HIGH TECH

Р. С. МАРТИН,
М. МАРТИН

Принципы, паттерны и методики гибкой разработки на языке C#



СИМВОЛ®

Agile Principles, Patterns, and Practices in C#

Robert Martin, Micah Martin

PRENTICE HALL
Pearson Education

HIGH TECH

Принципы, паттерны и методики гибкой разработки на языке C#

Роберт Мартин, Мика Мартин

Символ®

Санкт-Петербург — Москва
2011

Серия «High tech»

Роберт Мартин, Мика Мартин

Принципы, паттерны и методики гибкой разработки на языке C#

Перевод А. Слинкина

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Е. Тульсанова*
Корректор	*С. Минин*
Верстка	*К. Чубаров*

Мартин Р., Мартин М.

Принципы, паттерны и методики гибкой разработки на языке C#. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 768 с., ил.

ISBN 978-5-93286-197-4

Цель данной книги – собрать воедино все методики гибкой разработки и показать их работоспособность. Основанная на богатом опыте известного специалиста, Роберта Мартина, книга охватывает как теорию, так и все аспекты практического применения гибкой разработки. Во вступительных главах излагаются основные принципы, а далее они демонстрируются в действии. Применяя объектно-ориентированный подход, авторы рассматривают конкретные паттерны, применяемые к проектированию приложений, описывают методы рефакторинга и способы эффективного использования различных видов UML-диаграмм. Взяв какую-либо реальную задачу, они показывают, какие ошибки и ложные ходы можно допустить в ходе ее решения и как применение правильных методик позволяет добиться успеха.

Основная идея гибкой разработки: успех зависит прежде всего от людей. Работайте с командой увлеченных программистов, применяйте упрощенные процессы, подстроенные под эту команду, непрерывно адаптируйтесь к задаче – и успех вам гарантирован.

Книга в равной мере подойдет и тем, кто еще только собирается практиковать гибкую разработку, и тем, кто желает усовершенствовать уже имеющиеся навыки. Издание содержит много примеров исходного кода, которые можно скачать с сайта авторов.

ISBN 978-5-93286-197-4
ISBN 978-0-13-185725-4 (англ)

© Издательство Символ-Плюс, 2010
Authorized translation of the English edition © 2007 Pearson Education Inc. This translation is published and sold by permission of Pearson Education Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 30.11.2010. Формат 70×100 1/16. Печать офсетная.
Объем 48 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Манифест гибкой разработки программ

Мы открываем лучшие способы разработки программного обеспечения, применяя их на практике и помогая в этом другим. В ходе своей работы мы научились ценить:

Людей и их взаимоотношения
больше, чем процессы и инструменты.

Работающую программу
больше, чем исчерпывающую документацию.

Плодотворное сотрудничество с заказчиком
больше, чем формальные договоренности по контракту.

Оперативное реагирование на изменения
больше, чем следование плану.

Иначе говоря, не отрицая значимость факторов, перечисленных во второй части предложений, мы больше ценим те, что в первой части.

Кент Бек
Майк Бидл
Ари ван Беннекум
Алистер Кокбэрн
Уорд Каннингэм
Мартин Фаулер
Джеймс Греннинг
Джим Хайсмит
Эндрю Хант
Рон Джеффрис
Йон Керн
Брайан Мэрик
Роберт К. Мартин
Стив Меллор
Кен Швабер
Джефф Сазерленд
Дэйв Томас

Принципы Манифеста гибкой разработки

Мы придерживаемся следующих принципов:

- Высшим приоритетом считать удовлетворение пожеланий заказчика посредством поставки полезного программного обеспечения в сжатые сроки с последующим непрерывным обновлением.
- Не игнорировать изменение требований, пусть даже на поздних этапах разработки. Гибкие процессы позволяют учитывать изменения и тем самым обеспечивать заказчику конкурентные преимущества.
- В процессе разработки предоставлять промежуточные версии ПО часто, с интервалом от пары недель до пары месяцев, отдавая предпочтение меньшим срокам.
- Заказчики и разработчики должны работать совместно на протяжении всего проекта.
- Проекты должны воплощать в жизнь целеустремленные люди. Создайте им условия, обеспечьте необходимую поддержку и верьте, что они доведут дело до конца.
- Самый эффективный и продуктивный метод передачи информации команде разработчиков и обмена мнениями внутри нее – разговор лицом к лицу.
- Работающая программа – основной показатель прогресса в проекте.
- Гибкие процессы способствуют долгосрочной разработке. Заказчики, разработчики и пользователи должны быть в состоянии поддерживать неизменный темп сколь угодно долго.
- Непрестанное внимание к техническому совершенству и качественному проектированию повышает отдачу от гибких технологий.
- Простота – искусство достигать большего, делая меньше, – основа основ.
- Самые лучшие архитектуры, требования и проекты выдают самоорганизующиеся команды.
- Команда должна регулярно задумываться над тем, как стать еще более эффективной, а затем соответственно корректировать и подстраивать свое поведение.

Методики экстремального программирования

Единая команда

Все участники XP-проекта, бизнес-аналитики, тестировщики и т. д. находятся в открытом рабочем помещении и являются членами одной команды. На стенах развешаны крупные, видные всем графики и другие свидетельства продвижения вперед.

Игра в планирование

Планирование непрерывно и поступательно. Каждые две недели разработчики оценивают стоимость функций, предложенных для реализации в течение следующих двух недель, а заказчик отбирает те функции, которые должны быть реализованы исходя из их стоимости и ценности для бизнеса.

Тесты пишет заказчик

При выборе каждой желаемой функции заказчик составляет автоматизированные приемочные тесты, демонстрирующие, что данная функция работает.

Простой дизайн

Команда разрабатывает такой дизайн, который отвечает текущей функциональности системы, – ни больше ни меньше. Он должен проходить все тесты, быть свободным от дублирования, выражать все, что хотел выразить автор, и содержать как можно меньше кода.

Парное программирование

Весь код пишется двумя программистами, сидящими бок о бок за одним компьютером.

Разработка через тестирование

Программисты работают очень короткими циклами, сначала добавляя тест, который не проходит, а затем код, при котором тест завершается успешно.

Улучшение дизайна

Плохой код не должен доживать до заката. Код все время должен оставаться максимально чистым и выразительным.

Непрерывная интеграция

Система все время остается интегрированной.

Коллективное владение кодом

Любая пара программистов вправе в любой момент усовершенствовать любой код.

Стандарты кодирования

Весь код выглядит так, будто его писал один – очень квалифицированный – человек.

Метафора

Команда поддерживает общее видение работы программы.

Умеренный темп

Команда собралась для длительной работы. Она работает усердно, но в таком темпе, который можно поддерживать сколь угодно долго. Команда бережет силы, рассматривая проект как марафон, а не спринт.

Принципы объектно-ориентированного проектирования

SRP	Принцип единственной обязанности *У класса должна быть только одна причина для изменения.*
OCP	Принцип открытости/закрытости *Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для модификации.*
LSP	Принцип подстановки Лисков *Должна быть возможность вместо базового типа подставить любой его подтип.*
DIP	Принцип инверсии зависимости *Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.*
ISP	Принцип разделения интерфейсов *Клиенты не должны вынужденно зависеть от методов, которыми не пользуются. Интерфейсы принадлежат клиентам, а не иерархиям.*
REP	Принцип эквивалентности повторного использования и выпуска *Единица повторного использования равна единице выпуска.*
CCP	Принцип общей закрытости *Все классы внутри пакета должны быть закрыты относительно изменений одного и того же вида. Изменение, затрагивающее пакет, должно затрагивать все классы в этом пакете и только в нем.*
CRP	Принцип совместного повторного использования *Все классы внутри компонента используются совместно. Если вы можете повторно использовать один класс, то можете использовать и все остальные.*
ADP	Принцип ацикличности зависимостей *В графе зависимостей между пакетами не должно быть циклов.*
SDP	Принцип устойчивых зависимостей *Зависимости должны быть направлены в сторону устойчивости.*
SAP	Принцип устойчивых абстракций *Пакет должен быть столь же абстрактным, сколь и устойчивым.*

Оглавление

Предисловие 21
Об авторах 24
Введение 25

Часть I. Гибкая разработка 35

Глава 1. Гибкие методики 37
Организация Agile Alliance 38
Люди и их взаимоотношения
важнее процессов и инструментов 39
Работающая программа
важнее исчерпывающей документации 40
Плодотворное сотрудничество с заказчиком
важнее формальных договоренностей по контракту 41
Оперативное реагирование на изменения
важнее следования плану 42
Принципы 43
Заключение 45
Библиография 46

Глава 2. Обзор экстремального программирования 47
Методики экстремального программирования 48
Единая команда 48
Пользовательские истории 48
Короткие циклы 49
Приемочные тесты 50
Парное программирование 50
Разработка через тестирование 51
Коллективное владение 52
Непрерывная интеграция 52
Умеренный темп 53
Открытое рабочее пространство 53
Игра в планирование 53
Простота 54
Рефакторинг 55
Метафора 56

Заключение 57
Библиография 57
Глава 3. Планирование 58
Первичное обследование 59
Объединение, разбиение и скорость 59
Планирование выпуска 60
Планирование итерации 61
Определения понятия «готово» 61
Планирование задач 62
Подведение итогов итераций 63
Мониторинг 63
Заключение 64
Библиография 65
Глава 4. Тестирование 66
Разработка через тестирование 67
Пример проектирования начиная с тестов 68
Изоляция тестов 69
Разделение в придачу 71
Приемочные тесты 72
Архитектура в придачу 74
Заключение 74
Библиография 75
Глава 5. Рефакторинг 76
Простой пример рефакторинга: генерация простых чисел 77
Автономное тестирование 79
Рефакторинг 80
Последнее прочтение 85
Заключение 89
Библиография 90
Глава 6. Эпизод программирования 91
Игра в боулинг 93
Заключение 137
Часть II. Гибкое проектирование 139
Глава 7. Что такое гибкое проектирование 141
Ароматы дизайна 142
Ароматы дизайна – запахи гниющей программы 142
Жесткость 143
Хрупкость 143
Косность 143

Вязкость ... 143
Ненужная сложность ... 144
Ненужные повторения ... 144
Непрозрачность ... 145
Почему программы загнивают ... 145
Программа Copy ... 146
Знакомый сценарий ... 146
Гибкий дизайн программы Copy ... 150
Заключение ... 152
Библиография ... 152
Глава 8. Принцип единственной обязанности (SRP) ... 153
Определение обязанности ... 155
Разделение связанных обязанностей ... 157
Обеспечение сохранности ... 157
Заключение ... 158
Библиография ... 158
Глава 9. Принцип открытости/закрытости (OCP) ... 159
Описание принципа OCP ... 160
Приложение Shape ... 162
Нарушение OCP ... 163
Решение, удовлетворяющее принципу OCP ... 165
Предвидение и «естественная» структура ... 167
Расстановка «точек подключения» ... 168
Применение абстракции для явного закрытия ... 169
Закрытие за счет применения таблицы данных ... 170
Заключение ... 171
Библиография ... 171
Глава 10. Принцип подстановки Лисков (LSP) ... 172
Нарушения принципа LSP ... 173
Простой пример ... 173
Более тонкое нарушение ... 175
Реальный пример ... 181
Факторизация вместо наследования ... 186
Эвристика и соглашения ... 188
Заключение ... 189
Библиография ... 189
Глава 11. Принцип инверсии зависимости (DIP) ... 190
Разбиение на слои ... 191
Инверсия владения ... 192
Зависимость от абстракций ... 194

Простой пример принципа DIP 195
Отыскание глубинных абстракций 196
Задача об управлении печью 197
Заключение 199
Библиография 199
Глава 12. Принцип разделения интерфейсов (ISP) 200
Загрязнение интерфейса 200
Разделение клиентов означает разделение интерфейсов 202
Интерфейсы классов и интерфейсы объектов 204
Разделение путем делегирования 204
Разделение путем множественного наследования 205
Пользовательский интерфейс банкомата 206
Заключение 212
Библиография 213
Глава 13. Обзор UML для программистов 214
Диаграммы классов 218
Диаграммы объектов 219
Диаграммы последовательности 220
Диаграммы кооперации 220
Диаграммы состояний 221
Заключение 222
Библиография 222
Глава 14. Работа с диаграммами 223
Зачем нужно моделировать 223
Зачем строить модели программ 224
Нужно ли проектировать систему до конца, прежде чем приступать к кодированию 224
Эффективное использование UML 225
Общение с другими людьми 225
Карты 227
Финальная документация 228
Что сохранить, а что выбросить 229
Итеративное уточнение 230
Сначала поведение 230
Проверяем структуру 232
Набрасываем код 234
Эволюция диаграмм 235
Когда и как рисовать диаграммы 236
Когда рисовать диаграммы и когда остановиться 236
CASE-средства 237

Ну а как насчет документации? ... 238
Заключение ... 239
Глава 15. Диаграммы состояний ... 240
Основные понятия ... 241
Специальные события ... 242
Суперсостояния ... 243
Начальное и конечное псевдосостояния ... 245
Использование диаграмм состояний ... 245
Заключение ... 247
Глава 16. Диаграммы объектов ... 248
Мгновенный снимок ... 249
Активные объекты ... 250
Заключение ... 254
Глава 17. Прецеденты ... 255
Записывание прецедентов ... 256
Альтернативные потоки событий ... 257
Что-нибудь еще? ... 258
Представление прецедентов на диаграммах ... 258
Заключение ... 259
Библиография ... 259
Глава 18. Диаграммы последовательности ... 260
Основные понятия ... 261
Объекты, линии жизни, сообщения и прочее ... 261
Создание и уничтожение ... 262
Простые циклы ... 264
Различные сценарии ... 264
Более сложные конструкции ... 268
Циклы и условия ... 268
Сообщения, занимающие время ... 269
Асинхронные сообщения ... 270
Несколько потоков ... 276
Активные объекты ... 276
Отправка сообщений интерфейсам ... 277
Заключение ... 278
Глава 19. Диаграммы классов ... 280
Основные понятия ... 281
Классы ... 281
Ассоциация ... 282
Наследование ... 282
Пример диаграммы классов ... 284

Детали ... 286
Стереотипы классов ... 286
Абстрактные классы ... 288
Свойства ... 288
Агрегирование ... 289
Композиция ... 290
Кратность ... 292
Стереотипы ассоциаций ... 292
Вложенные классы ... 294
Классы ассоциаций ... 294
Квалификаторы ассоциаций ... 295
Заключение ... 295
Библиография ... 296
Глава 20. Эвристика и кофе ... 297
Кофеварка Mark IV Special ... 298
Спецификация ... 298
Типичное, но никуда не годное решение ... 301
Иллюзорная абстракция ... 303
Улучшенное решение ... 305
Реализация абстрактной модели ... 310
Достоинства описанного дизайна ... 317
Объектно-ориентированный перебор ... 318
Библиография ... 331
Часть III. Задача о расчете заработной платы ... 333
Краткая спецификация системы расчета заработной платы ... 334
Упражнение ... 335
Прецедент 1: добавление нового работника ... 335
Прецедент 2: удаление работника ... 336
Прецедент 3: регистрация карточки табельного учета ... 336
Прецедент 4: регистрация справки о продажах ... 336
Прецедент 5: регистрация платежного требования от профсоюза ... 336
Прецедент 6: изменение сведений о работнике ... 337
Прецедент 7: расчет заработной платы на сегодня ... 337
Глава 21. Команда и Активный объект: многогранность и многозадачность ... 338
Простые команды ... 339
Транзакции ... 341
Разрыв физических и темпоральных связей ... 343
Разрыв темпоральных связей ... 343

Метод Undo ... 344
Активный объект ... 345
Заключение ... 350
Библиография ... 350

Глава 22. Шаблонный метод и Стратегия: наследование или делегирование ... 351
Шаблонный метод ... 352
Злоупотребление паттерном ... 355
Пузырьковая сортировка ... 356
Стратегия ... 359
Заключение ... 364
Библиография ... 364

Глава 23. Фасад и Посредник ... 365
Фасад ... 365
Посредник ... 367
Заключение ... 369
Библиография ... 369

Глава 24. Одиночка и Моносостояние ... 370
Одиночка ... 371
Достоинства ... 373
Недостатки ... 373
Одиночка в действии ... 373
Моносостояние ... 375
Достоинства ... 376
Недостатки ... 377
Моносостояние в действии ... 377
Заключение ... 382
Библиография ... 383

Глава 25. Null-объект ... 384
Описание ... 384
Заключение ... 387
Библиография ... 387

Глава 26. Система расчета заработной платы: первая итерация ... 388
Краткая спецификация ... 389
Анализ по прецедентам ... 390
Добавление работников ... 391
Удаление работников ... 392
Регистрация карточки табельного учета ... 393
Регистрация справки о продажах ... 393

Регистрация платежного требования от профсоюза 394
Изменение сведений о работнике 395
Расчетный день 397
Осмысление:
поиск основополагающих абстракций 399
Тарификация работника 400
График выплат 400
Способы платежа 401
Принадлежность к другим организациям 402
Заключение 403
Библиография 403
Глава 27. Система расчета заработной платы: реализация 404
Операции 405
Добавление работников 405
Удаление работников 411
Карточки табельного учета,
справки о продажах и плата за услуги 413
Изменение сведений о работнике 420
Что я курил? 431
Начисление зарплаты 434
Начисление зарплаты работникам с твердым окладом 437
Начисление зарплаты работникам с почасовой оплатой 439
Головная программа 449
База данных 450
Заключение 452
Об этой главе 452
Библиография 453

Часть IV. Пакетирование системы расчета заработной платы 455

Глава 28. Принципы проектирования пакетов и компонентов 456
Пакеты и компоненты 457
Принципы сцепленности компонентов: детальность 458
Принцип эквивалентности повторного использования
и выпуска (Reuse/Release Equivalence Principle – REP) 458
Принцип общей закрытости
(Common Closure Principle – CCP) 461
Резюме 462
Принципы связанности компонентов: устойчивость 462
Принцип ацикличности зависимостей
(Acyclic Dependencies Principle – ADP) 462
Принцип устойчивых зависимостей
(Stable-Dependencies Principle – SDP) 469

Принцип устойчивых абстракций
(Stable-Abstractions Principle – SAP) 474
Заключение 479

Глава 29. Фабрика 480
Проблема зависимости 483
Статическая и динамическая типизация 484
Взаимозаменяемые фабрики 485
Использование фабрик в тестовых фикстурах 485
Важность фабрик 487
Заключение 488
Библиография 488

Глава 30. Система расчета заработной платы: анализ пакетов 489
Структура компонентов и обозначения 490
Применение принципа общей закрытости (CCP) 492
Применение принципа эквивалентности
повторного использования и выпуска (REP) 494
Связанность и инкапсуляция 497
Метрики 498
Применение метрик к задаче о расчете зарплаты 500
Фабрики объектов 504
Еще раз о границах сцепленности 506
Окончательная структура пакетов 507
Заключение 510
Библиография 510

Глава 31. Компоновщик 511
Составные команды 513
Кратный или не кратный 514
Заключение 514

Глава 32. Наблюдатель: превращение в паттерн 515
Цифровые часы 516
Паттерн Наблюдатель 535
Модели 535
Связь с принципами ООП 536
Заключение 537
Библиография 538

Глава 33. Абстрактный сервер, адаптер и мост 539
Абстрактный сервер 540
Адаптер 542
Адаптер класса 543
Задача о модеме, адаптеры и принцип LSP 543
Мост 547

Заключение 550
Библиография 550

Глава 34. Заместитель и Шлюз: управление сторонними API 551
Заместитель 552
Реализация Заместителя 557
Резюме 570
Базы данных, ПО промежуточного уровня и прочие сторонние интерфейсы 571
Шлюз к табличным данным 573
Тестирование и шлюз к табличным данным в памяти 580
Тестирование шлюзов к базе данных 581
Другие паттерны работы с базами данных 584
Заключение 586
Блиблиография 586

Глава 35. Посетитель 587
Посетитель 588
Ациклический посетитель 592
Применения паттерна Посетитель 597
Декоратор 604
Объект расширения 610
Заключение 621
Библиография 621

Глава 36. Состояние 622
Вложенные предложения switch/case 623
Поле State с внутренним доступом 626
Тестирование действий 626
Достоинства и недостатки 627
Таблицы переходов 627
Интерпретация таблицы 628
Достоинства и недостатки 629
Паттерн Состояние 630
Паттерны Состояние и Стратегия 633
Достоинства и недостатки 634
Компилятор конечных автоматов (SMC) 634
Файл Turnstile.cs, сгенерированный SMC, и другие вспомогательные файлы 637
Виды приложений конечных автоматов 642
Высокоуровневые политики приложения и ГИП 642
Контроллеры взаимодействия с ГИП 644
Распределенная обработка 645

Заключение 646
Библиография 646
Глава 37. Система расчета заработной платы: база данных 647
Построение базы данных 648
Изъян в дизайне программы 649
Добавление работника 651
Транзакции 662
Выборка работника 668
Что осталось? 681
Глава 38. Система расчета заработной платы: Модель-Вид-Презентатор 682
Интерфейс 684
Реализация 686
Конструирование окна 696
Окно Payroll 703
Снимаем покрывало 715
Заключение 716
Библиография 716
Приложение A. Сказ о двух компаниях 717
Приложение B. Что такое проектирование программного обеспечения 734
Алфавитный указатель 748

Предисловие

От Криса Селлса

Моим дебютом на поприще профессионального программирования стало добавление некоторых функций в базу данных о вредителях – тле, саранче, гусеницах – по заказу отдела патологии растений экспериментальных ферм университета штата Миннесота. Код был написан энтомологом, который выучил dBase ровно настолько, чтобы суметь написать одну форму, а потом методом копирования распространить ее на все приложение. Расширяя функциональность приложения, я по мере возможности объединял различные функции, так чтобы можно было исправлять ошибки, внося изменения лишь в одном месте, улучшения тоже производить в одном месте и т. д. Это заняло у меня все лето, но в итоге я вдвое расширил функциональность приложения, одновременно вдвое уменьшив объем кода.

Спустя много-много лет у нас с приятелем выдался период, когда не было срочной работы, и мы решили вместе написать какую-нибудь программу (если быть точным, реализацию одного из интерфейсов IDispatch или IMoniker, которые в то время занимали все наши мысли). Сначала я набирал код, а он стоял сзади и указывал мне на ошибки. Потом мы менялись местами, и я поучал его, пока он не возвращал управление мне.

Так продолжалось часами, и лучшего опыта в моей практике программирования не было. Вскоре после этого приятель предложил мне занять место главного архитектора во вновь образованном отделе разработки ПО в его компании. Работая архитектором, я нередко писал клиентский код для еще не существовавших объектов и передавал его инженерам, которые реализовывали объекты так, чтобы этот код заработал.

Полагаю, что мои эксперименты с различными аспектами гибкой разработки не были уникальными, как не были уникальными занятия подростков, изучавших различные любовные техники на заднем сиденье Шевроле 1957 года в те времена, когда половое воспитание еще не стало частью стандартного учебного плана. В общем и целом, мои опыты с такими методами гибкой разработки, как рефакторинг, парное программирование и разработка через тестирование, оказались успешными, хотя я еще не знал, что именно делаю. Конечно, и до выхода этой книги были материалы по гибким методикам, но, образно говоря,

мне совсем не хочется осваивать народные танцы по старым номерам National Geographic. Технологии гибкой разработки интересуют меня лишь применительно к платформе .NET, с которой работает моя группа. Рассказывая о .NET, Роберт (хотя он ясно дает понять, что во многих случаях .NET ничем не лучше Java) говорит на моем языке, как те преподаватели старших классов, которые давали себе труд изучать слэнг подростков, понимая, что сообщение важнее носителя.

Но дело не только в .NET; я хотел бы, чтобы начало изучения было ненавязчивым, неторопливым, не отпугивало меня, но чтобы при этом я узнал все то, что знать необходимо. Именно так Роберт «Дядя Боб» Мартин и организовал эту книгу. В первых главах основы гибкой разработки излагаются постепенно, не заставляя читателя очертя голову бросаться в омут SCRUM, экстремального программирования и других конкретных методик, а давая возможность атомам собираться в такие молекулы, где они чувствуют себя наиболее комфортно. Но еще больше мне нравится в стиле Роберта то, как он демонстрирует описываемые приемы в действии. Взяв некоторую реальную задачу, он показывает, какие ошибки и ложные ходы можно допустить в ходе ее решения и как применение методов, за которые он ратует, позволяет вернуться на твердую почву.

Не знаю, существует ли в реальности мир, описываемый Робертом в этой книге; за всю мою жизнь мне удалось лишь мельком взглянуть на него. Но совершенно ясно, что все «крутые ребята» живут именно в нем. Считайте «дядю Боба» своим личным «доктором Рут»[1] в мире гибкой разработки, чья единственная цель – научить вас выполнять эту работу хорошо, чтобы все чувствовали удовлетворенность своей деятельностью.

Крис Селлс

От Эриха Гаммы

Я пишу это предисловие сразу после выпуска очередной основной версии проекта Eclipse с открытым исходным кодом. Я еще не оправился от напряженного труда, и мозг немного затуманен. Но одну вещь я понимаю яснее, чем когда-либо: ключ к своевременной поставке продукта – не процессы, а люди. Наш рецепт успеха прост: работать с людьми, по-настоящему увлеченными программированием, применять в разработке упрощенные процессы, подстроенные под конкретную команду, и непрестанно адаптироваться.

Если «дважды щелкнуть» по любому разработчику из наших команд, то обнаружится личность, считающая программирование центром разработки. Они не просто пишут код, а еще и постоянно продумывают его с позиций понимания работы системы в целом. Проверка проекта с по-

[1] Рут Вестхаймер, американский сексопатолог, ведущая теле- и радиопередач, автор множества книг. – *Прим. перев.*

мощью кода дает обратную связь, без которой невозможно получить уверенность в отсутствии ошибок проектирования. Но в то же время наши разработчики понимают важность паттернов, рефакторинга, тестирования, инкрементной поставки, частого выполнения сборки и прочих методик экстремального программирования, изменивших наш взгляд на методологию разработки ПО.

Навыки в таком стиле разработки – необходимое условие успеха в проектах с высоким техническим риском и изменяющимися требованиями. Гибкая разработка молчит, когда речь идет о формальностях и проектной документации, но возвышает голос, когда дело доходит до повседневных практических методик, которые действительно имеют значение. Собрать эти методики воедино и заставить их работать – вот цель данной книги.

Роберт уже давно является активным членом сообщества объектно-ориентированного программирования, внесшим немалый вклад в практическое применение C++, паттерны проектирования и принципы объектно-ориентированной разработки в целом. Он с самого начала активно отстаивал методы экстремального программирования и гибкой разработки. Эта книга, основанная на его богатом опыте, охватывает все аспекты практического применения гибкой разработки. Амбициозная задача, ничего не скажешь. Решая ее, Роберт приводит разнообразные примеры, сопровождая их кодом, как и подобает адепту гибкой разработки. Он учит программированию и проектированию на практике.

Эта книга буквально напичкана мудрыми советами, как разрабатывать ПО. Она с равным успехом подойдет и тому, кто еще только собирается практиковать гибкую разработку, и тому, кто желает усовершенствовать уже имеющиеся навыки. Я с нетерпением ждал ее выхода и не был разочарован.

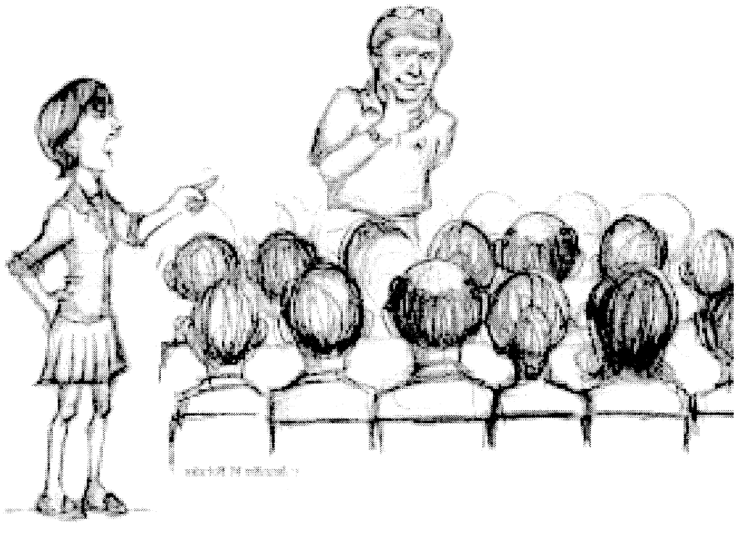
Эрих Гамма,
Object Technology International

Об авторах

Роберт К. Мартин («Дядя Боб») – основатель и президент международной компании Object Mentor Inc. со штаб-квартирой в Гурни, штат Иллинойс, предлагающей консультативные услуги по совершенствованию процесса разработки, объектно-ориентированному проектированию, обучению и повышению квалификации разработчиков крупным компаниям по всему миру. Он автор книг «Designing Object Oriented C++ Applications Using the Booch Method» и «Agile Software Development Principles, Patterns, and Practices» (обе вышли в издательстве Prentice Hall), а также «UML for Java Programming» (Addison-Wesley). В период с 1996 по 1999 год был главным редактором журнала «C++ Journal». Известен своими выступлениями на международных конференциях и промышленных выставках.

Мика Мартин трудится в компании Object Mentor в качестве разработчика, консультанта и наставника по различным предметам, начиная от объектно-ориентированных принципов и паттернов и кончая методиками гибкой разработки ПО. Мика – сооснователь и ведущий разработчик проекта FitNesse с открытым исходным кодом. Он также автор ряда печатных работ, регулярно выступает на конференциях.

Введение



Но Боб, ты же говорил, что закончишь книгу в ***прошлом*** *году.*

Клаудиа Фрерс, UML World, 1999

От Боба

Прошло уже семь лет с момента заявления этой справедливой претензии Клаудии, и думается, что я искупил свою вину. Опубликовать *три* книги – по одной каждые два года – и при этом руководить консалтинговой фирмой, писать уйму кода, преподавать, обучать, выступать на конференциях, писать статьи, вести блоги и колонки в разных журналах, не говоря уже об обязанностях по отношению к своей семье и родителям, – это нелегко. Но мне нравится такая жизнь.

Гибкая разработка – это умение быстро разрабатывать ПО в условиях постоянно изменяющихся требований. Чтобы достичь такой гибкости, необходимо освоить методики, диктующие определенную дисциплину и установление обратной связи. Мы должны применять принципы проектирования, благодаря которым программы будут оставаться гибки-

ми и удобными для сопровождения. Нам также нужны проверенные практикой паттерны проектирования, которые позволяют применить эти принципы к конкретным проблемам. Эта книга – попытка связать все три концепции в единое работающее целое.

Сначала описываются принципы, паттерны и методики, а затем на десятках различных примеров демонстрируется их применение. Важнее, однако, что примеры представлены в виде не завершенных работ, а разворачивающегося *процесса проектирования*. Вы увидите, что проектировщики тоже совершают ошибки, станете свидетелями того, как они выявляют эти ошибки и в конечном итоге исправляют их. Вы увидите, как проектировщики бьются над головоломными задачами, как они добиваются ясности и определенности и как вырабатывают компромиссы. Вы увидите само *действо* проектирования.

От Мики

В начале 2005 года я был членом небольшой группы разработчиков, которая начала работать над приложением .NET на языке C#. Обязательным условием было применение гибких методик разработки, в частности поэтому меня и пригласили. Хотя мне доводилось раньше писать на C#, лучше всего я владел языками Java и C++. Я не думал, что работа на платформе .NET будет чем-то существенно отличаться; так оно и оказалось.

Через два месяца работы над проектом была выпущена первая версия. Она содержала лишь часть запланированных функций, но при этом была в достаточной мере работоспособной. И с ней таки работали. Спустя всего два месяца организация-заказчик уже пожинала плоды наших трудов. Руководство было настолько поражено, что попросило нанять еще людей, чтобы мы могли приступить к новым проектам.

Принимая на протяжении многих лет участие в жизни сообщества гибкой разработки, я был знаком со многими приверженцами этой технологии, которые могли бы нам помочь. Я предложил им присоединиться к проекту. Но ни один из моих знакомых так и не влился в команду. Почему? Быть может, по той простой причине, что мы вели разработку на платформе .NET.

Почти все практики гибкой разработки имели опыт работы на Java, C++ или Smalltalk. Но о гибкой разработке в .NET тогда почти не слыхали. Возможно, мои друзья не приняли всерьез мои слова о том, что мы занимаемся гибкой разработкой в среде .NET, а быть может, не захотели связываться с .NET. Это оказалось серьезной проблемой. И я столкнулся с ней не впервые.

Проведение недельных курсов по различным аспектам программирования позволяет мне встречаться с самыми разными разработчиками по всему миру. Многие мои слушатели работали с .NET, но не меньше было

и программистов на Java или C++. И буду откровенен: мой опыт показывает, что программисты .NET зачастую слабее тех, что пишут на Java или C++. Понятно, что бывают исключения. Однако раз за разом наблюдая за своими слушателями, я вынужден был заключить, что программисты .NET обычно хуже разбираются в методах разработки ПО, паттернах и принципах проектирования и т. п. Нередко случалось, что присутствовавшие программисты .NET вообще не слыхали об этих фундаментальных концепциях. *Такое положение должно быть изменено.*

Первое издание этой книги, «Agile Software Development: Principles, Patterns, and Practices», написанной моим отцом, Робертом К. Мартином, вышло в 2002 году и получило премию Jolt Award 2003 года. Это замечательная книга, восторженно принятая многими разработчиками. К сожалению, она не оказала большого влияния на сообщество разработчиков в среде .NET. Хотя изложенные в книге идеи вполне применимы к .NET, немногие программирующие на этой платформе прочли ее.

Я надеюсь, что это издание, ориентированное на .NET, перекинет мост между разработчиками .NET и остальным сообществом. Я надеюсь, что программисты прочтут ее и увидят, что существуют лучшие способы построения программ. Я надеюсь, что они начнут использовать лучшие методики, создавать лучшие проекты и поднимут планку качества при разработке .NET-приложений. Я надеюсь, что программисты .NET завоюют новый статус в сообществе разработчиков, так что программисты на Java смогут с гордостью присоединиться к команде, работающей на платформе .NET.

Работая над этой книгой, я не раз сомневался, ставить ли свое имя на обложке книги, посвященной .NET. Я спрашивал себя, хочу ли я, чтобы меня ассоциировали с .NET и со всеми негативными представлениями, связанными с этой платформой. Но что толку отрицать? Я программист .NET. Нет! Я гибкий программист .NET. И горжусь этим.

Об этой книге

Немного истории

В начале 1990-х годов я (Боб) написал книгу «Designing Object-Oriented C++ Applications Using the Booch Method». Для меня она стала чем-то вроде magnum opus[1], и я был очень доволен результатом и тем, как книга продавалась.

Книга, которую вы сейчас держите в руках, была задумана как второе издание «Designing», но все обернулось иначе. От оригинальной книги осталось очень мало. В новую перешло чуть меньше трех глав, да и этот материал подвергся существенной переработке. Цель книги, характер

[1] Труд всей жизни. – *Прим. перев.*

изложения и многие уроки остались прежними. Но за десять лет, прошедших с момента выхода «Designing», я узнал очень много нового о проектировании и разработке ПО. И этот новый опыт нашел отражение в книге.

Ах, что это было за десятилетие! Книга «Designing» вышла еще до того, как Интернет покорил всю планету. С тех пор количество акронимов, с которыми мы сталкиваемся, удвоилось. EJB, RMI, J2EE, XML, XSLT, HTML, ASP, JSP, ZOPE, SOAP, C# и .NET... А к ним в придачу паттерны проектирования, Java, сервлеты и серверы приложений. Поверьте, наполнить главы этой книги актуальным материалом было непросто.

Сотрудничество с Бучем. В 1997 году Гради Буч попросил меня помочь ему в работе над третьим изданием его поразительно успешной книги «Object-Oriented Analysis and Design with Applications»[1]. Я и раньше работал с Гради над некоторыми проектами и с удовольствием читал его труды, в частности о UML, и вносил в них свой посильный вклад. Поэтому я с восторгом согласился и попросил своего старого друга Джима Ньюкирка присоединиться к этому проекту.

В последующие два года мы с Джимом написали несколько глав для книги Буча. Конечно, из-за этого я не смог уделять своей книге столько времени, сколько хотел бы, но мне казалось, что книга Буча стоит затраченных усилий. К тому же в то время эта книга представлялась мне просто вторым изданием «Designing», и душа у меня к ней не лежала. Уж если что-то говорить, то новое и незатертое.

К сожалению, книга Буча так и не вышла. Трудно найти время для написания книги без отрыва от основной работы. А уж в те годы стремительного развития интернет-компаний это было почти невозможно. Гради был безумно занят компанией Rational и другими венчурными предприятиями, такими как Catapulse. Поэтому проект застопорился. Я испросил у Буча и издательства Addison-Wesley разрешение включить написанные мною и Джимом главы в эту книгу. Они любезно согласились. Так появились некоторые примеры и главы, посвященные UML.

Влияние экстремального программирования. В конце 1998 года возникло экстремальное программирование (XP) и тут же бросило вызов нашим устоявшимся взглядам на разработку программного обеспечения. Нужно ли рисовать кучу UML-диаграмм перед тем, как приступить к написанию кода? Или лучше отказаться от всех диаграмм и просто писать код, много кода? Нужно ли составлять объемную документацию, в которой описывается проект? Или лучше сделать сам *код* настолько самодокументированным и ясным, что никакие дополнительные документы не понадобятся? Нужно ли программировать

[1] **Гради Буч «Объектно-ориентированный анализ и проектирование с примерами приложений на C++». – Пер. с англ. – Бином, Невский Диалект, 1998.**

вдвоем? Нужно ли писать тесты еще до написания основного кода? Что вообще нужно делать?

Для этой революции был выбран самый подходящий момент. С середины и до конца 1990-х годов фирма Object Mentor консультировала различные компании по вопросам объектно-ориентированного проектирования и управления проектами. Мы помогали реализовывать проекты, в частности внедряли в коллективы разработчиков свои представления и методы. К сожалению, они не были сформулированы в письменном виде. Это была просто устная традиция, передававшаяся от нас заказчикам.

К 1998 году я осознал, что необходимо описать наши процессы и методики, чтобы было проще излагать их заказчикам. Тогда я написал много статей в журнал *C++ Report*[1]. Но это был выстрел мимо цели. Статьи были информативными и даже иногда занимательными, но вместо того чтобы систематизировать те представления и методы, которые использовались нами в проектах, они оказались невольным компромиссом с теми концепциями, которые были навязаны мне за последние десятилетия. Указал мне на это Кент Бек.

Сотрудничество с Беком. В конце 1998 года, как раз тогда, когда я занимался систематизацией процесса разработки, принятого в компании Object Mentor, я наткнулся на работу Кента по экстремальному программированию (XP). Я нашел ее на сайте википедии[2] Уорда Каннингэма среди статей многих других авторов. И немного потрудившись, мне удалось вычленить основную мысль Кента. Я был заинтригован, но отнесся к ней скептически. Кое-какие положения XP идеально ложились на мою концепцию процесса разработки. Но некоторые моменты, например отсутствие четких шагов проектирования, озадачивали.

