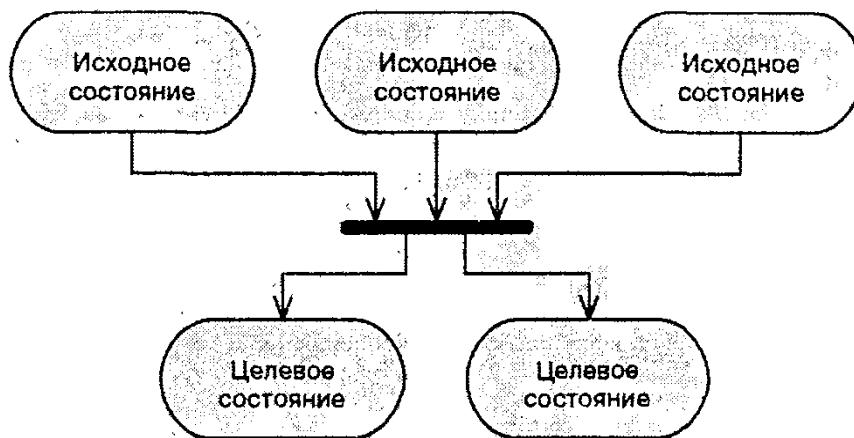
риваться как этап принятия решения. Но зачастую имеет смысл особым образом отмечать решения, которые обозначаются пиктограммой в виде ромба.



Переход входит в ромб, а защищенные переходы покидают его. И в этом случае совокупности булевых значений исходящих средств защиты необходимо присвоить значение true.

На рис. Б.7 представлена разновидность ромба, который больше походит на ромб, используемый в блок-схемах. Булево значение указывается в середине ромба, и два исходящих перехода помечаются ярлыками "Yes" и "No".

### Сложные переходы



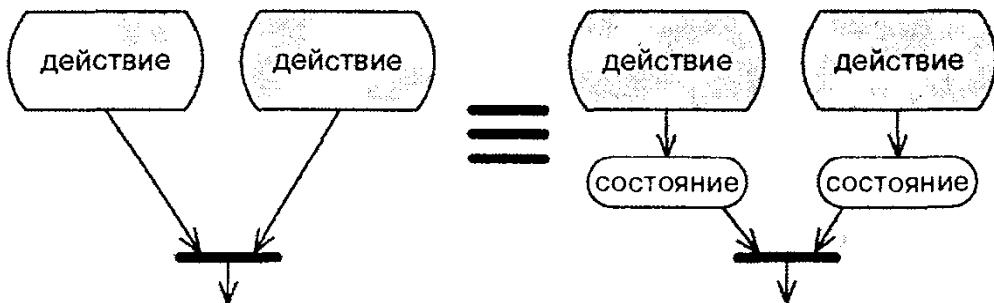
Сложные переходы демонстрируют разделение и соединение нескольких потоков управления. Такие переходы обозначаются темным прямоугольником, который называется прямоугольником *асинхронизации*. Состояния соединяются с прямоугольником *асинхронизации* стрелками. Состояния, ведущие к прямоугольнику, называются прямоугольниками *источника*, а состояния, ведущие в направлении от прямоугольника, называются состояниями *пункта назначения*.

Вся совокупность стрелок, ведущих к прямоугольнику и от него, образует *единий переход*. Стрелки не имеют ни ярлыков, указывающих на события, ни ярлыков, указывающих на условия режима защиты. Переход выполняется тогда, когда все состояния источника заняты (т.е. когда три независимых потока находятся в соответствующих состояниях). Более того, состояния источника должны быть истинными состояниями, а не состояниями действия (т.е. они должны поддерживать режим ожидания).

После начала выполнения перехода происходит выход из всех состояний источника, а также вход во все состояния пункта назначения. Если количество состояний пункта назначения превышает количество состояний источника, это свидетельствует о создании новых потоков управления. Если же преобладает число состояний источника, это указывает на присоединение нескольких потоков.

Каждый вход в состояние источника фиксируется счетчиком. Каждый раз при выполнении сложного перехода соответственно уменьшаются показания счетчиков в соответствующих им состояниях источника. Состояние источника считается занятым до тех пор, пока показание счетчика не достигнет нуля.

Для удобства обозначений истинный переход или переход действия может использоваться в качестве источника в прямоугольнике асинхронизации (см. рис. Б.7.) В таком случае предполагается, что фактический переход завершается в истинном состоянии без имени, которое является состоянием источника такого прямоугольника.



## Структура ПО протокола связи

В трех потоках управления используются одни и те же переменные. Очевидно, что функции, вызываемые этими потоками, должны представлять собой методы одного и того же класса. Но в большинстве потоковых систем, существующих на сегодняшний день, поток приравнивается к объекту. Это означает, что каждый поток имеет объект, который им управляет. В UML-записи такие объекты называются *активными объектами*. Поэтому предполагается, что класс, которому принадлежат методы протокола, также должен создавать активные объекты, управляющие потоками.

Применение потока таймера уместно в других частях системы (а не в протоколе), поэтому создание такого потока, очевидно, должно происходить в другой части системы, в то время как отправной и приемный потоки создаются объектом протокола.

На рис. Б.8 показана диаграмма объектов, которая описывает состояние системы сразу после инициализации объекта `CommunicationsProtocol`. Объект `CommunicationsProtocol` создал два объекта `Thread` и “несет ответственность” за “продолжительность жизни” этих объектов. В объектах `Thread` задей-

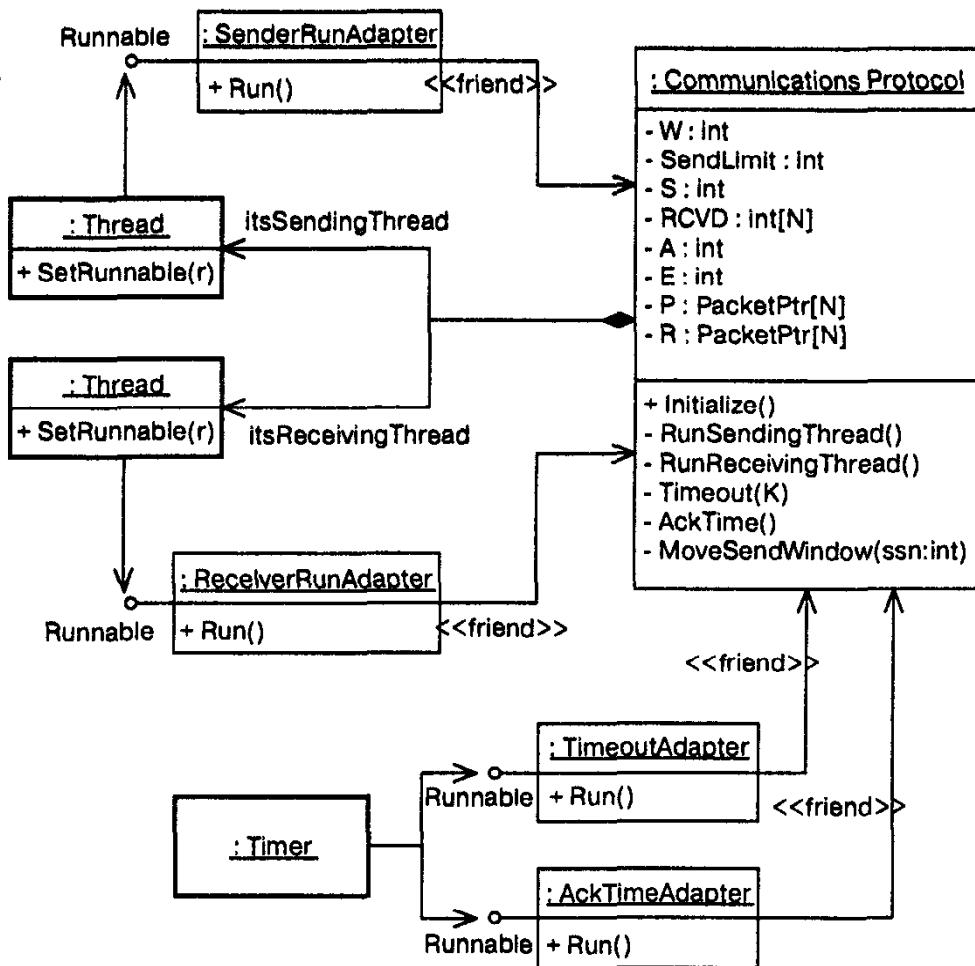


Рис. Б.8. Диаграмма объектов: после инициализации объекта протокола

ствован шаблон `Command`<sup>6</sup>, который обеспечивает начало выполнения вновь созданного потока управления. Каждый объект `Thread` содержит экземпляр объекта, который соответствует интерфейсу `Runnable`. Для привязки объектов `Thread` к соответствующим методам объекта `CommunicationsProtocol` используется шаблон `Adapter`<sup>7</sup>.

Подобный процесс можно наблюдать между объектами `Timer` и `CommunicationsProtocol`. Но в этом случае “продолжительность жизни” объекта `Timer` не контролируется объектом `CommunicationsProtocol`.

Так называемые “дружеские” отношения между объектами `Thread` и объектом `CommunicationsProtocol` существуют с той целью, чтобы методы объекта `CommunicationsProtocol` можно было вызвать только с помощью адаптера. Таким образом исключается возможность вызова этих методов другими способами, кроме адаптеров<sup>8</sup>.

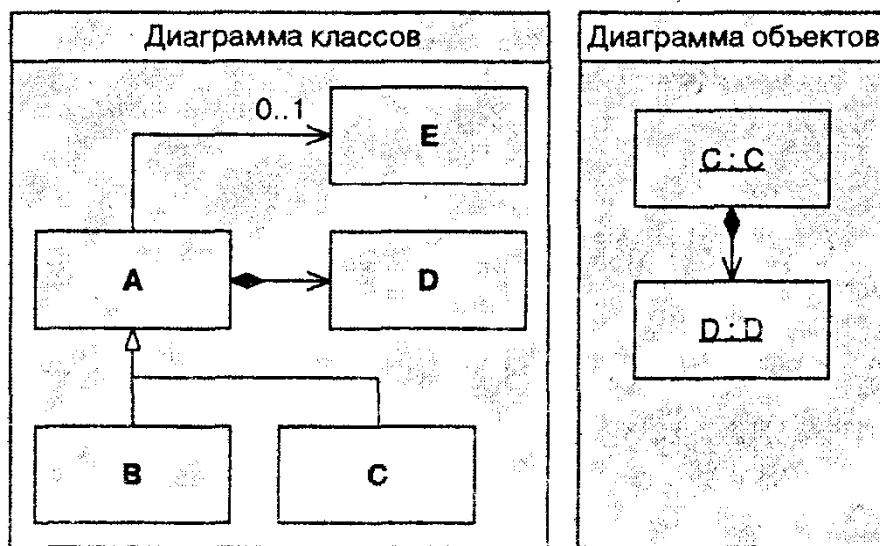
<sup>6</sup>Глава 13.

<sup>7</sup>Глава 25.

<sup>8</sup>Существует множество других способов для выполнения этого. Мы использовали внутренние классы Java.

## Диаграммы объектов

Диаграммы объектов описывают статические взаимосвязи, которые существуют между совокупностью объектов в определенный момент времени. Они отличаются от классовых диаграмм двумя аспектами. Во-первых, вместо *классов* они описывают *объекты*, а вместо *взаимосвязей* между классами — *каналы связи* между объектами. Во-вторых, классовые диаграммы отображают взаимосвязи и взаимозависимости в коде источника, в то время как диаграммы объектов отображают только те взаимосвязи и взаимозависимости рабочего цикла, которые существуют для экземпляра, определенного диаграммой объектов. Следовательно, диаграмма объектов отображает объекты и каналы связи между ними в момент пребывания системы в определенном состоянии.



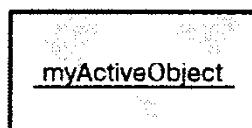
Предыдущая диаграмма представляет собой диаграмму класса и диаграмму объектов, которые отображают одно возможное состояние объектов и каналов связи. Такие объекты и каналы связи являются производными классов и взаимосвязей в классовой диаграмме. Обратите внимание, что объекты представлены в таком же виде, как и в последовательной диаграмме. Они представляют собой прямоугольники, каждому из которых присвоено двухкомпонентное имя с подчеркиванием. Также следует отметить, что взаимосвязь в диаграмме объектов отображена точно так же, как и в классовой диаграмме.

Взаимосвязь между двумя объектами называется каналом связи (link). Через канал связи происходит перемещение потока сообщений в доступном направлении. В данном случае сообщения могут перемещаться от объекта *theC* к объекту *theD*. Такой канал связи существует, поскольку между классом *A* и классом *D* существует взаимосвязь композиции, а также потому что класс *C* является производным класса *A*. Следовательно, экземпляр класса *C* может иметь каналы связи, которые являются производными базовых классов этого экземпляра.

Также обратите внимание, что взаимосвязь между классом А и классом Е не отображена на диаграмме объектов. Это объясняется тем, что диаграмма объектов описывает определенное состояние системы, на протяжении которого объекты С не ассоциируются с объектами Е.

## Активные объекты

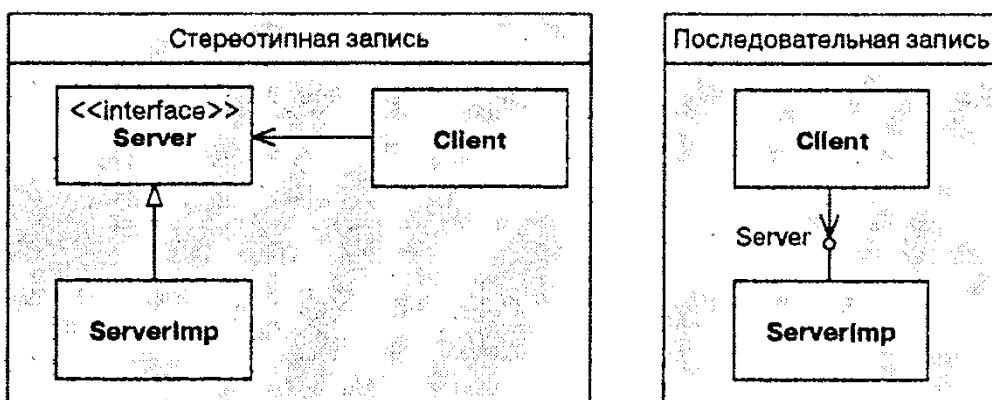
Активные объекты — это объекты, которые несут ответственность за поток управления в целом. Протекание потока управления внутри методов активного объекта не является необходимым условием. Более того, активный объект, как правило, вызывает другие объекты. Активный объект — это просто объект, в котором берет свое начало поток управления. Это также объект, который обеспечивает интерфейсы по управлению потоками, такие как `Terminate`, `Suspend` и `ChangePriority`.



Схематически активные объекты изображаются как обычные объекты, но выделяются по контуру жирной линией. Если активный объект включает в себя другие объекты, выполнение которых происходит в рамках его потока управления, такие объекты можно изобразить в середине активного объекта.

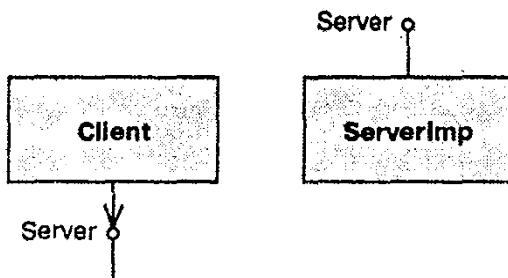
## Интерфейсы

Интерфейсы могут изображаться в виде классов со стереотипом «интерфейс» или же их можно обозначить специальной пиктограммой в виде “леденца” (последовательная форма).