Вероятно, дело в том, что мы с Кентом работали в разных условиях. Он был известным консультантом по языку Smalltalk, я – не менее известным консультантом по C++. Общение между этими двумя мирами затруднено. Между обеими парадигмами пролегает почти кунианская[3] пропасть.

---

[1] Эти статьи можно найти в разделе *Publications* на сайте www.objectmentor.com. Всего их четыре. Первые три озаглавлены «Iterative and Incremental Development» (I, II и III), последняя – «C.O.D.E Culled Object Development process».

[2] На сайте *http://c2.com/cgi/wiki* размещено множество статей на самые разные темы. Количество авторов исчисляется сотнями или тысячами. Говорили, что только Уорд Каннингэм способен вызвать социальную революцию, написав всего несколько строк на Perl.

[3] Термин *«кунианский»* должен был присутствовать в каждом значимом интеллектуальном труде, написанном между 1995 и 2001 годами. Он восходит к книге Томаса С. Куна (Thomas S. Kuhn) «The Structure of Scientific Revolutions», издательство University of Chicago Press, 1962.

При таких обстоятельствах я никогда не попросил бы Кента написать статью в журнал *C++ Report*. Но схожесть наших размышлений о процессе разработки помогла преодолеть языковую пропасть. В феврале 1999 года я встретился с Кентом в Мюнхене на конференции по ООП. Он рассказывал об XP в аудитории, расположенной как раз напротив той, где я докладывал о принципах объектно-ориентированной разработки. Поскольку послушать его доклад мне не удалось, то я разыскал Кента во время обеденного перерыва. Мы поговорили об XP, и я предложил ему написать статью в *C++ Report*. Это была блистательная статья о реальном случае, когда Кенту вместе с коллегой удалось внести кардинальное изменение в проект работающей системы, затратив на это около часа.

На протяжении нескольких следующих месяцев я медленно избавлялся от собственных страхов по поводу XP. Самым пугающим мне представлялось отсутствие явного этапа проектирования, предшествующего всему остальному. С этим я никак не мог примириться. Разве не считал я всегда своей обязанностью перед заказчиками и индустрией в целом разъяснять, что проектирование – вещь достаточно важная, чтобы тратить на нее время?

Но в конце концов я осознал, что и сам-то так не поступаю. Да, во всех своих статьях и книгах я писал о проектировании, о диаграммах Буча и UML-диаграммах, но при этом всегда использовал код как средство проверки осмысленности диаграмм. Консультируя своих заказчиков, я тратил час-другой на то, чтобы помочь им нарисовать диаграммы, а потом показывал, как наполнить их кодом. Я начал понимать, что, хотя фразеология XP в части проектирования выглядела для меня чуждой в кунианском смысле[1], стоящая за словами практика была мне знакома.

С другими опасениями, касающимися XP, справиться оказалось легче. Я всегда был тайным любителем парного программирования. XP дало мне возможность сбросить покров тайны и открыто заявить о своем желании программировать вместе с партнером. Рефакторинг, непрерывная интеграция, работа совместно с заказчиком – мне было нетрудно принять все это, поскольку такие идеи были очень близки к тому, что я и сам советовал своим клиентам.

Одна из методик XP стала для меня откровением. Идея разработки через тестирование (Test-driven development – TDD)[2] на первый взгляд

---

[1] Если имя Куна упомянуто в статье дважды, вы получаете дополнительные очки.

[2] Kent Beck «Test-Driven Development by Example», Addison-Wesley, 2003. (Кент Бек «Экстремальное программирование: разработка через тестирование». – Пер. с англ. – Питер, 2003.)

звучит совершенно безобидно: пишите тесты перед созданием основного кода. А основной код следует писать так, чтобы ранее не проходившие тесты завершались успешно. Эта идея полностью изменила мой подход к написанию программ – и изменила в лучшую сторону.

Таким образом, к осени 1999 года я пришел к убеждению, что компания Object Mentor должна принять XP в качестве основного процесса разработки и что мне следует отказаться от желания описать свой собственный процесс. Кент уже проделал отличную работу по облечению в слова методики и процесса XP, мои робкие потуги бледнели перед этим трудом.

.NET. Между крупнейшими корпорациями идет война. Корпорации борются за то, чтобы завоевать *ваши* симпатии. Корпорации полагают, что, владея неким языком, они владеют и программистами, пишущими на этом языке, и компаниями, в которых эти программисты работают.

Первым залпом в этой битве стал язык Java. Этот язык был специально создан крупной корпорацией с намерением завоевать умы программистов. Его успех оказался ошеломительным. Java глубоко проник в программистское сообщество и стал стандартом де факто для разработки современных многоуровневых ИТ-приложений.

Ответный залп последовал от корпорации IBM, которая благодаря среде разработки Eclipse отхватила значительный сегмент рынка Java. Открыли огонь и непревзойденные стратеги в корпорации Microsoft, которые подарили нам платформу .NET вообще и язык C# в частности.

Забавно, что провести различие между Java и C# весьма затруднительно. Оба языка семантически эквивалентны, а синтаксически настолько схожи, что многие фрагменты кода выглядят совершенно одинаково. Но если Microsoft и отстает в плане технической новизны, то с лихвой компенсирует это удивительной способностью играть в «догонялки» и выигрывать.

В первом издании этой книги в качестве языков программирования использовались Java и C++. А это издание ориентировано на язык C# и платформу .NET. Но не следует считать это сменой пристрастий. Мы не примыкаем ни к одной из воюющих сторон. Вообще, я полагаю, что война затихнет сама собой, когда через несколько лет появится какой-нибудь лучший язык и привлечет на свою сторону программистов, которых враждующие корпорации так старались удержать.

А причина появления версии этой книги для .NET в том, чтобы охватить аудиторию, работающую на этой платформе. Хотя сами описываемые принципы, паттерны и методики от языка не зависят, сказать то же самое о примерах уже нельзя. Программистам .NET привычнее читать код примеров на .NET-совместимом языке, а программистам Java – код на Java.

Дьявол кроется в деталях



В этой книге приведено *очень много* кода. Мы надеемся, что вы будете читать его внимательно, потому что в значительной степени именно код и является *сутью* данной книги. Именно в коде воплощено все, что мы хотели передать вам в этой книге.

Книга представляет собой последовательность примеров разного объема. Некоторые совсем небольшие, другие занимают несколько глав. Каждому примеру предшествует материал, имеющий целью подготовить читателя: описание принципов и паттернов объектно-ориентированного проектирования, используемых в данном примере.

Книга начинается с обсуждения методик и процессов разработки. Это обсуждение завершается рядом небольших примеров. Далее мы переходим к теме проектирования и его принципов, затем – к некоторым паттернам, потом снова к принципам – управления пакетами, и опять к паттернам. Изложение неизменно сопровождается конкретными примерами.

Поэтому приготовьтесь читать код и разбираться в UML-диаграммах. Книга, на которую вы сейчас смотрите, содержит *преимущественно* техническую информацию, и ее уроки, как и дьявол, кроются в деталях.

Структура книги

Книга состоит из четырех разделов и двух приложений.

В части I «Гибкая разработка» описывается идея гибкой разработки. Она начинается с *Манифеста гибкой разработки*, затем дается обзор экстремального программирования (XP), после чего на ряде небольших примеров иллюстрируются некоторые приемы XP, в особенности те, что влияют на способы проектирования и написания кода.

В части II «Гибкое проектирование» речь пойдет об объектно-ориентированном проектировании ПО: что это такое, постановка задачи об управлении сложностью и методы ее решения, принципы объектно-ориентированного проектирования классов. Завершается эта часть несколькими главами, посвященными описанию использования подмножества UML на практике.

В части III «Задача о расчете заработной платы» описывается объектно-ориентированный проект и реализация на C# простой пакетной системы расчета заработной платы. В начальных главах мы рассказываем о паттернах проектирования, встречающихся в этом примере. А последнюю главу занимает полный пример – самый большой и сложный в этой книге.

Часть IV «Пакетирование системы расчета заработной платы» начинается с описания *принципов проектирования объектно-ориентированных пакетов*, после чего мы переходим к иллюстрации этих принципов на примере постепенной компоновки в пакеты классов из предыдущего раздела. Завершается часть главами, касающимися проектирования базы данных и пользовательского интерфейса для приложения «Система расчета заработной платы».

В книге есть два приложения: «Сказ о двух компаниях» и статья Джека Ривза «Что такое проектирование программного обеспечения».

Как читать эту книгу

Если вы разработчик, то читайте книгу от корки до корки. Она написана преимущественно для разработчиков и содержит информацию о том, как писать программы, применяя гибкие методики. Читая книгу последовательно, вы сначала ознакомитесь с методиками, затем с принципами, с паттернами и наконец с примерами, где все это увязано воедино. Усвоение всех этих знаний поможет вам довести проект до успешного завершения.

Если вы менеджер или бизнес-аналитик, то прочитайте часть I «Гибкая разработка». В главах 1–6 вы найдете углубленное обсуждение принципов и методик гибкой разработки: от анализа требований до планирования, тестирования, рефакторинга и программирования. В части I приведены рекомендации по созданию команд и управлению проектами. Они помогут вам довести проект до успешного завершения.

Если вы хотите изучить UML, то начните с глав 13–19. Затем прочитайте все главы части III «Задача о расчете заработной платы». При такой последовательности чтения вы получите основательные знания о синтаксисе и использовании UML и сумеете перевести проект с языка диаграмм UML на C#.

Если вы хотите узнать о паттернах проектирования, прочитайте сначала материал о принципах проектирования в части II «Гибкое проектирование», а затем часть III «Задача о расчете заработной платы» и часть IV «Пакетирование системы расчета заработной платы». В них описаны все паттерны и показано, как применять их в типичных ситуациях.

Если вы хотите узнать о принципах объектно-ориентированного проектирования, прочитайте часть II «Гибкое проектирование», часть III «Задача о расчете заработной платы» и часть IV «Пакетирование системы расчета заработной платы». В них описаны принципы объектно-ориентированного проектирования и показано, как применять их на практике.

Если вы хотите узнать о методах гибкой разработки, прочитайте часть I «Гибкая разработка». Здесь описана методология гибкой разра-

ботки от анализа требований до планирования, тестирования, рефакторинга и программирования.

Если вы хотите немного развлечься, то прочитайте приложение А «Сказ о двух компаниях».

Благодарности

Мы благодарны Лоуэллу Линдстрому (Lowell Lindstrom), Брайану Баттону (Brian Button), Эрику Миду (Erik Meade), Майку Хиллу (Mike Hill), Майклу Фэзерсу (Michael Feathers), Джиму Ньюкирку (Jim Newkirk), Мике Мартину (Micah Martin), Анжелике Мартин (Angelique Martin), Сьюзен Россо (Susan Rosso), Талише Джефферсон (Talisha Jefferson), Рону Джеффрису (Ron Jeffries), Кенту Беку (Kent Beck), Джеффу Лэнгру (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 Эпту (Alan Apt), Полу Ходжеттсу (Paul Hodgetts), Филу Маркграфу (Phil Markgraf), Питу Макбрину (Pete McBreen), Х. С. Леману (H. S. Lahman), Дэйву Харрису (Dave Harris), Джеймсу Канзе (James Kanze), Марку Уэбстеру (Mark Webster), Крису Бигею (Chris Biegay), Алану Фрэнсису (Alan Francis), Джессике Д'Амико (Jessica D'Amico), Крису Гузиковски (Chris Guzikowski), Полу Петралиа (Paul Petralia), Мишель Хаусли (Michelle Housley), Дэвиду Челимски (David Chelimsky), Полу Пагелю (Paul Pagel), Тиму Оттингеру (Tim Ottinger), Кристоферу Хэдгейту (Christoffer Hedgate) и Нилу Рудину (Neil Roodyn).

Особая благодарность Гради Бучу и Полу Беккеру за разрешение включить главы, первоначально предназначавшиеся для третьего издания книги Гради *«Object-Oriented Analysis and Design with Applications»*. Также особая благодарность Джеку Ривзу за любезное разрешение воспроизвести текст его статьи «Что такое проектирование программного обеспечения».

Прекрасные, а иногда просто восхитительные иллюстрации принадлежат Дженнифер Конке (Jennifer Kohnke) и моей дочери Анджеле Брукс (Angela Brooks).

I

Гибкая разработка



Человеческие взаимоотношения сложны
и никогда не обходятся без шероховатостей,
но значат они больше, чем любая другая сторона работы.

Том Демарко и Тимоти Листер, «Peopleware»[1]

[1] Демарко Т., Листер Т. «Человеческий фактор. Успешные проекты и команды». – Пер. с англ. – Символ-Плюс, 2005.

Принципы, паттерны и методики важны, но в действие их приводят люди. Как говорил Алистер Кокбэрн[1], «процесс и технология оказывают на результат проекта эффект второго порядка. На первом месте стоят люди»[2].

Мы не можем управлять коллективами программистов, как будто это системы, составленные из компонентов в соответствии с некоторым технологическим процессом. По выражению Алистера Кокбэрна, люди – не «взаимозаменяемые программные модули». Чтобы проект был успешным, необходимо создавать самоорганизующиеся команды из готовых к сотрудничеству индивидов.

Компании, поощряющие формирование таких команд, будут иметь *гигантское* конкурентное преимущество перед теми, кто продолжает считать, что организация разработки ПО – не более чем сваливание в одну кучу мелких, неотличимых друг от друга людишек. Спаянный коллектив разработчиков – самый эффективный из всех существующих способов организации разработки ПО.

[1] Один из инициаторов движения за гибкую разработку ПО. – *Прим. перев.*

[2] Из частной беседы.

1

Гибкие методики



Петух на церковном шпиле, хоть и железный,
быстро сломался бы под ударами ветра,
если бы не овладел высоким искусством
поворачиваться, куда ветер дует.

Генрих Гейне

Многим из нас довелось пережить кошмар проекта, разрабатывавшегося без какой-либо методики. Отсутствие эффективной методики ведет к непредсказуемости, повторению одних и тех же ошибок и пустой трате сил. Заказчики недовольны постоянно переносимым графиком сдачи, растущим бюджетом и низким качеством. Разработчики приходят в уныние от того, что приходится сидеть на работе все дольше, а продукт становится только хуже.

Однажды потерпев такое фиаско, мы начинаем опасаться повторения подобного опыта. И потому возникает мысль создать технологический процесс, который налагал бы на нашу деятельность ограничения и заставлял производить определенные результаты и артефакты. Эти ограничения и результаты мы заимствуем из прошлого опыта, отбирая то, что доказало свою пригодность в предыдущих проектах. Мы надеемся, что эти механизмы будут работать и дальше, избавляя нас от страха.

Но проекты не настолько просты, чтобы немногие ограничения и артефакты могли надежно предотвратить ошибку. Ошибки продолжают возникать, мы их обнаруживаем и вводим все новые ограничения и артефакты, чтобы воспрепятствовать их появлению в будущем. По мере роста числа завершенных проектов у нас на руках оказывается тяжеловесный, неповоротливый процесс, который только мешает работать.

Такой громоздкий процесс способен лишь создавать те проблемы, для решения которых он был задуман. Он может замедлить работу команды до такой степени, что придется сдвигать график и увеличивать бюджет. Он может снизить способность команды реагировать на изменения настолько, что она всегда будет создавать не то, что нужно. К сожалению, многие команды из-за этого приходят к выводу, что процесс недостаточно всеобъемлющ. И потому делают его еще объемнее, что приводит к неконтролируемому разбуханию.

Неконтролируемое разбухание процесса – неплохое описание того состояния, в котором оказались многие компании по производству ПО где-то около 2000 года. Хотя многие команды все еще работали без всякого процесса, принятие объемных, тяжеловесных процессов быстро набирало темпы, особенно в крупных корпорациях.

Организация Agile Alliance

Наблюдая за тем, как команды разработчиков во многих фирмах и корпорациях тонули в трясине непрестанно расширяющихся процессов, группа экспертов, назвавшаяся *Agile Alliance* (сообщество профессионалов, занимающихся гибкими методологиями разработки), собралась в 2001 году, чтобы обсудить ту шкалу ценностей и принципы, которые позволили бы коллективам программистов быстро вести разработку и оперативно реагировать на изменения. В результате появился *Манифест гибкой разработки*.

Манифест гибкой разработки программ

Мы открываем лучшие способы разработки программного обеспечения, применяя их на практике и помогая в этом другим. В ходе своей работы мы научились ценить:

Людей и их взаимоотношения больше, чем процессы и инструменты.

Работающую программу больше, чем исчерпывающую документацию.

Плодотворное сотрудничество с заказчиком больше,
чем формальные договоренности по контракту.

Оперативное реагирование на изменения больше,
чем следование плану.

Иначе говоря, не отрицая значимость факторов, перечисленных в правой части предложений, мы больше ценим те, что слева.

Кент Бек	Джеймс Греннинг	Роберт К. Мартин
Майк Бидл	Джим Хайсмит	Стив Меллор
Ари ван Беннекум	Эндрю Хант	Кен Швабер
Алистер Кокбэрн	Рон Джеффрис	Джефф Сазерленд
Уорд Каннингэм	Йон Керн	Дэйв Томас
Мартин Фаулер	Брайан Мэрик	

Люди и их взаимоотношения важнее процессов и инструментов

Люди – важнейшая составная часть успеха. Самый лучший процесс не спасет проект от провала, если в команде нет сильных игроков. Однако плохой процесс может сделать даже сильнейших игроков неэффективными. И даже коллектив, состоящий сплошь из сильных игроков, может потерпеть фиаско, если его члены не будут работать как единая команда.

Сильный игрок – не обязательно ас программирования. Сильным игроком может быть средний программист, умеющий хорошо работать вместе с другими. Умение работать с коллегами – общаться и взаимодействовать – важнее, чем талант к программированию сам по себе. Команда средних программистов, умеющих работать совместно, скорее добьется успеха, чем группа суперзвезд, не способных быть командой.

Для успеха может оказаться чрезвычайно важен выбор подходящих инструментов. Компиляторы, интерактивные среды разработки (IDE), системы управления версиями исходного кода и пр. – все это существенно для функционирования команды разработчиков. Но значимость инструментов не стоит переоценивать. Чрезмерное изобилие больших и громоздких инструментов столь же плохо, как и их нехватка.

Наш совет – начинать с малого. Не считайте, что вы переросли инструмент, пока не опробовали его и не поняли, что он вам не подходит. Вместо того чтобы покупать новейшую супердорогую систему управления версиями, найдите бесплатную и работайте с ней, пока не станет ясно, что имеющихся в ней функций уже не хватает. Прежде чем приобретать групповую лицензию на самую лучшую систему автоматизированной разработки программ (CASE), попробуйте поработать с доской и миллиметровкой, пока не убедитесь, что вам нужно нечто большее.

Прежде чем прибегать к самой новой монструозной СУБД, попробуйте плоские файлы. Не думайте, что стоит заиметь большой и хороший инструмент, как ваша работа автоматически улучшится. Зачастую такие инструменты больше мешают, чем помогают.

Помните, что создание команды важнее, чем создание среды разработки. Многие команды и менеджеры впадают в одну и ту же ошибку: первым делом готовят среду и думают, что команда спаяется сама собой. А следовало бы сначала собрать команду, а потом позволить ей сконфигурировать среду так, как она сочтет нужным.

Работающая программа важнее исчерпывающей документации

Программа без документации – это ужас. Код – не самое подходящее место для описания назначения и структуры системы. Команда обязана подготовить понятные человеку документы, в которых описывалась бы система и обосновывались принятые проектные решения.

Однако слишком много документации хуже, чем слишком мало. На создание огромных томов документации уходит много времени и еще больше – на синхронизацию их с кодом. Если же документация не соответствует коду, то становится большой и запутанной ложью и источником дезинформации.

Очень неплохо, когда команда составляет и поддерживает в актуальном состоянии короткий документ, содержащий обоснования решений и описание структуры. Но он должен быть кратким и существенным. Под *«кратким»* я понимаю десяток-другой страниц, не более. Говоря *«существенный»*, я имею в виду, что в документе рассматриваются только общие особенности проекта и системные структуры самого верхнего уровня.

Но если у нас есть только краткий документ с обоснованием и описанием структуры, то как ввести новых членов команды в курс дела? Просто надо тесно сотрудничать с ними. Передавать знания, сидя рядом и оказывая необходимую помощь. Новые люди становятся членами команды в результате персонального обучения и личного общения.

Два документа, лучше всего приспособленных для передачи информации новым членам команды, – это код и сама команда. Код не лжет относительно того, что делает. Извлечь обоснование и назначение из кода не всегда легко, но только код может служить точным источником информации. Постоянно изменяющееся устройство системы находится в головах членов команды. Самый быстрый и эффективный способ перенести это устройство на бумагу и передать другим – общение между людьми.

Многие команды застревали на полпути, стремясь создать документацию вместо программы. Зачастую эта ошибка бывает фатальной. Вот простое правило, позволяющее избежать ее:

Первый закон документирования Мартина

Не создавайте документ, пока необходимость в нем не станет очевидной и срочной.

Плодотворное сотрудничество с заказчиком важнее формальных договоренностей по контракту

Программу нельзя заказать, как предмет потребления. Невозможно составить описание желаемой программы и надеяться, что кто-то разработает ее в оговоренные сроки по оговоренной цене. Сколько раз попытки трактовать программные проекты подобным образом терпели крах. Иногда неудачи оказывались весьма зрелищными.

У руководителей компаний возникает большое искушение сообщить разработчикам, что нужно сделать, и считать, что спустя некоторое время они явятся с готовой системой, удовлетворяющей всем требованиям. Но такой подход чреват низким качеством и срывом проекта.

Чтобы проект оказался успешным, необходимо регулярное и частое общение с заказчиком. Заказчик программы должен не полагаться на контракт или техническое задание, а плотно контактировать с командой разработчиков, регулярно высказывая свое мнение о том, что они сделали.

Контракт, в котором оговорены требования, сроки и стоимость проекта, – это фундаментальная ошибка. В большинстве случаев сформулированные в нем условия теряют смысл задолго до завершения проекта, а иногда даже задолго до подписания контракта! Самые лучшие контракты – те, что оговаривают способы взаимодействия заказчика с разработчиками.

Примером успешного контракта может служить тот, что я заключил в 1994 году на разработку большого, многолетнего проекта стоимостью в полмиллиона долларов. Нам, команде разработчиков, платили ежемесячно по сравнительно низкой ставке. Крупные платежи поступали, когда мы передавали в эксплуатацию большие функциональные блоки. Эти блоки не были детально специфицированы в контракте. Вместо этого в контракте оговаривалось, что блок будет оплачен после прохождения приемо-сдаточного испытания. Детали таких испытаний также не были оговорены в контракте.

По ходу работы над проектом мы тесно сотрудничали с заказчиком. Версии ПО выпускались чуть ли не каждую пятницу. К понедельнику или вторнику следующей недели мы получали список изменений, который следовало внести в программу. Мы совместно расставляли приоритеты и составляли график реализации изменений на следующие недели. Заказчик так плотно контактировал с нами, что приемо-сдаточные испытания никогда не вызывали никаких проблем. Он знал о готовно-

сти функционального блока, потому что еженедельно наблюдал за его разработкой.

Требования к проекту менялись постоянно. Не редкостью были и принципиальные изменения. Бывало, что исключались целые функциональные блоки, а вместо них вставлялись новые. И тем не менее контракт и проект выжили и успешно завершились. Ключом к успеху было тесное сотрудничество с заказчиком и контракт, в котором оговаривался именно порядок сотрудничества, а не детальные требования и сроки при фиксированной цене.

Оперативное реагирование на изменения важнее следования плану

Способность реагировать на изменения часто определяет успех или провал программного проекта. Строя планы, необходимо следить за тем, чтобы они были гибкими и могли адаптироваться к изменениям в бизнесе и технологиях.

Ход работы над программным проектом нельзя планировать на длительный срок. Во-первых, условия функционирования бизнеса вполне могут измениться, и требования придется пересматривать. Во-вторых, заказчики, увидев, что система заработала, сразу начинают просить что-то изменить. Наконец, даже если мы заранее знаем требования и уверены, что они не изменятся, трудно оценить, сколько времени уйдет на их реализацию.

Для неискушенных менеджеров очень соблазнительно выглядит идея нарисовать и повесить на стену симпатичную PERT-диаграмму или график Гантта по проекту в целом. У них даже может возникнуть иллюзия, будто такая диаграмма позволяет им управлять проектом. Они могут отслеживать отдельные задачи и убирать их с диаграммы по мере завершения. Можно сравнивать фактические сроки с плановыми и реагировать на расхождения.

Но *в действительности* диаграмма просто перестает быть актуальной. По мере того как команда больше узнает о системе, а заказчик – о потребностях команды, некоторые обозначенные на диаграмме задачи становятся ненужными. Но выявляются другие задачи, которые следует добавить. Короче говоря, изменяются не только даты, но и *форма* плана.

Более правильная стратегия планирования состоит в том, чтобы составлять подробные планы на следующую неделю, ориентировочные – на три месяца и совсем грубые – в более далекой перспективе. Необходимо знать, над какими задачами будут работать отдельные сотрудники в течение следующей недели. Нужно примерно представлять себе, какие требования будут реализовываться в течение следующих трех месяцев. И следует иметь общее представление о том, что сможет выполнять система через год.

Такая уменьшающаяся детализация плана означает, что мы тратим время на разработку подробного плана только для срочных задач. Детальный план изменить трудно, так как команда уже настроилась на его выполнение. Но поскольку этот план затрагивает всего одну неделю, планирование на более поздние сроки сохраняет гибкость.

Принципы

На основе описанной выше шкалы ценностей можно сформулировать следующие 12 принципов. Это те характеристики, которые отличают набор гибких методик от тяжеловесного процесса.

1. *Высшим приоритетом считать удовлетворение пожеланий заказчика посредством поставки полезного программного обеспечения в сжатые сроки с последующим непрерывным обновлением.* В журнале *MIT Sloan Management Review* был опубликован анализ методик разработки программ, которые помогают компаниям создавать высококачественные продукты.[1] Авторы статьи выявили ряд факторов, оказывающих значительное влияние на качество конечной системы. Одним из них является корреляция между качеством и ранней поставкой частично работающей системы. В статье отмечается, что *чем менее функциональна начальная версия, тем выше качество конечного продукта*. Отмечается также наличие сильной корреляции между качеством конечного продукта и частыми выпусками новых версий с постепенно расширяющейся функциональностью. *Чем чаще выпуски, тем выше качество конечного продукта*.

   Гибкие методики подразумевают быструю поставку начальной версии и частые обновления. Наша цель – поставить рудиментарную систему в течение нескольких недель с момента начала проекта. В дальнейшем системы с постепенно расширяющейся функциональностью должны поставляться каждые несколько недель. Заказчик может начать промышленную эксплуатацию системы, если сочтет, что она достаточно функциональна. А может просто ознакомиться с имеющейся функциональностью и сообщить о том, какие следует внести изменения.

2. *Не игнорировать изменение требований, пусть даже на поздних этапах разработки. Гибкие процессы позволяют учитывать изменения ради обеспечения конкурентных преимуществ заказчику.* Это выражение нашей позиции. Люди, практикующие гибкие процессы, не боятся изменений. Они считают изменения требований *благом*, так как любое изменение улучшает понимание командой потребностей заказчика.

---

[1] «Product-Development Practices That Work: How Internet Companies Build Software», *MIT Sloan Management Review*, зима 2001, номер репринта 4226.

Гибкая команда прилагает все силы к тому, чтобы сделать структуру программы подвижной и тем самым минимизировать влияние изменений на систему в целом. Ниже в этой книге мы будем обсуждать принципы, паттерны и методики объектно-ориентированного проектирования, позволяющие добиться такой гибкости.

3. *Поставлять новые работающие версии ПО часто, с интервалом от пары недель до пары месяцев, отдавая предпочтение меньшим срокам.* Мы поставляем *работающую* программу и делаем это быстро и часто. Мы не довольствуемся поставкой кипы документов или планов. Мы вообще не считаем последние объектом поставки. Наша цель – поставить программу, удовлетворяющую потребности пользователя.
4. *Заказчики и разработчики должны работать совместно на протяжении всего проекта.* Чтобы проект можно было назвать гибким, заказчики, разработчики и все заинтересованные стороны должны общаться часто и помногу. Программный проект – это не самонаводящийся снаряд. Программному проекту нужно постоянно уделять внимание.
5. *Проекты должны воплощать в жизнь целеустремленные люди. Создайте им условия, обеспечьте необходимую поддержку и верьте, что они доведут дело до конца.* Люди – важнейшее условие успеха. Все остальные факторы – процесс, среда разработки, управление и т. д. – вторичны и могут быть изменены, если негативно отражаются на людях.
6. *Самый эффективный и продуктивный метод передачи информации команде разработчиков и обмена мнениями внутри нее – разговор лицом к лицу.* В гибком проекте люди *разговаривают* друг с другом. Основной способ коммуникации – простое человеческое общение. Письменные документы создаются и обновляются постепенно по мере разработки ПО и только в случае необходимости.
7. *Работающая программа – основной показатель прогресса в проекте.* О приближении гибкого проекта к завершению судят по тому, насколько имеющаяся в данный момент программа отвечает требованиям заказчика. Прогресс не оценивается в терминах текущего этапа, количества документации или объема написанного инфраструктурного кода. Проект закончен на 30%, если присутствует 30% необходимой функциональности.
8. *Гибкие процессы способствуют долгосрочной разработке. Заказчики, разработчики и пользователи должны быть в состоянии поддерживать неизменный темп сколько угодно долго.* Гибкий проект – не забег на 50 ярдов, а, скорее, марафон. Команда не стартует на максимально возможной скорости, думая только о том, как бы дотянуть до конца дистанции. Нет, она бежит в высоком, но позволяющем долго продержаться темпе.

Слишком быстрый бег приводит к изнеможению, поиску коротких путей и неудаче. Гибкие команды не устраивают гонки. Они не позволяют себе слишком быстро уставать. Они не занимают завтрашнюю энергию, чтобы побольше успеть сегодня. Они работают в том темпе, который обеспечивает соблюдение высочайших стандартов качества на протяжении всего проекта.

9. *Непрестанное внимание к техническому совершенству и качественному проектированию повышает отдачу от гибких технологий.* Высокое качество – ключ к высокой скорости. Чтобы продвигаться быстро, нужно писать максимально чистый и надежный код. Поэтому все члены гибкой команды стремятся создавать только код самого высокого качества. Они не оставляют беспорядка, обещая себе прибраться, как только появится время. Порядок наводится незамедлительно.
10. *Простота – искусство достигать большего, делая меньше, – основа основ.* Гибкие команды не пытаются возводить системы-небоскребы. Напротив, они всегда ищут простейший путь, ведущий к цели. Они не придают первостепенного значения предвидению завтрашних проблем и не пытаются решить их сегодня. Вместо этого они решают сегодняшние задачи максимально просто и качественно, будучи уверены в том, что если завтра возникнет какая-то проблема, то можно будет без труда внести изменения.
11. *Самые лучшие архитектуры, требования и проекты выдают самоорганизующиеся команды.* Гибкая команда – самоорганизующийся коллектив. Задачи поручаются не отдельным членам команды извне, а команде в целом. Команда сама решает, как лучше всего справиться с порученными задачами.

    Члены гибких команд совместно работают над всеми аспектами проекта. Каждому участнику разрешено вносить свой вклад в общее дело. Нет такого члена команды, который единолично отвечал бы за архитектуру, требования или тесты. Ответственность лежит на команде в целом, и у каждого ее члена есть право голоса.
12. *Команда должна регулярно задумываться над тем, как стать еще более эффективной, а затем соответственно корректировать и подстраивать свое поведение.* Гибкая команда постоянно корректирует свою организацию, правила, соглашения, взаимоотношения и т. д. Она знает, что окружение непрестанно изменяется, и понимает, что должна изменяться вместе с ним, если хочет остаться гибкой.

Заключение

Профессиональная цель любого разработчика или команды разработчиков ПО – предложить заказчикам и работодателям продукт максимально высокого качества. И тем не менее количество неудачных проектов остается чересчур высоким. Раскручивающаяся спираль разбу-

хания процессов, пусть даже с самыми благими намерениями, – одна из причин таких неудач. Шкала ценностей и принципы гибкой разработки составлены таким образом, чтобы разорвать порочный круг разбухания технологического процесса и сосредоточиться на простых методах достижения целей.

Во время работы над этой книгой можно было выбирать из нескольких гибких процессов: SCRUM[1], Crystal[2], разработка по свойствам (Feature-driven development – FDD)[3], адаптивная разработка программ (Adaptive software development – ADP)[4] и экстремальное программирование (Extreme programming – XP)[5]. Однако большинство успешных гибких команд взяли что-то полезное из каждого процесса, подстроив это под свое представление о гибкости. Такие адаптации концентрируются вокруг SCRUM и XP, причем методики SCRUM применяются для управления несколькими командами, практикующими XP.

Библиография

[Beck99] Kent Beck «Extreme Programming Explained: Embrace Change», Addison-Wesley, 2004.

[Highsmith2000] James A. Highsmith «Adaptive Software Development: A Collaborative Approach to Managing Complex Systems», Dorset House, 2000.

[Newkirk2001] James Newkirk and Robert C. Martin «Extreme Programming in Practice», Addison-Wesley, 2001.

---

1 *www.controlchaos.com*

2 *crystalmethodologies.org*

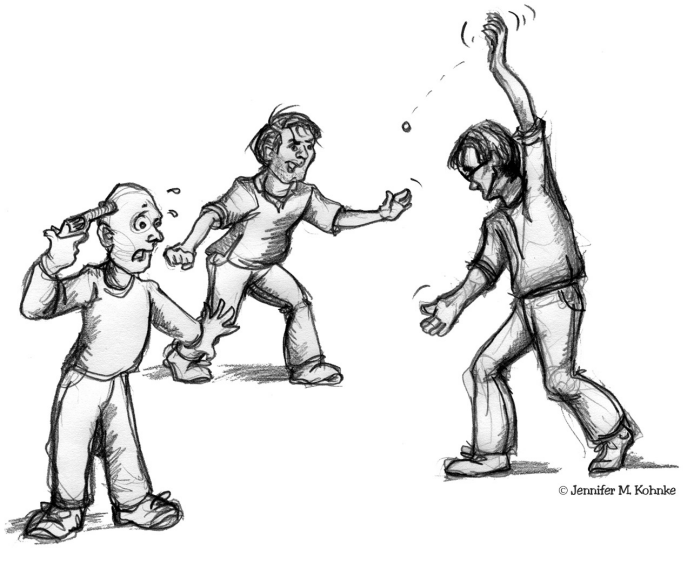
3 Peter Coad, Eric Lefebvre, Jeff De Luca «Java Modeling in Color with UML: Enterprise Components and Process», Prentice Hall, 1999.

4 [Highsmith2000]

5 [Beck99], [Newkirk2001]

2

Обзор экстремального программирования



Мы, разработчики, должны помнить,
что экстремальное программирование –
не единственная забава в городке.[1]

Пит Макбрин

[1] Отсылка к названию фильма Джорджа Стивенса «Only Game in Town». – *Прим. перев.*

В главе 1 мы кратко описали, что такое гибкая разработка программ, но не говорили о том, что конкретно нужно делать. Было сказано несколько общих фраз, поставлены некие цели, но не указано, в каком направлении двигаться. В этой главе мы исправим это упущение.

Методики экстремального программирования

Единая команда

Мы хотим, чтобы заказчики, менеджеры и разработчики тесно сотрудничали, знали о проблемах друг друга и совместно работали над их разрешением. Но кто такой заказчик? Заказчиком XP-команды является лицо или группа лиц, которая определяет функции ПО и назначает им приоритеты. Иногда в роли заказчика выступает группа бизнес-аналитиков, специалистов по контролю качества и/или маркетологов, работающих в той же организации, что и разработчики. Иногда заказчиком является представитель, делегированный сообществом пользователей. Бывает, что заказчик – это тот, кто платит реальные деньги. Но в XP-проекте заказчик, как бы его ни определять, является членом команды, доступным для общения.

Лучше всего, когда заказчик сидит в той же комнате, что разработчики. Если так не получается, то хорошо бы, чтобы он находился от этой комнаты не дальше чем в 30 метрах. Чем больше расстояние, тем труднее заказчику быть настоящим членом команды. Заказчика, который находится в другом здании или другом штате, включить в команду очень трудно.

Но что делать, если заказчик просто не может быть рядом? Мой совет – найти кого-нибудь, кто будет всегда неподалеку и может и хочет выступать от имени настоящего заказчика.

Пользовательские истории

Для планирования проекта мы должны иметь представление о требованиях, но не исчерпывающее. Достаточно знать о требовании лишь то, что необходимо для его оценки. Быть может, вы полагаете, что для оценки требования надо знать все детали? Это не совсем так. Нужно знать, что детали *существуют,* и примерно представлять себе, что это за детали, но досконально разбираться в специфике необязательно.

Конкретные детали требований могут изменяться со временем, особенно когда заказчик видит, как система начинает принимать форму. Ничто не способствует уточнению требований больше, чем зрелище нарождающейся системы. Поэтому сбор детальной информации о требовании задолго до его реализации скорее всего будет пустой и преждевременной тратой сил.

В экстремальном программировании мы получаем представление о деталях требований, обговаривая их с заказчиком. Но мы не фиксируем

эти детали. Вместо этого заказчик пишет на карточке несколько слов, которые, по общему согласию, будут напоминанием о состоявшемся разговоре. Разработчики записывают на той же карточке и примерно в то же время свою оценку трудозатрат. Оценка основывается на том представлении о деталях, которое было получено в беседе с заказчиком.

Пользовательская история (User story) – это памятные заметки о состоявшемся разговоре с заказчиком по поводу требования. Такие истории – инструмент планирования, с помощью которого заказчик составляет график реализации требования, исходя из его приоритета и оценочной стоимости.

Короткие циклы

В XP-проекте работающие версии программы выпускаются каждые две недели. На каждой итерации реализуются некоторые потребности заинтересованных сторон. В конце итерации система демонстрируется заказчику, чтобы тот мог высказать замечания и внести предложения.

План итерации. Обычно итерация занимает две недели и завершается поставкой второстепенной версии, которая может пойти, а может и не пойти в эксплуатацию. План итерации – это собрание пользовательских историй, отобранных заказчиком в соответствии с бюджетом, установленным разработчиками.

Разработчики формируют бюджет итерации, оценивая, сколько они сделали на предыдущей итерации. Заказчик может отобрать для итерации любое количество историй при условии, что их суммарная оценка не превышает бюджет.

После начала операции заказчик соглашается не изменять определение приоритетов историй на данной итерации. В течение этого времени разработчики вольны разбивать истории на *задачи* и реализовывать задачи в том порядке, который считают оптимальным с технической и деловой точки зрения.

План выпуска. XP-команда часто составляет план выпуска, охватывающий следующие шесть или около того итераций. Обычно выпуск является плодом трехмесячной работы. Он представляет собой очередную основную версию, которая, как правило, может быть запущена в эксплуатацию. План выпуска состоит из упорядоченного по приоритету набора пользовательских историй, отобранных заказчиком в соответствии с бюджетом, представленным разработчиками.

Разработчики формируют бюджет выпуска, оценивая, сколько они сделали в предыдущем выпуске. Заказчик может отобрать для выпуска любое количество историй при условии, что их суммарная оценка не превышает бюджет. Заказчик также определяет порядок реализации историй в выпуске. Если команда пожелает, то может распланировать первые несколько итераций выпуска, показав, какие истории будут завершены на каких итерациях.

Выпуски не являются чем-то жестко фиксированным. Заказчик может в любой момент изменить содержание выпуска: отменить некоторые истории, написать новые или установить для истории другой приоритет. Однако заказчик должен всячески воздерживаться от изменения *итераций*.

Приемочные тесты

Детали из пользовательских историй аккумулируются в виде приемочных тестов, определяемых заказчиком. Приемочные тесты для истории пишутся сразу перед или даже одновременно с реализацией этой истории на языке сценариев, который позволяет прогонять их автоматически и повторно.[1] В совокупности они позволяют удостовериться, что система ведет себя так, как специфицировал заказчик.

Приемочные тесты пишутся бизнес-аналитиками, специалистами по контролю качества и тестировщиками в течение итерации. Язык, на котором они пишутся, должен быть прост и понятен программистам, заказчикам и представителям бизнеса. Именно из этих тестов программисты извлекают настоящие детали историй, которые им предстоит реализовать. Тесты и выступают в роли документа, описывающего требования к проекту. Каждая деталь каждой функции описана в приемочных тестах, поэтому они являются окончательным авторитетом в вопросе о том, правильно ли реализована данная функция.

Если приемочный тест проходит, то он добавляется в состав корпуса прошедших приемочных тестов, после чего он никогда не должен завершаться неудачно. Растущий корпус приемочных тестов прогоняется несколько раз в день, при каждой сборке системы. Если приемочный тест не проходит, то сборка объявляется неудачной. Таким образом, если некоторое требование реализовано, то оно уже никогда не будет нарушено. Система переходит из одного работоспособного состояния в другое и не должна оставаться неработоспособной дольше нескольких часов.

Парное программирование

Код пишется двумя программистами, использующими одну и ту же рабочую станцию. В каждый момент времени один программист занимает клавиатуру и вводит код. В это время другой программист следит за тем, что вводится, ищет ошибки и предлагает улучшения.[2] Оба тесно общаются. И оба полностью поглощены процессом написания кода.

Роли часто меняются. Если пишущий устал или застрял, его партнер садится за клавиатуру и начинает вводить код. Клавиатура переходит от одного к другому несколько раз в течение часа. Проектировщиками

---

[1] См. *www.fitnesse.org*.

[2] Я встречал пары, в которых один программист узурпирует клавиатуру, а другой – мышь.

и авторами кода считаются оба партнера. Ни один не может претендовать больше чем на половину заслуг.

Состав пар часто меняется. Разумно менять пары по крайней мере один раз в день, чтобы в течение дня каждый программист успел поработать хотя бы в двух разных парах. На протяжении итерации каждый член команды должен поработать в паре со всеми остальными и принять участие в разработке всего запланированного на данную итерацию.

Парное программирование радикально ускоряет распространение знаний в команде. Хотя специализация остается и задачи, требующие специальных знаний, обычно распределяются между соответствующими специалистами, эти специалисты сходятся в парах почти со всеми членами команды. В результате специальные знания распространяются по команде, так что другие члены могут в трудную минуту заменить специалиста. Исследования Вильямса[1] и Носека[2] показывают, что объединение в пары не снижает эффективность работы программистов, но значительно уменьшает частоту ошибок.

Разработка через тестирование

В главе 4 эта тема обсуждается подробно. Здесь же мы приведем лишь краткое введение.

Весь код пишется так, чтобы ранее не проходивший автономный тест завершился успешно. Сначала мы пишем автономный тест, который завершается неудачно, так как тестируемая функциональность еще не реализована. Затем пишется код, который приведет к успешному завершению данного теста.

Промежуток времени между написанием тестов и кода очень краток, порядка минуты. Тесты и код эволюционируют вместе, причем тесты немного опережают код. (Пример см. в главе 6.)

В результате по мере написания кода формируется самый полный корпус тестов. Они позволяют проверять, как работает программа. Пара, запрограммировавшая небольшое изменение, может выполнить все тесты и убедиться, что никаких сбоев не возникло. Такая методика очень упрощает *рефакторинг* (обсуждается в следующей главе).

Код, который пишется для того, чтобы прошел некоторый тест, по определению поддается тестированию. Более того, создается сильная мотивация для разбиения программы на модули, чтобы каждый модуль можно было тестировать независимо. Поэтому проект, разрабатываемый таким способом, оказывается существенно менее связанным. В обеспечении несвязанности важную роль играют принципы объектно-ориентированного проектирования (см. раздел II).

---

[1] [Williams2000], [Cockburn2001]

[2] [Nosek98]

Коллективное владение

У пары есть полное право снять с учета *любой* модуль и улучшить его. Ни один программист не несет личной ответственности за какой-то конкретный модуль или технологию. Над графическим интерфейсом пользователя (ГИП) работают все.[1] Над промежуточным уровнем – тоже все. И над базой данных все. Ни у кого нет больших прав на какой-то модуль или технологию, чем у всех остальных.

Это не означает, что XP отрицает специализацию. Если вы специализируетесь на разработке ГИП, то скорее всего будете работать над задачами, относящимися к ГИП. Но вас могут также попросить поработать над промежуточным уровнем или базой данных. Если вы захотите освоить вторую специальность, то можете попроситься на соответствующие задачи и поработать со специалистами, которые научат вас. Вы не привязаны к одной какой-то специальности.

Непрерывная интеграция

Программисты ставят свой код на учет и интегрируют его несколько раз в день. Правило простое: кто первым запустил постановку на учет, тот в выигрыше; остальные должны объединять изменения.

В XP-командах применяются неблокирующие системы управления версиями. Это означает, что программисту разрешено ставить на учет любой модуль в любое время, даже если кто-то другой снял его с учета. При обратной постановке модуля на учет после модификации программист должен быть готов объединить свои изменения с теми, что были поставлены на учет раньше. Чтобы избежать существенных затрат времени на объединение, необходимо ставить модули на учет как можно чаще.

Пара работает над одной задачей час-два. Она создает тесты и промышленный код. В какой-то удобный момент, как правило, задолго до завершения задачи, пара решает поставить свой код на учет. Но сначала следует убедиться, что все тесты проходят. Затем новый код интегрируется в систему. При необходимости производится объединение изменений. Если нужно, пара советуется с программистами, которые успели поставить на учет свои изменения раньше. Прогоняются все имеющиеся в системе тесты, в том числе приемочные, которые в данный момент работоспособны. Если что-то из работавшего ранее повредилось, ошибка исправляется. Когда все тесты успешно завершились, постановка на учет может считаться законченной.

Таким образом, система собирается много раз в день. При этом система собирается *целиком*, от начала до конца.[2] Если конечным результатом

---

[1] Я здесь не агитирую за трехуровневую архитектуру, а просто выбрал три широко распространенных слоя программного обеспечения.

[2] Как сказал Рон Джеффрис, «путь от начала до конца длиннее, чем вам кажется».

сборки должна быть система на CD-ROM, то она записывается на CD-ROM. Если же конечным результатом является веб-сайт, то устанавливается этот сайт, вероятно, на тестовом сервере.

Умеренный темп

Программный проект – это не спринт, а марафон. Команда, которая срывается со старта и мчится что есть духу, может выдохнуться задолго до финиша. Чтобы быстро финишировать, нужно бежать в умеренном темпе, экономя силы и резвость. Команда должна сознательно выбрать умеренный, но постоянный темп.

Правило XP гласит, что переработки *запрещены*. Единственным исключением может стать последняя неделя выпуска, когда команда, находясь в непосредственной близости от цели выпуска, может сделать финальный рывок и поработать больше, чем положено.

Открытое рабочее пространство

Команда работает совместно в открытом помещении. На каждом столе стоит две-три рабочих станции. Перед каждой рабочей станцией два кресла. На стенах развешаны диаграммы текущего состояния, разбиение на задачи, UML-диаграммы и т. д.

В комнате стоит приглушенный шум разговоров. Каждая пара находится в пределах слышимости от всех остальных пар. У каждого члена команды есть возможность услышать, что кто-то оказался в затруднении. Каждый знает, что происходит у других. Программисты могут интенсивно общаться между собой.

Может возникнуть опасение, что такое окружение отвлекает. Что из-за постоянного шума и разговоров ничего не удастся сделать. Но все не так страшно. Более того, в такой обстановке «боевого командного пункта» продуктивность не только не снижается, а, как показывает исследование Мичиганского университета, может даже *возрасти* вдвое.[1]

Игра в планирование

В главе 3 мы будем подробно говорить об игре в планирование в экстремальном программировании. Здесь я приведу лишь краткое описание.



Смысл игры в планирование состоит в разделении ответственности между заказчиками и разработчиками. Заказчик решает, насколько важна некоторая функция, а разработчики – сколько будет стоить ее реализация.

[1] *www.sciencedaily.com/releases/2000/12/001206144705.htm*

В начале каждого выпуска и каждой итерации разработчики сообщают заказчикам бюджет. Заказчики выбирают истории, суммарная стоимость которых равна бюджету, но выходить за пределы бюджета им запрещено. Разработчики формируют бюджет, исходя из того, сколько им удалось сделать на предыдущей итерации или в предыдущем выпуске.

При таких простых правилах и принимая во внимание короткие итерации и частые циклы выпуска, заказчики и разработчики очень скоро войдут в ритм проекта. Заказчики будут представлять себе, насколько быстро могут продвигаться разработчики. А это позволит им примерно оценить, сколько времени займет работа над проектом и во что он обойдется.

Простота

XP-команда стремится к максимально простому и выразительному дизайну. Кроме того, она сосредотачивает внимание только на тех историях, которые запланированы на текущей итерации, не заботясь о тех, что будут потом. Структура системы меняется от итерации к итерации, и в итоге получает оптимальный проект для уже реализованных историй.

Это означает, что XP-команда, скорее всего, не начнет работу с подготовки инфраструктуры, с выбора СУБД или ПО промежуточного уровня. Вместо этого команда реализует первую порцию историй *самым простым из возможных способов*. Инфраструктура же будет расширяться только тогда, когда того потребует очередная история.

Разработчик руководствуется тремя мантрами XP.

1. *Выбирай самый простой способ, который будет работать*. XP-команды всегда пытаются отыскать простейший вариант реализации текущего пакета историй. Если для текущих историй можно обойтись плоскими файлами, то СУБД, возможно, и не понадобится. Если для текущих историй достаточно простого соединения через сокет, то ни к чему ни брокер объектных запросов (ORB), ни веб-служба. Если для текущих историй не нужна многопоточность, то и включать ее не стоит. Мы всегда пытаемся найти самый простой способ реализации текущих историй. А затем выбираем решение, настолько близкое к идеалу простоты, насколько это *практически* возможно.
2. *Тебе это не понадобится*. Да, мы *знаем*, что в какой-то момент СУБД будет нужна. Мы *знаем*, что рано или поздно возникнет необходимость в ORB. Мы *знаем*, что в будущем придется поддерживать нескольких пользователей. Поэтому предусмотреть все это нужно уже *сейчас*, правда?

   XP-команда серьезно обдумывает, что случится, если она воспротивится искушению добавить инфраструктуру до того, как та ста-

нет реально необходимой. Команда начинает с допущения, что эта инфраструктура вообще не понадобится. И добавляет ее, только если доказано или, по крайней мере, имеются очень веские доводы в пользу того, что включить инфраструктуру сейчас обойдется дешевле, чем ждать до последнего.

3. *Один и только один раз.* XP-команды не выносят дублирования кода. Обнаружив дублирование, они сразу же избавляются от него.

Источников дублирования кода много. Самые очевидные – копирование фрагмента с помощью мыши и вставка его в разные места. Встретив такое, мы создаем функцию или базовый класс. Но иногда два или более алгоритмов могут быть быть очень похожи и все же иметь тонкие различия. Тогда мы превращаем их в функции или пользуемся паттерном Шаблонный метод (Template Method) (см. главу 22). В общем, каким бы ни был источник дублирования, мы устраняем его.

Самый лучший способ избавиться от избыточности – создавать абстракции. В конце концов, если два предмета похожи, то на некотором уровне абстракции их можно унифицировать. Поэтому требование устранения избыточности заставляет команду создавать много абстракций и тем самым еще уменьшать связанность.

Рефакторинг

В главе 5 рефакторинг рассматривается более подробно[1], а ниже приводится краткое описание.

Код имеет тенденцию «протухать». По мере того как добавляются все новые функции и исправляются ошибки, структура кода ухудшается. Если не обращать на это внимания, то такая деградация приведет к неразберихе, которую невозможно сопровождать.

XP-команды борются с деградацией посредством частого рефакторинга. Рефакторинг – это проведение ряда мелких трансформаций, которые улучшают общую структуру системы, не влияя на ее поведение. Каждая трансформация в отдельности тривиальна, ее вроде бы и делать не стоит. Но вместе они существенно преобразуют структуру и архитектуру системы.

После каждой крохотной трансформации мы прогоняем автономные тесты, чтобы удостовериться, что ничто не повредилось. Потом производим следующую трансформацию, и еще одну, и еще, каждый раз прогоняя тесты. Таким образом, изменяя структуру системы, мы сохраняем ее работоспособность.

Рефакторинг производится постоянно, а не только в конце проекта, выпуска или итерации и даже не в конце рабочего дня. Это делается каждый час, а то и каждые полчаса. Применяя рефакторинг, мы поддерживаем элегантность, простоту и выразительность кода.

---

[1] [Fowler99]

Метафора

Метафора – единственная методика XP, не имеющая конкретного и ясного воплощения. Эта часть XP понята хуже всего. Адепты XP в душе прагматики, и такое отсутствие конкретного определения повергает их в смущение. Собственно, пропагандисты XP при обсуждении этой технологии часто не упоминают метафору как методику. И тем не менее метафора – одна из самых важных методик экстремального программирования.

Возьмите, к примеру, пазл. Откуда вы знаете, как соединять кусочки? Понятно, что каждый кусочек стыкуется с другими, и его форма должна идеально подходить к тем, которые к нему примыкают. Если вы слепы, но обладаете развитым осязанием, то могли бы перебирать один кусочек за другим, пытаясь поместить их в разные места.

Но есть нечто куда более важное, чем формы кусочков пазла: картинка. Она-то и является настоящим путеводителем. Картинка настолько важна, что если у двух соседних кусочков картинки формы не соответствуют друг другу, то вы точно *знаете*, что изготовитель пазла ошибся.

Вот это и есть метафора. Это большая картинка, собирающая все части системы воедино. Это взгляд на систему, делающий очевидными места и формы отдельных модулей. Если форма модуля не соответствует метафоре, значит, модуль не годится.

Часто метафора сводится к системе имен. Имена составляют словарь всех элементов системы и помогают выявить их взаимосвязи.

Например, как-то мне довелось работать над системой, которая передавала текст на экран со скоростью 60 символов в секунду. При такой скорости на заполнение экрана уходило заметное время. Поэтому мы решили, что программа, генерирующая текст, должна писать в некий буфер. Когда буфер заполнялся, мы выгружали программу на диск. Когда буфер был почти пуст, мы загружали программу с диска и давали ей возможность еще немного поработать.

Мы рассуждали о системе, как о мусоровозах, вывозящих мусор на свалку. Буферы были мусоровозами, экран дисплея – свалкой, программа – производителем мусора. Все имена были согласованы, и это помогало нам представлять систему в целом.

Другой пример – когда-то я работал над системой анализа сетевого трафика. Каждые 30 минут она опрашивала десятки сетевых адаптеров и получала от них данные мониторинга. Каждый адаптер возвращал небольшой блок данных, содержащий несколько переменных. Эти блоки мы называли «ломтиками». Ломтики представляли собой «сырые» исходные данные, подлежащие анализу. Программа анализа «поджаривала» ломтики и поэтому была названа «тостером». Отдельные переменные внутри ломтиков мы называли «крошками». В целом, это была полезная и забавная метафора.

Разумеется, метафора – больше, чем просто система имен. Это взгляд на систему. Метафора вынуждает всех разработчиков выбирать подходящие имена и соответствующие места для функций, создавать подходящие классы и методы и т. д.

Заключение

Экстремальное программирование – это набор простых и конкретных методик, в совокупности складывающихся в гибкий процесс разработки. XP – хороший универсальный метод разработки программ. Многие команды смогут взять эту технологию на вооружение без изменений. Другим будет удобнее адаптировать ее, добавив или модифицировав некоторые методики.

Библиография

[ARC97] Alistair Cockburn, The Methodology Space, *Humans and Technology*, technical report HaT TR.97.03 (датирован 03.10.97), http://members.aol.com/acockburn/papers/methyspace/methyspace.htm.

[Beck99] Kent Beck «Extreme Programming Explained: Embrace Change», Addison-Wesley, 2004.

[Beck2003] Kent Beck «Test-Driven Development by Example», Addison-Wesley, 2003.

[Cockburn2001] Alistair Cockburn and Laurie Williams «The Costs and Benefits of Pair Programming», XP2000 Conference in Sardinia, reproduced in Giancarlo Succi and Michele Marchesi, Extreme Programming Examined, Addison-Wesley, 2001.

[DRC98] Daryl R. Conner «Leading at the Edge of Chaos», Wiley, 1998.

[EWD72] D. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare «Structured Programming», Academic Press, 1972.

[Fowler99] Martin Fowler «Refactoring: Improving the Design of Existing Code», Addison-Wesley, 1999.[1]

[Newkirk2001] James Newkirk and Robert C. Martin «Extreme Programming in Practice», Addison-Wesley, 2001.

[Nosek98] J. T. Nosek «The Case for Collaborative Programming», *Communications of the ACM*, 1998, pp. 105–108.

[Williams2000] Laurie Williams, Robert R. Kessler, Ward Cunningham, Ron Jeffries «Strengthening the Case for Pair Programming», *IEEE Software*, July–Aug. 2000.

[1] Мартин Фаулер «Рефакторинг. Улучшение существующего кода». – Пер. с англ. – СПб.: Символ-Плюс, 2002.

3

Планирование



Если вы можете измерять и выражать в числах то,
о чем говорите, то об этом предмете вы кое-что знаете;
если же вы не можете сделать этого,
то ваши познания скудны и неудовлетворительны.

Лорд Кельвин, 1883

В этой главе мы опишем игру в планирование из практики экстремального программирования[1] Она похожа на подход к планированию в нескольких других методиках гибкой[2] разработки: SCRUM[3], Crystal[4], разработка по свойствам (FDD)[5] и адаптивная разработка ПО (ADP)[6]. Однако ни в одном из этих процессов описываемая методика не сформулирована так детально и строго.

Первичное обследование

В начале работы над проектом разработчики и заказчики обсуждают новую систему, чтобы выявить наиболее существенные функции. Однако они не пытаются идентифицировать *все вообще* функции. По мере развертывания работ заказчики будут обнаруживать все новые и новые функции. Поток функций не иссякнет вплоть до момента завершения проекта.

Вновь выявленная функция разбивается на одну или несколько *пользовательских историй*, которые записываются на учетной карточке или чем-то подобном. Много писать не надо, достаточно названия истории (например, «Вход в систему», «Добавление пользователя», «Удаление пользователя», «Изменение пароля»). На этой стадии мы не пытаемся зафиксировать все детали, а просто хотим оставить памятку о состоявшемся разговоре по поводу функций системы.

Разработчики совместно оценивают истории. Эти оценки относительны, а не абсолютны. Мы записываем на карточке число «баллов», то есть относительную стоимость истории. Мы не знаем точно, сколько времени потребуется на реализацию истории, но можем сказать, что на историю стоимостью 8 баллов уйдет в два раза больше времени, чем на историю стоимостью 4 балла.

Объединение, разбиение и скорость

Слишком длинные или слишком короткие истории трудно оценивать. Разработчики обычно недооценивают длинные истории и переоценивают короткие. Если история слишком велика, то ее следует разбить на меньшие части. Если история слишком мала, ее следует объединить с другими, такими же маленькими.

---

1 [Beck99], [Newkirk2001]

2 *www.AgileAlliance.org*

3 *www.controlchaos.com*

4 [Cockburn2005]

5 Peter Coad, Eric Lefebvre, Jeff De Luca «Java Modeling in Color with UML: Enterprise Components and Process», Prentice Hall, 1999.

6 [Highsmith2000]

Например, история «Пользователи могут безопасно переводить деньги на свой счет, со своего счета или с одного счета на другой» чересчур велика. Оценить ее будет трудно, а результат, скорее всего, окажется неточным. Но мы можем разбить ее на несколько меньших историй, оценить которые проще:

- Пользователь может войти в систему
- Пользователь может выйти из системы
- Пользователь может положить деньги на свой счет
- Пользователь может снять деньги со своего счета
- Пользователь может перевести деньги с любого из своих счетов на другой

После разбиения или объединения историю нужно заново оценить. Было бы неправильно просто сложить или вычесть оценки. Весь смысл разбиения и объединения историй состоит в том, чтобы получить размер, при котором оценка точна. Нередко бывает, что история, первоначально оцененная в 25 баллов, раскладывается на истории, которые в сумме дают 30 баллов! Тридцать – более точная оценка.

Каждую неделю мы завершаем реализацию нескольких историй. Сумма оценок завершенных историй дает метрику, называемую *скоростью*. Если на прошлой неделе мы завершили истории общей стоимостью 42 балла, то наша скорость составляет 42.

После трех-четырех недель мы получим довольно объективное представление о своей средней скорости. С помощью этой величины можно прогнозировать, какой объем работы мы сумеем выполнить в последующие недели. Отслеживание скорости – один из самых важных инструментов управления XP-проектом.

В самом начале проекта разработчики плохо представляют свою скорость. Поэтому нужно сделать некое начальное предположение, пользуясь соображениями, которые, по общему мнению, дадут оптимальный результат. В этот момент особая точность не нужна, поэтому не стоит тратить на это слишком много времени. Вообще, вполне подойдет и старый добрый «метод научного тыка».[1]

Планирование выпуска

Зная скорость, заказчик может получить представление о стоимости каждой истории, а также о ее значимости для бизнеса и о приоритете. Это позволяет заказчику решить, какие истории реализовывать первыми. Решение не сводится к приоритетности. Реализацию важной, но дорогой истории можно отложить в пользу менее важной, но более

[1] В оригинале это называется SWAG – Scientific Wild-Assed Guess. – *Прим. перев.*

дешевой. Это решения такого же рода, как принимаемые в *бизнесе*. Заказчик решает, какие истории дадут максимальную отдачу на вложенный капитал.

Разработчики и заказчик согласуют дату первого выпуска проекта. Обычно это занимает 2–4 месяца. Заказчик указывает, какие истории реализовать в этом выпуске и примерный порядок реализации. Не разрешается выбирать больше историй, чем укладывается в текущую скорость. Поскольку начальная оценка скорости неточна, то и выбор оказывается довольно грубым. Но в этот момент точность не так существенна. План выпуска можно будет откорректировать по мере уточнения оценки скорости.

Планирование итерации

Затем разработчики и заказчик определяют длительность итерации: обычно 1 или 2 недели. И снова заказчик решает, какие истории реализовать на первой итерации, но не может выбрать больше историй, чем позволяет текущая скорость.

Порядок реализации историй в пределах итерации – вопрос чисто технический, решаемый самими разработчиками. Можно работать последовательно, завершая одну историю за другой, а можно разделить множество историй на части и работать над ними параллельно. Тут разработчикам никто не указ.

Заказчик не может изменять истории после начала итерации. Он вправе изменять или переупорядочивать любые истории, кроме тех, над которыми разработчики уже трудятся.

Итерация заканчивается в заранее оговоренный день, даже если не все истории реализованы. Оценки всех завершенных историй суммируются, и вычисляется скорость на данной итерации. Эта величина используется для планирования следующей итерации. Правило очень простое: скорость, запланированная для последующей итерации, равна скорости, достигнутой на предыдущей. Если на последней итерации команда реализовала историй на 31 балл, то на следующую запланирует истории на те же 31 балл.

Такая обратная связь по скорости помогает синхронизировать планирование с возможностями команды. Если команда накапливает опыт, то скорость соизмеримо возрастает. Если из команды кто-то уходит, то скорость падает. По мере развития архитектуры, упрощающей разработку, скорость растет.

Определения понятия «готово»

История не считается реализованной, пока не пройдут *все* приемочные тесты. Эти тесты автоматизированы. Пишут их заказчики, бизнес-

аналитики, специалисты по контролю качества, тестировщики и даже программисты в начале каждой итерации. Тесты определяют детали истории и являются неоспоримым авторитетом в вопросе о поведении историй. В следующей главе мы еще будем говорить о приемочном тестировании.

Планирование задач

В начале новой итерации разработчики и заказчики вырабатывают план. Разработчики разбивают истории на задачи. *Задачей* называется объем работы, с которым один разработчик может справиться за 4–16 часов. Истории анализируются совместно с заказчиком, и задачи формулируются максимально полно.

Список задач записывается в лекционном блокноте, на доске или на любом другом носителе. Затем разработчики выбирают себе задачи, которые хотели бы реализовать, присваивая каждой задаче произвольную оценку в баллах.[1]

Разработчик может выбрать любую задачу. Специалисты по базе данных не обязаны ограничиваться только задачами, связанными с базой. Специалисты по разработке ГИП могут при желании выбрать задачу, касающуюся базы данных. Хотя на первый взгляд это может показаться неэффективным, но выгода очевидна: чем больше каждый разработчик знает о проекте *в целом*, тем успешнее и информированнее вся команда. Мы добиваемся, чтобы за проект отвечала команда целиком, независимо от специализации.

Каждый разработчик знает, сколько баллов он заработал на задачах на предыдущей итерации; это число составляет *бюджет* разработчика. Никто не берет себе задач на большую сумму, чем его бюджет.

Распределение задач продолжается до тех пор, пока не исчерпаются либо задачи, либо бюджеты всех разработчиков. Если какие-то задачи остались, то разработчики на общем обсуждении распределяют их, исходя из имеющихся навыков и квалификации. Если все задачи распределить все равно не удается, то разработчики просят заказчика изъять некоторые задачи или истории из данной итерации. Если же все задачи распределены, а бюджеты разработчиков не исчерпаны, то заказчика просят добавить несколько историй.



В середине итерации команда устраивает рабочее совещание. В этот мо-

[1] Многие разработчики считают удобным использовать в качестве оценки количество «идеальных часов программирования».

мент должна быть реализована половина *историй*, запланированных на данную итерацию. Если это не так, то команда пытается перераспределить задачи и ответственность таким образом, чтобы к концу итерации все истории были реализованы. Если разработчики не могут прийти к соглашению по этому поводу, то следует уведомить заказчика. Заказчик может принять решение об изъятии какой-то задачи или истории из итерации. В крайнем случае он хотя бы назовет задачи и истории с самым низким приоритетом, чтобы разработчики не тратили на них время.

Предположим, к примеру, что заказчик выбрал восемь историй на общую сумму 24 балла. Пусть они были разбиты на 42 задачи. Ожидается, что в середине итерации будет завершена 21 задача и набрано 12 баллов. Эти 12 баллов должны соответствовать полностью закрытым историям. Наша цель – завершить реализацию историй, а не просто задач. Кошмарная ситуация возникает, когда к концу итерации завершено 90% задач, но ни одной истории. В середине пути мы хотим видеть завершенные истории на сумму, равную половине общего числа баллов, запланированного на данную итерацию.

Подведение итогов итераций

Раз в две недели текущая итерация заканчивается и начинается новая. В конце каждой итерации заказчику демонстрируется текущая версия системы. Заказчика просят оценить внешний вид и функциональность проекта. Заказчик выражает свое мнение в виде набора новых пользовательских историй.

Заказчика часто информируют о ходе работы над проектом. Он может измерить скорость. Он может оценить, как быстро продвигается команда, и запланировать высокоприоритетные истории на ранней стадии. Короче говоря, у заказчика есть все данные и средства контроля, которые необходимы для управления проектом по его усмотрению.

Мониторинг

Мониторинг и управление XP-проектом сводятся к записи результатов каждой итерации и использованию этих данных для прогнозирования следующих итераций. Рассмотрим, например, рис. 3.1. Этот график называется *диаграммой скорости*. Обычно она вывешивается на стене «боевого командного пункта» проекта.

На диаграмме показано, сколько историй завершено, то есть прошло все автоматизированные тесты, в конце каждой недели. Хотя между неделями наблюдается некоторый разброс, видно, что в целом команда набирает в неделю примерно 42 балла.

Рассмотрим также график на рис. 3.2. Это *диаграмма выгорания*; она показывает, сколько баллов оставалось набрать в конце каждой недели

до следующей важной контрольной точки. Наклон этого графика – разумный индикатор даты завершения.

Обратите внимание, что разность между соседними столбиками на диаграмме выгорания не равна высоте столбиков на диаграмме скорости. Причина в том, что в проект добавляются новые истории. Или в том, что разработчики пересматривали оценки историй.

Если обе диаграммы висят на стене в комнате разработчиков, то, взглянув на них, всякий тут же скажет, каково состояние работы над проектом. Можно оценить, когда будет достигнута очередная контрольная точка и в какой мере расходятся объем работы и оценки. Эти диаграммы – основа основ XP и всех гибких методик. По существу, это и есть самая надежная управленческая информация.

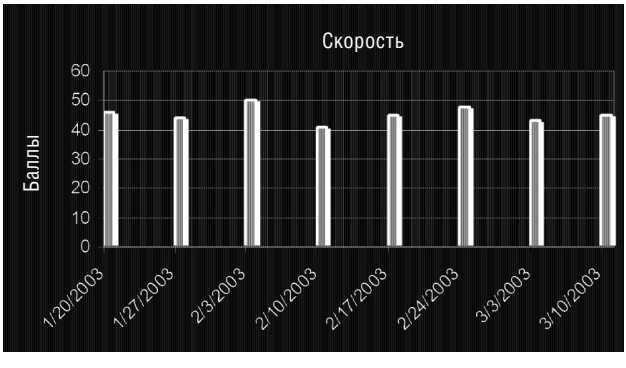


Рис. 3.1. *Диаграмма скорости*



Рис. 3.2. *Диаграмма выгорания*

Заключение

По мере перехода от одной итерации к другой и от одного выпуска к другому устанавливается предсказуемый и комфортный ритм работы над проектом. Каждый знает, чего и когда ожидать. Заказчику часто

демонстрируется, как идет работа. Вместо ноутбуков, забитых диаграммами и планами, заказчик видит работающую программу, может «потрогать ее руками» и высказать свои пожелания.

Разработчики видят разумный план, основанный на их собственных оценках и управляемый скоростью, которую они сами измерили. Разработчики выбирают те задачи, которые им нравятся, и добиваются высокого качества работы.

Менеджеры получают данные на каждой итерации и используют их для руководства и управления проектом. Не нужно ни прибегать к давлению и угрозам, ни взывать к чувству долга во имя достижения результатов к какой-то произвольно назначенной и совершенно нереалистичной дате.

Но не думайте, что это какая-то молочная река с кисельными берегами. Заинтересованные стороны не всегда бывают довольны данными на выходе процесса, особенно поначалу. Применение гибких методик не означает, что заказчик обязательно получит то, что хочет. Они просто позволяют управлять командой так, чтобы получить ценный для бизнеса продукт за минимальные деньги.

Библиография

[Beck99] Kent Beck «Extreme Programming Explained: Embrace Change», Addison-Wesley, 2004.

[Cockburn2005] Alistair Cockburn «Crystal Clear: A Human-Powered Methodolgy for Small Teams», Addison-Wesley, 2005.

[Highsmith2000] James A. Highsmith «Adaptive Software Development: A Collaborative Approach to Managing Complex Systems», Dorset House, 2000.

[Newkirk2001] James Newkirk and Robert C. Martin «Extreme Programming in Practice», Addison-Wesley, 2001.

4

Тестирование



Золото испытывают огнем,
сильных мужчин – невзгодами.

Сенека (приблизительно 3 г. до н. э. – 65 г.)

Написание автономного теста – скорее акт проектирования и документирования, чем верификации. Это действо замыкает многочисленные петли обратной связи, из которых к верификации функции относится лишь меньшая.

Разработка через тестирование

Допустим, что мы придерживались трех простых правил:

1. Не писать код, пока не будет написан автономный тест, который не проходит.
2. Не писать автономный тест большего объема, чем необходимо для неудачного завершения или неудачной компиляции.
3. Не писать больше кода, чем необходимо для того, чтобы ранее не проходивший тест завершился успешно.

При такой работе циклы будут очень короткими. Мы пишем автономный тест так, чтобы он не прошел, и ни на гран больше. А затем пишем ровно столько кода, чтобы заставить этот тест пройти. На переход от одного шага к другому уходит всего минута-другая.

Первый и самый очевидный эффект состоит в том, что для каждой функции программы имеются верифицирующие ее тесты. Комплект тестов служит опорой для дальнейшей разработки. Он поможет обнаружить случайные нарушения функциональности. Мы можем добавлять в программу новые функции или изменять ее структуру, не опасаясь при этом испортить что-то важное. Тесты показывают, продолжает ли программа работать правильно. Поэтому мы можем смело вносить изменения и улучшения.

Более важное, но менее очевидное следствие заключается в том, что написание тестов в самом начале заставляет нас сменить точку зрения. Мы должны рассматривать код, который собираемся написать, с точки зрения того, кто его вызывает. И значит, приходится задумываться не только о его функциональности, но и об интерфейсе. Написание теста до кода способствует проектированию кода, который было бы *удобно вызывать*.

Более того, если мы пишем сначала тесты, то заставляем себя проектировать программу так, чтобы она была *тестопригодна*. А удобство вызова и тестирования программ – чрезвычайно важное качество. Проектируя программу таким образом, мы разрываем ее связи с окружением. Поэтому предварительное написание тестов *заставляет нас писать слабо связанные программы*!

Еще одно важное следствие предварительного написания тестов – тот факт, что тесты могут выступать как чрезвычайно полезная документация. Если вы хотите знать, как вызвать некоторую функцию или создать объект, загляните в тест. Тесты служат примерами, помогающими другим программистами понять, как работать с кодом. Это компилируемая и исполняемая документация. Она всегда актуальна. Она не лжет.

Пример проектирования начиная с тестов

Забавы ради я недавно написал вариант игры *«Охота на Вампуса»*. Это простенькая сюжетная игра, в которой игрок перемещается по пещере, пытаясь убить Вампуса прежде, чем того его съест. Пещера представляет собой ряд залов, соединенных проходами. В каждом зале могут быть проходы, ведущие на север, юг, восток и запад. Чтобы перейти из одного зала в другой, игрок сообщает компьютеру, в каком направлении двигаться.

Одним из первых написанных для этой программы тестов был `testMove` (листинг 4.1). Этот метод создавал новый объект `WumpusGame`, соединял залы 4 и 5 проходом в восточном направлении, помещал игрока в зал 4, выдавал команду перемещения на восток, а затем проверял, находится ли игрок в зале 5.

Листинг 4.1

```
[Test]
public void TestMove()
{
  WumpusGame g = new WumpusGame();
  g.Connect(4,5,"E");
  g.GetPlayerRoom(4);
  g.East();
  Assert.AreEqual(5, g.GetPlayerRoom());
}
```

Весь этот код был написан до того, как я приступил к кодированию класса `WumpusGame`. Я последовал совету Уорда Каннингэма и написал тест так, как следует читать. Я полагал, что смогу обеспечить успешное прохождение этого теста, написав код в соответствии со структурой, подразумеваемой тестом. Такой подход называется *ментальным программированием* (Intentional programming). Вы должны выразить свои намерения в тесте еще до его реализации, сделав их максимально простыми и удобочитаемыми. Идея в том, что простота и ясность будут способствовать написанию программы с хорошей структурой.

Ментальное программирование подвело меня к интересному проектному решению. В тесте класс `Room` вообще не используется. Действие *соединения* (Connect) одного зала с другим выражает мое намерение. При этом я не вижу никакой необходимости в классе `Room`. Для представления залов можно использовать просто целые числа.

Это может показаться противоречащим здравому смыслу. Ведь вся программа построена на залах: мы переходим из зала в зал, выясняем, что находится в зале, и т. д. Быть может, вытекающий из моих намерений дизайн порочен, раз он не содержит класса `Room`?

Я мог бы возразить, что идея соединений гораздо важнее для игры, чем идея зала. Я мог бы добавить, что начальный тест указал хороший способ решения задачи. И я действительно так думаю, но сейчас дело

не в этом, а в том, что тест высветил важный аспект дизайна на очень ранней стадии. *Предварительное написание тестов – это акт выбора проектных решений.*

Обратите внимание, что тест рассказывает о том, как работает программа. Исходя из этой простой спецификации, было бы несложно написать все четыре упомянутых в тесте метода класса `WumpusGame`. Без особого напряжения можно было бы поименовать и написать команды перемещения в других направлениях. Если впоследствии мы захотим узнать, как соединить два зала или переместиться в определенном направлении, то тест даст недвусмысленные ответы на эти вопросы. Этот тест выступает в роли компилируемого и исполняемого документа, описывающего программу.

Изоляция тестов

Написание тестов до кода нередко выявляет части программы, которые нуждаются в разъединении. Например, на рис. 4.1 приведена простая UML-диаграмма приложения для расчета заработной платы. Класс `Payroll` использует класс `EmployeeDatabase` для получения объекта `Employee`, затем просит `Employee` вычислить свою зарплату, передает величину зарплаты объекту `CheckWriter`, чтобы тот выписал чек, и напоследок передает сумму платежа объекту `Employee` и записывает объект обратно в базу данных.

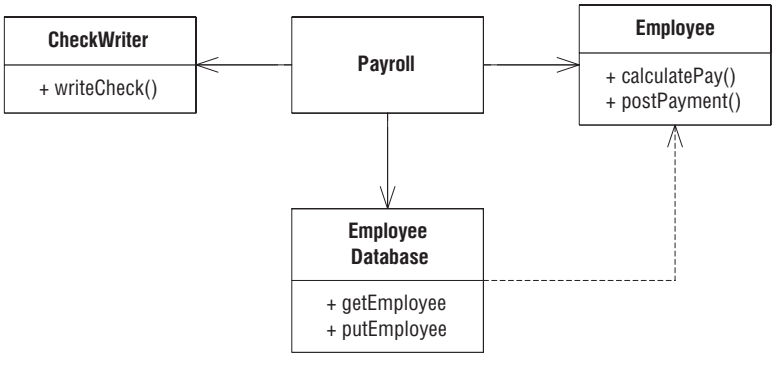


Рис. 4.1. *Связанная модель расчета заработной платы*

Предположим, что весь этот код еще не написан. Пока это всего лишь диаграмма, нарисованная на доске в ходе эскизного проектирования.[1] Теперь необходимо написать тесты, описывающие поведение объекта `Payroll`. Но при попытке сделать это возникает ряд проблем. Во-первых, какую СУБД мы будем использовать? Ведь приложение должно читать данные из некоторой базы. Необходимо ли иметь полнофункциональную базу данных перед тем, как тестировать класс `Payroll`? Какие

[1] [Jeffries2001]

данные в нее загрузить? Во-вторых, как проверить, что напечатан правильный чек? Не можем же мы написать автоматизированный тест, который будет смотреть на напечатанный принтером чек и сверять проставленную сумму!