Обе диаграммы в этом окне интерпретируются одинаково. Экземпляры класса `Client` используют интерфейс `Server`. Класс `ServerImp` реализует интерфейс `Server`.

Любую из двух взаимосвязей, ведущих к “леденцу”, можно опустить, как показано на представленной ниже диаграмме.



## Процесс инициализации

Отдельные этапы обработки, используемые для инициализации объекта CommunicationsProtocol, изображены на рис. Б.9. На этом рисунке представлена диаграмма сотрудничества.

Процесс инициализации начинается сообщением под номером 1. Объект CommunicationsProtocol получает сообщение инициализации от какого-либо неизвестного источника. В ответ он отсылает сообщения 1.1 и 1.2 и создает объект SendingThread и ассоциированный с ним объект SenderRunAdapter. Затем в сообщениях 1.3 и 1.4 объект CommunicationsProtocol выполняет привязку адаптера с потоком и запускает этот поток.

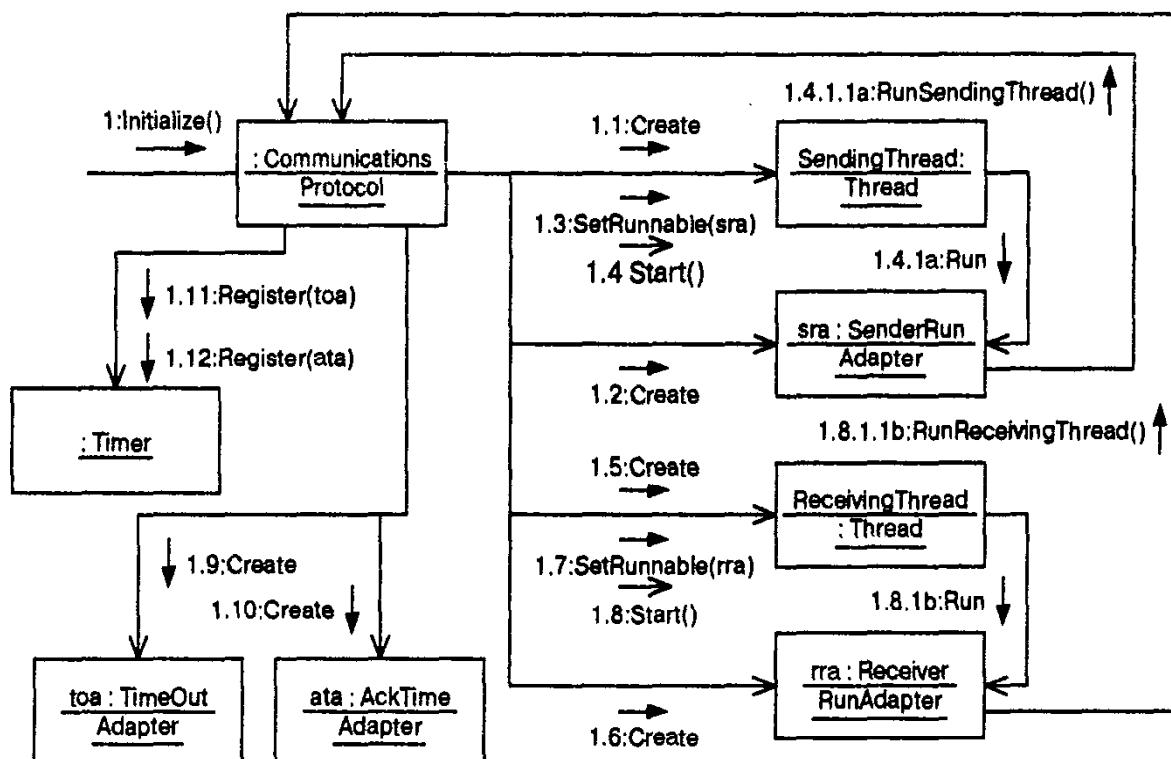
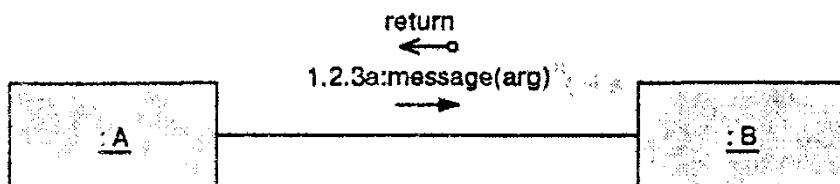


Рис. Б.9. Диаграмма сотрудничества: инициализация объекта CommunicationsProtocol

Обратите внимание, что сообщение 1.4 является асинхронным, поэтому процесс инициализации продолжается путем отсылки сообщений 1.5–1.8, которые выполняют эту же процедуру для создания потока `ReceivingThread`. Тем временем сообщение 1.4.1а запускает отдельный поток управления, который вызывает метод `Run` в объекте `SenderRunAdapter`. В результате адаптер отсылает сообщение 1.4.1.1а: `RunSendingThread` объекту `CommunicationsProtocol`. Это дает начало обработке отправного потока. Похожая цепочка событий запускает приемный поток. И наконец, сообщения 1.9 и 1.12 создают адаптеры таймера и регистрируют их в таймере.

## Диаграммы сотрудничества

Диаграммы сотрудничества аналогичны диаграммам объектов за исключением того, что они демонстрируют дальнейшее развертывание состояния системы во времени. Сообщения, которые отсылаются между объектами, изображаются вместе с относящимися к ним аргументами и возвращаемыми значениями. Каждое сообщение отмечено порядковым номером относительно других сообщений.



Сообщения изображаются маленькими стрелками, которые расположены возле канала связи между двумя объектами. Каждая отдельная стрелка указывает на объект, который получает сообщения. Сообщение отмечено именем и порядковым номером сообщения:

Порядковый номер отделен от имени сообщения двоеточием. За именем сообщения следуют круглые скобки, в которые заключен список разделенных запятыми аргументов сообщения. Порядковый номер представляет собой перечень разделенных точками чисел, после которого следует необязательный идентификатор потока.

Числа, входящие в состав порядкового номера, указывают на расположение сообщения в порядке очередности и его место в иерархии вызовов. Сообщение под номером 1 — это первое сообщение, предназначенное для отправки.

Если процедура, вызванная сообщением 1, вызывает два других сообщения, этим сообщениям будут присвоены соответственно номера 1.1 и 1.2. После их возвращения и выполнения сообщения под номером 1 следующему сообщению присваивается номер 2. Такая система разделительных точек позволяет полностью описать порядок и формирование гнезда сообщений.

Идентификатор потока — это имя потока, в котором выполняется сообщение. Если идентификатор потока опущен, это означает, что сообщение выполняется в потоке без имени. Если сообщение 1.2 порождает новый поток с именем "t", первое сообщение этого нового потока будет под номером 1.2.1t.

Возвращаемые значения и аргументы можно изобразить, используя символ знака "данные" (маленькая стрелка с окружностью на конце). Или же возвращаемые значения можно изобразить, используя в имени сообщения синтаксис привязки, а именно:

1.2.3 : c:=message(a,b)

В этом случае возвращаемое значение сообщения "message" будет содержаться в переменной с именем "c".

→ Сообщение, в котором используется заполненная размерная стрелка, как изображено слева, представляет синхронный вызов функции. Сообщение не возвратится до тех пор, пока не возвратятся все другие синхронные сообщения, вызванные из относящейся к нему процедуры. Данное сообщение принадлежит к классу сообщений для языков C++, Smalltalk, Eiffel или Java и т.п.

→ Размерная стрелка, изображенная слева, представляет асинхронное сообщение. Благодаря этому сообщению новый поток управления приступает к выполнению вызванного метода, после чего тот час же следует возврат сообщения. Таким образом, сообщение возвращается до того, как метод будет выполнен. Сообщения, отсылаемые методом, должны содержать в себе идентификатор потока, поскольку они выполняются в другом, отличном от вызванного, потоке.

## Состояние гонок в рамках протокола

Протоколу, представленному на рис. Б.7, присущ такой интересный феномен, как *состояние гонок*. О состоянии гонки говорят в том случае, когда невозможно предсказать заранее, в каком порядке будут происходить два отдельных события, но состояние системы является чувствительным к такому порядку. Состояние системы в конечном итоге зависит от того, какое событие выигрывает гонку.

Программист пытается обеспечить надлежащее поведение системы независимо от расстановки событий. Однако состояния гонки тяжело определить. Необнаруженные состояния гонки могут привести к перемежающимся ошибкам, которые трудно поддаются диагностике.

В качестве примера состояния гонки рассмотрим, что происходит при отправке пакета отправным потоком (рис. Б.10.) Этот вид диаграммы называется диаграммой последовательности сообщений. Локальный отправитель отправляет пакет S и запускает для него время ожидания. Удаленный получатель получает этот пакет и уведомляет удаленного отправителя об успешном получении пакета S. Удаленный отправитель либо посыпает явно выраженное подтверждение о получении

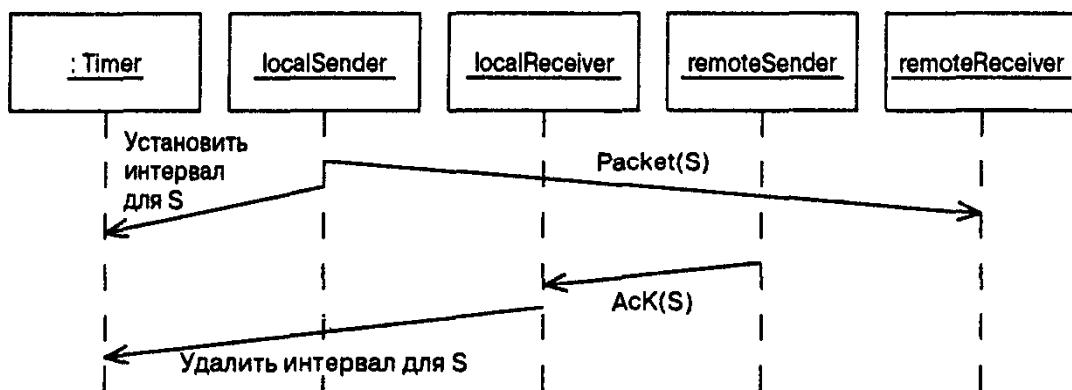


Рис. Б.10. Подтверждение о получении пакета: стандартная ситуация

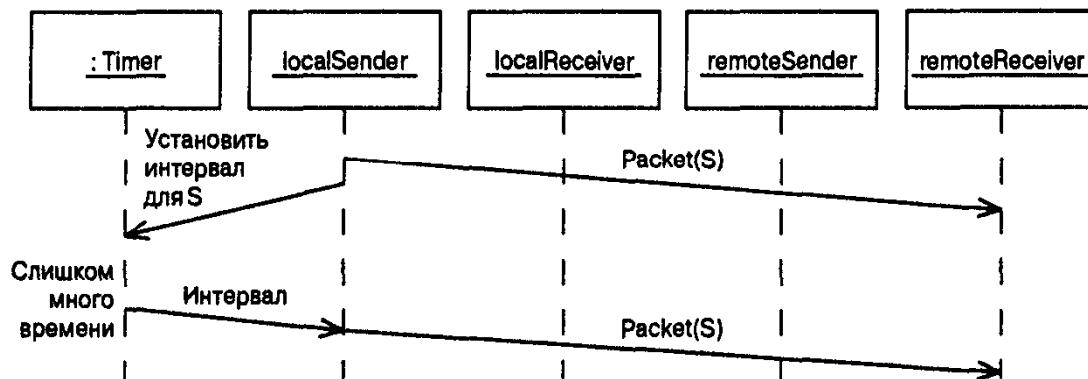


Рис. Б.11. Потеря подтверждения: повторная передача

пакета, либо совмещает отправку такого подтверждения с отправкой следующего пакета. Локальный получатель получает подтверждение и аннулирует время ожидания.

Иногда подтверждение не поступает. В таком случае время ожидания истекает и пакет передается повторно. Этот случай продемонстрирован на рис. Б.11.

Между этими двумя крайними вариантами существует состояние гонки. Может случиться, что время таймера истечет как раз в момент отправки подтверждения. Такой сценарий представлен на рис. Б.12. Обратите внимание на пересекающиеся линии. Они представляют гонку. Отправка и получение пакета S прошли успешно. Более того, в ответ на поступивший пакет было отправлено соответствующее подтверждение. Однако подтверждение поступило после истечения времени ожидания. Следовательно, даже несмотря на получение подтверждения пакет будет отправлен повторно.

Логическая схема, представленная на рис. Б.7, является корректным решением такой гонки. Удаленный получатель понимает, что второе поступление пакета S – это резервная копия пакета, и устраняет его.

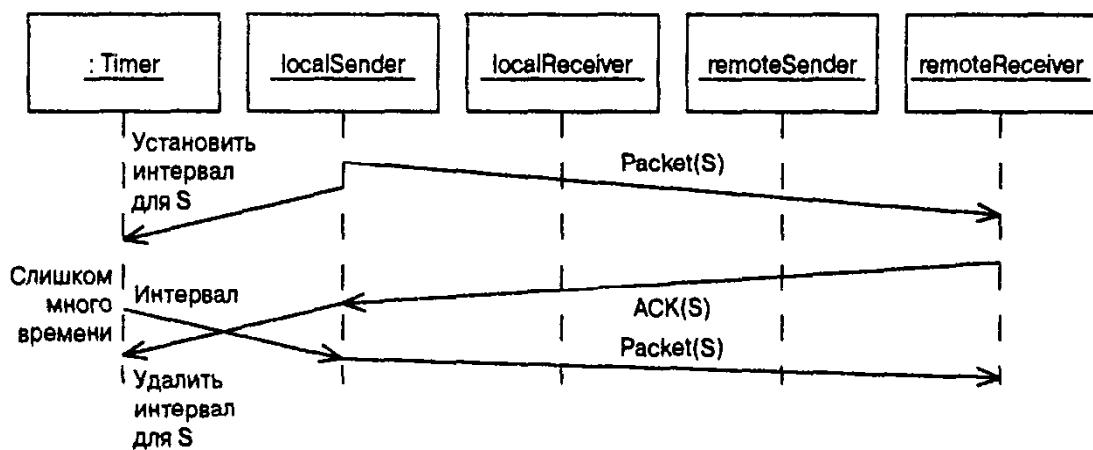


Рис. Б.12. Состояние гонки подтверждения/повторной передачи

## Диаграммы последовательности сообщений

Диаграммы последовательности сообщений — это специальная форма последовательных диаграмм. Первостепенная отличительная черта таких диаграмм заключается в том, что стрелки, обозначающие сообщения, закруглены вниз. Это указывает на то, что между отправкой и получением сообщение может пройти определенный промежуток времени. В диаграмме последовательности событий могут присутствовать все остальные части последовательной диаграммы, включая активации и порядковые номера.

Цель использования диаграмм последовательности сообщений состоит прежде всего в том, чтобы обнаруживать и документировать состояния гонки. Эти диаграммы очень эффективны в случаях, когда требуется показать временные соотношения между определенными событиями, а также продемонстрировать, как порядок выполнения событий может по-разному определяться в двух отдельных процессах.

Рассмотрим рис. Б.12. Объект Timer “полагает”, что событие Timeout предшествует событию Kill Timeout. Однако объект localSender воспринимает эти события в обратном порядке.