Решение этих проблем дает паттерн Объект-имитация (Mock Object).[1] Мы можем вставить между всеми классами, сотрудничающими с `Payroll`, интерфейсы, а затем создать тестовые заглушки, реализующие эти интерфейсы.

Такая структура показана на рис. 4.2. Теперь класс `Payroll` применяет интерфейсы для взаимодействия с `EmployeeDatabase`, `CheckWriter` и `Employee`. Для реализации интерфейсов было создано три объекта-имитации. Объект `PayrollTest` опрашивает их, чтобы удостовериться в том, что `Payroll` управляет ими правильно.

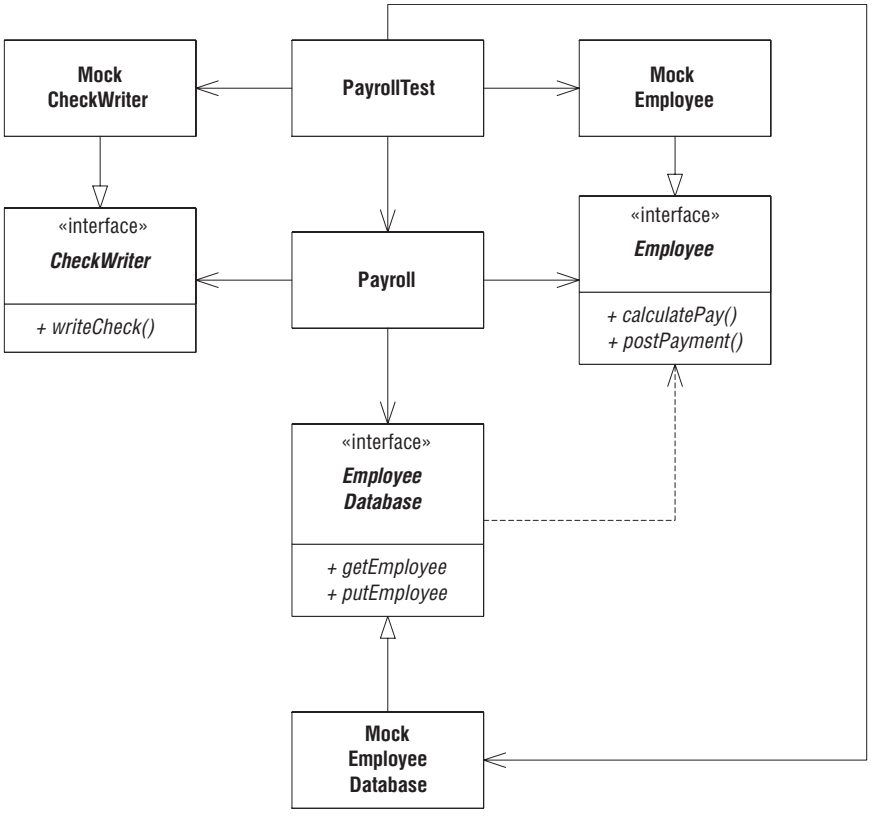


***Рис. 4.2.** Класс Payroll с разорванными связями, для тестирования используются объекты-имитации*

[1] [Mackinnon2000]

В листинге 4.2 отражено намерение теста. Мы создаем подходящие объекты-имитации, передаем их объекту Payroll, требуем, чтобы объект Payroll начислил зарплату всем работникам, а затем просим имитации проверить, что все чеки правильно выписаны, а платежи занесены и зарегистрированы.

Конечно, этот тест просто проверяет, что Payroll вызвал все нужные методы, передав им правильные данные. Тест не контролирует правильность выписки чеков и обновления настоящей базы данных. Но он проверяет, что класс Payroll в изоляции от всего остального ведет себя, как положено.

Листинг 4.2. TestPayroll

```
[Test]

public void TestPayroll()
{
  MockEmployeeDatabase db = new MockEmployeeDatabase();
  MockCheckWriter w = new MockCheckWriter();
  Payroll p = new Payroll(db, w);
  p.PayEmployees();
  Assert.IsTrue(w.ChecksWereWrittenCorrectly());
  Assert.IsTrue(db.PaymentsWerePostedCorrectly());
}
```

Может возникнуть вопрос, зачем нужен класс MockEmployee. Вроде бы можно воспользоваться настоящим классом Employee вместо имитации. Будь это так, я бы и воспользовался без зазрения совести. Но в данном случае я предположил, что класс Employee сложнее, чем необходимо для проверки работы Payroll.

Разделение в придачу

Разрыв связей класса Payroll с окружением – это прекрасно. Теперь мы можем подставлять разные базы данных и выписыватели чеков для тестирования и обобщения приложения. Интересно отметить, что такое разделение было обусловлено потребностями тестирования. Как видим, необходимость изолировать тестируемый модуль заставила нас разорвать его связи способом, который благотворно повлиял на структуру программы в целом. *Предварительное написание тестов улучшает дизайн программы.*

Значительная часть этой книги посвящена принципам управления зависимостями, то есть рекомендациям и приемам разрыва связей между классами и пакетами. Вы убедитесь в полезности этих принципов, если будете применять их как часть стратегии автономного тестирования. Именно автономные тесты дают импульс и направление для разрыва связей.

Приемочные тесты



Автономные тесты – необходимый, но не достаточный инструментарий верификации. С их помощью можно лишь убедиться, что небольшие элементы системы работают, как ожидается, но проверить функционирование системы в целом они не в состоянии. Автономные тесты – это белые ящики[1], которые проверяют отдельные механизмы работы системы. Приемочные тесты – это черные ящики[2], которые проверяют, удовлетворены ли требования заказчика.

Приемочные тесты пишутся людьми, которые не знают о внутреннем устройстве системы. Это может быть сам заказчик, бизнес-аналитик или специалист по контролю качества. Приемочные тесты автоматизированы. Обычно они составляются на специальном языке спецификаций, понятном людям, не обладающим техническими навыками.

Приемочные тесты – вершина документирования функции. После того как заказчик написал приемочные тесты, проверяющие, что некоторая функция работает правильно, программисты могут, прочитав их текст, полностью разобраться в назначении функции. Поэтому как автономные тесты служат компилируемой и исполняемой документацией внутреннего устройства системы, так приемочные тесты являются компилируемой и исполняемой документацией функций системы. Короче говоря, *приемочные тесты играют роль требований к системе.*

Заодно акт предварительного написания приемочных тестов оказывает существенное влияние на архитектуру. Чтобы система поддавалась тестированию, связи следует разрывать на верхних уровнях архитектуры. Например, пользовательский интерфейс следует отделить от бизнес-правил, так чтобы приемочные тесты могли получить доступ к бизнес-правилам, минуя ГИП.

На первых итерациях проекта возникает искушение выполнять приемочные тесты вручную. Это нежелательно, так как лишает эти первые итерации стимула к разрыву связей, который и появляется-то в связи необходимостью автоматизировать приемочное тестирование. Если с самой первой итерации вы непреложно знаете, что должны автоматизировать приемочные тесты, то будете совершенно по-другому подходить к выбору архитектуры. Как автономные тесты приводят к отличным проектным решениям в малом, так приемочные тесты способствуют выбору лучшей архитектуры в целом.

---

[1] Тест, который знает о внутренней структуре тестируемого модуля и зависит от нее.

[2] Тест, который не знает о внутренней структуре тестируемого модуля и не зависит от нее.

Снова рассмотрим приложение для расчета зарплаты. На первой итерации должна быть реализована возможность добавлять и удалять записи о работниках в базе данных. Мы также должны уметь создавать платежные чеки для имеющихся в базе данных работников. К счастью, нам придется иметь дело только с работниками на твердом окладе. Работников, тарифицируемых иначе, мы отложим до более поздней итерации.

Пока что мы не написали никакого кода и не потратили время на его проектирование. Самое время начать думать о приемочных тестах. И снова на помощь приходит ментальное программирование. Нам следует писать приемочные тесты так, как они, на наш взгляд, должны выглядеть, и соответственно проектировать систему.

Я хочу, чтобы приемочные тесты было удобно писать и легко изменять. Я хочу хранить их в какой-то системе общего пользования во внутренней сети, чтобы их можно было выполнить в любой момент. Поэтому я воспользуюсь инструментом с открытым исходным кодом FitNesse[1], который позволяет писать приемочные тесты в виде простых веб-страниц и запускать их из броузера.

На рис. 4.3 показан пример приемочного теста, написанного в FitNesse. На первом шаге теста в систему добавляются два работника. На втором шаге им начисляется зарплата. На третьем шаге проверяется, что чеки выписаны правильно. Мы предполагаем, что подоходный налог начисляется по плоской шкале и составляет 20%.

Сначала добавляем двух работников

Add employees.		
id	name	salary
1	Jeff Languid	1000.00
2	Kelp Holland	2000.00

Затем начисляем им зарплату

Create paychecks.	
pay date	check number
1/31/2001	1000

Проверяем, что вычтен 20-процентный налог

Inspect paychecks.		
id	gross pay	net pay
1	1000	800
2	2000	1600

Рис. 4.3. *Пример приемочного теста*

[1] *www.fitnesse.org*

Очевидно, что заказчику очень удобно читать и писать такие тесты. Но вдумайтесь, какие это влечет последствия для структуры системы. Первые две таблицы теста – это функции приложения для расчета зарплаты. Если бы мы писали такое приложение в виде повторно используемого каркаса, то они соответствовали бы функциям API. И действительно, чтобы FitNesse могла вызвать эти функции, такой API должен быть написан.[1]

Архитектура в придачу

Отметим то давление, которое приемочные тесты оказывают на архитектуру всей системы для расчета зарплаты. С самого начала задумавшись о приемочных тестах, мы пришли к необходимости API для функций системы. Понятно, что ГИП сможет использовать этот API для достижения своих целей. Отметим также, что печать чеков следует отделить от функции `Create Paychecks` (Создать чеки). И то и другое – хорошие архитектурные решения.

Заключение

Чем проще выполнить комплект тестов, тем чаще следует это делать. Чем чаще проводятся тесты, тем раньше обнаруживаются любые отклонения. Если можно выполнять все тесты несколько раз в день, то система никогда не будет находиться в неработоспособном состоянии более нескольких минут. Это разумная цель. Мы просто не позволяем системе скатиться по наклонной плоскости. Если мы достигли некоторого уровня работоспособности, то никогда не опустимся на более низкий уровень.

И все же верификация – лишь одно из преимуществ написания тестов. И автономные, и приемочные тесты – своего рода формы документации. Компилируемой и исполняемой, а потому точной и надежной. К тому же тесты пишутся на не допускающем неоднозначного толкования языке, понятном соответствующей аудитории. Программисты могут прочитать автономные тесты, потому что они написаны на знакомом им языке программирования. Заказчики могут прочитать приемочные тесты, потому что они написаны на простом табличном языке.

Быть может, самое важное достоинство тестирования – это его влияние на архитектуру и дизайн. Чтобы модуль или приложение поддавались тестированию, необходимо разорвать его связи с окружением. Чем лучше тестируется программа, тем меньше имеется связей. Намерение

[1] Описание способа вызова функций API из системы FitNesse выходит за рамки данной книги. Дополнительную информацию см. в документации по FitNesse. См. также [Mugridge2005].

иметь полный набор приемочных и автономных тестов весьма положительно сказывается на структуре программы.

Библиография

[Jeffries2001] Ron Jeffries «Extreme Programming Installed», Addison-Wesley, 2001.

[Mackinnon2000] Tim Mackinnon, Steve Freeman, and Philip Craig «Endo-Testing: Unit Testing with Mock Objects». В сборнике Giancarlo Succi and Michele Marchesi «Extreme Programming Examined», Addison-Wesley, 2001.

[Mugridge2005] Rick Mugridge and Ward Cunningham «Fit for Developing Software: Framework for Integrated Tests», Addison-Wesley, 2005.

5

Рефакторинг



Единственный фактор, становящийся дефицитным в мире изобилия, – человеческое внимание.

Кэвин Кэлли, журнал *Wired*

Это глава о человеческом внимании, о том, что нужно с вниманием относиться к тому, что вы делаете, стараясь выполнить свою работу как можно лучше. Это глава о том, что есть разница между просто сделанным и сделанным правильно. Это глава о ценности, которой обладает структура нашего кода.

В ставшей классической книге «Refactoring» Мартин Фаулер определяет рефакторинг, как «процесс изменения программной системы таким образом, что ее внешнее поведение не изменяется, а внутренняя структура улучшается».[1] Но зачем улучшать структуру работающей программы? Как насчет поговорки «От добра добра не ищут»?

У каждого программного модуля есть три функции. Во-первых, та функция, которую он реализует во время выполнения. Ради нее модуль и пишется. Во-вторых, модуль должен допускать изменение. Почти все модули на протяжении своей жизни изменяются, и задача разработчика – обеспечить возможность и простоту внесения изменения. Модуль, который трудно изменять, следует считать непригодным и нуждающимся в исправлении, даже если он работает. В-третьих, модуль должен быть доступным для понимания. Разработчики, не знакомые с модулем, должны иметь возможность прочитать и понять его, не прилагая сверхъестественных умственных усилий. Модуль, который не информативен, также непригоден и нуждается в исправлении.

Что необходимо для того, чтобы модуль было легко читать и изменять? Значительная часть этой книги посвящена принципам и паттернам, основная цель которых – научить вас создавать гибкие и адаптируемые модули. Но требуется и нечто большее, чем одни лишь принципы и паттерны. Нужно еще внимание. И дисциплина. И стремление создавать красивые вещи.

Простой пример рефакторинга: генерация простых чисел

Рассмотрим код в листинге 5.1. Эта программа генерирует простые числа. Она представляет собой одну большую функцию, содержащую много переменных с однобуквенными именами и комментарии, помогающие разобраться в ее устройстве.

Листинг 5.1. GeneratePrimes.cs, версия 1

```
/// <remark>
/// Этот класс генерирует простые числа, не превышающие заданного
/// пользователем порога. В качестве алгоритма используется решето
/// Эратосфена.
///
/// Эратосфен Киренский, 276 г. до н. э., Кирена, Ливия --
/// 194 г. до н. э., Александрия. Впервые измерил окружность Земли.
/// Известен также работами по составлению календаря с високосными
/// годами. Работал в Александрийской библиотеке.
///
/// Сам алгоритм прост. Пусть дан массив целых чисел, начинающийся
/// с 2. Вычеркиваем все кратные 2. Находим первое невычеркнутое
```

[1] [Fowler99], стр. xvi

```
/// число и вычеркиваем все его кратные. Повторяем, пока не
/// дойдем до квадратного корня из максимального значения.
///
/// Написал Роберт К. Мартин 9.12.1999 на языке Java
/// Перевел на C# Мика Мартин 12.01.2005.
///</remark>

using System;

/// <summary>
/// автор: Роберт К. Мартин
/// </summary>

public class GeneratePrimes
{
  ///<summary>
  /// Генерирует массив простых чисел.
  ///</summary>
  ///
  /// <param name="maxValue">Верхний порог.</param>
  public static int[] GeneratePrimeNumbers(int maxValue)
  {
    if (maxValue >= 2) // единственный допустимый случай
    {
      // объявления
      int s = maxValue + 1; // размер массива
      bool[] f = new bool[s];
      int i;

      // инициализировать элементы массива значением true.
      for (i = 0; i < s; i++)
        f[i] = true;

      // исключить заведомо не простые числа
      f[0] = f[1] = false;

      // решето
      int j;
      for (i = 2; i < Math.Sqrt(s) + 1; i++)
      {
        if(f[i]) // если i не вычеркнуто, вычеркнуть его кратные.
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // кратное - не простое число
        }
      }

      // сколько оказалось простых чисел?
      int count = 0;
      for (i = 0; i < s; i++)
      {
```

```
        if (f[i])
         count++; // увеличить счетчик
      }

      int[] primes = new int[count];

      // поместить простые числа в результирующий массив
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i]) // если простое
          primes[j++] = i;
      }
      return primes; // вернуть простые числа
    }
    else // maxValue < 2
      return new int[0]; // если входные данные не корректны,
                         // возвращаем пустой массив
  }
}
```

Автономное тестирование

Автономный тест для класса `GeneratePrimes` приведен в листинге 5.2. Мы применили выборочный подход и проверяем, что генератор порождает простые числа до 0, 2, 3 и 100. В первом случае простых чисел вообще не должно быть. Во втором случае должно быть одно простое число – 2. В третьем случае должно быть два простых числа – 2 и 3. А в четвертом – 25 простых чисел, последнее из которых равно 97. Если все эти тесты проходят, я считаю, что генератор работает правильно. Конечно, это небезупречное предположение, но я не могу представить себе ситуацию, когда эти тесты проходили бы, а функция тем не менее не работала.

Листинг 5.2. GeneratePrimesTest.cs

```
using NUnit.Framework;

[TestFixture]
public class GeneratePrimesTest
{
  [Test]
  public void TestPrimes()
  {
    int[] nullArray = GeneratePrimes.GeneratePrimeNumbers(0);
    Assert.AreEqual(nullArray.Length, 0);

    int[] minArray = GeneratePrimes.GeneratePrimeNumbers(2);
    Assert.AreEqual(minArray.Length, 1);
    Assert.AreEqual(minArray[0], 2);

    int[] threeArray = GeneratePrimes.GeneratePrimeNumbers(3);
```

```
    Assert.AreEqual(threeArray.Length, 2);
    Assert.AreEqual(threeArray[0], 2);
    Assert.AreEqual(threeArray[1], 3);

    int[] centArray = GeneratePrimes.GeneratePrimeNumbers(100);
    Assert.AreEqual(centArray.Length, 25);
    Assert.AreEqual(centArray[24], 97);
  }
}
```

Рефакторинг

Для рефакторинга этой программы я воспользовался Visual Studio с надстройкой *ReSharper* компании *JetBrains*. Этот инструмент позволяет безо всякого труда извлекать методы и переименовывать переменные и классы.

Довольно ясно, что главную функцию следует разбить на три. В первой должны инициализироваться все переменные и решето. Во второй мы просеиваем массив через решето, а в третьей загружаем результаты просеивания в массив целых чисел. Чтобы проявить эту структуру, я превратил эти функции в отдельные методы (листинг 5.3). Я также удалил лишние комментарии и переименовал класс в `PrimeGenerator`. Все тесты по-прежнему работают.

Выделение трех функций заставило меня преобразовать некоторые локальные переменные в статические поля класса. В результате стало гораздо яснее, какие переменные действительно локальны, а у каких область видимости шире.

Листинг 5.3. PrimeGenerator.cs, версия 2

```
///<remark>
/// Этот класс генерирует простые числа, не превышающие заданного
/// пользователем порога. В качестве алгоритма используется решето
/// Эратосфена.
/// Пусть дан массив целых чисел, начинающийся с 2:
/// Находим первое невычеркнутое число и вычеркиваем все его
/// кратные. Повторяем, пока в массиве не останется кратных.
///</remark>
using System;

public class PrimeGenerator
{
  private static int s;
  private static bool[] f;
  private static int[] primes;

  public static int[] GeneratePrimeNumbers(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
```

```
  else
  {
    InitializeSieve(maxValue);
    Sieve();
    LoadPrimes();
    return primes; // вернуть простые числа
  }
}

private static void LoadPrimes()
{
  int i;
  int j;

  // сколько оказалось простых чисел?
  int count = 0;
  for (i = 0; i < s; i++)
  {
    if (f[i])
      count++; // увеличить счетчик
  }

  primes = new int[count];

  // поместить простые числа в результирующий массив
  for (i = 0, j = 0; i < s; i++)
  {
    if (f[i]) // если 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, вычеркнуть его кратные.
    {
      for (j = 2 * i; j < s; j += i)
        f[j] = false; // кратное - не простое число
    }
  }
}

private static void InitializeSieve(int maxValue)
{
  // объявления
  s = maxValue + 1; // размер массива
```

```
      f = new bool[s];
      int i;

      // инициализировать элементы массива значением true.
      for (i = 0; i < s; i++)
        f[i] = true;

      // исключить заведомо не простые числа
      f[0] = f[1] = false;
    }
  }
```

Метод `InitializeSieve` получился несколько беспорядочным, поэтому я его подчистил (листинг 5.4). Во-первых, я заменил все вхождения переменной `s` выражением `f.Length`. Затем я изменил имена всех трех методов, сделав их более выразительными. И наконец, реорганизовал текст метода `InitializeArrayOfIntegers` (бывший `InitializeSieve`), чтобы его было проще читать. Все тесты по-прежнему завершаются успешно.

Листинг 5.4. PrimeGenerator.cs, версия 3 (неполная)

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

  public static int[] GeneratePrimeNumbers(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      InitializeArrayOfIntegers(maxValue);
      CrossOutMultiples();
      PutUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void InitializeArrayOfIntegers(int maxValue)
  {
    // объявления
    f = new bool[maxValue + 1];
    f[0] = f[1] = false; //не простые числа и не кратные
    for (int i = 2; i < f.Length; i++)
      f[i] = true;
  }
}
```

Далее я обратился к методу `CrossOutMultiples`. В этом и в других методах есть предложения вида `if(f[i] == true)`. Идея была в том, чтобы про-

верить, вычеркнуто ли число i, поэтому я изменил имя f на unCrossed. Но тогда получаются уродливые предложения вида unCrossed[i] = false. Я не люблю двойного отрицания. Поэтому я изменил имя массива на isCrossed и поменял соответственно интерпретацию всех булевских значений. Тесты по-прежнему проходят.

Я избавился от инициализации isCrossed[0] и isCrossed[1] значением true и просто гарантировал, что нигде нет обращений к элементам массива isCrossed с индексом меньше 2. Я преобразовал внутренний цикл метода CrossOutMultiples в метод CrossOutMultiplesOf. Я решил также, что запись if (isCrossed[i] == false) недостаточно понятна, поэтому завел метод NotCrossed и переписал это предложение if в виде if (NotCrossed(i)). Тесты по-прежнему проходят.

Я потратил некоторое время на написание комментария, в котором объясняется, почему достаточно итерировать только до квадратного корня из размера массива. Это побудило меня перенести вычисление в отдельный метод, где можно было бы разместить комментарий. Попутно я осознал, что этот квадратный корень не меньше любого простого множителя находящихся в массиве целых чисел. Поэтому я выбрал соответствующее имя для переменной и работающего с ней метода. Результаты всех этих изменений показаны в листинге 5.5. Тесты по-прежнему проходят.

Листинг 5.5. PrimeGenerator.cs, версия 4 (неполная)

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

  public static int[] GeneratePrimeNumbers(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      InitializeArrayOfIntegers(maxValue);
      CrossOutMultiples();
      PutUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void InitializeArrayOfIntegers(int maxValue)
  {
    isCrossed = new bool[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 + 1; i++)
    {
      if(NotCrossed(i))
        CrossOutputMultiplesOf(i);
    }
  }

  private static int CalcMaxPrimeFactor()
  {
    // Вычеркиваем все кратные p, где p – простое число. Таким
    // образом, любое вычеркнутое число разлагается в произведение
    // множителей p и q. Если p > sqrt из размера массива, то q не
    // может быть больше 1. Таким образом, p – максимальный простой
    // множитель всех чисел в массиве и одновременно верхний предел
    // итераций.
    double maxPrimeFactor = Math.Sqrt(isCrossed.Length) + 1;
    return (int) maxPrimeFactor;
  }

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

  private static bool NotCrossed(int i)
  {
    return isCrossed[i] == false;
  }
}
```

Последний метод, который нуждается в рефакторинге, – `PutUncrossedIntegersIntoResult`. Он состоит из двух частей. В первой подсчитывается количество невычеркнутых целых чисел в массиве и создается результирующий массив такого размера. Во второй невычеркнутые числа помещаются в результирующий массив. Я перенес первую часть в отдельный метод и выполнил кое-какую очистку (листинг 5.6). Тесты по-прежнему проходят.

Листинг 5.6. PrimeGenerator.cs, версия 5 (неполная)

```
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++; // увеличить счетчик
    }
    return count;
  }
```

Последнее прочтение



Напоследок я еще раз просмотрел всю программу от начала до конца, как если бы читал доказательство геометрической теоремы. Это важный шаг. До сих пор я подвергал рефакторингу отдельные фрагменты. Теперь я хочу понять, воспринимается ли программа как единое *гармоничное* целое.

Я сразу же понял, что мне не нравится имя `InitializeArrayOfIntegers`. Инициализируется на самом деле не массив целых чисел, а массив булевских значений. Но `InitializeArrayOfBooleans` трудно назвать улучшением. В действительности мы в этом методе помечаем все целые числа как невычеркнутые, чтобы позже можно было вычеркнуть кратные простых чисел. Поэтому я изменил имя на `UncrossIntegersUpTo`. Кроме того, мне не понравилось имя `isCrossed` для массива булевских значений, и я переименовал его в `crossedOut`. Тесты по-прежнему проходят.

Кому-то может показаться, что я увлекся всякими переименованиями, но, имея инструмент рефакторинга, такие вольности можно себе позволить, ведь они практически ничего не стоят. Впрочем, и без инструмента простой поиск и замена обходятся довольно дешево. А тесты гарантируют, что мы ничего случайно не испортили.

Не знаю, что я курил, когда писал всю эту чушь про `maxPrimeFactor`. Черт! Ведь квадратный корень из размера массива необязательно является простым числом. Этот метод *не* вычисляет максимальный простой множитель. И пояснительный комментарий попросту *неверен*. Поэтому я переписал комментарий, чтобы лучше объяснить, откуда взялся

квадратный корень, и соответственно переименовал переменные.[1] Тесты по-прежнему проходят.

А какого дьявола тут делает +1? Наверное, это просто паранойя. Я испугался, что дробный квадратный корень после преобразования в целое число окажется недостаточным для ограничения числа итераций. Но это же глупость. Истинный ограничитель итераций – это наибольшее простое число, меньшее или равное квадратному корню из размера массива. Избавлюсь от +1.

Все тесты проходят, но последнее изменение что-то меня смущает. Я понимаю, зачем нужен квадратный корень, но меня терзает опасение, что могли остаться какие-то граничные случаи, не покрываемые тестами. Поэтому напишу-ка я еще тесты, проверяющие, что в списке простых чисел от 2 до 500 не встречается ни одного составного числа. (См. метод `TestExhaustive` в листинге 5.8.) Новый тест проходит, и мои страхи улеглись.

Остаток программы не вызывает вопросов. Думаю, что на этом можно закончить. Окончательная версия представлена в листингах 5.7 и 5.8.

Листинг 5.7. PrimeGenerator.cs (окончательная версия)

```
///<remark>
/// Этот класс генерирует простые числа, не превышающие заданного
/// пользователем порога. В качестве алгоритма используется решето
/// Эратосфена.
/// Пусть дан массив целых чисел, начинающийся с 2:
/// Находим первое невычеркнутое число и вычеркиваем все его
/// кратные. Повторяем, пока в массиве не останется кратных.
///</remark>
using System;
public class PrimeGenerator
{
  private static bool[] crossedOut;
  private static int[] result;

  public static int[] GeneratePrimeNumbers(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      UncrossIntegersUpTo(maxValue);
      CrossOutMultiples();
```

[1] Мне довелось наблюдать, как ту же программу подвергал рефакторингу Кент Бек. Он вообще обошелся без квадратного корня. И объяснял это тем, что применение квадратного корня понять трудно, а тесты будут проходить, даже если итерировать до конца массива. Но я не могу заставить себя не заботиться об эффективности. Полагаю, корни растут из моего увлечения языком ассемблера.

```
    PutUncrossedIntegersIntoResult();
    return result;
  }
}

private static void UncrossIntegersUpTo(int maxValue)
{
  crossedOut = new bool[maxValue + 1];
  for (int i = 2; i < crossedOut.Length; i++)
    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++; // увеличить счетчик
  }
  return count;
}

private static void CrossOutMultiples()
{
  int limit = DetermineIterationLimit();
  for (int i = 2; i <= limit; i++)
  {
    if(NotCrossed(i))
      CrossOutputMultiplesOf(i);
  }
}

private static int DetermineIterationLimit()
{
  // У каждого составного числа в этом массиве есть простой
  // множитель, меньший или равный квадратному корню из размера
  // массива, поэтому необязательно вычеркивать кратные, большие
  // корня.
  double iterationLimit = Math.Sqrt(crossedOut.Length);
  return (int) iterationLimit;
```

```
    }

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

    private static bool NotCrossed(int i)
    {
      return crossedOut[i] == false;
    }
  }
```

Листинг 5.8. GeneratePrimeTest.cs (окончательная версия)

```
using NUnit.Framework;

[TestFixture]
public class GeneratePrimesTest
{
  [Test]
  public void TestPrimes()
  {
    int[] nullArray = PrimeGenerator.GeneratePrimeNumbers(0);
    Assert.AreEqual(nullArray.Length, 0);

    int[] minArray = PrimeGenerator.GeneratePrimeNumbers(2);
    Assert.AreEqual(minArray.Length, 1);
    Assert.AreEqual(minArray[0], 2);

    int[] threeArray = PrimeGenerator.GeneratePrimeNumbers(3);
    Assert.AreEqual(threeArray.Length, 2);
    Assert.AreEqual(threeArray[0], 2);
    Assert.AreEqual(threeArray[1], 3);

    int[] centArray = PrimeGenerator.GeneratePrimeNumbers(100);
    Assert.AreEqual(centArray.Length, 25);
    Assert.AreEqual(centArray[24], 97);
  }

  [Test]
  public void TestExhaustive()
  {
    for (int i = 2; i<500; i++)
      VerifyPrimeList(PrimeGenerator.GeneratePrimeNumbers(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.IsTrue(n%factor != 0);
  }
}
```

Заключение

Окончательная версия выглядит и читается гораздо лучше, чем изначальная. Да и работает она чуть быстрее. Я доволен результатом. Программу стало намного проще понять и модифицировать. Кроме того, в новой структуре отдельные части изолированы друг от друга. Это тоже облегчает модификацию.

Быть может, вас беспокоит, что вынесение в отдельные методы кода, который вызывается только один раз, отрицательно сказывается на производительности. Но я думаю, что понятность в большинстве случаев стоит нескольких наносекунд. Мой совет таков: считайте, что эта стоимость пренебрежимо мала, и подождите, пока не будет доказано противное.

Стоит ли результат затраченных усилий? Ведь программа-то и так работала. Но я настоятельно рекомендую *всегда* подвергать рефакторингу *все* модули, которые вы пишете или сопровождаете. Временные затраты малы по сравнению с тем, что вы сэкономите себе и другим в ближайшем будущем.

Рефакторинг – это как уборка со стола после обеда. В первый раз, когда вы решаете не убирать, обед заканчивается быстрее. Но отсутствие чистой посуды и места для работы приведет к тому, что на следующий день обед придется готовить дольше. И поэтому вы снова не захотите делать уборку. В общем, если не убирать за собой, то *сегодня* удастся пообедать быстрее. Но беспорядок накапливается. И в конце концов вы станете тратить непомерно много времени на поиск нужной утвари, соскабливание присохшей к тарелкам еды, отмывание их от жира и т. д. На то, чтобы пообедать, будет уходить вечность. Так что отказ от уборки не ускоряет обед.

Цель рефакторинга, как следует из этой главы, состоит в том, чтобы устраивать уборку в своем коде ежедневно, ежечасно, ежеминутно. Мы не хотим, чтобы беспорядок накапливался. Мы не хотим долотом отскребать присохшие за долгое время крошки. Мы хотим расширять и модифицировать свои системы с минимальными усилиями. И самое важное условие для этого – поддерживать чистоту кода.

Эту мысль невозможно переоценить. Все принципы и паттерны, описываемые в этой книге, сведутся к нулю, если применять их к замусоренному коду. Прежде чем приступать к принципам и паттернам, займитесь чистотой кода.

Библиография

[Fowler99] Martin Fowler «Refactoring: Improving the Design of Existing Code», Addison-Wesley, 1999.[1]

[1] Мартин Фаулер «Рефакторинг. Улучшение существующего кода». – Пер. с англ. – СПб.: Символ-Плюс, 2002.

6

Эпизод программирования



Проектирование и программирование –
виды человеческой деятельности;
стоит об этом забыть – и все пропало.

Бьярн Страуструп, 1991

Чтобы продемонстрировать технику гибкого программирования, Боб Косс (БК) и Боб Мартин (БМ) решили сообща написать небольшое приложение. А вы пока будете пассивным наблюдателем, подобно мухе на стене. Мы станем применять разработку через тестирование и многократно подвергать приложение рефакторингу. Описанное ниже – довольно достоверное воспроизведение эпизода программирования, имевшего место в гостиничном номере в конце 2000 года.

По ходу дела мы допустили массу ошибок. В коде, в логике, при проектировании и при формулировке требований. Вы увидите, как мы выпутываемся из всех этих ситуаций, сначала выявляя ошибки и недопонимание, а затем исправляя их. Это довольно беспорядочный процесс, как, впрочем, и все процессы с участием человека. Но порядок, оказавшийся результатом столь хаотического процесса, впечатляет.

Поскольку программа подсчитывает очки, набранные в партии в боулинг, то неплохо было бы знать правила этой игры. Если вы с ними не знакомы, обратитесь к врезке, которая приведена ниже.

Правила игры в боулинг

Игра в боулинг заключается в бросании шаров размером с футбольный мяч по узкой дорожке по направлению к десяти расставленным деревянным кеглям. Цель – сбить максимальное число кеглей за один бросок.

Игра состоит из десяти фреймов. В начале каждого фрейма все десять кеглей стоят. Игроку даются две попытки на то, чтобы их сбить.

Если игрок сбивает все кегли с первой попытки, то говорят, что он сделал «страйк», и на этом фрейм заканчивается. Если первым шаром сбить все кегли не получилось, но оставшиеся сбиты вторым шаром, то игрок сделал «спэа». После второго броска фрейм заканчивается, даже если какие-то кегли остались стоять.

Фрейм, который закончился *страйком*, добавляет к результату предыдущего фрейма десять очков плюс количество кеглей, сбитых следующими двумя бросками. Фрейм, закончившийся *спэа*, добавляет к результату предыдущего фрейма десять очков плюс количество кеглей, сбитых следующим броском. При любом другом исходе к результату предыдущего фрейма добавляется количество кеглей, сбитых двумя бросками в данном фрейме.

Если страйк выбит в десятом фрейме, то игрок может бросить еще два шара, чтобы завершить подсчет очков для данного страйка. Аналогично если в десятом фрейме выбит спэа, то игрок может бросить еще один шар. Таким образом, в десятом фрейме может быть брошено три, а не два шара.

1 4	4 5	6 ◢	5 ◢	■	0 1	7 ◢	6 ◢	■	2 ◢ 6
5	14	29	49	60	61	77	97	117	133

На карточке выше представлена типичная, хотя и не очень удачная, игра. В первом фрейме игрок сбил одну кеглю первым шаром и еще четыре вторым. Следовательно, счет игрока в этом фрейме

равен 5. Во втором фрейме первым шаром сбито четыре кегли, а вторым – еще пять. В сумме это дает девять очков, а вместе с результатом предыдущего фрейма – 14.

В третьем фрейме игрок сбил шесть кеглей первым шаром и все остальные – вторым. Очки за этот фрейм не начисляются, пока не будет брошен следующий шар.

В четвертом фрейме игрок сбил первым шаром пять кеглей. Это позволяет завершить подсчет очков для спэа в третьем фрейме. Счет для фрейма 3: 10 плюс счет во фрейме 2 (14) плюс количество кеглей, сбитых первым шаром во фрейме 4 (5). Итого 29. Последним шаром во фрейме 4 выбит спэа.

Фрейм 5 – страйк. Это дает возможность подсчитать очки во фрейме 4: 29 + 10 + 10 = 49.

Фрейм 6 – катастрофа. Первый шар скатился в желоб, не сбив ни одной кегли. Второй шар сбил всего одну кеглю. Счет для страйка во фрейме 5 равен 49 + 10 + 0 + 1 = 60.

Остальное можете подсчитать самостоятельно.

Игра в боулинг

БМ: Не поможешь мне написать программульку для подсчета очков в боулинге?

БК: *(размышляя про себя: этика парного программирования в XP говорит, что нельзя отказывать, когда тебя просят о помощи. Полагаю, что это особенно относится к случаю, когда просит начальник)* Конечно, Боб, с удовольствием.

БМ: Вот и ладушки. Мне хочется написать приложение, которое будет отслеживать положение дел в лиге боулинга. Оно должно хранить все игры, ранжировать команды, определять победителей и проигравших во всех еженедельных матчах и точно вести счет в каждой игре.

БК: Супер. Я когда-то неплохо играл в боулинг. Так что повеселимся. Ты уже наверняка накатал несколько пользовательских историй. С какой начнем?

БМ: А давай с подсчета очков в одной игре.

БК: Договорились. И что там будет? Какие у этой истории входные и выходные данные?

БМ: Я думаю, что вход – это просто последовательность бросков. Бросок будет представлен целым числом, равным количеству сбитых кеглей. Выход – счет в каждом фрейме.

БК: Давай ты будешь в этом упражнении заказчиком. Так в каком виде будем представлять входы и выходы?

БМ: Ладно, я заказчик. Нам нужна функция, которая будет добавлять броски, и еще одна для получения счета. Что-то в этом роде:

```
ThrowBall(6);
ThrowBall(3);
Assert.AreEqual(9, GetScore());
```

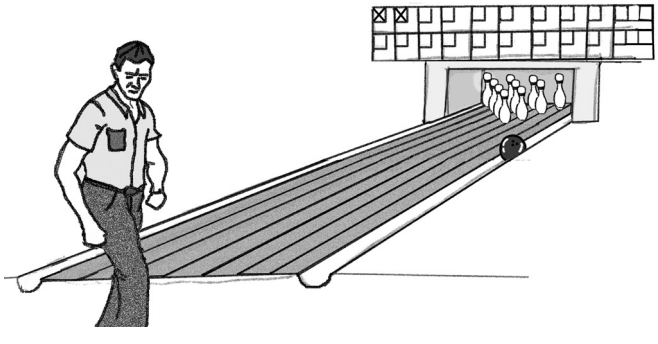
БК: Хорошо, но понадобятся какие-то тестовые данные. Сейчас нарисую карточку участника *(см. рис. 6.1)*.

1	4	4	5	6	5		0	1	7	6		2	6
5		14		29	49	60	61		77	97	117	133	

Рис. 6.1. *Типичная карточка участника в боулинге*

БМ: Да, этого парня не назовешь образцом стабильности.

БК: Или выпил малек. Зато какой отличный приемочный тест получится.



БМ: Нам понадобятся и другие, но с этим погодим. С чего начнем? Будем проектировать систему?

БК: Я бы нарисовал UML-диаграмму с понятиями из предметной области, которые показаны на карточке участника. Это даст нам объекты-кандидаты, а потом мы поглядим на них в коде.

БМ: *(принимая позу великого объектного проектировщика)* Ну так, понятно, что игра состоит из десяти фреймов. В каждом объекте-фрейме может быть один, два или три броска.

БК: Не, ну все великие умы думают одинаково. Дай-ка я это быстренько нарисую (см. рис. 6.2).



Рис. 6.2. *UML-диаграмма карточки участника игры в боулинг*

БК: Ну выбирай класс, какой хочешь. Может, начать с конца цепочки зависимостей, а потом пятиться назад? Так тестировать проще будет.

БМ: Конечно, почему бы и нет. Давай создадим тест для класса `Throw`.

БК: *(начинает печатать)*

```
//ThrowTest.cs--------------------------------
using NUnit.Framework;

[TestFixture]
public class ThrowTest
{
  [Test]
  public void Test???
}
```

БК: А ты имеешь представление, как должен вести себя объект `Throw`?

БМ: Он хранит количество кеглей, сбитых игроком.

БК: Ага, ты только что сказал, правда, другими словами, что он вообще ничего не делает. Может, возьмем для начала какой-нибудь объект с поведением, а не простое хранилище данных?

БМ: Гм. Ты полагаешь, что без класса `Throw` можно и обойтись?

БК: Ну, раз у него нет поведения, то для чего он нужен? Пока не знаю, должен он существовать или нет. Мне кажется, было бы полезнее поработать над объектом, у которого есть что-то помимо методов чтения и записи. Но если ты хочешь постучать... *(пододвигает клавиатуру к БМ)*.

БМ: Ладно, давай сместимся по цепочке зависимостей к `Frame` и посмотрим, нет ли каких-нибудь тестов, которые позволили бы нам покончить с `Throw` *(двигает клавиатуру обратно к БК)*.

БК: *(недоумевая, что задумал БМ: то ли хочет завести меня в тупик, чтобы чему-то научить, то ли вправду согласен со мной)* Поехали, новый файл, новый тест.

```
//FrameTest.cs-------------------------------------
using NUnit.Framework;

[TestFixture]
public class FrameTest
{
```

```
  [Test]
  public void Test???
}
```

БМ: Так, мы это набиваем уже второй раз. Есть у тебя на уме какие-нибудь интересные тесты для `Frame`?

БК: `Frame` должен вести счет, хранить количество кеглей на каждом броске, сообщать, имел ли место страйк или спэа...

БМ: Код покажи.

БК: *(печатает)*

```
//FrameTest.cs-----------------------------
using NUnit.Framework;

[TestFixture]
public class FrameTest
{
  [Test]
  public void TestScoreNoThrows()
  {
    Frame f = new Frame();
    Assert.AreEqual(0, f.Score);
  }
}
//Frame.cs---------------------------------------
public class Frame
{
  public int Score
  {
    get { return 0; }
  }
}
```

БМ: Хорошо, тест проходит. Но свойство `Score` какое-то дурацкое. Оно даст ошибку, если мы добавим бросок в `Frame`. Давай напишем тест, который добавляет несколько бросков, а потом проверяет счет.

```
//FrameTest.cs--------------------------------

[Test]
public void TestAddOneThrow()
{
  Frame f = new Frame();
  f.Add(5);
  Assert.AreEqual(5, f.Score);
}
```

БМ: Это не откомпилируется. В классе `Frame` нет метода `Add`.

БК: Бьюсь об заклад, что если ты добавишь этот метод, то откомпилируется ;-)

БМ: *(печатает)*

```
//Frame.cs--------------------------------------
public class Frame
{
  public int Score
  {
    get { return 0 };
  }

  public void Add(Throw t)
  {
  }
}
```

БМ: *(размышляя вслух)* Это не откомпилируется, потому что мы еще не написали класс `Throw`.

БК: Давай это обсудим, Боб. Тест передает целое число, а метод ожидает объект `Throw`. Нужно выбрать что-то одно. Но перед тем, как снова возвращаться к `Throw`, может, опишешь его поведение?

БМ: Черт! А я и не заметил, как написал `f.Add(5)`. Надо было бы написать `f.Add(new Throw(5))`, но это же уродство какое-то. На самом деле я хочу писать *именно* `f.Add(5)`.

БК: Уродство, не уродство, давай оставим эстетику на потом. Ты можешь описать хоть какое-нибудь поведение объекта `Throw`? Да или нет, двоичный ответ, Боб.

БМ: 10110101101010010l. Я не знаю, есть ли какое-нибудь поведение у `Throw`; начинаю думать, что `Throw` – это просто `int`. Но пока не надо с этим заморачиваться, мы можем написать метод `Frame.Add`, так чтобы он принимал `int`.

БК: Тогда мне кажется, что так и надо сделать, хотя бы потому, что это проще. Когда наткнемся на неприятности, подумаем о чем-нибудь похитрее.

БМ: Договорились.

```
//Frame.cs--------------------------------------
public class Frame
{
  public int Score
  {
    get { return 0};
  }

  public void Add(int pins)
  {
```

```
    }
  }
```

БМ: Вот и хорошо, это компилируется, и тест не проходит. Давай заставим его пройти.

```
//Frame.cs---------------------------------------
public class Frame
{
  private int score;

  public int Score
  {
    get { return score; }
  }

  public void Add(int pins)
  {
    score += pins;
  }
}
```

БМ: Компилируется, и тесты проходят. Но как-то уж больно простенько. Какой у нас следующий тест?

БК: Может, сперва перерывчик?

----------Перерыв----------

БМ: Полегчало. `Frame.Add` – хрупкая функция. Что если передать ей 11?

БК: Ну, она может исключение возбудить в таком случае. Но кто ее вызывает? Если этим приложением будут пользоваться тысячи людей, то надо предусмотреть защиту от таких вещей. А если ты и только ты? Тогда просто не зови ее с аргументом 11 (хихикает).

БМ: Точно подмечено; тесты в остальной части системы будут ловить недопустимые аргументы. Если наткнемся, то сможет потом вставить проверку.

Итак, функция `Add` пока не обрабатывает страйки и спэа. Напишем тест, который будет это выражать.

БК: Гм. Если мы вызовем `Add(10)`, чтобы представить страйк, то что вернет `GetScore()`? Не знаю, как написать утверждение. Быть может, мы не тот вопрос задаем? Или тот вопрос, но не тому объекту?

БМ: Если вызвать `Add(10)` или `Add(3)`, а потом `Add(7)`, то вызывать `Score` от `Frame` бессмысленно. `Frame` должен видеть последующие экземпляры `Frame`, чтобы вычислить свой счет. А если последующих экземпляров `Frame` еще нет, то ему придется возвращать что-то дурацкое, например –1. Я не хочу возвращать –1.

БК: Да, мне эта мысль про –1 тоже по душе. Но ты упомянул о том, что Frame'ы знают о других Frame'ах. А кто хранит эти разные объекты Frame?

БМ: Объект Game.

БК: Стало быть, Game зависит от Frame, а Frame – от Game. Не нравится мне это.

БМ: Frame'ы не обязаны зависеть от Game, их можно поместить в связанный список. Каждый Frame будет хранить указатели на следующий и предыдущий. Чтобы получить счет, Frame должен будет вернуться назад и получить счет от предыдущего Frame, а потом заглянуть вперед и посмотреть, не было ли страйка или спэа.

БК: Ой, что-то я туплю, не могу себе это представить. Покажи-ка мне код.

БМ: Покажу. Но сначала тест.

БК: Для Game или еще один тест для Frame?

БМ: Думаю, понадобится тест для Game, потому что именно Game создает Frame'ы и связывает их в список.

БК: Ты что, хочешь, чтобы мы прекратили работу над Frame и мысленно перескочили на Game? Или просто написали объект MockGame, который будет делать все, что необходимо для работы Frame?

БМ: Нет, давай пока оставим Frame и поработаем с Game. Тесты для Game должны доказать, что нам действительно нужен связанный список Frame'ов.

БК: Не понимаю, как они докажут необходимость списка. Код нужен.

БМ: *(печатает)*

```
//GameTest.cs-----------------------------------------
using NUnit.Framework;

[TestFixture]
public class GameTest
{
  [Test]
  public void TestOneThrow()
  {
    Game game = new Game();
    game.Add(5);
    Assert.AreEqual(5, game.Score);
  }
}
```

БМ: Так пойдет?

БК: Пойдет, но я все еще жду доказательства по поводу списка Frame'ов.

БМ: Я тоже. Пойдем-ка дальше по этим тестам и посмотрим, куда они нас заведут.

```
//Game.cs-----------------------------------
public class Game
{
  public int Score
  {
    get { return 0; }
  }

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

БМ: Отлично, компилируется, и тест не проходит. Сделаем так, чтобы прошел.

```
//Game.cs-----------------------------------
public class Game
{
  private int score;

  public int Score
  {
    get { return score; }
  }

  public void Add(int pins)
  {
    score += pins;
  }
}
```

БМ: Вот так проходит. Хорошо.

БК: Верно, но я все еще не вижу гениального доказательства необходимости в связанном списке объектов `Frame`. А мы только ради него и завели `Game`.

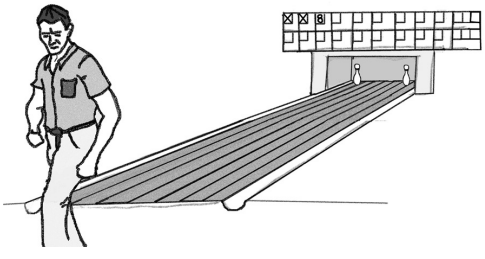
БМ: Ага, вот и я о том же. Но есть подозрение, что, когда мы начнем писать тесты для страйков и спэа, придется создавать `Frame`'ы и связывать их в список. Но пока программа не потребовала, я этого делать не хочу.

БК: Ну и правильно. Будем продвигаться по `Game` мелкими шажками. Как насчет еще одного теста, который проверит два броска без спэа?

БМ: Ладно, он как раз сейчас должен пройти. Пробуем.

```
//GameTest.cs-------------------------------
```

```
[Test]
public void TestTwoThrowsNoMark()
{
  Game game = new Game();
  game.Add(5);
  game.Add(4);
  Assert.AreEqual(9, game.Score);
}
```



БМ: Прошел. Теперь попробуем четыре шара, но без особо успешных бросков.

БК: И этот прошел. Не ожидал, признаться. Получается, что можно добавлять броски, а `Frame` так и не понадобился. Но мы еще не рассмотрели страйки и спэа. Может быть, там его придется ввести.

БМ: На это я и рассчитываю. Но давай-ка рассмотрим такой тест:

```
//TestGame.cs----------------------------------------
[Test]
public void TestFourThrowsNoMark()
{
  Game game = new Game();
  game.Add(5);
  game.Add(4);
  game.Add(7);
  game.Add(2);
  Assert.AreEqual(18, game.Score);
  Assert.AreEqual(9, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
}
```

БМ: Разумно выглядит?

БК: Вполне. Я и забыл, что мы должны уметь показывать счет в каждом фрейме. А, черт, я ведь на карточку участника поставил стакан с колой. Потому и забыл.

БМ: *(вздыхая)* Ну ладно, для начала добавим метод `ScoreForFrame` в `Game`, так чтобы этот тест не проходил.

```
//Game.cs---------------------------------
public int ScoreForFrame(int frame)
```

```
  {
    return 0;
  }
```

БМ: Отлично, компилируется и не проходит. Ну а как сделать, чтобы он прошел?

БК: Можно начать создавать объекты `Frame`. Но нет ли какого-нибудь способа попроще?

БМ: А почему бы просто не создать в `Game` массив целых чисел? Тогда при каждом вызове `Add` в этот массив можно было бы добавлять новое число. А при вызове `ScoreForFrame` мы пробежались бы по массиву и вычислили счет.

```
//Game.cs---------------------------------
public class Game
{
  private int score;
  private int[] throws = new int[21];
  private int currentThrow;

  public int Score
  {
    get { return score; }
  }

  public void Add(int pins)
  {
    throws[currentThrow++] = pins;
    score += pins;
  }

  public int ScoreForFrame(int frame)
  {
    int score = 0;
    for(int ball = 0;
      frame > 0 && ball < currentThrow;
      ball+=2, frame--)
    {
      score += throws[ball] + throws[ball + 1];
    }
    return score;
  }
}
```

БМ: *(очень довольный собой)* Вот так работает.

БК: А откуда это магическое число 21?

БМ: А это максимально возможное число бросков в игре.

БК: Ага, я догадался. Ты в молодости хакерил в UNIX'е и был ужасно горд, когда удавалось запихать всю программу в одно предло-

жение, в котором никто не мог разобраться. `ScoreForFrame()` надо рефакторить, чтобы сделать более понятным. Но сначала один вопрос. Разве `Game` – самое подходящее место для этого метода? На мой взгляд, `Game` нарушает принцип единственной обязанности (см. главу 8). Он принимает количество бросков *и* знает, как вычислять счет в каждом фрейме. Может, заведем объект `Scorer`?

БМ: *(строго грозит пальцем)* Пока что я не знаю, где какие функции обретаются, сейчас я хочу, чтобы счет заработал. Вот справимся с этим и *тогда* подебатируем насчет SRP[1]. Но по поводу UNIX я тебя понимаю, давай-ка упростим этот цикл.

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

  return score;
}
```

БМ: Так лучше, но у выражения `score+=` есть побочные эффекты. Здесь это все равно, потому что неважно, в каком порядке вычисляются слагаемые. (Или важно? Может ли быть так, что оба инкремента вычисляются перед операциями доступа к массиву?)

БК: Можно было бы провести эксперимент и убедиться, что побочных эффектов нет, но от этого функция не станет обрабатывать страйки и спэа. Будем и дальше улучшать читабельность или поработаем над функциональностью?

БМ: Такой эксперимент мог бы иметь смысл только для некоторых компиляторов. А другие могут вычислять в ином порядке. Не думаю, что это так уж важно, но давай избавимся от потенциальной зависимости от порядка вычислений, а потом займемся другими тестами.

```
public int ScoreForFrame(int theFrame)
{
  int ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
```

[1] Single Responsibility Principle – принцип единственной обязанности. – *Прим. перев.*

```
    currentFrame++)
  {
    int firstThrow = throws[ball++];
    int secondThrow = throws[ball++];
    score += firstThrow + secondThrow;
  }

  return score;
}
```

БМ: Отлично, следующий тест. Попробуем-ка спэа.

```
[Test]
public void TestSimpleSpare()
{
  Game game = new Game();
}
```

БМ: Который раз это пишу, устал уже. Давай поместим создание игры в функцию SetUp.

```
//GameTest.cs--------------------------------
using NUnit.Framework;

[TestFixture]
public class GameTest
{
  private Game game;

  [SetUp]
  public void SetUp()
  {
    game = new Game();
  }

  [Test]
  public void TestOneThrow()
  {
    game.Add(5);
    Assert.AreEqual(5, game.Score);
  }

  [Test]
  public void TestTwoThrowsNoMark()
  {
    game.Add(5);
    game.Add(4);
    Assert.AreEqual(9, game.Score);
  }

  [Test]
  public void TestFourThrowsNoMark()
```

```
  {
    game.Add(5);
    game.Add(4);
    game.Add(7);
    game.Add(2);
    Assert.AreEqual(18, game.Score);
    Assert.AreEqual(9, game.ScoreForFrame(1));
    Assert.AreEqual(18, game.ScoreForFrame(2));
  }

  [Test]
  public void TestSimpleSpare()
  {
  }
}
```

БМ: Вот так-то лучше, а теперь напишем тест для спэа.

```
[Test]
public void TestSimpleSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  Assert.AreEqual(13, game.ScoreForFrame(1));
}
```

БМ: Замечательно, тест не проходит. Исправим это дело.

БК: Дай я постучу.

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

    int frameScore = firstThrow + secondThrow;

    // для обработки спэа необходим первый бросок в следующем
    // фрейме
    if ( frameScore == 10 )
      score += frameScore + throws[ball++];
    else
      score += frameScore;
  }
```

```
    return score;
  }
```

БК: Отлично! Работает!

БМ: *(хватая клавиатуру)* Так-то оно так, но мне кажется, что инкремента `ball` в ветке `frameScore==10` быть не должно. И вот тест, который это доказывает.

```
[Test]
public void TestSimpleFrameAfterSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  game.Add(2);
  Assert.AreEqual(13, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.Score);
}
```

БМ: Ха! Гляди, не проходит. А теперь уберем этот гадкий лишний инкремент...

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

БМ: Хм, все равно не проходит. Может, метод `Score` неправильный? Сейчас проверю, только изменю тест, поставив `ScoreForFrame(2)`.

```
[Test]
public void TestSimpleFrameAfterSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  game.Add(2);
  Assert.AreEqual(13, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
}
```

БМ: Та-а-а-к, а этот проходит. Видно, в свойстве `Score` что-то напутали. Поглядим на него.

```
public int Score
{
  get { return score; }
}

public void Add(int pins)
{
  throws[currentThrow++] = pins;
  score += pins;
}
```

БМ: Ну конечно, в этом все и дело. Свойство `Score` просто возвращает сумму сбитых кеглей, а не настоящий счет. А нам нужно, чтобы `Score` вызывало `ScoreForFrame()` для текущего фрейма.

БК: Но мы не знаем, что такое текущий фрейм. Давай добавим это свойство во все наши тесты, по одному, разумеется.

БМ: Правильно.

```
//GameTest.cs--------------------------------

[Test]
public void TestOneThrow()
{
  game.Add(5);
  Assert.AreEqual(5, game.Score);
  Assert.AreEqual(1, game.CurrentFrame);
}

//Game.cs-----------------------------------

public int CurrentFrame
{
  get { return 1; }
}
```

БМ: Славненько, работает. Но в таком виде неинтересно. Возьмем следующий тест.

```
[Test]
public void TestTwoThrowsNoMark()
{
  game.Add(5);
  game.Add(4);
  Assert.AreEqual(9, game.Score);
  Assert.AreEqual(1, game.CurrentFrame);
}
```

БМ: Тоже неинтересно, следующий.

```
[Test]
public void TestFourThrowsNoMark()
{
  game.Add(5);
  game.Add(4);
  game.Add(7);
  game.Add(2);
  Assert.AreEqual(18, game.Score);
  Assert.AreEqual(9, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
  Assert.AreEqual(2, game.CurrentFrame);
}
```

БМ: Этот не проходит. Давай добьемся, чтобы прошел.

БК: Думаю, что алгоритм тривиален. Просто делим количество бросков на 2, потому что во фрейме разрешается два броска. Если только это не страйк. Но страйков у нас пока нет, поэтому про них можно забыть.

БМ: *(пробует добавлять и вычитать 1, пока наконец не заработало)*[1]

```
public int CurrentFrame
{
  get { return 1 + (currentThrow - 1) / 2; }
}
```

БМ: Что-то не больно мне это нравится.

БК: А что, если не производить вычисления каждый раз? Можно же корректировать член `currentFrame` после каждого броска.

БМ: Давай попробуем.

```
//Game.cs----------------------------------
private int currentFrame;
private bool isFirstThrow = true;

public int CurrentFrame
{
  get { return currentFrame; }
}

public void Add(int pins)
{
  throws[currentThrow++] = pins;
  score += pins;

  if (isFirstThrow)
  {
    isFirstThrow = false;
    currentFrame++;
  }
  else
  {
    isFirstThrow=true;
  }
}
```

БМ: Работает. Но отсюда следует, что текущий фрейм – это тот, в котором был брошен последний шар, а не тот, в котором будет брошен следующий. Если об этом помнить, то все будет хорошо.

[1] Дэйв Томас и Энди Хант называют это *программированием наугад* (Programming by coincidence).

БК: Ну у меня не такая хорошая память, поэтому надо бы сделать понятнее. Но прежде чем этим заняться, давай перенесем часть кода из `Add()` в закрытый метод, который назовем `AdjustCurrentFrame()` или как-то в этом роде.

БМ: Отличная мысль.

```
public void Add(int pins)
{
  throws[currentThrow++] = pins;
  score += pins;
  AdjustCurrentFrame();
}

private void AdjustCurrentFrame()
{
  if (isFirstThrow)
  {
    isFirstThrow = false;
    currentFrame++;
  }
  else
  {
    isFirstThrow=true;;
  }
}
```

БМ: А теперь переименуем переменную и метод, чтобы было яснее. Подойдет `currentFrame`?

БК: Да мне это имя нравится. Просто думаю, что мы инкремент не туда поставили. Для меня текущим является фрейм, в котором я бросаю шары. Поэтому увеличивать его номер надо сразу после последнего броска во фрейме.

БМ: Согласен. Давай поменяем тесты, чтобы отразить это решение, а потом я поправлю `AdjustCurrentFrame`.

```
//GameTest.cs--------------------------------
[Test]
public void TestTwoThrowsNoMark()
{
  game.Add(5);
  game.Add(4);
  Assert.AreEqual(9, game.Score);
  Assert.AreEqual(2, game.CurrentFrame);
}

[Test]
public void TestFourThrowsNoMark()
{
  game.Add(5);
  game.Add(4);
```

```
  game.Add(7);
  game.Add(2);
  Assert.AreEqual(18, game.Score);
  Assert.AreEqual(9, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
  Assert.AreEqual(3, game.CurrentFrame);
}

//Game.cs---------------------------------

private int currentFrame = 1;

private void AdjustCurrentFrame()
{
  if (isFirstThrow)
  {
    isFirstThrow = false;
  }
  else
  {
    isFirstThrow=true;
    currentFrame++;
  }
}
```

БМ: Работает. Теперь проверим `CurrentFrame` на двух тестах со спэа.

```
[Test]
public void TestSimpleSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  Assert.AreEqual(13, game.ScoreForFrame(1));
  Assert.AreEqual(2, game.CurrentFrame);
}

[Test]
public void TestSimpleFrameAfterSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  game.Add(2);
  Assert.AreEqual(13, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
  Assert.AreEqual(3, game.CurrentFrame);
}
```

БМ: И это работает. Возвращаемся к исходной задаче. Нам нужно работающее свойство `Score`. Теперь можно вызывать из `Score` метод `ScoreForFrame(CurrentFrame-1)`.

```
[Test]
public void TestSimpleFrameAfterSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  game.Add(2);
  Assert.AreEqual(13, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
  Assert.AreEqual(18, game.Score);
  Assert.AreEqual(3, game.CurrentFrame);
}

//Game.cs---------------------------------

public int Score()
{
  return ScoreForFrame(CurrentFrame - 1);
}
```

БМ: Теперь не проходит тест `TestOneThrow`. Поглядим на него.

```
[Test]
public void TestOneThrow()
{
  game.Add(5);
  Assert.AreEqual(5, game.Score);
  Assert.AreEqual(1, game.CurrentFrame);
}
```

БМ: После одного броска фрейм еще не закончен. Из `Score` вызывается `ScoreForFrame(0)`. Это ляп.

БК: Может, так, а может, и нет. Для кого мы пишем программу и кто будет вызывать `Score`? Разумно ли предположить, что это свойство не будет вызываться для незаконченного фрейма?

БМ: Ну да. Но меня это беспокоит. Чтобы обойти проблему, мы должны убрать из теста `TestOneThrow` обращение к `Score`. А мы этого хотим?

БК: Можно и так. Можно даже вообще исключить `TestOneThrow`. Мы и ввели-то его только как трамплин к более интересным тестовым случаям. А сейчас он еще имеет какой-то смысл? У нас и так есть полное покрытие остальными тестами.

БМ: Понимаю тебя. Ладно, вычеркиваем *(редактирует код, прогоняет тесты, получает зеленую полосу)*. Ну вот, так-то лучше.



А теперь пора заняться тестом страйков. Хотим мы наконец связать все эти объекты `Frame` в список или не хотим? *(фыркает)*

```
[Test]
public void TestSimpleStrike()
{
  game.Add(10);
  game.Add(3);
  game.Add(6);
  Assert.AreEqual(19, game.ScoreForFrame(1));
  Assert.AreEqual(28, game.Score);
  Assert.AreEqual(3, game.CurrentFrame);
}
```

БМ: **Компилируется и не проходит – что и следовало ожидать. Сейчас мы его заставим работать.**

```
//Game.cs-----------------------------------
public class Game
{
  private int score;
  private int[] throws = new int[21];
  private int currentThrow;
  private int currentFrame = 1;
  private bool isFirstThrow = true;

  public int Score
  {
    get { return ScoreForFrame(GetCurrentFrame() - 1); }
  }

  public int CurrentFrame
  {
    get { return currentFrame; }
  }

  public void Add(int pins)
  {
    throws[currentThrow++] = pins;
    score += pins;
```

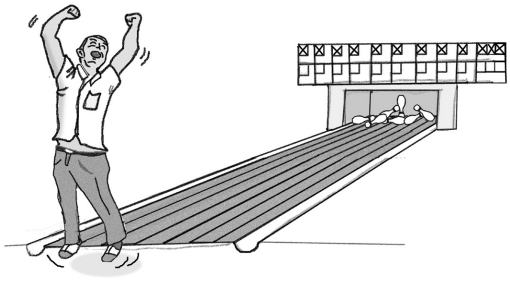
```
    AdjustCurrentFrame(pins);
  }

  private void AdjustCurrentFrame(int pins)
  {
    if (isFirstThrow)
    {
      if(pins == 10) // страйк
        currentFrame++;
      else
        isFirstThrow = false;
    }
    else
    {
      isFirstThrow=true;
      currentFrame++;
    }
  }

  public int ScoreForFrame(int theFrame)
  {
    int ball = 0;
    int score=0;
    for (int currentFrame = 0;
      currentFrame < theFrame;
      currentFrame++)
    {
      int firstThrow = throws[ball++];
      if(firstThrow == 10) // страйк
      {
        score += 10 + throws[ball] + throws[ball+1];
      }
      else
      {
        int secondThrow = throws[ball++];
        int frameScore = firstThrow + secondThrow;
        // для обработки спэа необходим первый бросок в следующем
        // фрейме
        if ( frameScore == 10 )
          score += frameScore + throws[ball];
        else
          score += frameScore;
      }
    }

    return score;
  }
}
```

БМ: Не так уж это было и трудно. Посмотрим, как обсчитается идеальная игра.



```
[Test]
public void TestPerfectGame()
{
  for (int i=0; i<12; i++)
  {
    game.Add(10);
  }
  Assert.AreEqual(300, game.Score);
  Assert.AreEqual(10, game.CurrentFrame);
}
```

БМ: Так-с, говорит, что счет равен 330. С чего бы это?

БК: Потому что номер текущего фрейма увеличивался аж до 12.

БМ: Ну конечно! Надо ограничить его десятью.

```
private void AdjustCurrentFrame(int pins)
{
  if (isFirstThrow)
  {
    if(pins == 10) // страйк
      currentFrame++;
    else
      isFirstThrow = false;
  }
  else
  {
    isFirstThrow=true;
    currentFrame++;
  }
  if(currentFrame > 10)
    currentFrame = 10;
}
```

БМ: Черт, а теперь говорит, что счет равен 270. Что опять не так?

БК: Боб, свойство `Score` вычитает 1 из `SetCurrentFrame` и поэтому выдает счет для фрейма 9, а не 10.

БМ: Что? Ты хочешь сказать, что я должен ограничить номер текущего фрейма числом 11, а не 10? Попробую.

```
if(currentFrame > 11)
  currentFrame = 11;
```

БМ: Ну вот, теперь счет правильный, но тест все равно не проходит, потому текущий фрейм равен 11, а не 10. Зараза! От этой затеи с текущим фреймом одни неприятности. Мы хотим, чтобы текущим был фрейм, в котором игрок бросает шары, но что под ним понимать в конце игры?

БК: Может быть, вернуться к старой идее и считать, что текущим является фрейм, в котором был брошен последний шар?

БМ: А может, нам стоит ввести понятие последнего *завершенного* фрейма? В конце концов, счет игры в любой момент – это счет в последнем завершенном фрейме.

БК: Завершенным надо считать фрейм, для которого можно вычислить счет, так?

БМ: Ну да, фрейм, в котором был спэа, завершается после следующего шара. Фрейм, в котором был страйк, завершается после двух следующих шаров. Обычный фрейм завершается после второго шара в нем же.

Погоди-ка. Мы же пытаемся заставить работать свойство `Score`, верно? Тогда нам просто нужно вызывать из `Score` метод `ScoreForFrame(10)`, если игра закончилась.

БК: Но откуда мы знаем, что игра закончилась?

БМ: Как только `AdjustCurrentFrame` попытается сделать `currentFrame` больше 10, игра закончилась.

БК: Стоп. Ты говоришь, что если `CurrentFrame` возвращает 11, то игра закончилась. Но именно так программа сейчас и работает!

БМ: Гм. Ты хочешь сказать, что нам нужно изменить тест, приведя его в соответствие с кодом?

```
[Test]
public void TestPerfectGame()
{
  for (int i=0; i<12; i++)
  {
    game.Add(10);
  }
  Assert.AreEqual(300, game.Score);
  Assert.AreEqual(11, game.CurrentFrame);
}
```

БМ: Теперь работает. Но по ощущению что-то здесь не так.

БК: Может, попозже осенит. А прямо сейчас я вижу ошибку. Ты позволишь? *(придвигает к себе клавиатуру)*

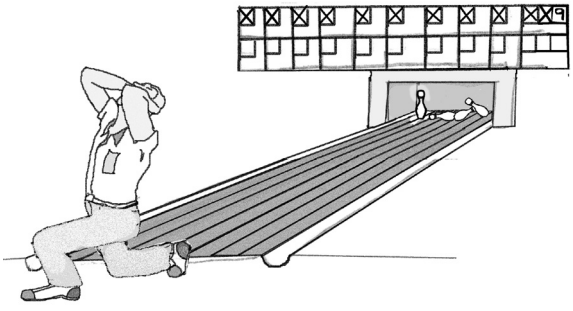
```
[Test]
public void TestEndOfArray()
{
  for (int i=0; i<9; i++)
  {
    game.Add(0);
    game.Add(0);
  }
  game.Add(2);
  game.Add(8); // спэа в 10-м фрейме
  game.Add(10); // страйк в последней позиции массива
  Assert.AreEqual(20, game.Score);
}
```

БК: Гм, не падает. Я думал, что раз в двадцать первой позиции массива страйк, то обсчитыватель попытается прибавить к счету 22-ю и 23-ю позиции. Но, похоже, это не так.

БМ: Все никак не расстанешься с обсчитывателем, да? Я понимаю, куда ты клонишь, но, поскольку `Score` никогда не вызывает `ScoreForFrame` с аргументом, большим 10, то последний страйк не считается страйком. За него просто начисляется 10 очков, чтобы завершить последний спэа. Мы никогда не выходим за границу массива.

БК: Ладно, давай-ка подсунем программе нашу исходную карточку.

```
[Test]
public void TestSampleGame()
{
  game.Add(1);
  game.Add(4);
  game.Add(4);
  game.Add(5);
  game.Add(6);
  game.Add(4);
  game.Add(5);
  game.Add(5);
  game.Add(10);
  game.Add(0);
  game.Add(1);
  game.Add(7);
  game.Add(3);
  game.Add(6);
  game.Add(4);
  game.Add(10);
  game.Add(2);
  game.Add(8);
  game.Add(6);
  Assert.AreEqual(133, game.Score);
}
```



БК: Работает. Еще какие-нибудь тесты можешь придумать?

БМ: А как же. Проверим еще кое-какие граничные случаи. Что если бедный парень выбил подряд 11 страйков, а в последнем фрейме сбил только 9 кеглей?

```
[Test]
public void TestHeartBreak()
{
  for (int i=0; i<11; i++)
    game.Add(10);
  game.Add(9);
  Assert.AreEqual(299, game.Score);
}
```

БК: Тоже работает. А как насчет спэа в десятом фрейме?

```
[Test]
public void TestTenthFrameSpare()
{
  for (int i=0; i<9; i++)
    game.Add(10);
  game.Add(9);
  game.Add(1);
  game.Add(1);
  Assert.AreEqual(270, game.Score);
}
```

БМ: *(со счастливой улыбкой глядя на зеленую полосу)* И это работает. Больше ничего не приходит в голову. А тебе?

БК: Тоже нет. Полагаю, что мы рассмотрели все случаи. А кроме того, у меня руки чешутся заняться рефакторингом и навести тут порядок. Я по-прежнему хочу куда-нибудь всунуть объект-обсчитыватель.

БМ: Согласен, метод `ScoreForFrame` какой-то неприглядный получился. Давай с него и начнем.

```
public int ScoreForFrame(int theFrame)
{
```

```
    int ball = 0;
    int score=0;
    for (int currentFrame = 0;
      currentFrame < theFrame;
      currentFrame++)
    {
      int firstThrow = throws[ball++];
      if(firstThrow == 10) // страйк
      {
        score += 10 + throws[ball] + throws[ball+1];
      }
      else
      {
        int secondThrow = throws[ball++];
        int frameScore = firstThrow + secondThrow;
        // для обработки спэа необходим первый бросок в следующем
        // фрейме
        if ( frameScore == 10 )
          score += frameScore + throws[ball];
        else
          score += frameScore;
      }
    }

    return score;
  }
```

БМ: Я бы вынес содержимое ветки `else` в отдельный метод `HandleSecondThrow`, но как это сделать, если тут используются локальные переменные `firstThrow` и `secondThrow`?

БК: Ну сделаем их переменными-членами.

БМ: Ага, и заодно это поможет реализовать твою идею о выделении отдельного объекта-обсчитывателя. Ладно, давай попробуем.

БК: *(завладевает клавиатурой)*

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

  public int ScoreForFrame(int theFrame)
  {
    ball = 0;
    int score=0;
    for (int currentFrame = 0;
      currentFrame < theFrame;
      currentFrame++)
    {
      firstThrow = throws[ball++];
      if(firstThrow == 10) // страйк
      {
```

```
      score += 10 + throws[ball] + throws[ball+1];
    }
    else
    {
      secondThrow = throws[ball++];
      int frameScore = firstThrow + secondThrow;
      // для обработки спэа необходим первый бросок в следующем
      // фрейме
      if ( frameScore == 10 )
        score += frameScore + throws[ball];
      else
        score += frameScore;
    }
  }

  return score;
}
```

БК: Работает. Теперь можно вынести ветку `else` в отдельный метод.

```
public int ScoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    firstThrow = throws[ball++];
    if(firstThrow == 10) // страйк
    {
      score += 10 + throws[ball] + throws[ball+1];
    }
    else
    {
      score += HandleSecondThrow();
    }
  }

  return score;
}

private int HandleSecondThrow()
{
  int score = 0;
  secondThrow = throws[ball++];
  int frameScore = firstThrow + secondThrow;
  // для обработки спэа необходим первый бросок в следующем фрейме
  if ( frameScore == 10 )
    score += frameScore + throws[ball];
  else
    score += frameScore;
```

```
    return score;
  }
```

БМ: Нет, ну ты посмотри на структуру `ScoreForFrame`! В псевдокоде она выглядит примерно так:

```
if strike
  score += 10 + NextTwoBalls;
else
  HandleSecondThrow.
```

БМ: А если написать так?

```
if strike
  score += 10 + NextTwoBalls;
else if spare
  score += 10 + NextBall;
else
  score += TwoBallsInFrame
```

БК: Ну супер! Это же правила подсчета очков в боулинге! А ну-ка посмотрим, сможем ли мы получить такую структуру в настоящем методе. Для начала изменим способ увеличения переменной `ball`, чтобы во всех трех случаях ею можно было манипулировать независимо.

```
public int ScoreForFrame(int theFrame)
{
  ball = 0;
  int score=0;
  for (int currentFrame = 0;
    currentFrame < theFrame;
    currentFrame++)
  {
    firstThrow = throws[ball];
    if(firstThrow == 10) // страйк
    {
      ball++;
      score += 10 + throws[ball] + throws[ball+1];
    }
    else
    {
      score += HandleSecondThrow();
    }
  }

  return score;
}

private int HandleSecondThrow()
{
  int score = 0;
```

```
    secondThrow = throws[ball + 1];
    int frameScore = firstThrow + secondThrow;
    // для обработки спэа необходим первый бросок в следующем фрейме
    if ( frameScore == 10 )
    {
      ball += 2;
      score += frameScore + throws[ball];
    }
    else
    {
      ball += 2;
      score += frameScore;
    }
    return score;
  }
```

БМ: *(хватает клавиатуру)* Так, а теперь избавимся от переменных `firstThrow` и `secondThrow` и заменим их подходящими методами.

```
  public int ScoreForFrame(int theFrame)
  {
    ball = 0;
    int score=0;
    for (int currentFrame = 0;
      currentFrame < theFrame;
      currentFrame++)
    {
      firstThrow = throws[ball];
      if(Strike())
      {
        ball++;
        score += 10 + NextTwoBalls;
      }
      else
      {
        score += HandleSecondThrow();
      }
    }

    return score;
  }

  private bool Strike()
  {
    return throws[ball] == 10;
  }

  private int NextTwoBalls
  {
    get { return (throws[ball] + throws[ball+1]); }
  }
```

БМ: **Этот шаг работает, идем дальше.**

```
private int HandleSecondThrow()
{
  int score = 0;
  secondThrow = throws[ball + 1];
  int frameScore = firstThrow + secondThrow;
  // для обработки спэа необходим первый бросок в следующем фрейме
  if (Spare())
  {
    ball += 2;
    score += 10 + NextBall;
  }
  else
  {
    ball += 2;
    score += frameScore;
  }
  return score;
}

private bool Spare()
{
  return throws[ball] + throws[ball+1] == 10;
}

private int NextBall
{
  get { return throws[ball]; }
}
```

БМ: **И это работает. Теперь займемся** `frameScore`**.**

```
private int HandleSecondThrow()
{
  int score = 0;
  secondThrow = throws[ball + 1];
  int frameScore = firstThrow + secondThrow;
  // для обработки спэа необходим первый бросок в следующем фрейме
  if ( IsSpare() )
  {
    ball += 2;
    score += 10 + NextBall;
  }
  else
  {
    score += TwoBallsInFrame;
    ball += 2;
  }
  return score;
}
```

```
private int TwoBallsInFrame
{
  get { return throws[ball] + throws[ball+1]; }
}
```

БК: Боб, ты инкремент `ball` делаешь непоследовательно. В случае спэа и страйка ты увеличиваешь перед вычислением `score`, а в `TwoBallsInFrame` – *после*. И код зависит от этого порядка! В этом есть какой-то смысл?

БМ: Извини, я должен был объяснить. Я планирую перенести инкременты в свойства `Strike`, `Spare` и `TwoBallsInFrame`. Тогда в методе `ScoreForFrame` их не останется, и этот метод будет выглядеть точно, как наш псевдокод.

БК: Ладно, поверю тебе, но помни – я слежу.

БМ: Ну, раз теперь никто не пользуется переменными `firstThrow`, `secondThrow` и `frameScore`, их можно убрать.

```
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;
  // для обработки спэа необходим первый бросок в следующем фрейме
  if ( Spare() )
  {
    ball += 2;
    score += 10 + NextBall;
  }
  else
```

```
    {
      score += TwoBallsInFrame;
      ball += 2;
    }
    return score;
  }
```

БМ: *(в глазах отражается зеленая полоса)* Ну а теперь, поскольку единственная переменная, связывающая эти три случая, — `ball` и поскольку в каждом случае она обрабатывается независимо, то все их можно объединить.

```
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: Прекрасно, теперь можно инкрементировать согласованно и назвать методы более явно *(тянется за клавиатурой)*.