Такое отличие в восприятии порядка событий может привести к возникновению дефектов логического характера, которые являются чрезмерно чувствительными к распределению интервалов времени и которые с большим трудом поддаются репродукции и диагностике. Диаграммы последовательности событий представляют собой очень эффективный инструментарий для обнаружения таких ситуаций до того, как они повлекут за собой разрушение системы.

## Резюме

В этом приложении представлено большинство динамических методов моделирования UML-языка. Мы рассмотрели машины состояний, диаграммы активности, диаграммы сотрудничества, а также диаграммы последовательности. Было продемонстрировано, каким образом эти диаграммы справляются с проблемами одного и нескольких потоков управления.

## Литература

1. Gamma и др. *Design Patterns*. Reading, MA: Addison-Wesley, 1995.

# B

## Сатира на деятельность двух компаний

### Крах проекта, фирма Rufus, Inc.

Вас зовут Боб. Сегодня 3-е января 2001 года. Вы до сих пор не можете справиться с головной болью после бурной встречи нового года и нового тысячелетия. Вы, несколько менеджеров и группа ваших коллег сидите в конференц-зале. Вы руководитель команды разработчиков проекта. На совещании также присутствует ваш шеф со всеми руководителями отделов.

Начальник вашего начальника, назовем его ВВ, говорит: “Нам необходимо разработать новый проект”. Его биополе настолько велико, что оно задевает потолок. Биополе вашего босса только начинает расти, но он нетерпеливо ожидает того дня, когда сможет оставлять пятна бриолина на потолке. ВВ говорит о новом рынке сбыта и продукте, который необходимо разработать для внедрения на этом рынке.

“Мы должны завершить и запустить в работу этот новый проект к четвертому кварталу, до 1-го октября,” — требует ВВ. — “С сегодняшнего дня для нас не существует ничего более важного, кроме этого проекта. В связи с этим все работы над текущим проектом прекращаются”.

Собравшиеся в зале реагируют ошеломляющей тишиной. Перечеркнута вся кропотливая многомесячная работа. Постепенно по комнате шепотом начинают разноситься недовольные реплики.

Аура ВВ вспыхивает зловещим зеленым заревом, когда он вонзает свой взгляд в каждого присутствующего в комнате. Этот хитрый взгляд превращает всех присутствующих в один сплошной кусок дрожащей протоплазмы. Этим красноречивым взглядом ВВ дает всем понять, что не потерпит каких-либо возражений по этому вопросу.

Когда в зале снова воцаряется тишина, ВВ говорит: “Мы должны начать работу незамедлительно. Сколько времени займет анализ?”

Вы поднимаете руку. Ваш шеф пытается вас остановить, однако вы не замечаете его испепеляющего взгляда и игнорируете его попытки остановить вас.

“Сэр, мы не можем сказать, сколько времени займет анализ, пока не получим требования”.

“Документ с требованиями будет готов не раньше чем через три или четыре недели,” — говорит ВВ, и его флюиды вибрируют от раздражения. — “Представьте, что у вас уже есть требования. Сколько времени вам потребуется для анализа?”

Все сидящие в зале затаили дыхание. Коллеги смотрят друг на друга в надежде, что у кого-то созрела какая-то мысль.

“Если анализ будет завершен после 1-го апреля, то это чревато проблемами. Сможете ли вы завершить анализ до этого времени?”

Видно, что ваш шеф призывает все свое мужество, чтобы воскликнуть: “Мы сделаем это, сэр!”

“Ладно”, — улыбается в ответ ВВ. — “Ну, а сколько времени займет разработка проекта?”

“Сэр”, — снова начинаете вы. Ваш шеф бледнеет на глазах. “Не выполнив анализ, трудно сказать, сколько времени уйдет на проект”.

Выражение лица ВВ становится более чем суровым. “А вы ПРЕДСТАВЬТЕ, что у вас уже есть результаты анализа!” — говорит он, сверля вас своими пустыми глазами-бусинками. “Сколько времени в таком случае у вас займет разработка проекта?”

Да, две таблетки тайленола здесь уже явно не помогут. Ваш шеф в отчаянной попытке сохранить свой новый рост бормочет: “Я думаю, сэр, если до завершения проекта остается только шесть месяцев, то разработку не следует затягивать более чем на три месяца”.

“Рад, что вы согласны со мной, Смитерс!” — сияет ВВ. Ваш шеф облегченно вздыхает. Он уверен, что его аура теперь в безопасности. Спустя некоторое время он начинает что-то напевать себе под нос.

ВВ продолжает: “Итак, анализ будет завершен к 1-му апреля, разработка проекта завершится к 1-му июля, поэтому у вас есть три месяца для осуществления проекта. Это совещание продемонстрировало нам, насколько важно соблюдать единодушие и согласованность действий в нашей работе. Теперь идите и начнайте работать. Надеюсь на следующей неделе увидеть на своем столе ТQM-планы и QIT-задания. И еще, не забудьте, что мы будем собирать многофункциональные групповые совещания и сдавать отчеты для проведения аудита качества в следующем месяце”.

“Тайленол здесь уже не поможет”, — думаете вы, возвращаясь в свой маленький кабинет. — “Бурбон — именно то, что мне сейчас необходимо”.

Заметно возбужденный, ваш шеф подходит к вам со словами: “Черт побери, это “важное” совещание. Мне кажется, что весь мир должен перевернуться, лишь

бы осуществить этот проект". Вы одобрительно киваете головой, слишком мерзко чувствуя себя для проявления каких-либо других эмоций.

"О, чуть не забыл", — продолжает ваш шеф, вручая вам какой-то документ из тридцати страниц. — "Не забудь, что на следующей неделе приезжают ребята из института разработки ПО для проведения оценочных работ. Это руководство по проведению оценочных работ. Тебе нужно его прочитать, изучить и разорвать. Он подскажет тебе ответ на любой вопрос, задаваемый аудиторами из института. В этом справочнике также сказано о том, куда тебе их можно пускать, а в куда — нет. В июне мы станем организацией, относящейся к уровню 3 модели СММ!"

\* \* \*

Вы и ваши коллеги начинаете работать над анализом нового проекта. Работа нелегкая, поскольку у вас нет требований. Однако, руководствуясь десятиминутной презентацией, проведенной ВВ этим злополучным утром, вы вырабатываете некоторые соображения о том, что должен подразумевать собой новый продукт.

Корпоративный процесс требует начинать работу с создания документа, описывающего примеры практического применения. Вы со своей командой начинаете перечислять все эти примеры и рисовать овальные и штриховые диаграммы.

Внутри команды разгораются философские споры. Несогласие возникает по вопросу того, должны те или иные примеры практического использования быть связаны с отношениями "расширяет" или "включает". Создаются конкурирующие модели, однако никто не знает, как их оценивать. Неутихающие споры эффективно тормозят рабочий процесс.

Спустя неделю кто-то обнаруживает Web-страницу, [iceberg.com](http://iceberg.com), рекомендующую размещение только "расширений" и "включений" и замену их на "предшествует" и "использует". Документы этого Web-узла, автором которых является Дон Сенгруа (Don Sengroix), описывают метод, известный как анализ по методу Столвортса. Этот анализ утверждается как пошаговый метод преобразования примеров практического использования в диаграммы проекта.

С помощью этой новой схемы создаются более конкурентные модели примеров практического использования; однако и в этот раз у команды нет единого мнения о том, как их оценивать. Пробуксовка продолжается.

Все чаще и чаще совещания по поводу примеров практического использования проходят больше под влиянием эмоций, чем здравого смысла. Если бы все дело было не в требованиях, которых у вас нет, то вы наверняка расстроились бы из-за отсутствия прогресса в вашей работе.

Документ с требованиями приходит только 15-го февраля. А потом еще один — 20-го, третий — 25-го, и так каждую неделю поступает по одному документу. Каждая новая версия требований противоречит предыдущим. Этот факт красноречиво свидетельствует о том, что маркетологи, разрабатывающие требования, все же не могут найти консенсус.

Одновременно членами команды предлагаются несколько новых конкурентных шаблонов документов, касающихся практического применения примеров. Каждый разработчик представляет свой собственный, исключительно творческий способ торможения прогресса. Страсти накаляются.

1-го марта Персивалю Патридженсу (Percival Putrigence), “надзирателю” процесса, удается собрать все конкурентные формы и шаблоны документов, описывающих примеры практического применения, в одно целое. Одна лишь пустая форма занимает 15 страниц! Ему удалось включить все поля, которые отображаются в используемых шаблонах. Кроме того, этот “надзиратель” представил документ на 159 страницах, в которых говорится о заполнении формы документа, описывающего примеры практического применения. В связи с этим все текущие документы должны быть переписаны согласно новому стандарту.

Вы мысленно прикидываете, что теперь вам нужно будет изучить 15 страниц инструкции по заполнению этой новой формы, чтобы ответить на вопрос: “Что должна делать система в случае, если пользователь нажимает кнопку возврата?”

Корпоративный процесс (автором которого является L. E. Ott, написавший знаменитую книгу *Глобальный анализ: прогрессивная диалектика для разработчиков ПО*) предполагает обнаружение всех исходных примеров использования, 87% всех второстепенных и 36,274% всех третичных примеров до завершения анализа и начала фазы проекта. У вас нет ни малейшего представления о том, что такое третичный документ, описывающий примеры практического использования. Поэтому в попытках удовлетворить это требование вы добиваетесь, чтобы ваш документ был пересмотрен коммерческим отделом. Может быть, там знают, что такое третичный документ, описывающий примеры практического использования.

К сожалению, коммерческий отдел слишком занят вопросами продвижения продаж для того, чтобы уделить вам внимание. Действительно, с самого момента начала проекта у вас не было ни единой возможности встретиться с ребятами из коммерческого отдела. Самое лучшее, что они могли бы предложить, — это непрекращающийся поток документов с постоянно изменяющимися и противоречими друг другу требованиями.

В то время, пока одна команда беспрерывно ломала голову над документами, описывающими примеры применения, другая занималась разработкой модели предметной области. Эта команда производит на свет бесконечные вариации UML-документов. Каждую неделю работа переделывается. Члены команды никак не могут решить, какие термины следует использовать в модели — «интерфейсы» или «типы». Огромные несогласия возникают по вопросу соответствующего синтаксиса и применения OCL. Другие члены команды только вернулись из пятидневного обучающего семинара по “катализму” и разрабатывают неимоверно подробные и загадочные диаграммы, непонятные никому, кроме них.

К 27-му марта, за неделю до срока завершения проекта, у вас готова целая куча документов и диаграмм, однако обоснованный анализ проблемы не продвинулся дальше уровня 3-го января.

\* \* \*

Но вдруг происходит чудо.

\* \* \*

В субботу 1-го апреля вы у себя дома по электронной почте получаете копию докладной записки своего шефа на имя ВВ. В ней ясно говорится о том, что команда завершила анализ.

Вы звоните своему шефу и возмущенно спрашиваете: "Как вы могли доложить ВВ, что мы уже сделали этот анализ?"

"А вы уже смотрели на календарь сегодня?" — отвечает он. — "Сегодня же 1-е апреля!"

Иронический подтекст этой даты все же не в состоянии вывести вас из состояния реальности. "Но нам же нужно еще так много подумать об этом, так много проанализировать! Мы ведь даже еще не определились с тем, что лучше использовать, — "расширяется" или "продолжается"!"

"А чем вы можете доказать, что вы не завершили анализ?" — спрашивает раздраженно ваш шеф.

"Ну..."

Однако он не дает вам договорить: "Анализ может длиться вечно, когда-нибудь его нужно остановить. А поскольку сегодня именно та дата, когда по плану он должен быть завершен, мы его и останавливаем. В понедельник я хочу, чтобы вы собрали все имеющиеся материалы по анализу и положили их в итоговую папку. Передайте эту папку Персивалю, чтобы он мог загрузить ее в СМ-систему к полудню понедельника. А потом начинайте работу над проектом".

Поговорив с шефом, вы вдруг понимаете, что неплохо всегда иметь под рукой в нижнем ящике стола бутылочку бурбона.

\* \* \*

В честь вовремя завершенной фазы анализа руководство организовало небольшой банкет. ВВ произнес воодушевляющую речь о согласованности в работе. Ваш шеф, аура которого выросла еще на 3 мм, также поздравил свою команду с замечательной демонстрацией единства и коллективной работы. И наконец, выступил главный управляющий по информации, сообщив, что аудиторская проверка, проведенная институтом техники программного обеспечения, прошла очень хорошо, в связи с чем он благодарит всех за изучение и детальный анализ сданных инструкций по оценке. Теперь, наверняка, 3 уровень достигнут, а в июне коллектив получит премию.

(Скатлбат говорит, что, если институт присвоит нам уровень 3, то руководители уровня ВВ и выше должны получить солидные премии.)

В течение последующих недель вы и ваша команда работаете над проектом системы. Вы, конечно, понимаете, что анализ, по которому предположительно создается проект, несовершенен... нет, скорее, бесполезен..., вернее, более, чем бесполезен. Но когда вы говорите своему шефу, что вам необходимо вернуться и немного доработать слабые звенья анализа, он просто дает понять, что фаза анализа завершена, и что единственная возможная сейчас деятельность – это проект, к которому и отправляет вас.

Поэтому вы и ваша команда как можно усерднее трудитесь над проектом, сомневаясь в правильности анализа требований. Конечно, это не важно, поскольку документ с требованиями обновляется еженедельно, а коммерческий отдел по-прежнему отказывается от встречи с вами.

Работа над проектом превращается в кошмар. Ваш шеф, прочитав недавно книгу *The Finish Line*, неправильно истолковал мысль автора Марка ДеТомазо (Mark DeThomas) о том, что документы по проекту должны записываться детально до кодового уровня.

“Если мы будем работать на уровне деталей, то почему бы вместо этого просто не записывать код?”

“Потому что тогда вы, конечно, не будете заниматься проектом. Напоминаю, что единственная разрешенная деятельность в фазе проекта – это проект!”

“Кроме того, – продолжает он, – мы только что приобрели лицензию для всей компании на продукт Dandelion! Этот инструмент позволяет выполнять проектирование с нуля! Вы должны преобразовать все диаграммы проекта в этот инструмент. Он будет автоматически генерировать код! Благодаря этому инструменту диаграммы проекта будут также синхронизироваться с кодом!”

Ваш шеф вручает вам яркую, обернутую в целлофан коробочку, в которой находится продукт Dandelion. Ничего не чувствующими руками вы берете ее и направляетесь в свой кабинет. Спустя двенадцать часов после восьми аварийных ситуаций с системой, переформатирования диска и прочих прелестей вы наконец-то устанавливаете этот инструмент на своем сервере. Еще одна неделя предположительно уйдет на проведение инструктажа для команды по использованию инструмента Dandelion. Затем, улыбаясь, вы думаете про себя: “Любая неделя хороша, когда меня здесь нет”.

Команда создает одну диаграмму проекта за другой. В программе Dandelion сложно нарисовать эти диаграммы. Нужно правильно заполнить несколько десятков глубоко посаженных диалоговых окон с забавными текстовыми полями и окошками метки. И, кроме того, вырисовывается проблема перемещения классов между пакетами...

Вначале эти диаграммы выводятся из документов, описывающих примеры практического применения. Однако вследствие того, что требования изменяются слишком часто, эти документы очень быстро утрачивают свою значимость.

И снова разгораются споры. На этот раз вокруг проблемы, какие шаблоны проекта лучше использовать, — *Visitor* или *Decorator*. Один из разработчиков отказывается использовать *Visitor* в любой его форме, утверждая, этот шаблон не является правильным объектно-ориентированным структурным компонентом. Другой отказывается использовать множественное наследование, поскольку, по его мнению, это порождение дьявола.

Текущие совещания быстро превращаются в споры о значении объектного ориентирования, сопоставления анализа с проектом или о том, в каких случаях следует использовать агрегирование вместо ассоциации.

В середине фазы проекта коммерческий отдел сообщает о том, что они заново пересмотрели фокус системы. Их новый документ с требованиями полностью переделан. Они выбросили несколько пунктов с основными свойствами и заменили их теми, которые, на их взгляд, будут более соответствовать требованиям заказчика.

Вы объясняете своему шефу, что эти изменения повлекут за собой новый анализ и повторное проектирование большей части системы. На что он возражает: “Фаза анализа завершена. Единственная разрешенная деятельность — проект. Вот идите и работайте над ним”.

Вы предполагаете, что неплохо было бы создать простой прототип, чтобы показать его ребятам из коммерческого отдела и даже некоторым заказчикам. Однако ваш шеф непреклонен: “Фаза анализа завершена. Единственная дозволенная деятельность — проект. Идите и работайте над ним”.

Черт побери! Вы пытаетесь создать определенный документ по проекту, который фактически отражал бы документ с новыми требованиями. Тем не менее, даже после изменения требования не становятся идеальными. На самом деле, ужасные колебания в документах с требованиями только увеличиваются по частоте и амплитуде. А вам все равно надо упорно над этим работать.

15-го июня база данных *Dandelion* выходит из строя. Очевидно, что это повреждение прогрессировало. Обнаруженные в базе данных незначительные ошибки постепенно накапливались и через несколько месяцев превратились в более грубые ошибки. В конечном итоге CASE-инструмент просто перестал работать. Конечно, вяло текущее разрушение характерно для всех систем резервного копирования.

На звонки в службу технической поддержки компании *Dandelion* никто не отвечает в течение нескольких дней. Наконец, вы получаете короткое электронное сообщение о том, что об этой проблеме уже знают, и что выходом в такой ситуации является приобретение новой версии программы (которая, по их словам,

будет готова в следующем квартале), после чего необходимо заново ввести все диаграммы вручную.

\* \* \*

1-го июля происходит еще одно чудо. Проект готов!

Вместо обращения с жалобами и претензиями к шефу, вы загружаете в средний ящик своего стола бутылку водки.

\* \* \*

Компания организовала банкет по случаю вовремя завершенной фазы проекта и перехода на 3 уровень СММ. На этот раз речь ВВ кажется вам настолько "вдохновенной", что вы спешите в туалет, пока она не началась.

На вашем рабочем месте появляются новые баннеры и плакаты, на которых изображены орлы и альпинисты, которые говорят о командной работе и поручительстве. Все эти слоганы воспринимаются лучше после нескольких глотков шотландского виски. Вы вдруг вспомнили, что вам нужно очистить свой карточный шкаф, чтобы поставить туда бутылочку бренди.

Вы и ваша команда приступаете к созданию кода. Однако вдруг обнаруживается, что проект пробуксовывает в нескольких важных местах. На самом деле этот проект вообще не имеет никакого смысла. Вы созываете совещание по проекту в одном из залов заседаний, пытаясь разобраться с некоторыми наиболее насущными проблемами. Однако ваш шеф вылавливает вас на этом совещании и закрывает его со словами: "Фаза проекта завершена. Единственная дозволенная деятельность — это кодирование. Вот идите и работайте над этим".

Код, разработанный компанией Dandelion, действительно ужасен. Оказывается, что вы и ваша команда в конечном счете неправильно использовали ассоциацию и агрегирование. Чтобы исправить эти упущения, необходимо отредактировать генерированный код. Редактирование этого кода — задача весьма затруднительная, поскольку он сопровождается ужасными блоками комментариев со специальным синтаксисом. Этот синтаксис необходим инструменту Dandelion для синхронизации диаграммы с кодом. Если вы случайно измените один из этих блоков комментариев, то диаграммы будут регенерироваться неправильно. Оказывается, что разработка "с нуля" требует значительных затрат сил и энергии.

Чем больше вы пытаетесь создать совместимый с Dandelion код, тем больше ошибок выдает эта программа. Вы бросаете эту затею и начинаете обновлять диаграммы вручную. Секунду спустя вы решаете, что нет вообще никакого смысла в обновлении диаграмм. Кроме того, у вас нет на это времени.

Ваш шеф берет на работу консультанта, который должен заниматься созданием инструментов учета количества строк производимого кода. Он водружаet на стене огромный график с термометром, на вершине которого значится 1 000 000. Каждый день он рисует на нем красную линию, что означает количество добавленных строк кода.

Через три дня после того, как на стене появился термометр, ваш шеф останавливает вас в холле. “Этот график не слишком быстро растет. До 1-го октября мы должны генерировать миллион строк кода”.

“Мы даже не уверены в том, что для проекта необходим миллион строк”, — говорите вы, запинаясь.

“Но, тем не менее, нам нужно генерировать миллион строк до 1-го октября”, — снова повторяет шеф. Его биополе еще больше выросло, а греческий рецепт, по которому он его готовит, создает ауру авторитетности и компетентности. “Вы уверены, что ваши блоки комментариев слишком велики?”

Затем в припадке директорской проницательности он говорит: “Я придумал! Я хочу, чтобы вы внедрили новую политику среди проектировщиков. Ни одна строка кода не должна превышать 20 символов. Любая такая строка должна быть разделена на две или большее количество частей, желательно, на несколько частей. Все уже существующие коды необходимо переделать согласно этому стандарту. И тогда кривая на нашем графике будет быстро подниматься вверх!”

Вы решаете не говорить ему, что такое новшество потребует двух незапланированных человеко-месяцев. Вы решаете вообще ничего не говорить ему. Вам кажется, что внутреннее вливание чистого спирта в этом случае — единственный выход. И вы соответственно начинаете к этому готовиться.

Черт побери! Вы и ваша команда медленно сходите с ума от этого кода. 1-го августа ваш шеф, хмуро глядя на график, висящий на стене, вводит обязательную 50-часовую рабочую неделю.

Черт побери! К 1-му сентября график отображает уже 1,2 миллиона строк, в результате чего ваш шеф просит написать докладную записку о превышении финансирования во время фазы кодирования на 20%. Он вводит обязательные субботы и требует, чтобы проект вернули к запланированному 1 миллиону строк. Вы начинаете кампанию по разделению строк кода.

Страсти накаляются: проектировщики завершают свою работу, а отдел контроля качества засыпает вас тревожными отчетами. Заказчики требуют произвести установку и предоставлять руководство пользователю: продавцы требуют проведение предварительных демонстраций для специальных заказчиков: документ с требованиями все еще не готов; коммерческий отдел жалуется, что продукт не имеет ничего общего с тем, что они предусматривали, а винный магазин уже не принимает вашу кредитную карточку. Нужно что-то делать. 15-го сентября ВВ созывает совещание.

При входе в зал аура ВВ испускает клубы пара. Когда он говорит, басовые нотки его тщательно поставленного голоса выворачивают наизнанку ваш желудок. “Руководитель отдела качества сказал, что этот проект содержит менее половины требуемых свойств. Он также сказал мне, что система все время дает сбои, выдает недостоверные результаты и к тому же ужасно медленно работает. Он также сказал

мне, что не может справиться с непрекращающимися ежедневными выпусками, каждый из которых содержит намного больше ошибок, чем предыдущий!"

Он замолкает на несколько секунд, заметно пытаясь взять себя в руки. "Руководитель отдела качества подсчитал, что при такой скорости разработки мы не сможем до декабря предоставить заказчикам продукт!"

На самом деле, думаете вы, что не раньше марта, хотя вслух об этом ничего не говорите.

"До декабря!" — ВВ так неистово орет, что люди прячут свои головы, как будто от наставленного на них ружья. "Декабрь — и абсолютно никаких возражений. Руководители отделов, завтра на столе у меня должны лежать новые расчеты. Сим до завершения проекта я ввожу 65-часовую рабочую неделю. А еще лучше завершить его к 1-му ноября!"

Слышно, как он, выходя из кабинета, бормочет: "Коллективизм — да уж!"

\* \* \*

Ваш шеф облысел; его аура водружена на стене в кабинете ВВ. Люминесцентный свет, отражающийся от его макушки, на мгновение ослепляет вас.

"У вас есть что-нибудь выпить?" — спрашивает он. Поскольку вы уже прикончили последнюю бутылку Boone's Farm, вы вытаскиваете из книжного шкафа бутылку Thunderbird и наливаете его в кофейную чашку шефа. "Что нужно сделать, чтобы сдать проект к назначенному сроку?" — спрашивает он.

"Нам необходимо "заморозить" требования, проанализировать, разработать их и только затем применять", — говорите вы без всякого энтузиазма.

"К 1-му ноября?" — восклицает ваш шеф скептически. "Ни в коем случае! Возвращайтесь к кодированию этой проклятой программы". Он бушует, почесывая свою рассеянную голову.

Несколько дней спустя вы узнаете о том, что вашего шефа перевели в отдел корпоративных исследований. Текучка кадров происходит с реактивной скоростью. Заказчики, в последнюю минуту узнав о том, что их заказы не могут быть выполнены вовремя, начали их аннулировать. Коммерческий отдел проводит переоценку соответствия данного продукта целям компании и т.д. Договора летают, головы катятся, политики изменяются и вообще, все выглядит довольно ужасно.

Наконец, к марта, спустя столько 65-часовых рабочих недель, выходит весьма сомнительная версия программы. На местах показатель обнаруженных ошибок очень высок, а служба технической поддержки заходит в тупик в попытках справиться с жалобами и требованиями раздраженных заказчиков. Никому не сладко.

В апреле ВВ решает проблему, купив лицензионный продукт, предлагаемый компанией Rupert industries, и перераспределяет его среди заказчиков. Заказчики успокоены, коммерческий отдел остался доволен, а вы уволены.

## Проект Альфа, фирма Rupert Industries

Вас зовут Роберт. Сегодня 3-е января 2001 года. Вы спокойно провели праздник в кругу семьи и посвежевшим приступили к дальнейшей работе. Вы сидите в конференц-зале со своими коллегами по работе. Совещание собрал руководитель отдела.

“У нас возникли некоторые соображения по поводу нового проекта”, — говорит начальник отдела. Назовем его Рас. Это нервный и легковозбудимый британец, энергия которого во много раз превышает энергию ядерного реактора. Он амбициозен, но в то же время понимает ценность командной работы.

Рас говорит о новом рынке сбыта и представляет вас руководителю коммерческого отдела Джою, который отвечает за определение продуктов.

Обращаясь к вам, Джей говорит: “Мы бы хотели как можно быстрее начать распространение нашего продукта. Когда вы и члены вашей команды можете встретиться со мной?”

Вы отвечаете: “Мы завершим текущую фазу разработки продукта в эту пятницу. До этого времени, я думаю, мы сможем уделить вам некоторое время. После этого мы выделим часть людей из команды для работы с вами. Мы немедленно начнем подыскивать им замену и нанимати новых людей для вашей команды”.

“Хорошо”, — говорит Рас. “Но я хочу, чтобы вы поняли, насколько важно для нас представить какой-нибудь продукт на промышленной выставке, которая будет проходить в июле. Если мы не покажем там ничего стоящего внимания, то потеряем все свои возможности”.

“Я понимаю”, — отвечаете вы. “Хотя я до сих пор так и не понял, что вы имеете в виду, но уверен, что до июля сможем выполнить это задание. Я пока еще не могу определенно сказать, что это будет. В любом случае вы и Джей можете полностью контролировать работу разработчиков, поэтому можете быть уверены, что до июля эти самые важные разработки будут завершены, чтобы можно было их представить на выставке”.

Рас одобрительно кивает. Он знает, как это все происходит. Ваша команда всегда держала его в курсе событий и позволяла руководить их разработками. Он абсолютно уверен в том, что ваша команда сначала будет работать над самыми важными разработками и что производимая вами продукция будет самого высокого качества.

“Итак, Роберт”, — говорит Джей на первом совещании, — “как вы смотрите на то, чтобы разделить вашу команду?”

“Мы уже привыкли работать друг с другом”, — отвечаете вы, — “но многие из нас порядком устали, работая над последним проектом, и поэтому были бы рады любому изменению. Итак, над чем же сейчас “варятся” ваши ребята?”.

Джей сияет. “Вы, наверное, слышали, с какими проблемами сейчас сталкиваются наши заказчики...” Далее в течение получаса он посвящает вас в проблемы заказчиков и возможные пути их разрешения.

“Хорошо, секундочку” — отвечаете вы. “Мне нужно немного в этом разобраться” Итак, вы с Джем обсуждаете вопрос о работе системы. Некоторые идеи вашего собеседника звучат не совсем завершенно. Вы предлагаете возможные варианты решения проблемы, некоторые из которых он находит интересными. Обсуждение проблемы продолжается.

В ходе обсуждения каждой из новых тем Джей заполняет карточки, включающие описание пользовательских историй. В каждой карточке представлены определенные свойства, которыми должна обладать новая система. Постепенно на столе перед вами собирается огромное количество этих карточек. По мере обсуждения историй, вы с Джем делаете на них заметки. Эти карточки являются мощными мнемоническими приемами, которые можно использовать для представления сложных и некорректно сформулированных идей.

В конце совещания вы говорите: “OK, теперь я имею общее представление о том, что именно вам нужно. Я должен обсудить эту тему со своей командой. Мне кажется, ребята захотят поэкспериментировать с различными структурами баз данных и форматами представления. На следующей нашей встрече, когда я буду представлять уже групповое мнение, мы сможем начать определение самых важных свойств системы”.

На следующей неделе ваша формирующаяся команда встречается с Джемом. Они раскладывают на столе уже имеющиеся карточки с описанием пользовательских историй и начинают вникать в отдельные детали системы.

Совещание проходит очень динамично. Джей представляет истории в порядке их важности. Каждая из них подробно обсуждается. Разработчикам важно, чтобы истории были как можно более короткими, чтобы можно было их оценить и протестировать. Поэтому они постоянно просят Джая разбивать одну большую историю на несколько небольших. Для Джая важно, чтобы каждая история имела четкую деловую ценность и приоритетность, поэтому, разбивая их, он подтверждает правильность своего выбора.

Карточки с историями постепенно накапливаются на столе. Джей их заполняет, но разработчики делают на них пометки по мере необходимости. Никто даже и не пытается зафиксировать все, о чем говорит Джей. Карточки не предназначены для того, чтобы туда все подряд записывать, они служат только как средство напоминания о разговоре.

Получив более подробную информацию о пользовательских историях, разработчики начинают писать на карточках свои расчеты. Эти расчеты сырье и представляют собой большей частью бюджетные сметы, но благодаря им Джей уже может иметь представление о том, во что обойдется та или иная история.

В конце совещания оказывается, что остается еще много историй, достойных обсуждения. Понятно также, что рассмотрены самые важные истории, работа над которыми займет несколько месяцев. Джей закрывает совещание, забирая с собой карточки и обещая утром выдвинуть свои предложения по первой версии программы.

Следующим утром вы снова созываете совещание. Джей выбирает пять карточек и кладет их на стол.

“Согласно вашим оценкам, в этих карточках представлено около пятидесяти пунктов, над которыми стоит поработать. На последнем этапе предыдущего проекта нам удалось выполнить пятьдесят пунктов за три недели. Если мы выполним эти пять историй за три недели, то сможем предоставить результаты Расу. Он должен по достоинству оценить наш прогресс”.