```
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
  {
    get { return (throws[ball+1] + throws[ball+2]); }
  }

  private int NextBallForSpare
  {
    get { return throws[ball+2]; }
  }
```

БМ: Ну ты погляди на метод `ScoreForFrame`! Прямо-таки правила боулинга, сформулированные максимально кратко.

БК: Но, Боб, а что случилось со связанным списком `Frame`'ов? *(фыркает)*

БМ: *(вздыхает)* Нас сбили с толку демоны чрезмерного увлечения диаграммами. Боже мой, всего три квадратика на салфетке – `Game`, `Frame` и `Throw`, – а как все оказалось усложнено и просто неверно.

БК: Мы ошиблись, начав с класса `Throw`. Надо было начинать с `Game`!

БМ: Точно! В следующий раз будем идти вниз, начиная с самого верхнего уровня.

БК: *(выпучив глаза)* Проектирование сверху вниз?!!

БМ: Поправка: проектирование сверху вниз *с предварительными тестами*. Честно говоря, не знаю, можно ли это считать универсальным правилом. Просто в данном случае оно помогло бы. Поэтому в следующий раз я собираюсь ему последовать и посмотреть, что получится.

БК: Ну тогда ладно. Но в любом случае у нас еще осталось место для рефакторинга. Переменная `ball` – это просто закрытый итератор

для ScoreForFrame и иже с ним. Все они должны быть перенесены в другой объект.

БМ: Ну конечно, в твой любимый Scorer. Но вообще-то ты прав. Давай сделаем.

БК: *(тянет к себе клавиатуру и мелкими шагами вносит изменения, прерывая их прогоном тестов)*

```
//Game.cs---------------------------------
public class Game
{
  private int score;
  private int currentFrame = 1;
  private bool isFirstThrow = true;
  private Scorer scorer = new Scorer();

  public int Score
  {
    get { return ScoreForFrame(GetCurrentFrame() - 1); }
  }

  public int CurrentFrame
  {
    get { return currentFrame; }
  }

  public void Add(int pins)
  {
    scorer.AddThrow(pins);
    score += pins;
    AdjustCurrentFrame(pins);
  }

  private void AdjustCurrentFrame(int pins)
  {
    if (isFirstThrow)
    {
      if(pins == 10) // страйк
        currentFrame++;
      else
        isFirstThrow = false;
    }
    else
    {
      isFirstThrow = true;
      currentFrame++;
    }
    if(currentFrame > 11)
      currentFrame = 11;
  }
```

```
  public int ScoreForFrame(int theFrame)
  {
    return scorer.ScoreForFrame(theFrame);
  }
}

//Scorer.cs---------------------------------
public class Scorer
{
  private int ball;
  private int[] throws = new int[21];
  private int currentThrow;

  public void AddThrow(int pins)
  {
    throws[currentThrow++] = 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 int NextTwoBallsForStrike
  {
    get { return (throws[ball+1] + throws[ball+2]); }
  }
```

```
    private int NextBallForSpare
    {
      get { return throws[ball+2]; }
    }

    private bool Strike()
    {
      return throws[ball] == 10;
    }

    private int TwoBallsInFrame
    {
      get { return throws[ball] + throws[ball+1]; }
    }

    private bool Spare()
    {
      return throws[ball] + throws[ball+1] == 10;
    }
  }
```

БК: Вот так гораздо лучше. Теперь `Game` только отслеживает фреймы, а `Scorer` только вычисляет счет. Принцип единственной обязанности во всей красе!

БМ: Как бы это ни называлось, но так действительно лучше. Ты заметил, что переменная `score` больше нигде не используется?

БК: Ха! Ты прав. Давай-ка ее изничтожим. *(радостно начинает стирать)*

```
public void Add(int pins)
{
  scorer.AddThrow(pins);
  AdjustCurrentFrame(pins);
}
```

БК: Неплохо. А теперь не почистить ли нам `AdjustCurrentFrame`?

БМ: Ну давай посмотрим.

```
private void AdjustCurrentFrame(int pins)
{
  if (isFirstThrow)
  {
    if(pins == 10) // страйк
      currentFrame++;
    else
      isFirstThrow = false;
  }
  else
  {
```

```
      isFirstThrow = true;
      currentFrame++;
    }
    if(currentFrame > 11)
      currentFrame = 11;
  }
```

БМ: Так, для начала вынесем инкременты в отдельный метод, который заодно проследит, чтобы номер фрейма не превышал 11. (Бррр, ну не нравится мне это 11.)

БК: Боб, 11 означает конец игры.

БМ: Да. Бррр. *(берет клавиатуру и вносит парочку изменений, перемежая их прогоном тестов)*

```
private void AdjustCurrentFrame(int pins)
{
  if (isFirstThrow)
  {
    if(pins == 10) // страйк
      AdvanceFrame();
    else
      isFirstThrow = false;
  }
  else
  {
    isFirstThrow = true;
    AdvanceFrame();
  }
}

private void AdvanceFrame()
{
  currentFrame++;
  if(currentFrame > 11)
    currentFrame = 11;
}
```

БМ: Уже получше. А теперь вынесем случай страйка в отдельный метод. *(вносит несколько изменений, прогоняя каждый раз тесты)*

```
private void AdjustCurrentFrame(int pins)
{
  if (isFirstThrow)
  {
    if(AdjustFrameForStrike(pins) == false)
      isFirstThrow = false;
  }
  else
  {
    isFirstThrow = true;
```

```
      AdvanceFrame();
    }
  }

  private bool AdjustFrameForStrike(int pins)
  {
    if(pins == 10)
    {
      AdvanceFrame();
      return true;
    }
    return false;
  }
```

БМ: Почти хорошо. Только вот это число 11...

БК: Эк ты его ненавидишь-то.

БМ: Ага, ты глянь на свойство `Score`.

```
public int Score
{
  get { return ScoreForFrame(GetCurrentFrame() - 1); }
}
```

БМ: Эта `-1` выглядит странно. Это единственное место, где мы по-настоящему используем `CurrentFrame`, и при этом все-таки должны корректировать возвращенное значение.

БК: А ведь ты прав. Сколько же раз мы уже к этому возвращались?

БМ: Слишком много. А что делать? Программа хочет, чтобы `currentFrame` представляла фрейм, в котором был брошен последний шар, а не тот фрейм, в котором мы только собираемся бросать.

БК: Ох, но ведь так мы порушим почти все наши тесты.

БМ: Вообще-то я думаю, что нужно убрать `CurrentFrame` из всех тестов, да и само свойство `CurrentFrame` тоже убрать. Никто его реально не использует.

БК: Понял тебя. Я сделаю это и положу конец страданиям хромой лошадки. *(берет клавиатуру)*

```
//Game.cs-----------------------------------
public int Score
{
  get { return ScoreForFrame(currentFrame); }
}

private void AdvanceFrame()
{
  currentFrame++;
  if(currentFrame > 10)
    currentFrame = 10;
}
```

БМ: Ну просто плакать хочется. Ты хочешь сказать, что мы все время мучились из-за *этого*? Всего-то и надо было изменить порог с 11 на 10 и убрать -1. Ну и дела!

БК: Да, дядя Боб. Не стоило оно таких мучений.

БМ: Не нравится мне побочный эффект в AdjustFrameForStrike(). Хочу от него избавиться. Как тебе такой вариант?

```
private void AdjustCurrentFrame(int pins)
{
  if ((isFirstThrow && pins == 10) || (!isFirstThrow))
    AdvanceFrame();
  else
    isFirstThrow = false;
}
```

БК: Идея хорошая, и тесты проходят, но вот этот длинный if мне не по душе. А если так?

```
private void AdjustCurrentFrame(int pins)
{
  if (Strike(pins) || (!isFirstThrow))
    AdvanceFrame();
  else
    isFirstThrow = false;
  }

private bool Strike(int pins)
{
  return (isFirstThrow && pins == 10);
}
```

БМ: Да, так лучше. Но можно пойти еще дальше:

```
private void AdjustCurrentFrame(int pins)
{
  if (LastBallInFrame(pins))
    AdvanceFrame();
  else
    isFirstThrow = false;
}

private bool LastBallInFrame(int pins)
{
  return Strike(pins) || (!isFirstThrow);
}
```

БК: Совсем хорошо!

БМ: Ну что ж, похоже, мы закончили. Давай еще раз прочитаем всю программу и посмотрим, так ли она проста и понятна, как должна быть.

```
//Game.cs---------------------------------
public class Game
{
  private int currentFrame = 0;
  private bool isFirstThrow = true;
  private Scorer scorer = new Scorer();

  public int Score
  {
    get { return ScoreForFrame(currentFrame); }
  }

  public void Add(int pins)
  {
    scorer.AddThrow(pins);
    AdjustCurrentFrame(pins);
  }

  private void AdjustCurrentFrame(int pins)
  {
    if (LastBallInFrame(pins))
      AdvanceFrame();
    else
      isFirstThrow = false;
  }

  private bool LastBallInFrame(int pins)
  {
    return Strike(pins) || (!isFirstThrow);
  }

  private bool Strike(int pins)
  {
    return (isFirstThrow && pins == 10);
  }

  private void AdvanceFrame()
  {
    currentFrame++;
    if(currentFrame > 10)
      currentFrame = 10;
  }

  public int ScoreForFrame(int theFrame)
  {
    return scorer.ScoreForFrame(theFrame);
  }
}

//Scorer.cs-------------------------------
public class Scorer
```

```
{
  private int ball;
  private int[] throws = new int[21];
  private int currentThrow;

  public void AddThrow(int pins)
  {
    throws[currentThrow++] = 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 int NextTwoBallsForStrike
  {
    get { return (throws[ball+1] + throws[ball+2]); }
  }

  private int NextBallForSpare
  {
    get { return throws[ball+2]; }
  }

  private bool Strike()
  {
    return throws[ball] == 10;
```

```
  }

  private int TwoBallsInFrame
  {
    get { return throws[ball] + throws[ball+1]; }
  }

  private bool Spare()
  {
    return throws[ball] + throws[ball+1] == 10;
  }
}
```

БМ: Выглядит очень прилично. Я больше ничего придумать не могу.

БК: Да, вполне. Ну уж заодно посмотрим и на тесты.

```
//GameTest.cs--------------------------------
using NUnit.Framework;

[TestFixture]
public class GameTest
{
  private Game game;

  [SetUp]
  public void SetUp()
  {
    game = new Game();
  }

  [Test]
  public void TestTwoThrowsNoMark()
  {
    game.Add(5);
    game.Add(4);
    Assert.AreEqual(9, game.Score);
  }

  [Test]
  public void TestFourThrowsNoMark()
  {
    game.Add(5);
    game.Add(4);
    game.Add(7);
    game.Add(2);
    Assert.AreEqual(18, game.Score);
    Assert.AreEqual(9, game.ScoreForFrame(1));
    Assert.AreEqual(18, game.ScoreForFrame(2));
  }
```

```
[Test]
public void TestSimpleSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  Assert.AreEqual(13, game.ScoreForFrame(1));
}

[Test]
public void TestSimpleFrameAfterSpare()
{
  game.Add(3);
  game.Add(7);
  game.Add(3);
  game.Add(2);
  Assert.AreEqual(13, game.ScoreForFrame(1));
  Assert.AreEqual(18, game.ScoreForFrame(2));
  Assert.AreEqual(18, game.Score);
}

[Test]
public void TestSimpleStrike()
{
  game.Add(10);
  game.Add(3);
  game.Add(6);
  Assert.AreEqual(19, game.ScoreForFrame(1));
  Assert.AreEqual(28, game.Score);
}

[Test]
public void TestPerfectGame()
{
  for (int i=0; i<12; i++)
  {
    game.Add(10);
  }
  Assert.AreEqual(300, game.Score);
}

[Test]
public void TestEndOfArray()
{
  for (int i=0; i<9; i++)
  {
    game.Add(0);
    game.Add(0);
  }
  game.Add(2);
```

```
    game.Add(8); // спэа в 10-м фрейме
    game.Add(10); // страйк в последней позиции массива
    Assert.AreEqual(20, game.Score);
  }

  [Test]
  public void TestSampleGame()
  {
    game.Add(1);
    game.Add(4);
    game.Add(4);
    game.Add(5);
    game.Add(6);
    game.Add(4);
    game.Add(5);
    game.Add(5);
    game.Add(10);
    game.Add(0);
    game.Add(1);
    game.Add(7);
    game.Add(3);
    game.Add(6);
    game.Add(4);
    game.Add(10);
    game.Add(2);
    game.Add(8);
    game.Add(6);
    Assert.AreEqual(133, game.Score);
  }

  [Test]
  public void TestHeartBreak()
  {
    for (int i=0; i<11; i++)
      game.Add(10);
    game.Add(9);
    Assert.AreEqual(299, game.Score);
  }

  [Test]
  public void TestTenthFrameSpare()
  {
    for (int i=0; i<9; i++)
      game.Add(10);
    game.Add(9);
    game.Add(1);
    game.Add(1);
    Assert.AreEqual(270, game.Score);
  }
}
```

БК: Вроде все покрывается. Можешь придумать еще какие-нибудь осмысленные тесты?

БМ: Нет, я думаю, что набор годится. В данный момент нет ничего такого, что я хотел бы исключить.

БК: Тогда все сделано.

БМ: И мне так кажется. Огромное спасибо за помощь.

БК: Да не за что. Мне понравилось.

Заключение

Написав эту главу, я опубликовал ее на сайте компании Object Mentor.[1] Ее прочитало и прокомментировало много посетителей. Некоторым не понравилось, что почти не применялось объектно-ориентированное проектирование. Я нахожу такую реакцию любопытной. Так ли уж необходимо применять объектно-ориентированное проектирование в любом приложении? В данной программе такой необходимости просто не возникло. В самом деле, класс `Scorer` был единственной уступкой ОО, но даже он введен ради простого разделения функций, а не ООП как такового.

Другие считают, что класс `Frame` все-таки должен быть. Один читатель даже создал свою версию программы с классом `Frame`. Она оказалась гораздо длиннее и сложнее, чем приведенная выше.

Кое-кто считал, что мы не справедливы к UML. Ведь мы начали писать код, не закончив проектирования. Нельзя же считать маленькую смешную диаграмму, нарисованную на салфетке (см. рис. 6.2), полным проектом; диаграммы последовательности отсутствуют. Это довольно странный аргумент. Не думаю, что включение в рис. 6.2 диаграмм последовательности заставило бы нас отказаться от классов `Throw` и `Frame`. Напротив, я полагаю, что мы только утвердились бы во мнении об их необходимости.

Хочу ли я сказать, что диаграммы не нужны? Конечно, нет. Хотя, пожалуй, в каком-то смысле – да. В этой программе диаграммы ничем не помогли бы, а только отвлекали бы внимание. Последуй мы им – и получилась бы программа, куда более сложная, чем необходимо. Вы можете возразить, что такую программу было бы проще сопровождать, но я и с этим не согласен. Представленная программа проста для понимания, а значит, и для сопровождения. В ней нет плохо управляемых зависимостей, делающих ее слишком жесткой или слишком хрупкой.

Таким образом, да, диаграммы иногда бесполезны. А когда именно? Когда вы создаете их без кода, который мог бы подтвердить их пра-

[1] *www.objectmentor.com*

вильность, *а затем намереваетесь им строго следовать*. Нет ничего дурного в том, чтобы нарисовать диаграмму для анализа идеи. Но, нарисовав диаграммы, не нужно думать, что на них представлен лучший дизайн из возможных. Оптимальный дизайн может выявиться по мере выполнения небольших шагов при наличии предварительно написанных тестов.

В подтверждение этого вывода позвольте привести слова генерала Дуайта Дэвида Эйзенхауэра: *«Готовясь к сражению, я неизменно обнаруживал, что планы бесполезны, но само планирование совершенно необходимо»*.

II

Гибкое проектирование

Если *гибкость* означает построение программы мелкими шажками, то как вообще можно *проектировать* такую программу? Какой этап позволяет убедиться, что программа имеет хорошую структуру, что она гибкая, удобная для сопровождения и повторного использования? Если программа создается инкрементно, то не готовим ли мы площадку для мусора и переработки во имя факторинга? Не упустим ли мы из виду общую картину?

В гибкой команде общая картина вырисовывается по ходу создания программы. На каждой итерации команда улучшает дизайн системы, так чтобы он оставался максимально хорошим на *данной* стадии развития. Команда не тратит время на анализ будущих требований и потребностей. И не пытается сегодня построить инфраструктуру, которая завтра, быть может, не понадобится. Вместо этого команда сосредотачивает усилия на *текущей* структуре системы, стараясь сделать ее как можно лучше.

Но это не отказ от архитектуры и проектирования, а, скорее, постепенная эволюция в сторону наиболее подходящей архитектуры и дизайна системы. Это также способ гарантировать, что дизайн и архитектура будут оставаться на достойном уровне по мере роста и развития системы. При гибкой разработке процесс проектирования и выработки архитектуры *непрерывен*.

Но откуда мы знаем, что дизайн программной системы хорош? В главе 7 перечисляются и описываются признаки плохого дизайна. Этот «дурной запашок» часто пропитывает всю структуру программы. В этой главе описывается, как такие симптомы накапливаются, и объясняется, как их избежать.

Перечислим эти признаки:

- *Жесткость*: дизайн трудно поддается изменению.

- *Хрупкость*: дизайн легко разрушается.
- *Косность*: дизайн трудно использовать повторно.
- *Вязкость*: трудно добиться желаемого.
- *Ненужная сложность*: избыточное проектирование.
- *Ненужные повторения*: чрезмерное использование копирования и вставки.
- *Непрозрачность*: плохо выраженная цель.

По своей природе эти признаки аналогичны «дурно пахнущему» коду, но на более высоком уровне. Эти ароматы, пропитывают структуру программы в целом, а не просто отдельный участок кода.

Нездоровую ауру дизайна как признак можно оценить если не объективно, то хотя бы субъективно. Часто причиной является нарушение одного или нескольких принципов проектирования. В главах 8–12 описываются принципы объектно-ориентированного проектирования, позволяющие избавиться от признаков плохого дизайна – его душка́ – и создать наилучший дизайн для данного набора функций.

Перечислим эти принципы:

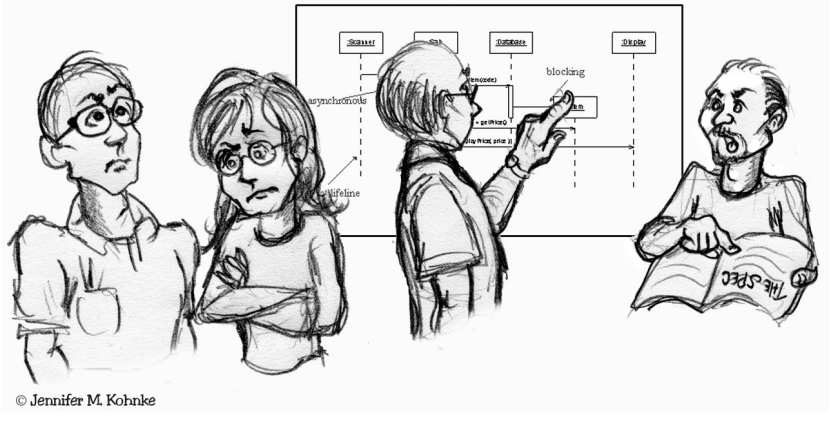
- Глава 8: принцип единственной обязанности (Single-Responsibility Principle – SRP).
- Глава 9: принцип открытости/закрытости (Open/Closed Principle – OCP).
- Глава 10: принцип подстановки Лисков (Liskov Substitution Principle – LSP).
- Глава 11: принцип инверсии зависимости (Dependency-Inversion Principle – DIP).
- Глава 12: принцип разделения интерфейсов (Interface Segregation Principle – ISP).

Эти принципы были выработаны ценой больших усилий за многие десятилетия развития технологии программного обеспечения. Это не плод трудов одного человека, а продукт объединения мыслей и работ большого числа разработчиков и исследователей. Хотя в этой книге они представлены как принципы объектно-ориентированного проектирования, на самом деле это лишь частный случай устоявшихся принципов технологии создания программного обеспечения.

Гибкие команды применяют эти принципы только, чтобы устранить запашок; когда ничем плохим не пахнет, то и принципы не применяются. Было бы ошибкой безоговорочно придерживаться некоторого принципа просто потому, что это принцип. Принципы существуют для того, чтобы помогать в устранении дурных запахов. Это не духи, которыми надо обильно поливать всю систему. Чрезмерная приверженность принципам ведет к пороку ненужной сложности.

7

Что такое гибкое проектирование



Проанализировав жизненный цикл разработки
программного обеспечения так, как я его понимал,
я пришел к выводу, что единственная документация,
которая хоть как-то удовлетворяет
критериям инженерного проектирования, –
это листинги с исходным кодом.

Джек Ривз

В 1992 году Джек Ривз опубликовал концептуальную статью «What Is Software Design?» в журнале *C++ Journal*.[1] В ней Ривз доказывал, что дизайн программной системы документирован главным образом ее исходным кодом, что диаграммы, описывающие исходный код, – это до-

[1] [Reeves92] Это замечательная работа. Настоятельно рекомендую прочитать. Ее текст включен в приложение B к этой книге.

полнение к дизайну, а не сам дизайн. Как оказалось, статья Джека стала предвестником технологий гибкой разработки.

Ниже мы часто будем говорить о «дизайне». Не следует думать, что речь идет о комплекте UML-диаграмм, отдельных от кода. Диаграммы могут представлять некоторые части дизайна, но это не *сам* дизайн. Дизайн программной системы – абстрактная концепция. Он касается общего вида и структуры программы, а также детального вида и структуры каждого модуля, класса и метода. Дизайн можно представить в различных формах, но окончательным его воплощением является исходный код. В конечном итоге исходный код и является дизайном.

Ароматы дизайна

Если вам повезло, то вы начинаете работу над новым проектом с четким представлением о том, какая система должна получиться. Дизайн системы запечатлен у вас в голове. Если вам исключительно повезло, то эта ясность дизайна сохраняется вплоть до выпуска первой версии.

Но потом все идет наперекосяк. Программа начинает гнить, как кусок испорченного мяса. Проходит время, а процесс загнивания продолжается. В коде накапливаются гнойные язвы и нарывы, сопровождать его становится все труднее. И в конце концов усилия, необходимые для внесения даже простейшего изменения, становятся столь непомерными, что и программисты, и их менеджеры начинают требовать перепроектирования.

Но такое перепроектирование редко приносит успех. Хотя поначалу проектировщики исполнены самых благих намерений, постепенно обнаруживается, что они стреляют по движущейся мишени. Старая система продолжает развиваться и изменяться, а новый дизайн не должен отставать. Язвы и фурункулы накапливаются в новом дизайне еще до выпуска первой версии.

Ароматы дизайна – запахи гниющей программы

О том, что программа начинает загнивать, можно узнать по появлению следующих ароматов:

- Жесткость
- Хрупкость
- Косность
- Вязкость
- Ненужная сложность
- Ненужные повторения
- Непрозрачность

Жесткость

Жесткость – это характеристика программы, затрудняющая внесение в нее изменений, даже самых простых. Дизайн жесткий, если единственное изменение вызывает целый каскад других изменений в зависимых модулях. Чем больше модулей приходится изменять, тем жестче дизайн.

Большинству разработчиков приходилось так или иначе сталкиваться с подобной ситуацией. Человека просят внести, казалось бы, простенькое изменение. Он анализирует его и сообщает разумную оценку трудоемкости. Но по ходу работы выявляются непредвиденные обстоятельства. Разработчик вынужден прослеживать эффект изменения, копаясь в огромных фрагментах кода, модифицировать куда больше модулей, чем предполагалось, и обнаруживать целые пласты других изменений, которые надо не забыть сделать. В конце концов работа занимает гораздо больше времени, чем планировалось изначально. Когда программисту задают вопрос, почему оценка оказалась такой недостоверной, он отвечает традиционной жалобой разработчиков: «Это оказалось гораздо сложнее, чем я думал!»

Хрупкость

Хрупкость – это свойство программы повреждаться во многих местах при внесении единственного изменения. Зачастую новые проблемы возникают в частях, не имеющих концептуальной связи с той, что была изменена. Исправление одних проблем ведет к появлению новых, и команда разработчиков уподобляется собаке, пытающейся укусить себя за хвост.

По мере роста хрупкости модуля вероятность того, что изменение породит неожиданные проблемы, приближается к 100 процентам. Каким бы абсурдным это ни казалось, но такие модули далеко не редкость. Это те модули, что пребывают в состоянии перманентного исправления и никогда не исчезают из списка замеченных ошибок. Разработчики знают, что их надо бы перепроектировать, но никто не хочет браться за это безнадежное дело. Такие модули становятся тем *хуже*, чем старательнее их пытаются исправить.

Косность

Дизайн является косным, если он содержит части, которые могли бы оказаться полезны в других системах, но усилия и риски, сопряженные с попыткой отделить эти части от оригинальной системы, слишком велики. Печальная, но часто встречающаяся ситуация.

Вязкость

Вязкость проявляется в двух формах: вязкость программы и вязкость окружения. Сталкиваясь с необходимостью внести изменение, разра-

ботчик обычно находит несколько способов сделать это. Одни сохраняют дизайн, другие – нет (то есть являются, по существу, «хаком»). Если сохраняющие дизайн подходы оказывается реализовать труднее, чем «хак», то вязкость дизайна высока. Решить задачу неправильно легко, а правильно – трудно. Мы хотим проектировать наши программы так, чтобы легко было вносить изменения, сохраняющие дизайн.

Вязкость окружения проявляется, когда среда разработки медленная и неэффективная. Например, если время компиляции очень велико, то у разработчика появляется искушение внести изменение так, чтобы не требовалось массивной перекомпиляции, даже если при этом нарушается дизайн. Если постановка нескольких файлов на учет в систему управления версиями занимает несколько часов, то разработчик будет стремиться изменить как можно меньше файлов, наплевав на сохранение дизайна.

В обоих случаях проект, дизайн которого трудно сохранить при изменениях, следует считать вязким. Мы хотим создавать такие системы и среды разработки, чтобы сохранять и улучшать дизайн было легко.

Ненужная сложность

Дизайн попахивает ненужной сложностью, если содержит элементы, неиспользуемые в текущий момент. Это часто случается, когда разработчики стараются предвидеть будущие изменения требований и вставлять в программу средства для их поддержки. На первый взгляд, вполне похвальное желание. Ведь готовность к будущим изменениям должна сделать код гибким и предотвратить кошмар последующих переделок.

К сожалению, эффект нередко оказывается прямо противоположным. Готовясь к самым разным ситуациям, мы засоряем дизайн конструкциями, которые никогда не будут востребованы. Некоторые прогнозы оправдываются, большинство – нет. А между тем дизайн обременен всеми этими неиспользуемыми элементами. В результате программа становится сложной и малопонятной.

Ненужные повторения

Копирование и вставка полезны при редактировании текста, но могут оказывать разрушительное влияние при редактировании кода. Слишком часто системы включают десятки, а то и сотни повторяющихся фрагментов кода. Происходит это так: Ивану нужно написать некий код, который *курдячит бокренка*[1]. Полазив по разным частям программы, где, как он подозревает, бокрят уже не раз курдячили, он находит подходящий фрагмент. Копирует его, вставляет в свой модуль и вносит необходимые изменения.

[1] Под этой бессмысленной фразой подразумевается функция программы, назначение которой неясно и сомнительно.

Но Иван не знает, что код, который он извлек мышкой, поместил туда Петр, взявший его из модуля, написанного Светой. Свете первой пришлось курдячить бокренка, но она знала, что этот процесс очень похож на курдячение бармаглотов. Она где-то отыскала код, который курдячит бармаглота, скопировала его в свой модуль и модифицировала.

Когда один и тот же код в слегка различающихся формах появляется снова и снова, разработчики утрачивают представление об абстракции. Поиск всех повторов и замена их подходящей абстракцией стоят в списке приоритетов не слишком высоко, хотя это сделало бы систему гораздо понятнее и удобнее для сопровождения.

Если в системе есть дублирующийся код, то задача ее изменения может потребовать значительных усилий. Ошибки, обнаруженные в повторяющемся блоке, должны быть исправлены во всех его копиях. Но, поскольку повторения немного отличаются друг от друга, то и исправления будут разными.

Непрозрачность

Непрозрачность – это трудность модуля для понимания. Код может быть ясным и выразительным или темным и запутанным. Код, эволюционирующий со временем, постепенно становится все более и более непрозрачным. Дабы свести непрозрачность к минимуму, нужно постоянно следить, чтобы он оставался ясным и выразительным.

Когда разработчик пишет первую версию модуля, она может казаться ему вполне ясной. Ведь он с головой погрузился в код, любит его и знает до мельчайших деталей. Но прошло время, любовь забылась, разработчик возвращается к модулю и недоумевает, как он мог написать такое чудовище. Чтобы не оказаться в подобной ситуации, разработчик должен примерять на себя роль читателя и прилагать сознательные усилия к рефакторингу кода таким образом, чтобы тот был понятен читателям. Кроме того, код необходимо отдавать на рецензию коллегам.

Почему программы загнивают

В негибком окружении дизайн деградирует, потому что требования изменяются непредвиденным образом. Часто изменения нужно вносить быстро, и делают это разработчики, не знакомые с философией исходного дизайна. Поэтому изменение хотя и работает, но в каком-то отношении расходится с первоначальным дизайном. Постепенно изменения накапливаются, и в конце концов образуется злокачественная опухоль.

Но мы не можем считать причиной деградации дизайна лишь изменяющиеся требования. Мы, разработчики, отлично знаем, что требования всегда изменяются. Более того, мы понимаем, что требования – самая непостоянная часть проекта. Если дизайн рушится под напором непрекращающегося потока изменяющихся требований, то виноваты в том сам дизайн и применяемая методология. Необходимо найти способ сде-

лать дизайн устойчивым к изменениям и применять методики, защищающие программу от загнивания.

Гибкая команда только расцветает, сталкиваясь с изменениями. Она не прикладывает слишком много усилий заранее, поэтому не боится устаревания первоначального дизайна. Вместо этого команда стремится сделать дизайн максимально ясным и простым, страхуя себя множеством автономных и приемочных тестов. В результате дизайн остается эластичным и допускает модификацию без труда. Команда пользуется этой гибкостью, чтобы постоянно улучшать дизайн; поэтому каждая итерация завершается системой, дизайн которой идеально соответствует требованиям, сформулированным на данной итерации.

Программа Copy

Знакомый сценарий

Проиллюстрируем высказанные выше соображения, понаблюдав за тем, как дизайн загнивает. Допустим, что в одно прекрасное утро понедельника начальник просит вас написать программу, которая выводит непосредственно на принтер текст, вводимый с клавиатуры. Прокрутив задачу в голове, вы приходите к выводу, что программа займет меньше десяти строк кода. На проектирование и кодирование уйдет меньше часа. Но если принять во внимание всякие совещания межотдельских групп, занятия по повышению качества, ежедневные планерки, да еще три текущих неотложных сообщения об ошибках от эксплуатационщиков, то потребуется неделя – и то если засиживаться вечерами. Но вы, как положено, умножаете свою оценку на 3.

«Три недели», сообщаете вы начальнику. Он хмыкает и уходит, оставляя вас заниматься своим делом.

Первоначальный дизайн. У вас выдалась свободная минутка перед совещанием по разбору процесса, поэтому вы решили прикинуть дизайн программы. Будучи приверженцем структурного проектирования, вы рисуете структурную диаграмму, изображенную на рис. 7.1.

Приложение будет состоять из трех модулей, или подпрограмм. Модуль `Copy` вызывает остальные два. Программа `Copy` получает символы от модуля `Read Keyboard` и передает их модулю `Write Printer`.

Вы смотрите на свой дизайн и видите, что это хорошо. Улыбаетесь и идете на совещание. Там, по крайней мере, можно будет немного вздремнуть.

Во вторник вы приходите на работу пораньше, чтобы закончить с программой `Copy`. К несчастью, ночью как раз дала о себе знать ошибка, о которой твердили эксплуатационщики, и вам нужно бежать на стенд, чтобы помочь в ее отладке. Во время обеденного перерыва, на который вам наконец удалось вырваться в 3 часа дня, вы все-таки набираете код программы `Copy`. Результат показан в листинге 7.1.

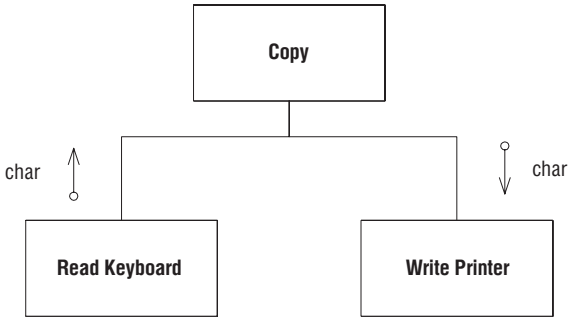


Рис. 7.1. *Структурная диаграмма программы Copy*

Листинг 7.1. Программа Copy

```
public class Copier
{
  public static void Copy()
  {
    int c;
    while((c=Keyboard.Read()) != -1)
      Printer.Write(c);
  }
}
```

Сохранив результаты редактирования, вы вспоминаете, что уже опаздываете на совещание по качеству. А оно очень важно, там собираются говорить о стоимости разработки без дефектов. Поэтому вы наскоро заглатываете пирожное, запиваете его колой и мчитесь на совещание.

В среду вы снова приходите рано утром, и на этот раз, кажется, ничто не должно помешать. Вы загружаете исходный код программы `Copy` и начинаете его компилировать. О чудо, компилируется с первого раза без ошибок! И это хорошо, потому что тут звонит начальник и приглашает на внеплановое совещание по поводу экономии тонера для лазерных принтеров.

В четверг, провисев четыре часа на телефоне (вместе с сотрудником службы поддержки из Роки-Маунт в Северной Каролине вы занимались удаленной отладкой и протоколированием ошибок в одном из малопонятных компонентов системы), вы закусываете шоколадкой и приступаете к тестированию программы `Copy`. Работает, ну надо же! И это хорошо еще и потому, потому что студент-стажер ухитрился стереть главный каталог с исходными текстами и вам предстоит разыскивать резервную копию и восстанавливать все с нее. Разумеется, последняя полная копия снималась аж три месяца назад, так что на нее надо будет еще накатить 94 инкрементных.

На пятницу ничего не назначено. Удачно, потому что постановка программы `Copy` на учет в систему управления версиями заняла целый день.

Разумеется, программа имеет бешеный успех и внедрена во всех подразделениях компании. Ваша репутация классного программиста получила еще одно подтверждение, и вы купаетесь в лучах славы. Если все хорошо сложится, то, возможно, удастся написать за этот год целых 30 строк кода!

А требования-то изменяются. Спустя несколько месяцев приходит начальник и говорит, что программа `Copy` должна еще читать с перфоленты. Вы скрипите зубами и закатываете глаза. Ну почему они всегда меняют требования?! Программа же не проектировалась для чтения перфолент! Вы предупреждаете шефа, что такие изменения разрушат стройность дизайна. Но шеф твердо заявляет, что пользователям время от времени необходимо читать символы с перфолент.

С тяжелым вздохом вы начинаете планировать модификации. Надо бы добавить в функцию `Copy` булевский аргумент. Если он равен `true`, то будем читать с перфоленты, а если `false`, то с клавиатуры, как и раньше. К несчастью, уже столько программ пользуются функцией `Copy`, что изменить ее интерфейс невозможно. На повторную компиляцию и тестирование ушло бы много недель. Вас линчевали бы системные тестировщики, не говоря уже о семи парнях из группы по управлению конфигурацией. А уж у службы контроля за процессами будет настоящий праздник – еще бы, предстоит ревизия кода каждого модуля, вызывающего `Copy`!

Нет, изменение интерфейса исключается. Но тогда как программа `Copy` узнает, что должна читать с перфоленты? Эврика! Воспользуемся глобальной переменной! А заодно самым лучшим и полезным средством в языках, берущих начало от C, оператором `?:`! Результат представлен в листинге 7.2.

Листинг 7.2. Первая модификация программы Copy

```
public class Copier
{
  //не забудьте сбросить этот флаг
  public static bool ptFlag = false;
  public static void Copy()
  {
    int c;
    while((c=(ptFlag ? PaperTape.Read()
                     : Keyboard.Read())) != -1)
    Printer.Write(c);
  }
}
```

Если программа хочет вызвать `Copy` для чтения с перфоленты, то должна сначала установить для переменной `ptFlag` значение `true`. А после завершения программы `Copy` нужно сбросить этот флаг, иначе следующий обратившийся будет читать с перфоленты, а не с клавиатуры. Чтобы напомнить программистам о необходимости сбрасывать флаг, вы включили комментарий.

И вот новая версия программы предложена на суд публики. Успех еще более оглушительный, толпе восхищенных программистов не терпится ею воспользоваться. Жизнь прекрасна.

Только дай ему волю. Проходит несколько недель. И ваш начальник – а он все еще начальник, несмотря на три корпоративных реорганизации за последние три месяца, – радует вас известием о том, что заказчики иногда хотели бы использовать программу `Copy` для вывода на перфоленту. Ох уж эти заказчики! Всегда-то они портят ваш дизайн. *Насколько проще было бы писать программы, если бы не заказчики.* Вы говорите шефу, что эти бесконечные изменения резко отрицательно сказываются на элегантности вашего дизайна, и предупреждаете, что если новые требования и дальше будут поступать в таком темпе, то к концу года программу будет невозможно сопровождать. Шеф понимающе кивает, но требует, чтобы изменения все-таки были внесены.

Эта модификация аналогична предыдущей. Надо лишь создать еще одну глобальную переменную и снова воспользоваться оператором `?:`! В листинге 7.3 показаны плоды ваших усилий.

Особенно вы гордитесь тем, что не забыли изменить комментарий. Но как-то беспокоит тот факт, что структура программы начинает расползаться. Стоит добавить еще одно устройство ввода – и придется полностью менять структуру условия в цикле `while`. Не пора ли стряхнуть пыль со своего резюме?

Листинг 7.3. Вторая модификация программы Copy

```
public class Copier
{
  //не забудьте сбросить эти флаги
  public static bool ptFlag = false;
  public static bool punchFlag = false;
  public static void Copy()
  {
    int c;
    while((c=(ptFlag ? PaperTape.Read()
                     : Keyboard.Read())) != -1)
      punchFlag ? PaperTape.Punch(c) : Printer.Write(c);
  }
}
```

Ожидайте изменений. Предоставляю вам самим решать, сколько в этом рассказе сатирического преувеличения. Но смысл его в том, чтобы продемонстрировать, как быстро дизайн программы может деградировать при наличии изменений. Первоначальный дизайн программы `Copy` был простым и элегантным. Но после всего двух изменений появились признаки жесткости, хрупкости, косности, сложности, дублирования и непрозрачности. Такая тенденция, безусловно, продолжится, и программа превратится в хаос.

Можно откинуться на спинку стула и возложить всю вину на изменения. Можно поплакаться, что программа была идеально спроектирована для первоначальных спецификаций, а последующие изменения привели к деградации дизайна. Но тем самым вы игнорируете фундаментальную особенность разработки ПО: *требования изменяются всегда*!

Следует помнить, что самой изменчивой частью в большинстве программных проектов являются именно требования. Они находятся в постоянном движении. И мы, разработчики, должны с этим фактом смириться! *Мы живем в мире изменяющихся требований, и наша задача состоит в том, чтобы написанные нами программы смогли успешно пережить эти изменения.* Если дизайн программы деградирует из-за изменения требования, то ни о какой гибкой разработке говорить не приходится.

Гибкий дизайн программы Copy

Гибкая команда разработчиков могла бы начать с того же кода, что показан в листинге 7.1.[1] Если бы начальник попросил, чтобы программа могла читать с перфоленты, разработчики отреагировали бы внесением в дизайн таких поправок, чтобы он был эластичен к подобным изменениям. Результат мог бы выглядеть, как показано в листинге 7.4.

Листинг 7.4. Вторая модификация программы Copy