Джей начинает “напирать”. По его выражению лица можно догадаться о том, что он тоже это осознает. Вы отвечаете: “Джей, это новая команда, работающая над новым проектом. Было бы слишком самонадеянно полагать, что мы будем работать с такой же скоростью, как и предыдущая команда. Тем не менее, я вчера встречался с командой, и мы все пришли к выводу, что фактически нашу начальную скорость можно определить как пятьдесят пунктов каждые три недели. Считайте, что вам повезло”.

“Однако не забывайте”, — продолжаете вы, — “что на данном этапе мы экспериментируем с оценками историй и скоростью работы над ними. У нас будет больше определенности, когда мы спланируем цикл и даже больше, когда начнем применять его”.

Джей смотрит на вас поверх своих очков, как будто хочет сказать: “Кто здесь начальник, в конце концов?” Но, улыбаясь, он лишь произносит: “Ну ладно, не волнуйтесь, я все понял”.

Затем Джей выкладывает на стол еще 15 карточек со словами: “Если мы проработаем все эти карточки до конца марта, то сможем передать систему заказчикам для бета-тестирования. У нас получится неплохая обратная связь”.

Вы отвечаете: “Мы определились с первым этапом, на следующие три этапа у нас тоже есть истории. По завершении этих четырех циклов мы получим первую версию программы”.

“Итак”, — говорит Джей, — “сможете ли вы реально проработать эти пять историй за три недели?”

“Я не уверен, Джей”, — отвечаете вы. “Давайте разобьем их на задачи и посмотрим, что у нас получится”.

Итак, Джей, вы и ваша команда в течение нескольких часов разбиваете каждую из выбранных Джоем историй для первого цикла на небольшие задачи. Разработчики быстро смекнули, что некоторые задачи можно применить ко многим историям, и что другие задачи также содержат много общего, чем и можно будет воспользоваться в работе. Разработчикам неожиданно на ум приходят потенциаль-

ные проекты. Время от времени на некоторых карточках они образуют небольшие дискуссионные узлы и набрасывают UML диаграммы.

Вскоре доска заполняется задачами, которые в завершенном виде будут выполнять пять историй первого цикла. Вы предлагаете команде расписать эти задания.

“Я возьмусь за разработку исходной базы данных”, — говорит Пит. “Я занимался этим при работе над предыдущим проектом, а этот, по-моему, мало чем от него отличается. Я сделаю его оценку за два дня”.

“OK, а я тогда возьму экран регистрации”, — говорит Джо.

“Ладно”, — говорит Элмо, самый юный член команды. “Я никогда не занимался графическим интерфейсом пользователя, но был бы не прочь попробовать себя в этой работе”.

“Ох, уж эта юношеская нетерпеливость”, — мудро говорит Джо, подмигивая вам. “Вы можете мне в этом помочь, Джеди”. Джо он говорит: “Думаю, у меня займет это около трех дней”.

Один за другим разработчики подписывают задачи и оценивают их. Вы достаточно хорошо понимаете, что не будете оспаривать результаты оценок разработчиков. Вы знаете этих парней и доверяете им. Вы уверены, что они выложатся на все сто.

Разработчики понимают, что не могут подписать больше заданий, чем они выполнили за прошлый цикл. Заполнив график цикла, каждый разработчик заканчивает подписывать задания.

В конечном итоге все разработчики завершили подписание задач. Но на доске еще остаются задачи.

“Я предполагал, что такое может случиться”, — говорите вы. “OK, остается еще одно, Джей. В этом цикле у нас очень много работы. От каких историй или заданий мы можем отказаться?”

Джей вздыхает. Он понимает, что это единственный выход в сложившейся ситуации. Внеурочное время в самом начале работы над проектом недопустимо, а те проекты, в работе над которыми это практиковалось, не показывали хороших результатов.

Поэтому Джей начинает удалять менее важные задания. “У нас нет пока особой необходимости в экране регистрации. Мы можем просто запускать систему в состоянии регистрации”.

“Черт!” — выкрикивает Элмо. “Я же так хотел заняться этим!”

“Успокойтесь, Кузнецик”, — говорит Джо. “Кто хочет переселиться в золотой дворец, оказывается у разбитого корыта”.

Эта фраза приводит Елмо в замешательство.

Все присутствующие также в замешательстве.

“Итак...”, — продолжает Джей. “Я думаю, мы также можем избавиться от...”

Таким образом, постепенно список заданий сокращается. Разработчики, потратившие свое задание, подписывают оставшиеся.

Совещание проходит небезболезненно. Несколько раз Джей срывается и демонстрирует нетерпение. Даже в момент наивысшего накала страсти Элмо предлагает “более усердно поработать, чтобы наверстать упущенное время”. Вы только собирались поправить его, когда, к счастью, Джо посмотрел в его глаза, сказав: “Береги честь смолоду...”.

В конечном итоге вы пришли к такому варианту цикла, который удовлетворяет Джая. Но это не то, чего он хотел. На самом деле этот вариант значительно слабее. Но в то же время именно с таким вариантом цикла может справиться команда за следующие несколько недель. И наконец, в этот вариант включены самые важные моменты, которые хотел бы видеть Джей.

“Итак, Джей”, — говорите вы, когда страсти немного улеглись. “Когда мы можем получить тесты приемлемости?”

Джей вздыхает. Это другая сторона медали. Для каждой разрабатываемой истории Джей должен предоставить набор тестов приемлемости, подтверждающих их корректную работу. А команде они нужны задолго до конца цикла, поскольку они, конечно же, будут отражать различия во взглядах Джая и разработчиков на поведение системы.

“Я предоставлю вам сегодня несколько примерных сценариев тестов”, — обещает Джей. “Каждый день я буду их дополнять. Полный набор тестов будет у вас к середине цикла”.

Цикл начинается в понедельник утром суматохой CRC-сеансов. К середине утра все разработчики быстро разбились по парам и приступили к кодированию.

“А теперь, мой юный ученик”, — говорит Джо Элмо, — “вы узнаете тонкости проектирования на основе тестов!”

“Звучит интригующе”, — отвечает Элмо. “Как вы это делаете?”

Джо сияет. Понятно, что он предвидел этот момент. “Ну, паренек, что сейчас делает код?”

“Что?” — переспрашивает Элмо. “Он вообще ничего не делает, его вообще здесь нет”.

“Итак, вспомните нашу задачу. Вы помните, что должен делать код?”

“Конечно”, — говорит Элмо с присущей ему юношеской уверенностью. “Сначала он должен подключиться к базе данных”.

“И что необходимо для подключения к базе данных?”

“Вы как будто говорите предсказания”, — смеется Элмо. “Мне кажется, нам надо извлечь объект базы данных из системного реестра и вызвать метод `Connect()`”.

“Очень проницательный молодой волшебник. Вы все правильно понимаете. Нам действительно нужен объект, с помощью которого можно кэшировать объект базы данных”.

“Есть ли такое слово «кэшировать»?”

“Если я говорю, значит есть! Итак, какого рода тест поможет проверить пригодность реестра базы данных?”

Элмо вздыхает. Он понимает, что ему остается только поддакивать. “Мы должны создать объект базы данных и передать его системному реестру в методе `Store()`. А затем мы должны вытянуть его из системного реестра с помощью метода `Get()` и удостовериться, что это именно тот объект”.

“Неплохо”.

“Здорово!”

“А теперь давайте создадим тестовую функцию, которая подтвердит все сказанное вами”.

“А разве мы не должны сначала создать объект базы данных и объект системного реестра?”

“Тебе еще слишком многому надо научиться, мой нетерпеливый мальчик. Сначала просто создайте тест”.

“Но он даже не будет компилироваться!”

“А вы уверены? А что, если будет?”

“Ну...”

“Просто создайте тест, Элмо. Поверьте мне”.

Итак, Джо, Элмо и все остальные разработчики начинают кодировать свои задания. Комната, в которой они работают парами, наполнена жужжанием голосов. Бормотание сменяется случайным возгласом, когда пара заканчивает задание или сложный тестовый случай.

Разработчики меняют партнеров раз или два раза в день. Каждый разработчик может увидеть, с какими трудностями сталкивается другой, поэтому вся команда вскоре получает определенное представление о коде.

Если пара заканчивает какой-то важный этап работы, будь то целую задачу или просто важную часть задачи, они подключают то, что у них получилось, к остальной части системы. Таким образом, база кода увеличивается день ото дня, а трудности интеграции сводятся к минимуму.

Ежедневно разработчики советуются с Джоем. Они обращаются к нему всякий раз при возникновении вопросов о функционировании системы или интерпретации случая теста приемлемости.

Джей, сдерживая свое слово, стабильно предоставляет команде поток сценариев тестов приемлемости. Команда тщательно изучает их и таким образом получает намного большее представление о том, что, по ожиданиям Джоя, должна выполнять система.

К началу второй недели, Джою нужно показать еще достаточно много функциональных свойств. Джей охотно рассматривает их по мере того, как демонстрация одного тестового случая сменяется другим.

“Действительно неплохо”, — говорит Джей по завершении демонстрации. “Однако, не похоже, что это третья часть заданий. Вы работаете медленнее, чем мы предполагали?”

Вы хмуритесь, ибо ожидали удобного случая, чтобы дать понять это Джою, но теперь он сам затронул этот вопрос.

“Да, к сожалению, мы работаем медленнее, чем ожидали. Оказалось, что новый сервер приложений, который мы используем, очень сложно конфигурируется. Его также необходимо постоянно перезагружать, и нам необходимо перегружать его всякий раз при малейшей попытке изменить его конфигурацию”.

Джей смотрит на вас с подозрением. Его головная боль от совещания, проведенного в прошлый понедельник, еще не совсем утихла. Он говорит: “Как же быть с нашим графиком? Мы не можем переделать его, просто не можем. Шеф этого не перенесет! Он затащит нас всех в сарай и разорвет на части!”

Вы смотрите Джою прямо в глаза. Преподносить новости таким образом не очень-то приятно. Вы решаетесь сказать: “Послушайте, если все так и будет продолжаться, то мы не сможем завершить все дела до следующей пятницы! Сейчас, возможно, нам придется подумать над тем, как можно работать быстрее. Но, честно говоря, я от этого не буду зависеть. Вы лучше подумайте, какие один или два задания можно исключить из цикла, чтобы это никак не отразилось на демонстрации работы для Раса. Случись конец света или всемирный потоп, мы должны провести эту демонстрацию в пятницу, и полагаю, вам не хотелось бы, чтобы мы выбирали задачи для удаления”.

“Ради всего святого!” Едва успев подавить выкрик последнего слова, Джей величаво уходит, покачивая головой.

Уже не в первый раз вы про себя отмечаете: “Никто и не обещал тебе, что управление проектом дается легко”. Вы достаточно уверены, что такая мысль не в последний раз приходит вам в голову.

Фактически дела пошли намного лучше, чем вы предполагали. Команда действительно должна исключить одну задачу из цикла, однако благодаря мудрому выбору Джоя демонстрация для Раса проходит без сучка и задоринки.

Рас явно не впечатлен прогрессом, но в то же время и не разочарован. Он просто говорит: “Довольно неплохо. Но не забывайте, что в июле нам нужно продемонстрировать эту систему на промышленной выставке, а с такой скоростью, похоже, нам это не удастся”.

Джой, отношение которого к проекту значительно изменилось к концу цикла, отвечает Расу: “Рас, эта команда работает достаточно много и неплохо. Я уверен, что в июле нам будет что показать на выставке. Не скажу, что работа будет представлена в завершенном виде, хотя, может быть, частично и будет, но все-таки у нас она будет”.

Каким бы болезненным ни был последний цикл, он подтвердил заданную вначале скорость вашей работы. Следующий цикл проходит намного спокойнее.

Не только потому, что команда делает намного больше, чем в предыдущем цикле, а просто потому, что им не приходится удалять задачи или истории в середине цикла.

К началу четвертого цикла устанавливается естественный ритм работы. Вы, Джей и вся ваша команда достаточно хорошо знаете, чего можете ожидать друг от друга. Команда работает усердно при устойчивом темпе. Вы уверены, что команда может работать в таком темпе год или более.

Количество непредвиденных ситуаций в графике приближается к нулю; тем не менее, количество сюрпризов в предъявляемых требованиях не уменьшается. Джей и Рас часто наблюдают за разрабатываемой системой и вносят рекомендации или изменения в существующие функциональные характеристики. Однако все понимают, что эти изменения занимают время и их необходимо вносить в график.

В марте проходит показательная демонстрация системы совету директоров. Система очень ограничена и еще не в той форме, чтобы быть представленной на промышленной выставке, однако прогресс стабилен, в результате чего совет директоров остается доволен.

Второй выпуск проходит еще более гладко, чем первый. К этому времени команда разработала способ автоматизации сценариев тестов приемлемости, выдаваемых Джеем. Они также провели рефакторинг проекта системы, чтобы было действительно проще добавлять новые свойства и изменять уже существующие.

Второй выпуск готов до конца июня, и его демонстрируют на выставке. В нем, конечно, меньше тех свойств, которых ожидали Рас и Джей, но, тем не менее, эта версия демонстрирует самые важные свойства системы. И несмотря на то, что заказчики все же замечают отсутствие некоторых свойств, они, тем не менее, остаются довольны. Вы, Рас и Джей покидаете выставку с улыбкой удовлетворения на лицах. У вас у всех такое ощущение, как будто ваш проект выиграл.

Много месяцев спустя вас разыскало руководство компании Rufus, Inc. Они работали над подобного рода системой для своих внутренних операций. Компания прекратила разработку этой системы после вымотавшего их мартовского проекта и ведет переговоры о приобретении прав на вашу технологию для своего окружения.

Действительно, жизнь прекрасна!

# Г

## Исходный код — это проект

Я до сих пор помню, каким образом пришел к пониманию некоторых вещей, в результате чего появилась эта статья. Это было летом 1986 года. Я выполнял функции временного консультанта в Центре *China Lake Naval Weapons Center* в Калифорнии. В это же время я воспользовался случаем и принял участие в групповом обсуждении, посвященном языку Ада. В один из моментов кто-то из аудитории задал типичный вопрос: “Являются ли разработчики ПО инженерами?” Я не помню точного ответа, но знаю, что толком на этот вопрос так никто и не ответил. Поэтому я мысленно возвращаюсь в прошлое и думаю том, как бы я ответил на этот вопрос сейчас. Я точно не знаю, как бы ответил, но что-то в процессе обсуждения заставило меня вспомнить статью, которую я прочел в журнале *Datamation* почти за 10 лет до состоявшегося обсуждения. Статья логически обосновывала тот факт, что инженеры должны быть хорошими писателями, а ключевым моментом, вынесенным мною из статьи, было утверждение автора о том, что конечным результатом инженерного процесса является документ. Другими словами, инженеры производят документы, а не вещи. Другие люди берут эти документы и на их основе производят материальные вещи. Таким образом, мой запутавшийся разум задал вопрос: “Из всей документации, которую обычно порождают программные проекты, можно ли найти хоть что-то, что могло бы действительно считаться инженерным документом?” Ответ, который пришел ко мне в голову, был следующим: “Да, существует такой документ, и только один — исходный программный код”.

Взгляд на исходный код как на инженерный документ изменил мое отношение к выбранной когда-то профессии. Изменился способ, с помощью которого я разглядывал окружающий мир. Чем больше думал я об этом, тем больше чувствовал, что эта точка зрения позволяет разрешить множество проблем, с которыми обычно сталкиваются многие программные проекты. Я почувствовал, что тот факт, что большинство людей не понимали этого различия или активно отвергали его, объяснял многое происходящее. Прошло несколько лет, прежде чем представилась возможность обнародовать мою точку зрения. Статья о разработке программных проектов в журнале *C++* побудила меня написать письмо редак-

*тору этого журнала. После обмена письмами редактор Ливлин Сайн (Livleen Singh) согласился опубликовать мою статью на эту тему.*

Джек Ривз, 22 декабря 2001.

## Определение программного проекта

© Джек У. Ривз, 1992 г.

Создается впечатление, что объектно-ориентированные методики, в частности, C++, переживают штурм, разразившийся в мире ПО. Появляются многочисленные статьи и книги, описывающие, как применять новые методы. Рекламная шумиха вокруг ОО-методов сменилась статьями, в которых описываются их преимущества и недостатки. Объектно-ориентированные методики используются не так уж и долго, но их растущая популярность кажется немного необычной. Почему к этим методам возник внезапный интерес? Предлагались самые различные объяснения этого феномена. На самом деле единственной причины, которая бы все объясняла, не существует. Возможно, сочетание факторов окончательно достигло “критической массы”, и сработал закон перехода количества в качество. Тем не менее, создается впечатление, что C++ сам по себе является главным фактором на этой последней фазе революции в сфере ПО. Опять же, возможно, существуют определенные причины, но я склонен предложить несколько иной ответ: C++ стал популярным потому, что он облегчил процесс проектирования и создания ПО.

Если этот комментарий кажется вам немного необычным, то это сделано преднамеренно. В этой статье я хочу исследовать взаимосвязь между программированием и разработкой программного проекта. На протяжении 10 лет у меня создавалось впечатление, что в индустрии разработки ПО “смазывается” тонкая грань между процессом разработки программного проекта и его внутренним содержанием. Как мне кажется, есть глубокий смысл в том, что программированием на C++ в настоящее время заняты лучшие инженеры-программисты. Это связано с тем, что программирование — не просто процесс конструирования ПО, а скорее процесс разработки программных проектов.

Несколько лет назад я принимал участие в семинаре, где возник вопрос о том, является ли разработка ПО инженерной дисциплиной. Я не припоминаю подробностей итогового обсуждения, хотя они навели меня на мысли о том, что индустрия ПО проводит некоторые мнимые параллели с проектированием аппаратных средств, причем некоторые фактические параллели при этом утрачиваются. В результате я пришел к выводу, что процесс разработки ПО трудно отнести к инженерной дисциплине, поскольку далеко не все программисты могут представить себе программный проект во всех деталях. И это убеждение еще больше окрепло со временем.

Конечная цель любой деятельности инженера — некоторый тип документации. После завершения этапа проектирования разработанная документация передается команде разработчиков. В состав этой команды входят специалисты, навыки которых совершенно отличаются от навыков команды разработчиков проекта. Если проектные документы адекватно отражают проект в целом, команда разработчиков может приступить к его созданию. Причем они могут разрабатывать большую часть программного кода без малейшего вмешательства в дальнейшем со стороны разработчиков проекта. После пересмотра жизненного цикла разработки ПО я пришел к выводу о том, что единственной программной документацией, которая действительно удовлетворяет требования инженерного проекта, является листинг исходного кода.

Конечно, существует множество аргументов “за” и “против” этой предпосылки, излагаемые в многочисленных статьях. Вообще говоря, я полагаю, что конечный первоначальный код является реальным программным проектом, а затем уже рассматриваются некоторые следствия этого предположения. Вряд ли можно привести адекватное обоснование этой точки зрения, но таким образом я надеюсь объяснить некоторые из явлений, присущих индустрии разработки ПО, включая феномен популярности C++.

Точка зрения, заключающаяся в рассмотрении кода в качестве программного проекта, влечет за собой одно следствие, которое является столь важным, что “перекрывает” все остальные. Причем его важность еще и подчеркивается тем, что в большинстве организаций, занимающихся разработкой ПО, его просто не замечают. Суть состоит в том, что процесс разработки ПО обходится очень дешево. Причем создается впечатление, что программы обходятся нам практически бесплатно. Если в качестве программного проекта рассматривается исходный код, фактическое построение ПО — это прерогатива компиляторов и компоновщиков. Процесс компиляции и связывания завершенной программной системы часто называют “созданием версии”. Капитальные инвестиции в оборудование, используемое для производства ПО, относительно невелики. Все, что реально требуется, — это компьютер, редактор исходного кода, компилятор и компоновщик. Если среда разработки настроена заранее, сам процесс создания ПО много времени не занимает. Конечно, компиляция программы на C++, объем которой составляет 50000 строк кода, может показаться вечностью, но представьте себе то количество времени, которое отнимает построение аппаратной системы, сложность которой идентична подобной программе на C++!

Другим следствием рассмотрения исходного кода в качестве программного проекта является тот факт, что процесс создания программного проекта является относительно простым. Написание (т.е. разработка) типичного программного модуля, содержащего от 50 до 100 строк кода, — плод усилий за пару дней (для полноты картины следует заметить, что достаточно много времени занимает отладка разработанного кода, но об этом — немного позже). Невольно напрашивается

вопрос: существует ли другая инженерная дисциплина, позволяющая за столь короткое время создавать проекты, сложность которых соизмерима со сложностью программ? Для начала следует сформулировать методы, обеспечивающие измерение и сравнение степени сложности. В любом случае очевидно, что программным проектам присуща тенденция ненормированного роста.

Присущая программным проектам способность быстрой изменяемости, а также относительная дешевизна приводят к тому, что программные проекты проявляют склонность к неумеренному росту и усложнению. На первый взгляд, это положение кажется очевидным, хотя достаточно часто оно игнорируется на практике. Сложность учебных проектов часто измеряется несколькими тысячами строк кода. Существуют программы, проекты которых содержат до 10000 строк кода. На самом деле интерес представляет ПО, размеры которого не слишком велики. Типичные коммерческие программы включают сотни тысяч строк кода. Многие программные проекты измеряются миллионами строк кода. К тому же, программным проектам присуща тенденция к практически постоянному расширению. Исходный проект может включать лишь несколько тысяч строк кода, но в течение "жизни" программного продукта он может многократно переписываться.

Можно привести множество примеров аппаратных проектов, сложность которых приближается к сложности программных проектов, но для начала обратите внимание на два факта, связанные с современным аппаратным обеспечением. Усилия, затраченные на проектирование аппаратного обеспечения, не всегда приводят к устранению всех технических дефектов, как пытаются убедить нас критики программных проектов. Микропроцессоры имеют ошибки в логике, мостам свойственно разрушаться, дамбам — прорываться, авиалайнеры терпят катастрофы... Можно вспомнить о тысячах автомобилей и других товаров широкого потребления, которые были не так давно отзваны с рынка вследствие ошибок, допущенных на этапе разработки проекта. Воплощение сложных проектов, целью которых является разработка аппаратных средств, достаточно дорого. Поэтому действительно сложные технические проекты способны производить лишь немногие компании. В случае с программами подобные ограничения не актуальны. Созданием ПО занимаются буквально сотни различных организаций, причем реально существуют тысячи весьма сложных программных систем. Количество и сложность подобных систем возрастает практически ежедневно. Это свидетельствует о том, что вряд индустрия по разработке программных средств в процессе разрешения своих проблем будет подражать разработчикам аппаратных средств. Если бы существовали аналоги CAD- и CAM-систем для аппаратных средств, позволяющие разрабатывать все более и более сложные проекты, разработка оборудования напоминала бы процесс проектирования и разработки программ.

Процесс разработки ПО — это своего рода упражнение в управлении степенью сложности (т.е. ее преодоление). Сложность проявляется внутри самого программного проекта, в подразделении, которое занимается разработкой программ,

а также в самой индустрии в целом. Проектирование программ весьма напоминает проектирование систем. В процессе создания проекта может использоваться множество технологий, а также различные поддисциплины. Спецификации ПО проявляют тенденцию к изменчивости, причем процесс изменения происходит быстро и часто в то время, когда продолжается процесс разработки ПО. Изменяется состав команд разработчиков ПО, причем зачастую на “полпути” процесса создания программ. Во многих отношениях ПО напоминает сложные социальные или органические системы, но никак не технические средства. В результате процесс разработки программ усложняется, а вероятность возникновения ошибок возрастает. Эти рассуждения вряд ли можно назвать оригинальными, но неоспорим тот факт, что уже 30 лет, прошедшие с момента начала революции в деле разработки ПО, этот процесс сродни некоему вольному творчеству, а не инженерному искусству.

Принято полагать, что в процессе разработки инженерами проекта, независимо от степени сложности последнего, они просто уверены в том, что все будет работать. Также специалисты совершенно уверены в работоспособности проекта в случае применения общепринятых методик разработки. Для того чтобы добиться этого на практике, технические инженеры очень много времени отводят на проверку правильности и уточнения собственных проектов. В качестве примера можно рассмотреть проект моста. Перед тем как действительно создать такой проект, инженеры выполняют структурный анализ — создают компьютерные модели, а также имитируют нагрузку, конструируют масштабные модели, проверяя их на устойчивость, или тестируют с помощью других методов. Короче говоря, разработчики проектов делают все возможное для того, чтобы создать хороший проект еще на предварительном этапе. В этом смысле проектировать новый авиаилайнер еще сложнее. В этом случае придется создавать полномасштабные прототипы, а также выполнять испытательные полеты, гарантирующие качество проектирования и сборки.

Большинство специалистов считает, что программные проекты не проходят через такое количество стадий инжиниринга, как технические проекты. Однако, если рассматривать в качестве проекта исходный код, нетрудно прийти к выводу о том, что разработчики программных проектов выполняют значительный объем работы, направленный на проверку корректности и совершенствование собственных проектов. В данном случае подобные процессы именуются тестированием и отладкой. Большинство почему-то не склонно рассматривать эти действия в качестве фактического “инжиниринга”, если эти люди не заняты в сфере разработки программ. Причина подобной ситуации заключается скорее в том, что индустрия по разработке программ отказывается рассматривать код в качестве проекта, а не в том, что процессы инжиниринга в этих двух случаях чем-то сильно отличаются. Макеты, прототипы и экспериментальные модели, скорее всего, имеют отношение к другим инженерным дисциплинам. Разработчики программных проектов

не располагают либо просто не используют формализованные методы проверки собственных проектов, руководствуясь соображениями экономии средств в производственном цикле разработки ПО.

Так что на самом деле дешевле и проще создать и протестировать проект, чем делать что-либо еще. В этом случае не нужно беспокоиться о количестве разрабатываемых версий — их стоимость приравнивается к нулевой, а использованные ресурсы могут быть востребованы обратно в случае, если отказаться от этой версии. Обратите внимание, что тестирование — это не просто вспомогательное действие, направленное на улучшение текущего проекта, а скорее органическая часть процесса детализации проекта. Инженеры, занимающиеся разработкой аппаратных средств, часто создают модели (или, по крайней мере, в визуальном виде воспроизводят свои проекты, используя компьютерную графику). Это позволяет им получать “ощущение” проекта, чего невозможно достичь с помощью простого просмотра самого проекта. В программных проектах создание упомянутых моделей просто невозможно, причем этого даже и не требуется. В этом случае всего лишь создается программный продукт. Даже если бы формальные доказательства, используемые в индустрии ПО, работали в автоматическом режиме, как и компиляторы, можно было бы как и раньше оставаться в рамках циклов создания версий/тестирования. Следовательно, формальные доказательства никогда не имели большой “вес” в индустрии разработки ПО.

Такова реальность сегодняшних дней, связанная с процессом разработки ПО. Более сложные программные проекты создаются с привлечением все большего количества людей и организаций. Превращение разработанных проектов в программный код осуществляется с помощью некоего языка программирования, а проверка действительности и уточнение осуществляется в цикле “создание версий/тестирование”. В ходе осуществления этого процесса могут появляться ошибки, причем изначально ему не присуща какая-либо строгость. Фактически, многие разработчики не хотят верить в то, что способ, с помощью которого осуществляется реализация процесса, представляет собой чрезвычайную проблему.

Большинство текущих процессов разработки ПО предусматривают выделение различных фаз программного проекта в виде отдельных модулей. Проект верхнего уровня должен быть завершен и “заморожен” до наступления этапа создания кода. Процесс тестирования и отладки требуется исключительно для устранения ошибок, связанных с процессом разработки ПО. Посредине находятся программисты, которые являются конструкторами индустрии по разработке ПО. Многие полагают, что если бы программисты меньше занимались “хакингом”, а создавали новые версии проектов в “правильном ключе” (и меньше бы делали ошибок при этом), то процесс разработки ПО вполне мог бы превратиться в инженерную дисциплину. Этого не произойдет до тех пор, пока в ходе осуществления процесса игнорируются инженерные и экономические реальности.

Например, ни в какой другой современной отрасли индустрии не допускается норма переработки продукта, равная 100%. Конструктор, который не сможет сконструировать что-либо правильно в первый раз, в большинстве случаев остается без работы. В процессе разработки ПО существует вероятность, что даже наименьший фрагмент кода будет исправлен или полностью переписан в процессе тестирования и отладки. Работа по уточнению кода, выполняемая во время осуществления творческого процесса, относится к проекту, а не к производственному процессу. Никто не ожидает того, что инженер сможет создать совершенный проект с “первой попытки”. Даже если инженеру удастся создать подобный проект, потребуются уточнения, в процессе которых проверяется совершенство проекта.

Вместо того чтобы форсировать разработку программного проекта в целях соответствия некорректной модели процесса, следует внимательно изучить его, чтобы добиться ускоренного производства ПО отличного качества. В данном случае идет речь о некой “лакмусовой бумажке” для процесса “программного инжиниринга”. Инжиниринг определяет методы осуществления процесса, а вовсе не то, требует ли конечный проектный документ CAD-системы для его производства.

Основная проблема, возникающая в процессе разработки ПО, заключается в том, что все является частью процесса проектирования. Процессы кодирования, тестирования и отладки являются составными частями процесса проектирования, причем сюда же входит все, что обычно называется программным проектом. Создание ПО может обходиться дешево, но его проектирование бывает невероятно дорогим. По причине повышенной сложности программ существует множество различных аспектов проектирования, а также взглядов на сам процесс проектирования. Проблему порождает взаимодействие различных аспектов (как и в случае проектирования аппаратных компонентов). Было бы идеально, если бы разработчики проектов верхнего уровня могли бы игнорировать детали модульного алгоритмического проектирования. Также было бы неплохо, если бы программисты не думали о вопросах, связанных с проектированием на верхнем уровне, при разработке внутренних алгоритмов, используемых модулем. К сожалению, аспекты, связанные с одним проектирования, часто “вторгаются” на другие уровни. Выбор алгоритмов для данного модуля может быть важен для достижения общего успеха программной системы, равно как и любой из аспектов проекта верхнего уровня. Не существует иерархия предпочтений, связанных с различными аспектами программного проекта. Некорректный проект на низшем модульном уровне может быть столь же пагубным, как и ошибка, допущенная на высшем уровне. Программный проект должен быть завершенным и корректным во всех своих ипостасях, иначе все программы, созданные на основе разработанного проекта, будут ошибочными.

В целях корректного решения вопросов, связанных с возникающей сложностью, используется проектирование ПО с помощью разбиения на уровни. Если программиста беспокоят вопросы, связанные с детализированным проектирова-

нием одного модуля, возможно, существуют сотни других модулей и тысячи деталей, о которых он не может беспокоиться одновременно. Например, существуют важные аспекты программного проекта программного обеспечения, которые не подпадают в категории структур данных и алгоритмов. В идеале программисты не должны были бы беспокоиться об этих аспектах в процессе разработки кода.