```
public interface Reader
{
  int Read();
}

public class KeyboardReader : Reader
{
  public int Read() {return Keyboard.Read();}
}

public class Copier
{
  public static Reader reader = new KeyboardReader();
  public static void Copy()
  {
    int c;
    while((c=(reader.Read())) != -1)
      Printer.Write(c);
  }
}
```

[1] На самом деле методика разработки через тестирование, скорее всего, естественно привела бы к достаточно гибкому дизайну, способному выдержать требования шефа без каких бы то ни было изменений. Но в данном примере забудем об этом.

Вместо того чтобы приспосабливать дизайн к новому требованию, команда пользуется возможностью улучшить дизайн так, чтобы он был эластичен к аналогичным изменениям в будущем. Если теперь начальник попросит добавить еще одно устройство ввода, то команда сможет сделать это без деградации программы `Copy`.

В данном случае команда следовала *принципу открытости/закрытости*, который мы опишем в главе 9. Он требует проектировать модули так, чтобы их можно было расширять без модификации. Именно так команда и поступила. Какое бы устройство ввода ни попросил добавить начальник, это можно сделать, не изменяя саму программу `Copy`.

Отметим, однако, что, начиная проектировать модуль, команда не пыталась предвидеть, в каких направлениях программа может изменяться. А просто написала его максимально просто. Только тогда, когда требования действительно поменялись, команда модифицировала дизайн модуля, учтя все изменения такого рода.

Можно было бы возразить, что команда сделала только половину работы. Учтя наличие различных устройств ввода, разработчики ничего не сделали для адаптации к разнообразию устройств вывода. Но ведь команда ничего не знала о том, будут ли устройства вывода когда-нибудь изменяться. Встраивать на этой стадии дополнительные механизмы было бы бессмысленно. Ясно, что если это когда-нибудь понадобится, то поддержку легко будет добавить. Поэтому делать это сейчас нет никаких причин.

Следование гибким методикам. В нашем примере гибкие разработчики создали абстрактный класс, защищающий от изменения устройства ввода. Откуда они знали, как это сделать? Ответ лежит в одной из фундаментальных доктрин объектно-ориентированного проектирования.

Первоначальный дизайн программы `Copy` был негибким из-за направления зависимостей. Взгляните еще раз на рис. 7.1. Обратите внимание, что модуль `Copy` прямо зависит от `KeyboardReader` и `PrinterWriter`. `Copy` — модуль верхнего уровня. Он определяет всю стратегию работы приложения. Он знает, как копировать символы. Но, к сожалению, он зависит от низкоуровневых элементов: клавиатуры и принтера. Любое изменение этих элементов сказывается на стратегии на верхнем уровне.

Столкнувшись с таким отсутствием эластичности, гибкие разработчики сразу поняли, что направление зависимости от модуля `Copy` к устройству ввода следует *инвертировать*, воспользовавшись принципом инверсии зависимости (глава 11), устранив тем самым зависимость `Copy` от устройства ввода. Затем они воспользовались паттерном Стратегия (Strategy), обсуждаемым в главе 22, чтобы реализовать желаемую инверсию.

Итак, гибкие разработчики знали, что делать, потому что выполнили следующие шаги:

1. Выявили проблему, следуя гибким методикам.
2. Диагностировали проблему, применив гибкие принципы.

3. Разрешили проблему, применив подходящий паттерн проектирования.

Взаимодействие этих трех аспектов разработки ПО и есть акт проектирования.

Поддержание дизайна в оптимальном виде. Гибкие разработчики всеми силами стремятся поддерживать дизайн адекватным и чистым. И это происходит не время от времени, когда выдастся свободная минутка. Гибкие разработчики не занимаются «уборкой» раз в несколько недель, а постоянно следят за чистотой, простотой и выразительностью программы – ежедневно, ежечасно, ежеминутно. Они никогда не говорят: «Попозже исправим». Они не позволяют начаться гниению.

Гибкие разработчики относятся к дизайну ПО так же, как хирург к процедуре стерилизации. Только благодаря стерилизации хирургия вообще *может существовать*. Не будь ее, риск инфицирования был бы недопустимо высок. Точно так же гибкие разработчики рассуждают о дизайне. Нельзя мириться с риском даже малейшего проявления загнивания.

Дизайн должен оставаться чистым. А поскольку исходный код – самое важное выражение дизайна, то и он должен оставаться чистым. Профессионализм требует, чтобы мы, разработчики программ, не мирились с загниванием кода.

Заключение

Итак, что такое гибкое проектирование? Это процесс, а не разовое событие. Это постоянное применение определенных принципов, паттернов и методик для улучшения структуры и понятности программы. Это стремление все время поддерживать дизайн максимально простым, чистым и выразительным.

В последующих главах мы будем исследовать принципы и паттерны проектирования ПО. По ходу чтения помните, что гибкий разработчик не применяет их, чтобы с самого начала предусмотреть в дизайне все на свете. Нет, они применяются на каждой итерации, чтобы поддерживать в чистоте код и воплощенный в нем дизайн.

Библиография

[Reeves92] Jack Reeves «What Is Software Design?», *C++ Journal*, (2), 1992. Доступна также по адресу *www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm*.

8

Принцип единственной обязанности (SRP)



Никто, кроме самого Будды, не должен брать на себя ответственность за сокровенные знания...

Э. Кобхэм Брюэр 1810–1897,
«Dictionary of Phrase and Fable» (1898)

Этот принцип впервые был описан в работе Тома Демарко[1] и Мейлира Пейдж-Джонса[2]. Они назвали его *сцепленностью* (Cohesion), определив ее как функциональную соотнесенность элементов модуля. В этой главе мы несколько переиначим смысл этого термина и будем понимать под сцепленностью силы, заставляющие модуль или класс изменяться.

> **Принцип единственной обязанности (Single-Responsibility Principle – SRP)**
>
> *У класса должна быть только одна причина для изменения.*

Рассмотрим игру в боулинг, о которой шла речь в главе 6. На протяжении почти всей разработки у класса `Game` было две различных обязанности: отслеживать текущий фрейм и вычислять счет. В конце концов БМ и БК разнесли эти обязанности по разным классам. Класс `Game` сохранил ответственность за отслеживание фреймов, а ответственность за ведение счета перешла к классу `Scorer`.

Почему так важно было поручить эти обязанности разным классам? Потому, что каждая обязанность – это ось изменения. Любое изменение требований проявляется в изменении распределения обязанностей между классами. Если класс берет на себя несколько обязанностей, то у него появляется несколько причин для изменения.

Если класс отвечает за несколько действий, то его обязанности оказываются связанными. Изменение одной обязанности может привести к тому, что класс перестанет справляться с другими. Такого рода связанность – причина хрупкого дизайна, который неожиданным образом разрушается при изменении.

Рассмотрим, к примеру, дизайн на рис. 8.1. У класса `Rectangle` есть два метода. Один рисует прямоугольник на экране, другой вычисляет площадь прямоугольника.

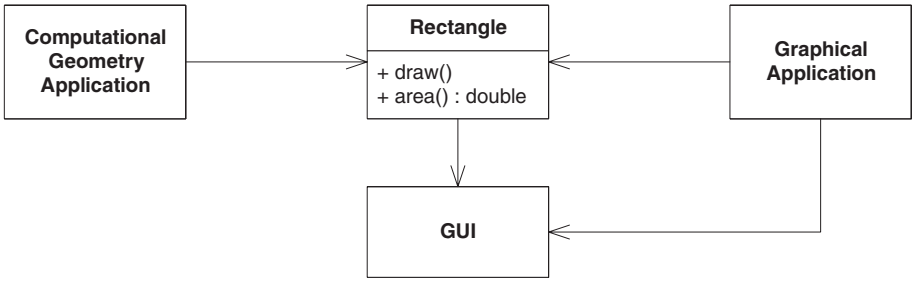


Рис. 8.1. *Более одной обязанности*

[1] [DeMarco79], стр. 310

[2] [PageJones88], стр. 82

Классом Rectangle пользуются два разных приложения. Одно относится к области вычислительной геометрии. Класс Rectangle в нем применяется для математических вычислений с геометрическими фигурами, но на экране оно ничего не рисует. Другое приложение графическое, оно может каким-то боком касаться и вычислительной геометрии, но совершенно точно выводит прямоугольник на экран.

Такой дизайн нарушает принцип SRP. У класса Rectangle две обязанности: предоставление математической модели прямоугольника и рисование прямоугольника в графическом интерфейсе пользователя (ГИП).

Нарушение SRP порождает ряд серьезных проблем. Во-первых, мы должны включать ГИП в приложение вычислительной геометрии. В .NET придется собирать относящуюся к ГИП сборку и развертывать ее вместе с приложением вычислительной геометрии.

Во-вторых, если изменение приложения GraphicalApplication по какой-то причине потребует изменить класс Rectangle, то нам придется заново собирать, тестировать и развертывать приложение ComputationalGeometryApplication. Если мы забудем об этом, то приложение может неожиданно перестать работать.

Более правильный подход к дизайну состоит в том, чтобы распределить обязанности по двум разным классам, как показано на рис. 8.2. Теперь вычислительная часть Rectangle помещена в класс GeometricRectangle и изменения в алгоритме рисования прямоугольников не могут повлиять на приложение ComputationalGeometryApplication.

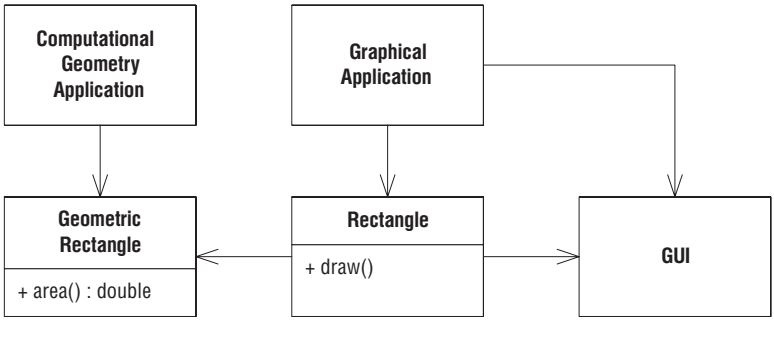


***Рис. 8.2.** Обязанности разделены*

Определение обязанности

В контексте принципа SRP мы будем называть обязанностью *причину изменения*. Если вы можете найти несколько причин для изменения класса, то у такого класса более одной обязанности. Иногда увидеть это трудно. Мы привыкли воспринимать обязанности группами. Рассмотрим, например, интерфейс Modem в листинге 8.1. Многие согласят-

ся, что выглядит он совершенно нормально. Все четыре объявленных метода, несомненно, относятся к модему.

Листинг 8.1. Modem.cs – нарушение SRP

```
public interface Modem
{
  public void Dial(string pno);
  public void Hangup();
  public void Send(char c);
  public char Recv();
}
```

Однако здесь присутствуют две обязанности: управление соединением (методы `Dial` и `Hangup`) и передача данных (методы `Send` и `Recv`).

Следует ли разделить эти обязанности? Все зависит от того, как именно изменяется приложение. Если модификация подразумевает изменение сигнатуры методов управления соединением, то дизайн начинает попахивать жесткостью, так как классы, вызывающие `Send` и `Recv`, придется повторно компилировать и развертывать чаще, чем хотелось бы. В таком случае обязанности следует разделить, как показано на рис. 8.3. Это защищает приложение-клиент от связанности двух обязанностей.

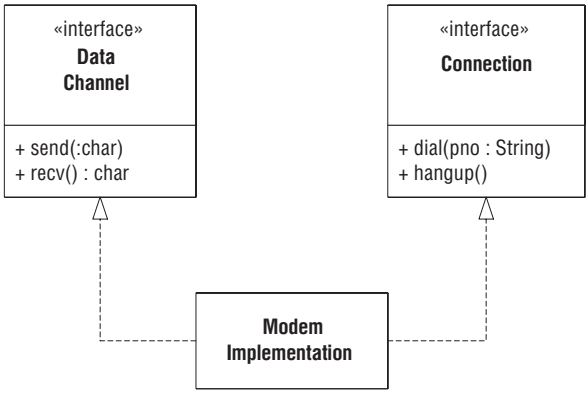


***Рис. 8.3.** Разделенный интерфейс модема*

С другой стороны, если приложение не модифицируют таким образом, что эти обязанности изменяются порознь, то и разделять их нет необходимости. Более того, разделение в этом случае попахивало бы ненужной сложностью.

Отсюда вытекает следствие. *Ось изменения становится таковой, только если изменение имеет место.* Неразумно применять SRP – как и любой другой принцип, – если для того нет причин.

Разделение связанных обязанностей

Обратите внимание, что на рис. 8.3 обе обязанности связаны в классе `ModemImplementation`. Это нежелательно, но может оказаться необходимым. Часто имеются причины, обычно относящиеся к деталям оборудования или операционной системы, которые вынуждают нас связывать такие вещи, которые связывать не хотелось бы. Но, разделив интерфейсы, мы развели эти концепции с точки зрения приложения.

Класс `ModemImplementation` можно считать вынужденным клуджем[1]; однако отметим, что все зависимости ведут *от* него. Ничто не обязано зависеть от этого класса. Ничто, кроме метода `Main`, не обязано знать о его существовании. Таким образом, мы поместили этого уродца за ограду. Его яд не проникнет наружу и не отравит остальное приложение.

Обеспечение сохранности

На рис. 8.4 показано распространенное нарушение принципа SRP. Класс `Employee` содержит как бизнес-правила, так и механизмы управления сохранением. Эти обязанности почти никогда не следует смешивать. Бизнес-правила имеют тенденцию часто изменяться, а механизмы сохранения изменяются реже и совершенно по другим причинам. Связывать бизнес-правила с подсистемой сохранения – значит напрашиваться на неприятности.

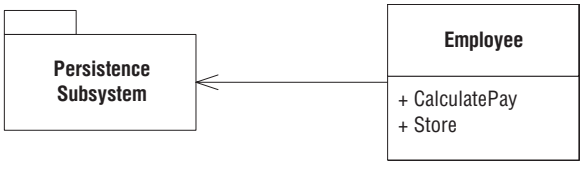


Рис. 8.4. *Связанный механизм сохранения*

К счастью, как мы видели в главе 4, методика разработки через тестирование обычно вынуждает разделять эти обязанности задолго до того, как в дизайне появляется душок. Однако если тесты не заставили это сделать, а аромат жесткости и хрупкости ощущается все сильнее, то дизайн следует подвергнуть рефакторингу, применяя для разделения обязанностей паттерны Фасад, DAO (Data Access Object – Объект доступа к данным) или Заместитель (Proxy).

[1] Программа или часть программы, которая теоретически не должна работать, но почему-то работает. – *Прим. перев.*

Заключение

Принцип единственной обязанности – один из самых простых, но при этом его трудно применять правильно. Сочетание обязанностей для нас выглядит совершенно естественно. Их выявление и разделение как раз и является одной из задач проектирования ПО. Мы еще будем неоднократно возвращаться к этой теме при обсуждении остальных принципов.

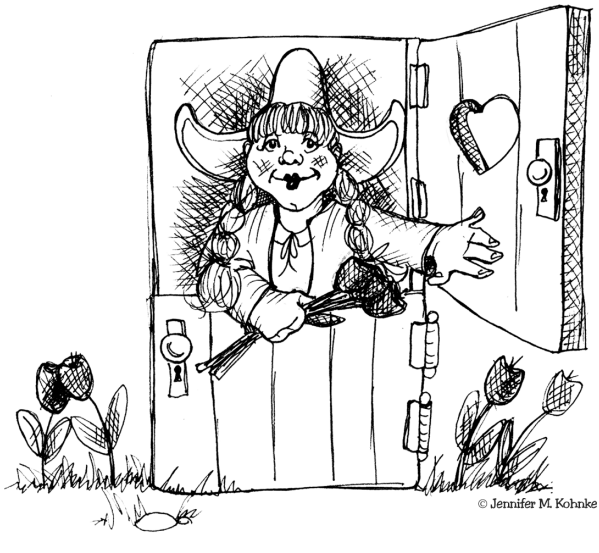
Библиография

[DeMarco79] Tom DeMarco «Structured Analysis and System Specification», Yourdon Press Computing Series, 1979.

[PageJones88] Meilir Page-Jones «The Practical Guide to Structured Systems Design», 2-d ed., Yourdon Press Computing Series, 1988.

9

Принцип открытости/закрытости (OCP)



Голландская дверь: существительное.
Дверь, разделенная на две части по горизонтали,
так что каждая створка может открываться
и закрываться независимо.

The American Heritage Dictionary
of the English Language, Fourth Edition, 2000

Айвар Джекобсон говорил: «Любая система на протяжении своего жизненного цикла претерпевает изменения. Об этом следует помнить, разрабатывая систему, которая предположительно переживет первую версию».[1] Как создавать системы, которые сохраняли бы стабильность перед лицом изменений и просуществовали бы дольше первой версии? Направление указал Бертран Мейер[2], который еще в 1988 году сформулировал ставший ныне знаменитым принцип открытости/закрытости. Вот как он звучит в слегка перефразированном виде:

> **Принцип открытости/закрытости (Open/Closed Principle — OCP)**
>
> *Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для модификации.*

Если единственное изменение в каком-то месте программы приводит к каскаду изменений в зависимых модулях, то дизайн попахивает жесткостью. Принцип OCP рекомендует переработать систему так, чтобы в будущем аналогичные изменения можно было реализовать путем добавления нового кода, а не изменения уже работающего. На первый взгляд это кажется недостижимым идеалом, но существуют относительно простые и эффективные способы *приблизиться* к нему.

Описание принципа OCP

У модулей, согласованных с принципом OCP, есть две основных характеристики.

1. Они *открыты для расширения*. Это означает, что поведение модуля можно расширить. Когда требования к приложению изменяются, мы добавляем в модуль новое поведение, отвечающее изменившимся требованиям. Иными словами, мы можем изменить состав функций модуля.
2. Они *закрыты для модификации*. Расширение поведения модуля не сопряжено с изменениями в исходном или двоичном коде модуля. Двоичное исполняемое представление модуля, будь то компонуемая библиотека, DLL или EXE-файл, остается неизменным.

Может показаться, что эти характеристики противоречивы. Обычно расширение поведения модуля предполагает изменение его исходного кода. Поведение модуля, который нельзя изменить, принято считать фиксированным.

---

[1] [Jacobson92], стр. 21

[2] [Meyer97]

Возможно ли изменить поведение модуля, не трогая его исходного кода? Как можно изменить состав функций модуля, не изменяя сам модуль?

С помощью *абстракции*. В C#, как и в любом другом объектно-ориентированном языке программирования, можно создавать абстракции, которые сами по себе фиксированы, но представляют неограниченное множество различных поведений. Абстракции – это абстрактные базовые классы, а поведения представляются производными от них классами.

Модуль может манипулировать абстракцией. Такой модуль можно сделать закрытым для модификации, поскольку он зависит от фиксированной абстракции. И тем не менее поведение модуля можно расширять, создавая новые производные от абстракции.

На рис. 9.1 изображен простой дизайн, нарушающий принцип OCP. Классы `Client` и `Server` конкретные. Класс `Client` *использует* класс `Server`. Если мы захотим, чтобы объект `Client` использовал другой серверный объект, то класс `Client` придется изменить, указав в нем имя нового серверного класса.

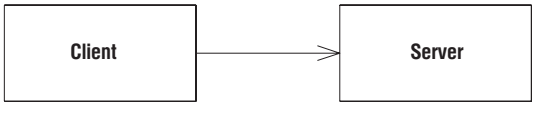


Рис. 9.1. *Класс Client не является открытым и закрытым*

На рис. 9.2 показан дизайн, который согласуется с принципом OCP за счет применения паттерна Стратегия (см. главу 22). В данном случае класс `ClientInterface` абстрактный и содержит только абстрактные методы. Класс `Client` использует эту абстракцию. Однако объекты класса `Client` будут пользоваться объектами производного класса `Server`. Если мы захотим, чтобы объекты `Client` пользовались другим серверным классом, то сможем создать новый класс, производный от `ClientInterface`. Сам класс `Client` при этом не изменится.

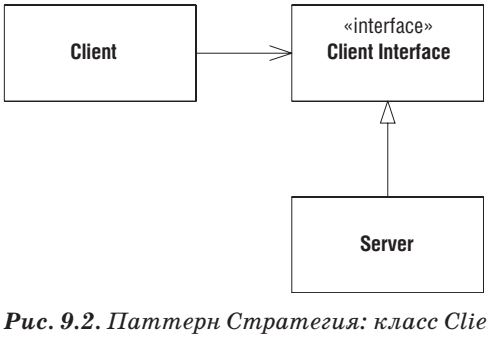


Рис. 9.2. *Паттерн Стратегия: класс Client является одновременно открытым и закрытым*

У класса `Client` есть определенные функции, которые можно описать в терминах абстрактного интерфейса `ClientInterface`. Подтипы `ClientInterface` могут реализовывать этот интерфейс, как сочтут нужным. Таким образом, поведение, специфицированное в классе `Client`, можно расширять и модифицировать путем создания новых подтипов `ClientInterface`.

Возможно, вы недоумеваете, почему я назвал `ClientInterface` именно так. Почему не `AbstractServer`, например? А дело в том, что, как мы увидим ниже, *абстрактные классы более тесно ассоциированы со своими клиентами, чем с реализующими их конкретными классами*.

На рис. 9.3 показана альтернативная структура на основе паттерна Шаблонный метод (Template Method) (см. главу 22). В классе `Policy` есть набор открытых конкретных методов, реализующих некоторую политику; они аналогичны методам класса `Client` на рис. 9.2. Как и раньше, эти методы описывают определенные функции в терминах абстрактных интерфейсов. Но теперь эти абстрактные интерфейсы являются частью самого класса `Policy`. В C# они описывались бы как абстрактные методы. Реализуются они в подтипах `Policy`. Таким образом, поведения, описанные внутри `Policy`, можно расширять или модифицировать путем создания классов, производных от `Policy`.

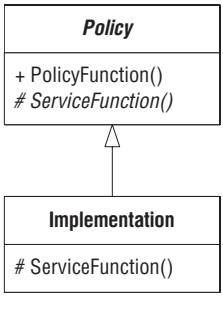


Рис. 9.3. *Паттерн Шаблонный метод: базовый класс является одновременно открытым и закрытым*

Эти два паттерна – наиболее распространенные способы удовлетворить принципу OCP. Они позволяют добиться четкого отделения общей функциональности от деталей ее реализации.

Приложение Shape

Пример `Shape` приводится во многих книгах по объектно-ориентированному проектированию. Обычно на нем иллюстрируют, как работает полиморфизм. Но на этот раз мы воспользуемся им для разъяснения принципа OCP.

У нас имеется приложение, которое должно рисовать круги и квадраты в стандартном ГИП. Рисовать их нужно в определенном порядке. Мы создадим список кругов и квадратов, а программа будет обходить этот список и рисовать встретившиеся фигуры.

Нарушение OCP

На языке C, используя процедурный подход, не согласованный с OCP, мы могли бы решить задачу, как показано в листинге 9.1. Здесь мы видим структуры данных, в которых первый элемент одинаков, а следующие различаются. Первый элемент содержит код типа, идентифицирующий структуру данных как `Circle` или `Square`. Функция `DrawAllShapes` пробегает по массиву указателей на эти структуры и в зависимости от кода вызывает одну из функций – `DrawCircle` или `DrawSquare`.

Листинг 9.1. Процедурное решение задачи о кругах и квадратах

```
--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;
    }
  }
}
```

Поскольку функция `DrawAllShapes` не может быть закрытой для добавления новых видов фигур, то она не удовлетворяет принципу OCP. Если бы я захотел расширить эту функцию так, чтобы она могла рисовать фигуры, в список которых входят треугольники, то должен был бы модифицировать ее код. И так для каждого нового вида фигур.

Разумеется, эта программа – всего лишь простой пример. На практике предложение `switch`, встречающееся в функции `DrawAllShapes`, повторялось бы снова и снова в различных частях приложения и каждый раз выполнялось бы что-то, слегка отличающееся: перетаскивание фигур, растяжение, перемещение, удаление и т. д. Чтобы добавить в такое приложение новую фигуру, пришлось бы отыскать все такие предложения `switch` (или цепочки `if/else`) и в каждое добавить новый вид фигуры.

И нет гарантии, что все предложения `switch` или цепочки `if/else` были бы так же удобно структурированы, как приведенное в функции `DrawAllShapes`. Гораздо вероятнее, что условия в предложениях `if` были бы соединены логическими операторами или что ветви `case` в предложениях `switch` комбинировались бы с целью «упростить» принятие решений. В некоторых патологических ситуациях функции могли бы делать с квадратами то же самое, что и с кругами. В таких функциях вообще не было бы ни предложений `switch/case`, ни цепочек `if/else`. И значит, задача нахождения всех мест, куда нужно добавить новую фигуру, оказалась бы отнюдь не тривиальной.

И еще подумаем о том, какие пришлось бы внести изменения. Нужно было бы добавить новый элемент в перечисление `ShapeType`. Поскольку определения фигур зависят от этого перечисления, то их все пришлось бы перекомпилировать.[1] А заодно и все модули, зависящие от `Shape`.

[1] В языках C/C++ изменение перечисления `enum` может привести к изменению размера содержащей его переменной. Поэтому нужно дважды подумать, принимая решение о том, что объявления фигур перекомпилировать необязательно.



Поэтому нам предстоит не только модифицировать исходный код всех предложений switch/case или цепочек if/else, но также изменить – путем перекомпиляции – двоичные файлы всех модулей, в которых используются структуры данных Shape. Изменение двоичных файлов означает, что все сборки, DLL-библиотеки или иные разновидности двоичных компонентов нужно повторно развернуть. Простое добавление новой фигуры привело к каскаду изменений в различных исходных файлах и даже в двоичных модулях и компонентах. Очевидно, что совершение этой операции вызывает слишком широкий резонанс.

Резюмируем. Решение, показанное в листинге 9.1, жесткое, потому что после добавления фигуры Triangle необходимо заново откомпилировать и развернуть файлы, содержащие определения Shape, Square, Circle и DrawAllShapes. Это решение хрупкое, потому что есть много других предложений switch/case или if/else, которые трудно отыскать и понять. Это решение косное, потому что всякий, кто попытается использовать функцию DrawAllShapes в другой программе, вынужден будет тащить за собой определения Square и Circle, даже если в этой программе они не используются. Короче говоря, от листинга 9.1 веет самыми разными ароматами плохого дизайна.

Решение, удовлетворяющее принципу OCP

В листинге 9.2 показано решение задачи о кругах и квадратах, удовлетворяющее OCP. В нем мы написали абстрактный класс Shape, в котором есть единственный абстрактный метод Draw. Классы Circle и Square являются производными от Shape.

Листинг 9.2. Объектно-ориентированное решение задачи о кругах и квадратах

```
public interface Shape
{
  void Draw();
}

public class Square : Shape
{
  public void Draw()
  {
    // нарисовать квадрат
  }
}
```

```
public class Circle : Shape
{
  public void Draw()
  {
    // нарисовать круг
  }
}

public void DrawAllShapes(IList shapes)
{
  foreach(Shape shape in shapes)
    shape.Draw();
}
```

Если мы захотим расширить поведение метода `DrawAllShapes` в листинге 9.2, чтобы он умел рисовать еще один вид фигур, то достаточно будет добавить новый класс, производный от `Shape`. Сам метод `DrawAllShapes` изменять не придется. Поэтому `DrawAllShapes` удовлетворяет принципу OCP. Его поведение можно расширить без модификации исходного кода. Более того, добавление класса `Triangle` *вообще не сказывается* ни на одном из приведенных выше модулей. Понятно, что какие-то части системы все же придется изменить для включения класса `Triangle`, но весь представленный в листинге 9.2 код останется неприкосновенным.

В реальном приложении в классе `Shape` было бы гораздо больше методов. И все равно добавление новой фигуры не вызывает сложностей, потому что нужно лишь создать новый производный класс и реализовать все его методы. Не требуется проверять все приложение, выискивая места, требующие изменений. Это решение не хрупкое.

И не жесткое. Не пришлось ни модифицировать существующие исходные тексты, ни пересобирать имеющиеся двоичные модули – за одним исключением. Следует изменить модуль, который создает экземпляры нового класса, производного от `Shape`. Обычно это делается в методе `Main`, в каком-нибудь методе, вызываемом из `Main`, или в методе некоторого объекта, созданного в `Main`.[1]

Наконец, это решение не является неподвижным. Метод `DrawAllShapes` можно повторно использовать в любом приложении, не таская за собой классы `Square` или `Circle`. Таким образом, не наблюдается никаких вышеупомянутых признаков плохого дизайна.

Эта программа удовлетворяет принципу OCP. *Для ее модификации нужно написать новый код, а не изменить существующий*. При этом не возникнет каскада изменений, характерных для не следующих этому принципу программ. Единственно необходимые изменения – добавление нового модуля и поправки в `Main`, позволяющие создавать объекты нового типа.

---

[1] Такие объекты называются *фабриками*, и мы еще будем говорить о них в главе 29.

Но подумайте, что произойдет с методом `DrawAllShapes` из листинга 9.2, если мы потребуем, чтобы *все круги рисовались раньше всех квадратов*. Метод `DrawAllShapes` не закрыт от такого рода изменения. Чтобы его реализовать, нам придется переделать его так, чтобы на первом проходе из списка выбирались все объекты `Circle`, а на втором – все объекты `Square`.

Предвидение и «естественная» структура

Если бы мы предвидели такие изменения, то смогли бы придумать абстракцию, защищающую от них. Абстракции в листинге 9.2 – скорее помеха, а не подмога для реализации такого изменения. Тут вы, наверное, удивитесь: в самом деле, что может быть естественнее базового класса `Shape` с производными от него `Square` и `Circle`? Почему эта естественная, хорошо соотносящаяся с реальным миром модель не оптимальна? Да просто такая модель *не является* естественной в системе, где упорядоченность связана с типом фигуры.

И это приводит нас к печальному выводу. В общем случае, каким бы «закрытым» ни был модуль, всегда найдется такое изменение, от которого он не закрыт. *Не существует моделей, естественных во всех контекстах!*

Поскольку от всего закрыться нельзя, то нужно мыслить стратегически. Иными словами, проектировщик должен решить, от каких изменений закрыть дизайн: определить, какие изменения наиболее вероятны, а затем сконструировать абстракции, защищающие от них.

Для этого требуется способность к предвидению, которая приходит только с опытом. Опытный проектировщик надеется, что достаточно хорошо знает пользователей и индустрию, чтобы оценить вероятность тех или иных изменений. Затем он призывает на помощь ООП, чтобы защититься от наиболее вероятных изменений.

Это непросто. Необходимо строить обоснованные гипотезы о том, с каким изменениями приложение может столкнуться в будущем. Если проектировщик угадывает верно, он вправе торжествовать. Если нет, это провал. И разумеется, догадки не всегда бывают правильными.

К тому же следование принципу OCP обходится дорого. На создание подходящих абстракций уходят время и силы разработчиков. Абстракции увеличивают сложность дизайна программы. Существует предел количеству абстракций, которые могут позволить себе разработчики. Очевидно, что мы хотели бы ограничить применение OCP только вероятными изменениями.



Но как узнать, какие изменения вероятны? С помощью исследования, задавая правильные вопросы и призывая на помощь свой опыт и здра-

вый смысл. И после всего этого *мы ничего не предпринимаем, пока изменение не произойдет*!

Расстановка «точек подключения»

Как же защититься от изменений? В прошлом веке мы сказали бы: «Предусматривайте точки подключения (Hooks)» для предполагаемых изменений – и считали бы свою программу гибкой.

Однако предусмотренные точки подключения зачастую оказывались неправильными. Хуже того, они отличались ненужной сложностью, которую приходилось поддерживать и сопровождать, даже если точки и не использовались. Это плохо. Мы не хотим перегружать дизайн множеством излишних абстракций. Лучше подождать, когда возникнет необходимость в некоторой абстракции, и уж тогда включить ее.

Обманул меня раз. «Обманул меня раз – позор тебе, обманул другой – позор мне». Это очень правильный подход к разработке ПО. Чтобы не перегружать программу ненужной сложностью, мы можем *один раз* позволить себе обмануться. Это означает, что первоначально код пишется без учета возможных изменений. Если же изменение происходит, то мы реализуем абстракции, которые в будущем защитят от *такого рода* изменений. Короче говоря, *первую пулю мы принимаем*, но от последующих пуль из того же ружья защищаемся.

Стимулирование изменений. Но лучше бы, чтобы пули прилетали пораньше и почаще. Мы хотим знать, какие изменения вероятны, прежде чем продвинемся в разработке слишком далеко. Чем дольше мы будем ждать прояснения ситуации с изменениями, тем сложнее потом будет создавать подходящие абстракции.

Поэтому изменения следует стимулировать. И применяются для этого средства, обсуждавшиеся в главе 2.

- Сначала пишем тесты. Тестирование – один из видов использования системы. Кодируя тесты в самом начале, мы по необходимости создаем систему, поддающуюся тестированию. Поэтому никакие сюрпризы в плане изменения тестопригодности нам не грозят. Мы всегда будем создавать абстракции так, чтобы система оставалась пригодной для тестирования. Скорее всего, обнаружится, что такие абстракции защищают и от других изменений.
- Разработку ведем очень короткими циклами, измеряемыми в днях, а не в неделях.
- Разрабатываем содержательные функции до инфраструктуры и часто демонстрируем их заинтересованным сторонам.
- Сначала реализуем наиболее важные функции.
- Первую версию программы выпускаем быстро, а последующие часто. Предлагаем их заказчикам и пользователям настолько оперативно, насколько возможно.

Применение абстракции для явного закрытия

Ну хорошо, первую пулю мы получили. Пользователь хочет, чтобы круги рисовались раньше квадратов. Как защититься от подобных изменений в будущем?

Как можно закрыть функцию `DrawAllShapes` от изменений в порядке рисования? Напомним, закрытие основано на абстракции. Следовательно, чтобы закрыть `DrawAllShapes` от изменения порядка, необходима некая «абстракция порядка». Это был бы абстрактный интерфейс, с помощью которого можно выразить любую стратегию упорядочения.

Стратегия упорядочения означает, что для любых двух объектов можно узнать, какой из них рисовать первым. В C# такая абстракция уже есть – интерфейс `IComparable` с одним-единственным методом `CompareTo`. Этот метод принимает в качестве параметра объект и возвращает –1, если объект, от имени которого метод вызван, меньше параметра, 0 – если равен параметру, и 1 – если больше параметра.

В листинге 9.3 показано, как мог бы выглядеть тип `Shape`, наследующий интерфейсу `IComparable`.

Листинг 9.3. Тип Shape, наследующий интерфейсу IComparable

```
public interface Shape : IComparable
{
  void Draw();
}
```

Теперь, зная, как определить относительный порядок двух объектов `Shape`, мы можем отсортировать их и нарисовать в нужном порядке. В листинге 9.4 приведен соответствующий код на C#.

Листинг 9.4. Функция DrawAllShapes с поддержкой упорядочения

```
public void DrawAllShapes(ArrayList shapes)
{
  shapes.Sort();
  foreach(Shape shape in shapes)
    shape.Draw();
}
```

Это дает нам возможность упорядочивать объекты и рисовать их в требуемом порядке. Но пока у нас еще нет нужной абстракции порядка. В данном случае все конкретные подклассы `Shape` должны тем или иным способом переопределить метод `CompareTo`. Как это будет работать? Как должен выглядеть метод `Circle.CompareTo`, чтобы все круги рисовались раньше квадратов? Взгляните на листинг 9.5.

Листинг 9.5. Упорядочение объектов Circle

```
public class Circle : Shape
{
  public int CompareTo(object o)
```

```
    {
      if(o is Square)
        return -1;
      else
        return 0;
    }
  }
```

Должно быть очевидно, что эта функция, как и все ей подобные в других классах, производных от `Shape`, не удовлетворяет принципу OCP. Их никак не удастся закрыть относительно появления новых производных классов. Стоит появиться такому классу, как мы должны будем изменить все ранее написанные функции `CompareTo()`.[1]

Конечно, это не имеет значения, если новые производные от `Shape` никогда не будут создаваться. С другой стороны, если они создаются часто, то такой дизайн приведет к повторению одной и той же работы. Первая пуля прилетела снова.

Закрытие за счет применения таблицы данных

Если необходимо, чтобы производные от `Shape` классы не знали друг о друге, то можно завести таблицу. Один такой вариант решения приведен в листинге 9.6.

Листинг 9.6. Механизм упорядочения типов с помощью таблицы

```
/// <summary>
/// Этот сравниватель просматривает хеш-таблицу типов фигур. В ней
/// хранятся приоритеты, определяющие относительный порядок фигур.
/// Отсутствующие фигуры предшествуют найденным.
/// </summary>
public class ShapeComparer : IComparer
{
  private static Hashtable priorities = new Hashtable();
  static ShapeComparer()
  {
    priorities.Add(typeof(Circle), 1);
    priorities.Add(typeof(Square), 2);
  }

  private int PriorityFor(Type type)
  {
    if(priorities.Contains(type))
      return (int)priorities[type];
    else
      return 0;
```

[1] Эту проблему можно решить с помощью паттерна Ациклический посетитель (Acyclic Visitor), описанного в главе 35. Но приводить это решение сейчас означало бы забегать вперед. В главе 35 я напомню, чтобы вы вернулись к этому месту.