Рассматриваемые вопросы не связаны с процессом проектирования программ, но в любом случае они весьма актуальны. Программный проект считается незавершенным до тех пор, пока он не будет закодирован и протестирован. Тестирование является фундаментальной частью процесса проверки корректности и уточнения проекта. Структурный проект верхнего уровня не является завершенным программным проектом; в данном случае скорее идет речь о некоем структурном каркасе, используемом для детализированного проекта. Мы располагаем весьма ограниченными возможностями для строгой проверки правильности проекта, относящегося к верхнему уровню. Процесс детализированного проектирования в конечном счете оказывает влияние (или должен влиять) на проект верхнего уровня, по крайней мере, так же сильно, как другие факторы. Уточнение всех аспектов, связанных с проектом, — это процесс, который должен “проходить красной нитью” через все фазы разработки проекта. Если какой-либо аспект проекта отделен от процесса уточнения, неудивительно, что конечный проект будет неудовлетворительным или даже нерабочим.

Было бы неплохо, если бы программный проект верхнего уровня относился бы к категории более строгих инженерных процессов, но реальный мир программных систем не является строгим. Программы являются слишком сложными и зависят от многих других факторов. Может случиться так, что некоторое аппаратное обеспечение работает вовсе не так, как задумали проектанты, или библиотечная подпрограмма имеет нездокументированное ограничение. Именно с подобными типами проблем приходится сталкиваться при разработке каждого программного проекта (раньше или позже). Эти виды проблем выявляются в процессе тестирования (при условии его тщательного проведения) по той простой причине, что раньше их обнаружить просто невозможно. После обнаружения этих проблем приходится выполнять соответствующие изменения в проекте. При определенной степени везения изменения будут носить локальный характер. Чаще происходит все с точностью дооборот, изменения затрагивают значительную часть программного проекта (закон Мерфи). Если часть затронутого проекта не может быть изменена в силу каких-либо причин, затрудняется процесс адаптации других частей проекта. Полученный в этом случае результат менеджеры часто называют “хакерством”, но он является реальностью процесса разработки ПО.

Например, я недавно работал над проектом, в процессе осуществления которого была открыта временная зависимость между содержимым модулей А и В. К сожалению, содержимое модуля А было скрыто за абстракцией, вследствие чего был исключен другой способ включения вызова модуля в состав соответ-

ствующей последовательности. Конечно, со временем проблема была решена, но было уж слишком поздно пытаться изменить абстракцию А. Как ожидалось, в результате получился невероятно сложный набор “исправлений”, примененных во внутреннем проекте А. Прежде чем была завершена установка версии 1, создавалось впечатление полного разрушения проекта. Каждое новое исправление приводило к разрушению старого исправления. Причем в данном случае идет речь об обычном программном проекте. В конечном итоге, мои коллеги и я спорили о возможных изменениях в проекте, но мы вызвались добровольно поработать бесплатно, чтобы решить проблемы менеджмента.

В любом программном проекте, имеющем типичный размер, практически всегда возникают подобные проблемы. Несмотря на все попытки предотвратить их появление, важные детали будут упущены из виду. Именно в этом заключается различие между ремеслом и проектированием. Опыт может вести нас в правильном направлении. Именно в этом и заключается суть ремесла. Опыт проводит нас на территорию, не отмеченную на карте. Затем следует оставить то, с чего мы начинали, и сделать его значительно лучше, воспользовавшись контролируемым процессом уточнения. В этом и состоит процесс проектирования.

В качестве маленького замечания: все программисты знают о том, что создание документов, сопровождающих программный проект — после завершения кодирования (а не до его начала), приводит к созданию более точных документов. Причина очевидна. Конечный проект в том виде, в котором он отражен в коде, уточняется лишь один раз во время цикла “разработка проекта/тестирование”. Вероятность того, что начальный проект будет неизменным во время этого цикла, обратно пропорциональна количеству модулей, а также количеству программистов, выполняющих проект. Эта вероятность быстро стремится к нулю.

В процессе разработки программ необходим отличный проект на всех уровнях. В частности, требуется хороший проект верхнего уровня. Чем лучше более ранний проект, тем легче будет разработать детализированный проект. Разработчики проектов могут использовать все необходимые вспомогательные материалы. В частности, пользуйтесь структурными схемами, диаграммами Буча, таблицами состояния, PDL и т.д. Однако следует помнить о том, что перечисленные инструменты и способы записи не являются программным проектом в полном смысле этого слова. Конечная цель заключается в разработке реального программного проекта, который должен быть реализован на некотором языке программирования. Поэтому следует приступать к немедленному кодированию сразу же после завершения разработки программного проекта. Просто следует быть готовым к дальнейшему уточнению разработанного проекта.

Не существует записи проекта, которая может быть в равной степени пригодной для использования как в проекте верхнего уровня, так и в детализированном проекте. Завершение проекта знаменуется кодированием с применением некоторого языка программирования. Это значит, что записи проектов верхнего уровня

должны быть переведены на целевой язык программирования до того, как начнется этап детализированной разработки проекта. Этап перевода занимает время, а также приводит к появлению ошибок. Вместо прямой трансляции с записи, в результате которой не может осуществляться точное отображение на используемый язык программирования, программистам часто приходится возвращаться к требованиям, а также переделывать проект верхнего уровня, кодируя его “на лету”. Это также является частью реальности, связанной с разработкой ПО.

Вероятно, лучше разрешить специалистам в области проектирования создавать исходный код, а не позволять кому-либо транслировать проект позднее на уровне, который не зависит от языка. Все, что в данном случае требуется, — единая запись проекта, приемлемая на всех уровнях проекта. Другими словами, требуется язык программирования, который соответствует концепциям проекта верхнего уровня. Вот где “на сцену выходит C++”. Язык программирования C++ пригоден для разработки реальных проектов, причем он также может использоваться в качестве выразительных средств программных проектов. Язык C++ позволяет непосредственно выражать информацию верхнего уровня о проектных компонентах. В результате значительно упрощается создание проекта, а также его уточнение в дальнейшем. Благодаря более строгой проверке типов облегчается процесс обнаружения ошибок, связанных с проектом.

В итоге программный проект должен быть представлен на некотором языке программирования, а затем проверен и уточнен с применением цикла создание версии/тестирование. Какое-либо другое действие будет неправомерным. А теперь вкратце рассмотрим популярные инструменты и методики, используемые в процессе разработки ПО. В свое время структурное программирование рассматривалось как крупное достижение. “Героем” тех времен был язык программирования Pascal. Новой “звездой” стало объектно-ориентированное проектирование и язык C++. Теперь подумаем о том, что может вызвать потенциальные проблемы. Инструменты CASE? Популярны, да; универсальны, нет. Структурные схемы? Об этом можно сказать то же самое. Также речь идет о диаграммах Уорнера-Орра (Warner-Orr), диаграммах Буча, объектных диаграммах и т.д. Каждый из этих объектов имеет свои сильные стороны, но в то же время одну фундаментальную слабость — они не являются программным проектом. Фактически единственной записью программного проекта, которая широко используется на практике, является диаграмма PDL.

Это говорит о том, что “коллективное подсознательное” в индустрии по разработке ПО инстинктивно “осознает”, что улучшения в методах программирования и в действительных языках программирования в частности являются чрезвычайно более важными, чем что-нибудь еще, имеющее отношение к бизнесу по разработке ПО. Это также свидетельствует о том, что программисты заинтересованы в разработке проектов. По мере того как становятся доступными более выразительные языки программирования, разработчики начинают перенимать их.

А теперь рассмотрим, каким образом изменяется процесс разработки ПО. Когда-то шла речь о каскадной модели разработки ПО. Теперь же идет речь о спиральной модели, а также о модели быстрого прототипирования. В то время, как упомянутые методы часто сопровождаются характеристиками “снижения риска” и “сокращения времени поставки продукта”, они в действительности обосновывают более раннее начало кодирования в жизненном цикле разработки ПО. И этот фактор является очень положительным. Это позволяет в процессе цикла “создание версий/тестирование” начать проверку корректности и уточнения проекта на ранних этапах. Это означает (что более вероятно), что разработчики программного проекта, которые разрабатывали проект верхнего уровня, останутся заняты детализированным проектом.

Как отмечалось ранее, проектирование — это набор указаний о том, каким образом осуществлять проект, а не описание внешнего вида конечного продукта. Участвуя в бизнесе по разработке ПО, мы вынуждены быть инженерами. Программирование и цикл “создание версий/тестирование” являются центральными в процессе разработки программных проектов. Также потребуется управлять этими компонентами. Процесс может быть усовершенствован, но отказаться от его осуществления просто невозможно.

Последнее замечание: целью любой разработки проекта является создание некой документации. Очевидно, что наиболее важны фактические проектные документы. Также весьма вероятно то, что позднее система будет изменяться и разрастаться в размерах. Это означает, что вспомогательная документация важна как для программного проекта, так и для проекта, предусматривающего разработку аппаратных средств. Игнорируемые на настоящем этапе руководства пользователя, инструкции по установке, а также некоторые другие документы связаны с процессом проектирования. Все еще существуют две важные потребности, которые должны решаться путем использования вспомогательных документов, сопровождающих проект.

Первая потребность: использование вспомогательной документации в целях получения важной информации из проблемной области. Программный проект включает изобретение концепций ПО, используемых для моделирования концепций в предметной области. Этот процесс требует разработки и понимания концепций, связанных с проблемной областью. Обычно это понимание включает информацию, влечет выполнение моделирования в области разработки ПО, но которая, тем не менее, помогает проектанту определить существенные концепции, а также то, каким наилучшим образом можно их моделировать. Эта информация должна сохраняться, чтобы быть примененной в случае потребности в дальнейшем изменении модели.

Второй важной потребностью является документирование тех аспектов проекта, сведения о которых трудно получить непосредственно из самого проекта. Сюда могут включаться аспекты проекта верхнего и нижнего уровней. Многие

из этих аспектов лучше всего изображаются в графическом виде. В результате затрудняется их включение в первоначальный код в качестве комментариев. Этот аргумент не актуален при графической записи программного проекта, используемой вместо языка программирования.

Никогда не забывайте о том, что фактический вид проекта определяет исходный код, а не вспомогательная документация. В идеале, должны быть доступны программные инструменты, которые позволяют произвести завершающую обработку проекта, представленного исходным кодом, а также генерировать вспомогательную документацию. Конечно, идеальный вариант не всегда осуществляется на практике. Неплохо, если бы некоторые инструменты позволяли программистам (или техническим писателям) извлекать специфическую информацию на основе исходного кода, которая затем документируется некоторым иным способом. Несомненно, что своевременное обновление подобной документации вручную затруднительно. И это еще один аргумент в пользу применения более выразительных языков программирования. Также лучше сводить объем вспомогательной документации к минимуму, причем следует придавать ей по возможности неформальный характер. Опять же, можно воспользоваться некоторыми лучшими инструментами; иначе придется вернуться к карандашу, бумаге и мелу.

Подводя итог, скажем следующее.

- Реальное ПО выполняется на компьютерах. Любая программа является последовательностью, состоящей из нулей и единиц, которые хранятся на некотором магнитном носителе. Программа не эквивалентна листингу на языке C++ (или любом другом языке программирования).
- Программный листинг является документом, представляющим программный проект. Программные проекты фактически создаются компиляторами и компоновщиками.
- Создание программ обходится очень дешево, причем это тенденция проявляется все сильнее с увеличением быстродействия компьютеров.
- Проектирование реальных программ обходится достаточно дорого. Это связано с невероятной сложностью ПО, а также с тем, что практически все этапы программного проекта являются частью процесса проектирования.
- Программирование — это деятельность в рамках проекта. Это признается при любом хорошо организованном процессе разработки программного проекта, поэтому отсутствуют колебания, связанные с необходимостью кодирования.
- Кодирование имеет смысл гораздо чаще, чем это может показаться на первый взгляд. Весьма часто процесс трансляции проекта в код приводит к обнаружению просчетов и к необходимости предпринимать дополнительные усилия в рамках проекта. Чем раньше это происходит, тем лучшим будет качество самого проекта.

- В силу дешевизны формальные инженерные методы проверки корректности нечасто используются в процессе реальной разработки ПО. Проще и дешевле создавать и тестировать проект, чем пытаться проверять его корректность.
- Действия по тестированию и отладке программ подобны процессам проверки корректности и уточнения проекта, характерных для других инженерных дисциплин.
- Существуют и другие взаимодействия, связанные с разработкой проекта — проект верхнего уровня, модульное проектирование, структурное проектирование, архитектурное проектирование, а также многие другие.
- Все действия, реализованные в рамках проекта, рассматриваются как интерактивные. Они характеризуют хорошо продуманный процесс разработки ПО, в котором разрешается изменение проекта (порой, радикальное) по мере того, как в этом возникает необходимость и по мере выполнения различных этапов проекта.
- Весьма полезным является ведение различных записей, используемых в программных проектах. Их можно считать вспомогательной документацией, облегчающей процесс разработки проекта.
- В любом случае разработка ПО — это скорее искусство, а не строгая инженерная дисциплина. Это связано с тем, что критические действия, направленные на проверку корректности и уточнение проекта, жестко не заданы.
- В конечном счете, фактические улучшения в процессе разработки ПО зависят от улучшений в методах программирования, которые, в свою очередь, зависят от улучшения в языках программирования. Улучшение подобного рода представляет язык программирования C++. Его популярность объясняется тем, что этот язык находится в “основном потоке” языков программирования, непосредственно поддерживающих лучшие программные проекты.

## Послесловие

*В процессе пересмотра написанного 10 лет назад материала у меня появилось несколько свежих идей. Первая (более уместная именно для этой книги) заключается в том, что на сегодняшний день я еще больше убедился в истинности ключевых моментов, изложенных в этой статье. Мое убеждение основано на популярных разработках в последующие годы. Наиболее очевидна (и, возможно, менее всего важна) популярность объектно-ориентированных языков программирования в современной практике разработки программных проектов. В настоящее время появилось множество объектно-ориентированных языков программирования, отличных от C++. К тому же, разработаны объектно-ориентированные форматы записи проектов, например, UML. Мое утверждение*

о том, что популярность объектно-ориентированных языков программирования объясняется тем, что более выразительные проекты могут непосредственно транслироваться в код, кажется уже несколько устаревшим.

Концепция рефакторинга — реструктуризация кода с целью придать ему свойства устойчивости и повторного использования — перекликается с моими формулировками о том, что все аспекты проекта должны быть гибкими, позволяющими выполнять изменения по мере выполнения проверки корректности проекта. Рефакторинг просто обеспечивает процесс, а также определяет основную “линию поведения”, определяющую методики улучшения проекта, которому присущи некие “слабые места”.

Вообще говоря, в настоящее время появилась целая концепция быстрой разработки ПО (*Agile Development*). Среди новых методик, присущих этой концепции, наиболее известно экстремальное программирование (*eXtreme Programming*). Наиболее общим является признание того факта, что исходный код является наиболее важным продуктом процесса разработки ПО.

С другой стороны, важность некоторых аспектов за истекшие годы значительно возросла. Первым из них является признание важности архитектуры, или проекта верхнего уровня. В статье было подчеркнуто, что архитектурная система является лишь частью проекта, поэтому она должна быть достаточно гибкой в процессе проверки корректности проекта в цикле “создания версий/тестирования”. Это положение остается истинным, но сейчас, оглядываясь назад, я думаю, что был несколько наивным. Тогда как цикл “создание версий/тестирование” может приводить к появлению проблем, связанных с архитектурой, большинство проблем обычно выявляется путем изменения требований. Разработка программных проектов “по большому счету” не так уж и проста, поэтому ни новые языки программирования, такие как Java или C++, ни графические записи, такие как UML-диаграммы, не принесут пользы для тех, кто не знает оптимальных методов их применения. Более того, однажды созданному проекту присуща надстройка из немалого кода вокруг архитектуры. Фундаментальное изменение архитектуры зачастую равноценно пересмотру проекта и возврату к началу, что эквивалентно полному “провалу” самого проекта. Даже в случае с проектами и организациями, которые на всех уровнях применяют концепцию рефакторинга, они неохотно берутся за что-либо, что выглядит как полное переписывание кода. Поэтому лучше полное переписывание проекта выполнять в самом начале, а не тогда, когда объемы проектов вырастают сверх всякой разумной меры. К счастью, в данном случае идет речь о той области, в которой могут с успехом применяться шаблоны программных проектов.

Особое внимание стоит уделять и вспомогательной документации, особенно касающейся архитектуры системы. В то время как исходный код может быть проектом, попытка сформировать архитектуру на основе начального кода может привести к печальным последствиям. В статье я выразил надежду, что

могут появиться инструменты по разработке ПО, позволяющие автоматизировать поддержку вспомогательной документации, сопровождающей исходный код. Теперь я отказался от этой идеи. Хорошо продуманная объектно-ориентированная архитектура может быть описана в нескольких диаграммах и на нескольких десятках страниц текста. Эти диаграммы (и текст) должны описывать ключевые классы и взаимосвязи проекта. К сожалению, я не вижу реальных признаков того, что инструменты по разработке ПО позволяют достаточно быстро извлекать эти важные аспекты из массы деталей исходного кода. Это означает, что неизбежно возникает потребность вручную создавать и поддерживать документацию. Тем не менее, я думаю, что лучше формировать ее после создания исходного кода, или, по крайней мере, одновременно, чем пытаться писать ее перед его созданием.

И наконец, учитывая то, что языки, конкурирующие с C++, не привнесли значительных изменений в искусство программирования, утверждение об особой роли языка C++ в настоящее время является еще более справедливым, чем когда-либо.

Джек Ривз, 1 января 2002.

# Предметный указатель

## А

А-метрика, 386  
Agile Alliance, 39

## М

Monostate, 278

## А

Абстракция, 172  
Автомат с конечным числом состояний, 581  
Агрегирование, 666  
Агрессивное программирование, 72  
Ассоциация, 666

## Б

Браузер Idea, 85  
Быстрая разработка ПО, 35

## В

Вариант использования, 294; 647  
Верификация, 76  
Вероятностная изоляция, 75

## Г

Главная последовательность, 389

## Д

Декомпозиция крупноячеистая, 379  
функциональная, 379  
Делегирование, 253  
Диаграмма STD, 581  
объектов, 695  
последовательности сообщений, 703

сотрудничества, 700

Диаграммы действий, 693  
Динамический полиморфизм, 216  
Дуальная ссылка, 546

## З

Запись UML, 645

## И

Изоляция теста, 73  
Исполнитель, 647

## К

Класс экземпляр, 273  
Компания Microburst, 502  
Компилятор SMC, 595  
Композиция, 667

## М

Метафора, 59  
Метрика управления зависимостями, 391  
Метрика устойчивости, 381  
Множественное наследование, 223  
Модель предметной области, 653  
Модульное тестирование, 76  
Моновалентность, 230

## Н

Наследование, 256

## П

Пакет, 366; 668  
TransactionImplementation, 417  
связывание , 367  
сцепление, 367

Парное программирование, 53  
План выпуска версий, 52  
Подсистема, 669  
Подсостояние, 686  
Поливалентность, 231  
Пользовательская история, 51; 295  
Приемочный тест, 52  
Признак плохого проекта  
    вязкость, 151  
    закрепощенность, 151  
    неоправданная сложность, 151  
    неоправданное повторение, 151  
    неопределенность, 151  
    неподвижность, 151  
    неустойчивость, 151  
Принцип  
    ADP, 372  
    CCP, 371  
    CRP, 370  
    DIP, 206  
    LSP, 185  
    OCP, 171  
    REP, 369; 407  
    SAP, 386  
    SDP, 380  
    SRP, 166  
Пробное испытание, 53  
Проводка, 344  
Программа расчета зарплаты, 293  
Проект Nimbus-LC, 503

## P

Разреженная матрица, 554  
Рефакторинг, 59; 85

## C

Система WMS, 502  
Системная граничная диаграмма, 652  
Служба ETS, 609  
Сортировка методом пузырька, 258  
Спупинг, 398  
Статистический мультиплексор, 678  
Статический полиморфизм, 216

Суперсостояние, 686  
Схематическое проектирование, 207

## T

Таблица  
    STT, 582  
Тестирование  
    методом белого ящика, 76  
    методом черного ящика, 76  
Транзакция, 242

## Ф

Фабрика, 396  
Фабрика объектов, 416  
Фирма Cloud Company, 502

## Ц

Цикл зависимости, 395

## Ш

Шаблон  
    Stairway to Heaven, 490  
    Abstract Server, 455  
    Active Object, 245  
    Acyclic Visitor, 550  
    Adapter, 456  
    Bridge, 462  
    Command, 240  
    Composite, 424  
    Decorator, 499; 562  
    Extension Object, 498; 568  
    Facade, 268; 499  
    Factory, 393  
    Mediator, 270  
    Null Object, 288  
    Observer, 428  
    Proxy, 397; 469  
    Singleton, 274  
    Strategy, 253; 261  
    Template Method, 253  
    Visitor, 499; 546

## Э

Экстремальное программирование, 50

# **Манифест альянса специалистов по быстрой разработке ПО**

Мы находимся в процессе поиска более эффективных методов разработки ПО, а также помогаем это делать другим пользователям. Особо пристальное внимание следует уделить таким вопросам:

- **индивиду и взаимодействия**, связанные с процессами и инструментальными средствами разработки;
- **рабочий программный продукт** и полный комплект документации;
- **совместная работа с заказчиком** и обсуждение условий контракта;
- **реакция на происходящие изменения** и соблюдение плана.

Наиболее важными являются компоненты, выделенные полужирным шрифтом, хотя и не следует пренебрегать остальными деталями.

*Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas*

## **принципы быстрой разработки ПО**

- Наша первостепенная задача — достичь соответствия требованиям заказчика, непрерывно поставляя ему в оптимальные сроки высококачественное ПО.
- Следует приветствовать изменения требований, даже если они происходят на позднем этапе разработки. Правила быстрой разработки ПО рассматривают изменения как преимущество в конкурентной борьбе.
- Следует чаще выпускать рабочие версии программ: от одного раза в две недели до одного раза в два месяца, стремясь к соблюдению более сжатых сроков.
- Менеджеры и разработчики обязаны совместно трудиться над проектом на протяжении всего процесса его выполнения.
- Выполняйте проекты с привлечением мотивированных индивидов. Представьте им соответствующую среду разработки и необходимую поддержку, причем доверьте им выполнение поставленных задач.
- Наиболее результативный и эффективный метод передачи информации команде, а также в рамках самой команды заключается в интерактивном общении.
- Рабочая программа — это первостепенный критерий процесса.
- Процесс быстрой разработки программ не терпит неравномерной работы. Спонсоры, разработчики и пользователи должны уметь поддерживать постоянный темп работы.
- Постоянный акцент на высоком техническом уровне качества и хорошем проекте повышает быстроту разработки.
- Простота, достигаемая путем минимизации ненужной работы, является определяющей.
- Наилучшие архитектуры, требования и проекты возникают у самоуправляемых команд.
- Периодически команда разработчиков делает выводы об эффективности своей работы, на основе которых корректирует свою линию поведения.

# **Практики экстремального программирования**

## **Командный принцип работы**

Все участники программного проекта (разработчики, бизнес-аналитики, тестеры и т. д.) работают совместно, являясь членами одной команды. Стены рабочей комнаты увешаны диаграммами, иллюстрирующими ход выполняемого проекта, а также обставлены стеллажами с документацией.

## **Игра в планирование**

Планирование является непрерывным и прогрессивным. Каждые две недели разработчики составляют план на следующие две недели. В этот документ включается описание разрабатываемых функций, которые предоставляются на рассмотрение заказчика. Последний осуществляет требуемый выбор, руководствуясь величинами затрат и экономической отдачи.

## **Испытания, выполняемые заказчиками**

Часть процесса выбора требуемой функции составляют автоматизированные приемочные испытания, производимые заказчиками. Благодаря этому проверяется работоспособность разработанных функций.

## **Простая структура**

Следует придерживаться структуры разработанного проекта, которая обеспечивает минимально достаточное количество системных функций. В частности, требуется успешный прогон тестов, отсутствие дублирования, внедрение всех запланированных функций, а также минимизация объема программного кода.

## **Парное программирование**

Все рабочее ПО создается двумя программистами, работающими за одним компьютером.

## **Разработка, управляемая тестированием**

Рекомендуется частое тестирование разработанных фрагментов программного кода. Неудачные результаты тестирования служат сигналом для переработки кода.

## **Улучшение проекта**

Не забывайте о том, что даже “на Солнце есть пятна”. Придерживайтесь максимальной чистоты и выразительности кода.

## **Непрерывная интеграция**

Команда разработчиков обязана придерживаться тактики максимальной интеграции системы.

## **Коллективное владение кодом**

Любая пара программистов может усовершенствовать разработанные фрагменты программ.

## **Стандарты кодирования**

Программный код в целом выглядит однообразно, соответствуя самым высоким требованиям к качеству.

## **Метафора**

Члены команды обладают единой точкой зрения на то, каким образом функционирует программа.

## **Постоянный темп**

Команда разработчиков работает длительный срок. Их тяжелая работа должна выполняться равномерно, с “марафонским” темпом.

# **Принципы объектно-ориентированного проектирования**

## **Принцип персональной ответственности (SRP, Single Responsibility Principle)**

Должна быть только одна причина для изменения класса.

## **Принцип “открытия-закрытия” (OCP, Open-Closed Principle)**

Программные объекты (классы, модули, функции и т. д.) должны быть открыты для расширения, но закрыты для изменения.

## **Принцип подстановки Лискоу (LSP, Liskov Substitution Principle)**

Подтипы должны наследовать базовые типы.

## **Принцип инверсии зависимостей (DIP, Dependency Inversion Principle)**

Абстракции не должны зависеть от деталей, а детали — от абстракций.

## **Принцип отделения интерфейса (ISP, Interface Segregation Principle)**

Клиенты не должны зависеть от методов, которые ими не используются. Интерфейсы принадлежат клиентам, но не иерархиям клиентов.

## **Принцип эквивалентности повторного применения/нового выпуска (REP, Release-Reuse Equivalency Principle)**

Повторное применение эквивалентно новому выпуску.

## **Принцип общего закрытия (CCP, Common Closure Principle)**

После выполнения некоторых типов изменений выполняется одновременное закрытие классов пакета. Изменения, влияющие на закрытый пакет, воздействуют на все классы этого пакета, но не затрагивают другие пакеты.

## **Принцип всеобщего повторного применения (CRP, Common Reuse Principle)**

Классы пакета повторно используются в синхронном режиме. Если повторно применяется один из классов пакета, в этот процесс вовлекаются другие классы из этого же пакета.

## **Принцип ациклических зависимостей (ADP, Acyclic Dependencies Principle)**

Не допускаются циклы в графе зависимостей пакета.

## **Принцип устойчивых зависимостей (SDP, Stable Dependencies Principle)**

Зависимость от направления устойчивости.

## **Принцип устойчивости абстракций (SAP, Stable Abstractions Principle)**

Пакет будет абстрактным в той же степени, в которой он является устойчивым.

*Научно-популярное издание*

Роберт С. Мартин

# Быстрая разработка программ: принципы, примеры, практика

Литературный редактор *С.Г. Татаренко*

Верстка *А.Н. Полинчик*

Художественный редактор *С.А. Чернокозинский*

Корректоры *З.В. Александрова,*

*Л.А. Гордиенко,*

*О.В. Мишутина,*

*Л.В. Чернокозинская*

Издательский дом “Вильямс”.

101509, Москва, ул. Лесная, д. 43, стр. 1.

Изд. лиц. ЛР № 090230 от 23.06.99

Госкомитета РФ по печати.

Подписано в печать 29.11.2003. Формат 70×100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 60,63. Уч.-изд. л. 35,9.

Тираж 3000 экз. Заказ № 1255.

Отпечатано с диапозитивов в ФГУП “Печатный двор”

Министерства РФ по делам печати,

телерадиовещания и средств массовых коммуникаций.

197110, Санкт-Петербург, Чкаловский пр., 15.

# БЫСТРАЯ РАЗРАБОТКА ПРОГРАММ

Принципы, примеры, практика

Роберт К. Мартин

а также Джеймс В. Ньюкирк и Роберт С. Косс

Роберт К. Мартин, автор множества бестселлеров, всемирно известный эксперт в области разработки ПО, демонстрирует методы решения проблем, возникающих в процессе деятельности разработчиков ПО, руководителей и менеджеров проектов. Перед вами исчерпывающее практическое руководство, написанное одним из основателей различных методик быстрой разработки ПО и экстремального программирования. В частности, здесь вы найдете:

- советы, помогающие разработчикам ПО и менеджерам своевременно выполнять проекты, не выходя за рамки утвержденного бюджета и используя весь арсенал средств быстрой разработки ПО;
- практические примеры, демонстрирующие методы планирования, тестирования, рефакторинга, а также парного программирования;
- решение проблем, связанных с ориентированными на заказчиков системами, с помощью диаграмм UML и шаблонов проектирования.

## Об авторе

Роберт К. Мартин является президентом компании Object Mentor Inc. В своей деятельности Мартин и его команда консультантов используют методы объектно-ориентированного проектирования, шаблоны, диаграммы UML, а также методы быстрой разработки ПО и экстремального программирования. Его перу принадлежат такие бестселлеры, как *Designing Object-Oriented C++ Applications Using the Booch Method* (Prentice Hall, 1995), а также многие другие книги. Доктор Мартин выполнял функции главного редактора во время издания книги *Pattern Languages of Program Design 3* (Addison Wesley, 1999), редактировал книгу *More C++ Gems* (Cambridge, 1999), а также был соавтором Джеймса Ньюкирка в книге *XP in Practice* (Addison-Wesley, 2001). Роберт был главным редактором журнала *C++ Report* с 1996 по 1999 год.

"Возможно, перед вами первая книга, в которой применяется комплексный подход, включающий описание методов и шаблонов быстрой разработки ПО, а также методик, составляющих основу современной практики разработки программ. Если говорит Боб Мартин, внимайте ему".

Джон Власциан,  
консультант и автор  
книги *Design Patterns  
and Pattern Hatching*

"Я очень долго ждал появления подобной книги. В этом труде Боб сумел преподать уроки, позволяющие достичь вершин в искусстве программирования".

Мартин Фаулер,  
консультант и автор  
книги *UML Distilled  
and Refactoring*

"В этой книге Боб Мартин демонстрирует незаурядный талант разработчика ПО и преподавателя. Он поражает глубиной практического подхода и очаровательным стилем изложения".

Кraig Larman,  
консультант и автор  
книги *Применение UML и шаблонов проектирования*



[www.williamspublishing.com](http://www.williamspublishing.com)

Pearson  
Education

[www.prenhall.com](http://www.prenhall.com)

ISBN 5-8459-0558-3

04003

9 785845 905581