```
    }

    public int Compare(object o1, object o2)
    {
      int priority1 = PriorityFor(o1.GetType());
      int priority2 = PriorityFor(o2.GetType());
      return priority1.CompareTo(priority2);
    }
  }

  public void DrawAllShapes(ArrayList shapes)
  {
    shapes.Sort(new ShapeComparer());
    foreach(Shape shape in shapes)
      shape.Draw();
  }
```

Этот подход позволил нам закрыть саму функцию `DrawAllShapes` от проблем упорядочения, а подклассы `Shape` – от создания новых подклассов и от изменения стратегии упорядочения объектов `Shape` по типу (например, такого изменения порядка, чтобы сначала рисовались квадраты).

Единственный элемент, не закрытый относительно порядка объектов `Shape`, – сама таблица. Но эту таблицу можно поместить в отдельный модуль, так что ее изменение не будет сказываться на других модулях.

Заключение

Во многих отношениях принцип открытости/закрытости – основа основ объектно-ориентированного проектирования. Именно следование этому принципу позволяет получить от ООП максимум обещанного: гибкость, возможность повторного использования и удобство сопровождения. Но чтобы удовлетворить ему, недостаточно просто использовать какой-нибудь ОО язык программирования. И бездумно применять абстракции ко всем вообще частям приложения тоже не стоит. Надо, чтобы разработчики осознанно применяли абстракции только к тем фрагментам программы, которые часто изменяются. *Отказ от преждевременного абстрагирования столь же важен, как и само абстрагирование.*

Библиография

[Jacobson92] Ivar Jacobson, Patrick Johnsson, Magnus Christerson, and Gunnar Övergaard «Object-Oriented Software Engineering: A Use Case Driven Approach», Addison-Wesley, 1992.

[Meyer97] Bertrand Meyer «Object Oriented Software Construction», 2-d ed., Prentice Hall, 1997.[1]

[1] Бертран Мейер «Объектно-ориентированное конструирование программных систем». – Пер. с англ. – Русская редакция, 2005.

10

Принцип подстановки Лисков (LSP)



Механизмы, лежащие в основе принципа открытости/закрытости, – абстрагирование и полиморфизм. В статически типизированных языках, к числу которых относится и C#, один из главных механизмов поддержки абстрагирования и полиморфизма – наследование. Именно наследование позволяет нам создавать производные классы, реализующие абстрактные методы, объявленные в базовых классах.

Какими правилами проектирования мы руководствуемся, когда решаем, как воспользоваться наследованием? Каковы характеристики наилучших иерархий наследования? Какие подводные камни приводят к иерархиям, не удовлетворяющим принципу OCP? На эти вопросы отвечает принцип подстановки Лисков (LSP).

Принцип подстановки Лисков (Liskov Substitution Principle).

Должна быть возможность вместо базового типа подставить любой его подтип.

Этот принцип был сформулирован Барбарой Лисков в 1988 году.[1] Она писала:

> Мы хотели бы иметь следующее свойство подстановки: если для каждого объекта o_1 типа S существует объект o_2 типа T, такой, что для любой программы P, определенной в терминах T, поведение P не изменяется при замене o_1 на o_2, то S является подтипом T.

Важность этого принципа становится очевидной, если рассмотреть последствия его нарушения. Предположим, что имеется функция *f*, принимающая в качестве аргумента ссылку на некоторый базовый класс *B*. Предположим также, что при передаче функции *f* ссылки на объект класса *D*, производного от *B*, она ведет себя неправильно. Тогда *D* нарушает принцип LSP. Понятно, что класс *D* оказывается хрупким в присутствии *f*.

У авторов *f* может возникнуть искушение включить некоторый тест для *D*, проверяющий, что *f* правильно ведет себя при передаче ей *D*. Но такой тест нарушает принцип *OCP*, потому что *f* теперь не закрыта от других классов, производных от *B*. Подобные тесты демонстрируют неопытность разработчиков или, что еще хуже, чрезмерно поспешную реакцию на нарушение LSP.

Нарушения принципа LSP

Простой пример

Нарушение принципа LSP часто влечет за собой использование проверки типов во время выполнения способом, который находится в вопиющем противоречии с принципом открытости/закрытости. В таком случае мы видим предложения `if` или `if/else`, служащие для того, чтобы выбрать подходящее поведение в зависимости от типа объекта. Рассмотрим листинг 10.1.

Листинг 10.1. Нарушение LSP, приводящее к отходу от OCP

```
struct Point {double x, y;}

public enum ShapeType {square, circle};

public class Shape
```

[1] [Liskov88]

```
{
  private ShapeType type;
  public Shape(ShapeType t){type = t;}
  public static void DrawShape(Shape s)
  {
    if(s.type == ShapeType.square)
      (s as Square).Draw();
    else if(s.type == ShapeType.circle)
      (s as Circle).Draw();
  }
}

public class Circle : Shape
{
  private Point center;
  private double radius;

  public Circle() : base(ShapeType.circle) {}
  public void Draw() {/* рисует круг */}
}

public class Square : Shape
{
  private Point topLeft;
  private double side;

  public Square() : base(ShapeType.square) {}
  public void Draw() {/* рисует квадрат */}
}
```

Очевидно, что метод `DrawShape` в листинге 10.1 нарушает принцип OCP. Он должен знать обо всех классах, производных от `Shape`, и при каждом появлении нового производного класса его придется изменять. И будет совершенно справедливо предать такой метод анафеме. Но что могло бы побудить программиста написать нечто подобное?

Рассмотрим инженера Васю. Он изучал объектно-ориентированные технологии и пришел к выводу, что затраты на полиморфизм слишком велики.[1] Поэтому он определил класс `Shape` без абстрактных методов. Классы `Square` и `Circle` являются производными от `Shape`, и в них есть методы `Draw()`, но они не переопределяют одноименный метод в классе `Shape`. Так как `Circle` и `Square` невозможно подставить вместо `Shape`, то `DrawShape` должна определять тип переданного ей параметра и вызывать подходящий метод `Draw`.

Тот факт, что `Square` и `Circle` нельзя подставить вместо `Shape`, – нарушение принципа LSP. И вызвано оно отходом от принципа OCP в методе `DrawShape`. Таким образом, *нарушение LSP – это латентное нарушение OCP.*

[1] На относительно быстрой машине затраты составляют примерно 1 нс на каждый вызов метода, так что согласиться с Васиной точкой зрения трудно.

Более тонкое нарушение

Разумеется, есть и другие, куда более тонкие, виды нарушения LSP. Рассмотрим приложение, в котором используется класс `Rectangle`, показанный в листинге 10.2.

Листинг 10.2. Класс Rectangle

```
public class Rectangle
{
  private Point topLeft;
  private double width;
  private double height;

  public double Width
  {
    get { return width; }
    set { width = value; }
  }

  public double Height
  {
    get { return height; }
   set { height = value; }
  }
}
```

Представьте себе, что это приложение нормально работает и установлено на многих площадках. Но, как всегда бывает с удачными программами, пользователи время от времени просят внести изменения. И в один прекрасный день пользователь возжелал, чтобы программа могла манипулировать не только прямоугольниками, но и *квадратами*.

Часто говорят, что наследование – это отношение *ЯВЛЯЕТСЯ* (*IS-A*). Иными словами, если про новый вид объекта можно сказать, что он ЯВЛЯЕТСЯ частным случаем некоего старого вида, то класс нового объекта должен быть производным от класса старого объекта.

Вроде бы квадрат во всех отношениях *является* прямоугольником. Поэтому логично считать класс `Square` производным от `Rectangle` (рис. 10.1).

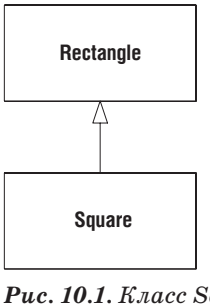


***Рис. 10.1.** Класс Square наследует Rectangle*

Такое использование отношения ЯВЛЯЕТСЯ иногда считают одним из основных приемов объектно-ориентированного анализа (термин, который часто применяют, но редко определяют). Квадрат является прямоугольником, поэтому класс `Square` должен быть производным от класса `Rectangle`. Однако такое рассуждение может привести к тонким, но весьма существенным проблемам, которые невозможно предвидеть, пока не столкнешься с ними в программе.

Первый признак, наводящий на мысль, что тут не все в порядке, – тот факт, что классу `Square` не нужны оба поля `height` и `width`. Однако же он наследует их от `Rectangle`. Понятно, что это расточительство. Во многих случаях таким расточительством можно пренебречь. Но если создаются сотни тысяч объектов `Square`, как, например, в САПР, где каждый контакт компонента изображается квадратиком, то оно становится существенным.

Ладно, предположим на минуточку, что эффективное использование памяти нас не очень волнует. Но у наследования `Square` от `Rectangle` есть и другие проблемы. Класс `Square` наследует свойства записи `Width` и `Height`. Однако это неправильно, потому что ширина и высота квадрата одинаковы. И это уже несомненный признак проблемы. Впрочем, ее можно обойти, переопределив свойства `Width` и `Height` следующим образом:

```
public new double Width
{
  set
  {
    base.Width = value;
    base.Height = value;
  }
}

public new double Height
{
  set
  {
    base.Height = value;
    base.Width = value;
  }
}
```

Теперь при любой попытке изменить ширину объекта `Square` высота тоже изменится. И то же самое относится к высоте. Следовательно, инварианты – условия, которые должны оставаться истинными в любом состоянии, – класса `Square` соблюдаются. Объект `Square` всегда остается математически правильным квадратом:

```
Square s = new Square();
s.SetWidth = 1; // К счастью, делает высоту тоже равной 1
s.SetHeight = 2; // Делает высоту и ширину равными 2. Хорошо.
```

Однако рассмотрим следующую функцию:

```
void f(Rectangle r)
{
  r.SetWidth = 32; // вызывает Rectangle.SetWidth
}
```

Если передать этой функции ссылку на объект `Square`, то он будет искажен, потому что высота не изменится. Это очевидное нарушение принципа LSP. Функция `f` не работает, когда в качестве аргумента передан объект производного класса. Причина в том, что свойства `Width` и `Height` не были объявлены виртуальными и поэтому не являются полиморфными.

Это легко исправить, добавив в объявление свойств записи ключевое слово `virtual`. Однако если при создании производного класса мы вынуждены вносить изменение в базовый класс, значит, в дизайне, скорее всего, есть изъян. Конечно же, он нарушает принцип OCP. На это можно возразить, указав, что изъян как раз в том, что мы с самого начала забыли сделать свойства `Width` и `Height` виртуальными, а теперь просто исправляем собственную ошибку. Но такое объяснение трудно принять, потому что задание высоты и ширины прямоугольника – самые примитивные операции. Какой ход мысли мог бы подсказать, что они должны быть виртуальными, если мы не предвидели необходимость класса `Square`?

Но все же предположим, что мы согласились с этим доводом и исправили классы. Тогда получится код, показанный в листинге 10.3.

Листинг 10.3. Согласованные классы Rectangle и Square

```
public class Rectangle
{
  private Point topLeft;
  private double width;
  private double height;

  public virtual double Width
  {
    get { return width; }
    set { width = value; }
  }

  public virtual double Height
  {
    get { return height; }
    set { height = value; }
  }
}

public class Square : Rectangle
{
```

```
  public override double Width
  {
    set
    {
      base.Width = value;
      base.Height = value;
    }
  }

  public override double Height
  {
    set
    {
      base.Height = value;
      base.Width = value;
    }
  }
}
```

Настоящая проблема. Классы `Square` и `Rectangle`, похоже, работают. Что бы ни делалось с объектом `Square`, он останется математически строгим квадратом. А что бы ни делалось с объектом `Rectangle`, он останется прямоугольником. Можно даже передать объект `Square` функции, которая принимает `Rectangle`, и `Square` все равно будет вести себя, как квадрат, не порождая никаких противоречий.

Стало быть, можно заключить, что теперь дизайн внутренне непротиворечив и правилен, да? Увы, такое заключение было бы чересчур поспешным. Внутренне непротиворечивый дизайн необязательно совместим с потребностями всех пользователей! Рассмотрим такую функцию `g`:

```
void g(Rectangle r)
{
  r.Width = 5;
  r.Height = 4;
  if(r.Area() != 20)
    throw new Exception("Неправильная площадь!");
}
```

Она вызывает свойства-члены `Width` и `Height` объекта, который считает экземпляром класса `Rectangle`. Функция прекрасно работает для `Rectangle`, но возбуждает исключение `Exception`, если ей передать объект `Square`. Вот где настоящая проблема: *автор* `g` *предполагал, что при изменении ширины* `Rectangle` *высота остается неизменной*.

Очевидно, предположение о том, что изменение ширины прямоугольника не приводит к изменению высоты, вполне разумно! Однако же не все объекты, которые можно передать как `Rectangle`, удовлетворяют этому предположению. Если передать экземпляр `Square` функции, подобной `g`, автор которой считал это предположение истинным, то функция будет вести себя некорректно. Функция *g* оказывается хрупкой относительно иерархии `Square`/`Rectangle`.



Пример функции g показывает, что существуют функции, принимающие в качестве аргументов объекты Rectangle, но работающие неправильно для объектов Square. Поскольку для таких функций невозможно подставить Square вместо Rectangle, то отношение, связывающее классы Square и Rectangle, не удовлетворяет принципу LSP.

Можно было бы возразить, что проблема заключается в самой функции g, что ее автор не имел права предполагать независимость ширины и высоты. Автор g с этим не согласился бы. Функция g принимает в качестве аргумента Rectangle. Существует ряд инвариантов, истинных высказываний, с очевидностью применимых к классу с именем Rectangle, и один из таких инвариантов гласит, что высота и ширина прямоугольника независимы. Автор g имел полное право предполагать наличие этого инварианта. Нарушил же его автор класса Square.

Интересно отметить, что автор Square не нарушал инвариант класса Square. Но, произведя Square от Rectangle, он нарушил инвариант Rectangle!

Правильность не является внутренне присущим свойством. Из принципа подстановки Лисков вытекает одно чрезвычайно важное следствие: *невозможно установить правильность модели, рассматриваемой изолированно.* Правильность модели можно выразить только в терминах ее клиентов. Например, рассматривая окончательную версию классов Square и Rectangle, взятых в изоляции от всего остального, мы решили, что они внутренне непротиворечивы и правильны. Но, взглянув на них с точки зрения программиста, сделавшего разумные предположения о базовом классе, мы обнаружили, что модель не годится.

Обдумывая вопрос о том, подходит ли конкретный дизайн, нельзя рассматривать решение в изоляции. Необходимо смотреть на него через призму разумных предположений со стороны пользователей данного дизайна.[1]

Откуда нам знать, какие предположения пользователей нашего дизайна разумны? Как правило, предвидеть их нелегко. Действительно, попытайся мы предвидеть все такие предположения, получилась бы система, обладающая ненужной сложностью. Поэтому, как и в случае всех остальных принципов, лучше предотвратить только самые очевидные случаи нарушения LSP, отложив остальные до того момента, как проявится хрупкость дизайна.

[1] Часто такие предположения выражаются в виде утверждений в автономных тестах, написанных для базового класса. И это еще один довод в пользу разработки через тестирование.

Отношение ЯВЛЯЕТСЯ – часть поведения. Так что же произошло? Почему разумная на первый взгляд модель, состоящая из классов `Square` и `Rectangle`, оказалась плохой? Разве квадрат – это не прямоугольник? Разве отношение ЯВЛЯЕТСЯ не имеет места?

С точки зрения автора функции `g` – нет! Квадрат может быть прямоугольником, но с точки зрения функции `g` объект `Square` определенно *не* является объектом `Rectangle`. Почему? Потому, что *поведение* объекта `Square` несовместимо с предположениями о поведении объекта `Rectangle`. С точки зрения поведения `Square` не есть `Rectangle`, а именно *поведение* и интересует любую программу. Принцип подстановки Лисков ясно показывает, что в объектно-ориентированном проектировании отношение ЯВЛЯЕТСЯ относится к *поведению*, относительно которого клиенты могут строить разумные предположения.

Проектирование по контракту. Многим разработчикам вряд ли придется по душе идея о поведении, относительно которого можно строить «разумные предположения». Как узнать, какие предположения клиентов разумны? Существует методика, позволяющая выразить разумные предположения явно и тем самым удовлетворить принципу LSP. Она называется *проектированием по контракту* (Design by Contract – DBC) и изложена Бертраном Мейером.[1]

Смысл DBC состоит в том, что автор класса явно формулирует контракт для данного класса. Контракт информирует авторов клиентского кода о том, на какое поведение можно полагаться. Спецификация контракта состоит из предусловий и постусловий для каждого метода. Чтобы метод мог начать выполнение, должны выполняться некоторые предусловия. По завершении метод гарантирует, что будут выполнены постусловия.

Постусловие для свойства записи `Rectangle.Width` можно сформулировать так:

```
assert((width == w) && (height == old.height));
```

где `old` – значение `Rectangle` перед установкой `Width`. Сформулированное Мейером правило пред- и постусловий для производных классов гласит: «Переопределенный (в производном классе) метод вправе заменить исходное предусловие эквивалентным или более слабым, а исходное постусловие – эквивалентным или более сильным».[2]

Иными словами, используя объект через интерфейс его базового класса, пользователь знает лишь о пред- и постусловиях базового класса. Следовательно, объекты производных классов не могут ожидать, что их клиенты будут подчиняться предусловиям, более сильным, чем того требует базовый класс. А значит, клиент должен принять все то, что готов принять базовый класс. Кроме того, производные классы обязаны гарантировать выполнение всех постусловий базового класса, то есть

[1] [Meyer97], стр. 331

[2] [Meyer97], стр. 573

их поведение и результат работы не должны нарушать ограничения, наложенные на базовый класс. Результаты работы производного класса не должны быть сюрпризом для клиентов базового класса.

Очевидно, что постусловие свойства записи `Square.Width` слабее[1] постусловия свойства `Rectangle.Width`, потому что не гарантируется соблюдение ограничения (`height == old.height`). Поэтому свойство `Width` класса `Square` нарушает контракт базового класса.

В некоторых языках, например Eiffel, имеется прямая поддержка пред- и постусловий. Их можно объявить, а исполняющая система будет их проверять. В C# такой возможности нет. Мы должны самостоятельно рассмотреть пред- и постусловия каждого метода и удостовериться в том, что правило Мейера не нарушено. Очень полезно документировать пред- и постусловия в виде комментариев к каждому методу.

Специфицирование контрактов в автономных тестах. Контракты можно также специфицировать, составляя автономные тесты. Тщательно проверяя поведение класса, автономные тесты делают это поведение явно выраженным. Авторы клиентского кода могут заглянуть в автономные тесты и узнать, какие предположения об используемых классах разумны.

Реальный пример

Но довольно квадратов и прямоугольников! Находит ли принцип LSP применение в реальных программах? Рассмотрим пример, взятый из проекта, над которым я работал несколько лет назад.

Мотивировка. В начале 1990-х годов я купил разработанную сторонней фирмой библиотеку, содержавшую несколько контейнерных классов.[2] Контейнеры примерно соответствовали классам `Bag` и `Set` в языке Smalltalk. Существовало по два варианта каждого класса. Первый назывался *ограниченным* и был основан на использовании массива. Второй назывался *неограниченным*, и в его основе лежал связанный список.

В конструкторе `BoundedSet` задавалось максимальное число элементов множества. Память для них заранее выделялась в виде массива и являлась частью объекта `BoundedSet`. Таким образом, в случае успешного создания объекта `BoundedSet` имелась полная уверенность в том, что памяти достаточно. Будучи основан на использовании массива, этот класс работал очень быстро. При выполнении стандартных операций память не выделялась. А поскольку память была распределена заранее, то работа с `BoundedSet` не могла привести к исчерпанию кучи. Однако имел

---

[1] Термин *слабее* может вызвать путаницу. Говорят, что *X* слабее *Y*, если *X* не соблюдает все ограничения *Y*. Не имеет значения, сколько новых ограничений вводит *X*.

[2] Она была написана на языке C++ задолго до появления контейнеров в стандартной библиотеке.

место непроизводительный расход памяти, поскольку необходимость во всей предварительно выделенной памяти возникала редко.

С другой стороны, класс `UnboundedSet` не налагал предварительных ограничений на число элементов в контейнере. Пока в куче оставалась доступная память, `UnboundedSet` был готов принимать новые элементы. Таким образом, этот класс был весьма гибким и одновременно экономичным, потому что потреблял лишь столько памяти, сколько было необходимо для хранения помещенных в него элементов. Но при этом он работал медленно, так как ему приходилось выделять и освобождать память. И конечно, была опасность исчерпать всю кучу.

Интерфейсы этих классов меня не устраивали. Я не желал, чтобы код моего приложения зависел от них, поскольку полагал, что впоследствии захочу заменить их классами получше. Поэтому я обернул контейнеры собственным абстрактным интерфейсом, показанным на рис. 10.2.

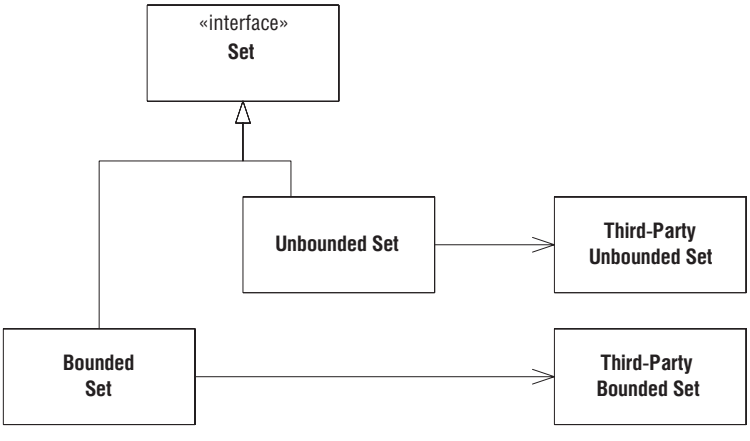


***Рис. 10.2.** Адаптер контейнерных классов*

В моем интерфейсе `Set` были абстрактные функции `Add`, `Delete` и `IsMember`, представленные в листинге 10.41. Тем самым ограниченная и неограниченная версии множества были унифицированы и доступ к ним производился через общий интерфейс. Это означало, что клиент мог принять аргумент типа `Set`, не заботясь о том, работает ли он с ограниченным или неограниченным контейнером. (См. функцию `PrintSet` в листинге 10.5.)

Листинг 10.4. Абстрактный класс Set

```
public interface Set
{
  public void Add(object o);
```

[1] Оригинальный код переведен на C#, чтобы его было проще понять программирующим на платформе .NET.

```
  public void Delete(object o);
  public bool IsMember(object o);
}
```

Листинг 10.5. Функция PrintSet

```
void PrintSet(Set s)
{
  foreach(object o in s)
    Console.WriteLine(o.ToString());
}
```

Не знать и не думать о том, с какой разновидностью `Set` мы работаем, очень удобно. Благодаря этому программист может решить, какой вариант `Set` больше подходит в каждом конкретном случае, а на клиентских функциях это никак не отразится. Если память дефицитна, а быстродействие не критично, то можно взять `UnboundedSet`, если же памяти много, но важно быстродействие, то `BoundedSet`. Клиентские функции будут манипулировать этими объектами через интерфейс базового класса `Set`, не зная и не интересуясь тем, с каким именно `Set` они работают.

Проблема. Я захотел добавить в эту иерархию класс `PersistentSet`. Сохраняемое множество можно было бы записать в поток, а впоследствии прочитать – возможно, в другом приложении. К несчастью, единственный предлагавший сохранение контейнерный класс, написанный третьей фирмой, к которому у меня был доступ, не годился. Он принимал лишь объекты, производные от абстрактного базового класса `PersistentObject`. Я создал иерархию, показанную на рис. 10.3.

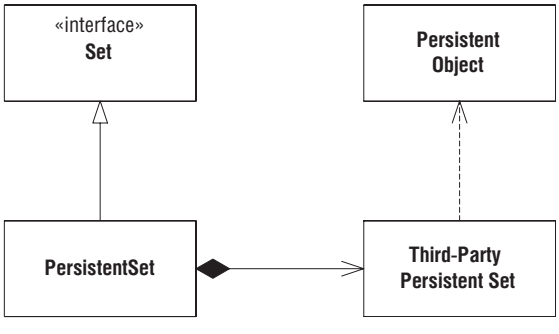


***Рис. 10.3.** Иерархия класса PersistentSet*

Обратите внимание, что `PersistentSet` содержит экземпляр стороннего класса сохраняемого множества, которому он делегирует все свои методы. Таким образом, вызов метода `Add` объекта `PersistentSet` просто делегируется подходящему методу внутреннего класса.

На первый взгляд все выглядит прекрасно. Однако имеется крайне неприятное обстоятельство. Объекты, добавляемые в сторонний контей-

нер, должны принадлежать классу, производному от PersistentObject. Так как класс PersistentSet просто делегирует работу стороннему классу, то любой объект, добавляемый в PersistentSet, тоже должен быть производным от PersistentObject. Однако же в интерфейсе класса Set такого ограничения нет.

Клиент, добавляющий объекты в экземпляр базового класса Set, не в курсе, работает он с Set или с PersistentSet. Следовательно, у клиента нет никакой возможности узнать, что добавляемые объекты должны быть производными от PersistentObject.

Рассмотрим код метода PersistentSet.Add() в листинге 10.6. Из него видно, что при попытке добавить в мой контейнер PersistentSet объект класса, не являющегося производным от PersistentObject, произойдет ошибка времени выполнения. Операция приведения типа возбудит исключение. Но ни один из существующих клиентов абстрактного базового класса Set не ожидает, что метод Add возбуждает исключения. Поскольку использование производного от Set класса может стать источником сюрпризов, то такое изменение иерархии нарушает принцип LSP.

Листинг 10.6. Метод Add класса PersistentSet

```
void Add(object o)
{
  PersistentObject p = (PersistentObject)o;
  thirdPartyPersistentSet.Add(p);
}
```

Следует ли считать это проблемой? Безусловно. Функции, которые раньше работали нормально, когда им передавали объекты, производные от Set, будут завершаться с ошибкой при получении объекта PersistentSet. Отлаживать программу с такими ошибками довольно трудно, поскольку они не являются логическими. Логический изъян появился тогда, когда принималось решение о передаче функции объекта PersistentSet или о добавлении в PersistentSet объекта, не являющегося производным от PersistentObject. В обоих случаях решение может отстоять на миллионы команд от вызова метода Add. Поиск ошибки может стать сложной задачей, а исправление – тяжким испытанием.

Решение, не удовлетворяющее принципу LSP. Как разрешить эту проблему? Несколько лет назад я решил ее, просто приняв некое соглашение, то есть на уровне исходного кода не сделал ничего. Соглашение состояло в том, чтобы скрыть классы PersistentSet и PersistentObject от приложения. Они были известны только одному модулю, который отвечал за чтение и запись всех контейнеров из постоянного хранилища. Когда контейнер нужно было записать, его содержимое копировалось в подходящие объекты, производные от PersistentObject, затем они добавлялись в экземпляр PersistentSet, который и сохранялся в потоке. Если нужно было прочитать контейнер из потока, то выполнялась об-

ратная процедура: объект PersistentSet считывался из потока, затем программа получала из него объекты PersistentObject по одному и копировала их в обычные, не сохраняемые объекты, которые помещались в обычный контейнер Set.

Такое решение может показаться чрезмерно ограничительным, но только так я сумел предотвратить появление объектов PersistentSet в интерфейсе функций, которые могли бы поместить в них не сохраняемые объекты. Заодно оно разрывало зависимость остального приложения от самой концепции сохранения.

Работало ли это решение? На самом деле нет. Принятое соглашение в нескольких частях приложения было нарушено программистами, не понимавшими его необходимости. Это проблема любых соглашений: их надо доводить до сведения каждого нового разработчика. Если разработчик не знает о соглашении или не принимает его, то соглашение будет нарушено. А одно-единственное нарушение может поставить под угрозу всю конструкцию.

Решение, удовлетворяющее принципу LSP. Как бы я разрешил эту проблему сейчас? Я бы признал, что класс PersistentSet не связан отношением ЯВЛЯЕТСЯ с классом Set, что он не является по-настоящему производным от Set. Раз так, то я разделил бы иерархии, но не полностью. У классов Set и PersistentSet есть общие особенности. На самом деле сложности с принципом LSP вызывает только метод Add. Поэтому я создал бы иерархию, в которой Set и PersistentSet находились бы на одном уровне и наследовали общему интерфейсу, допускающему проверку вхождения, обход и т. д. (см. рис. 10.4). Тем самым мы сумели бы обойти контейнер PersistentSet, проверить, есть ли в нем конкретный элемент и т. п., но не смогли бы добавить в PersistentSet объект класса, не являющегося производным от PersistentObject.

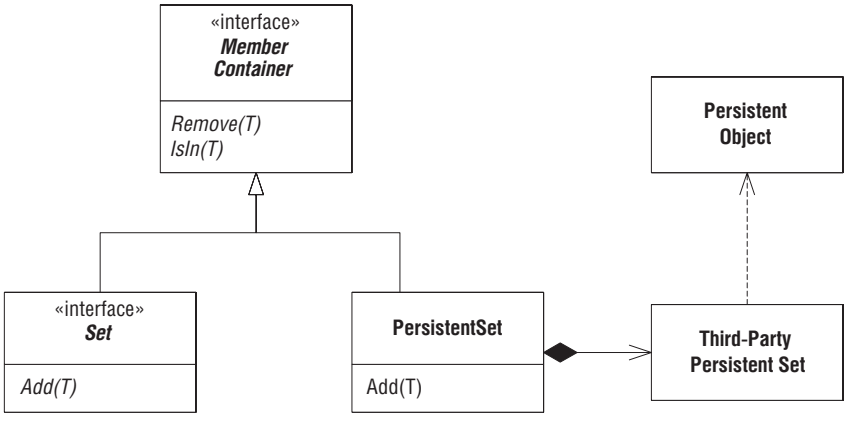


***Рис. 10.4.** Решение, удовлетворяющее принципу LSP*

Факторизация вместо наследования

Еще один интересный и озадачивающий случай наследования – классы `Line` (прямая) и `LineSegment` (отрезок прямой).[1] Взгляните на листинги 10.7 и 10.8. На первый взгляд оба класса – естественные кандидаты на наследование. Классу `LineSegment` необходимы все без исключения методы, объявленные в классе `Line`. А кроме того, `LineSegment` добавляет собственное свойство `Length` и переопределяет метод `IsOn`. И тем не менее эти классы нарушают принцип LSP, хотя и неочевидным образом.

Листинг 10.7. Line.cs

```
public class Line
{
  private Point p1;
  private Point p2;

  public Line(Point p1, Point p2){this.p1=p1; this.p2=p2;}

  public Point P1 { get { return p1; } }
  public Point P2 { get { return p2; } }
  public double Slope { get {/* код */} }
  public double YIntercept { get {/* код */} }
  public virtual bool IsOn(Point p) {/* код */}
}
```

Листинг 10.8. LineSegment.cs

```
public class LineSegment : Line
{
  public LineSegment(Point p1, Point p2) : base(p1, p2) {}
  public double Length { get {/* код */} }
  public override bool IsOn(Point p) {/* код */}
}
```

Пользователь класса `Line` вправе ожидать, что все точки, коллинеарные прямой, принадлежат ей. Например, свойство `YIntercept` возвращает точку, в которой прямая пересекается с осью *Y*. Поскольку эта точка лежит на прямой, то естественно предположить, что `IsOn(YIntercept) == true`. Однако для объектов класса `LineSegment` это утверждение верно далеко не всегда.

Почему это так важно? Почему бы просто не произвести `LineSegment` от `Line` и не смириться с наличием мелкой проблемы? Это зависит от ситуации. Существуют *редкие* случаи, когда целесообразно принять такой незначительный отход от полиморфного поведения и не пытаться привести дизайн в полное соответствие с принципом LSP. Согласиться на компромисс, не пытаясь достичь идеала, – распространенная инженер-

[1] Этот пример, хоть и похож на `Square/Rectangle`, взят из реального приложения и стал предметом обсуждения реальных проблем.

ная практика. Хороший инженер на опыте познает, когда компромисс *выгоднее* совершенства. Однако *бездумно отказываться от принципа LSP не стоит*. Гарантия того, что подкласс будет работать всюду, где работает базовый класс, – очень действенный способ управления сложностью. Если мы отказываемся от нее, то должны рассматривать каждый подкласс индивидуально.

В случае `Line` и `LineSegment` есть простое решение, иллюстрирующее один важный инструмент объектно-ориентированного проектирования. Если у нас имеется доступ к исходному коду обоих классов, то можно произвести *факторизацию*, то есть вынести их общие элементы в абстрактный базовый класс. В листингах 10.9, 10.10 и 10.11 показано, как факторизовать классы `Line` и `LineSegment`, образовав базовый класс `LinearObject`.

Листинг 10.9. LinearObject.cs

```
public abstract class LinearObject
{
  private Point p1;
  private Point p2;
  public LinearObject(Point p1, Point p2)
  {this.p1=p1; this.p2=p2;}

  public Point P1 { get { return p1; } }
  public Point P2 { get { return p2; } }
  public double Slope { get {/* код */} }
  public double YIntercept { get {/* код */} }
  public virtual bool IsOn(Point p) {/* код */}
}
```

Листинг 10.10. Line.cs

```
public class Line : LinearObject
{
  public Line(Point p1, Point p2) : base(p1, p2) {}
  public override bool IsOn(Point p) {/* код */}
}
```

Листинг 10.11. LineSegment.cs

```
public class LineSegment : LinearObject
{
  public LineSegment(Point p1, Point p2) : base(p1, p2) {}

  public double GetLength() { get {/* код */}}
  public override bool IsOn(Point p) {/* код */}
}
```

Представляя `Line` и `LineSegment`, класс `LinearObject` предоставляет почти всю функциональность и данные-члены обоих подклассов, за исключением метода `IsOn`, который является абстрактным. Клиенты `LinearObject` не вправе делать никаких предположений о протяженности используемого объекта. Тогда они смогут спокойно принимать экземпляры как

Line, так и LineSegment. Более того, клиенты Line никогда не будут иметь дело с LineSegment.

Факторизация – мощный инструмент. Если свойства некоего класса можно распределить по двум подклассам, то вполне вероятно, что в будущем появятся и другие классы, нуждающиеся в этих свойствах. Ребекка Вирфс-Брок, Брайан Уилкерсон и Лоран Винер говорят о факторизации следующее:

> Мы можем утверждать, что если все классы из некоторого набора поддерживают общую обязанность, то они должны наследовать ее от общего суперкласса.
>
> Если общий суперкласс еще не существует, создайте его и перенесите туда общие обязанности. Ведь полезность такого класса уже доказана – вы только что продемонстрировали, что его обязанности будут унаследованы какими-то классами. Не будет ли разумно предположить, что при последующем расширении системы появятся и другие подклассы, поддерживающие те же обязанности каким-то своим способом? Этот новый суперкласс, скорее всего, будет абстрактным.[1]

В листинге 10.12 показано, как атрибуты класса LinearObject можно использовать в ранее не предполагавшемся классе Ray (Луч). Объект Ray можно подставить вместо LinearObject, и все клиенты LinearObject смогут использовать его без каких бы то ни было проблем.

Листинг 10.12. Ray.cs

```
public class Ray : LinearObject
{
  public Ray(Point p1, Point p2) : base(p1, p2) {/* код */}
  public override bool IsOn(Point p) {/* код */}
}
```

Эвристика и соглашения

Существуют простые эвристические правила, способные подсказать, когда имеет место нарушение принципа LSP. Все они касаются производных классов, которые тем или иным способом *исключают* функциональность из базового класса. Производный класс, умеющий делать меньше, чем базовый, обычно нельзя подставить вместо базового, и потому он нарушает LSP.

Взгляните на листинг 10.13. Метод f класса Base реализован в классе Derived, но является в нем вырожденным. Быть может, автор Derived счел, что метод f в нем бесполезен. К несчастью, пользователи Base не знают, что не должны вызывать f, поэтому принцип подстановки нарушен.

[1] [Wirfs-Brock90], стр. 113

Листинг 10.13. Вырожденный метод в производном классе

```
public class Base
{
  public virtual void f() {/* какой-то код */}
}

public class Derived : Base
{
  public override void f() {}
}
```

Присутствие вырожденных методов в производных классах не всегда свидетельствует о нарушении LSP, но присмотреться к ним внимательнее стоит в любом случае.

Заключение

Принцип открытости/закрытости лежит в основе многих требований объектно-ориентированного проектирования. Если этот принцип соблюден, то приложение более надежно, лучше поддается сопровождению и пригодно для повторного использования. Принцип подстановки Лисков – один из основных инструментов реализации принципа OCP. Возможность подстановки подтипов позволяет без модификации расширять модуль, выраженный в терминах базового типа. И это то, на что разработчики вправе рассчитывать по умолчанию. Поэтому контракт базового типа должен быть хорошо и ясно понятен, если не явно навязан, из кода.

Термин ЯВЛЯЕТСЯ слишком широк, чтобы служить определением подтипа. Правильное определение подтипа – *заместим*, где заместимость определяется явным или неявным контрактом.

Библиография

[Liskov88] «Data Abstraction and Hierarchy», Barbara Liskov, *SIGPLAN Notices*, 23(5) (May 1988).

[Meyer97] Bertrand Meyer, «Object-Oriented Software Construction», 2-d ed., Prentice Hall, 1997.[1]

[Wirfs-Brock90] Rebecca Wirfs-Brock et al., «Designing Object-Oriented Software», Prentice Hall, 1990.

[1] Бертран Мейер «Объектно-ориентированное конструирование программных систем». – Пер. с англ. – Русская редакция, 2005.

11

Принцип инверсии зависимости (DIP)



© Jennifer M. Kohnke

И пусть никогда более высокие интересы Государства
не зависят от тысячи случайностей,
способных поколебать слабого человека.

Сэр Томас Нун Тэлфорд (1795–1854)

Принцип инверсии зависимости (Dependency-Inversion Principle – DIP)

A. *Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.*

B. *Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.*

На протяжении многих лет разные люди спрашивали, почему я употребляю в названии этого принципа слово *«инверсия»*. А причина в том, что в традиционных методологиях разработки, например в структурном анализе и проектировании, принято создавать программные конструкции, в которых модули верхнего уровня зависят от модулей нижнего уровня, а стратегия – от деталей. Собственно, одна из целей таких методологий состоит в том, чтобы определить иерархию подпрограмм, описывающую, как модули верхнего уровня обращаются к модулям нижнего уровня. Первоначальный дизайн программы `Copy` (см. рис. 7.1) дает хороший пример подобной иерархии. Но в правильно спроектированной объектно-ориентированной программе структура зависимостей «инвертирована» по отношению к той, что возникает в результате применения традиционных процедурных методик.

Посмотрим, к чему приводит зависимость модулей верхнего уровня от модулей нижнего уровня. Именно в модулях верхнего уровня инкапсулированы важные стратегические решения и бизнес-модели приложения. Эти модули и отличают одно приложение от другого. Но если они зависят от модулей нижнего уровня, то изменение последних может напрямую отразиться на модулях верхнего уровня и стать причиной их изменения.

Но ведь такое положение вещей – нелепость! Это модули верхнего уровня, определяющие стратегию, должны влиять на модули нижнего уровня, а не наоборот. Модули, которые содержат высокоуровневые бизнес-правила, должны быть приоритетнее модулей, определяющих детали реализации, и независимы от них. Модули верхнего уровня вообще никак не должны зависеть от модулей более низкого уровня.

Скажу больше, именно модули верхнего уровня, определяющие стратегию, мы и хотели бы использовать повторно. Мы уже поднаторели в повторном использовании низкоуровневых модулей, представленных в виде библиотек подпрограмм. Но если модули верхнего уровня зависят от модулей нижнего уровня, то повторно использовать первые в различных контекстах становится трудно. Если же такой зависимости нет, то повторное использование модулей верхнего уровня существенно упрощается. Этот принцип лежит в основе проектирования всех каркасов.

Разбиение на слои

Согласно Бучу, «в любой хорошо структурированной объектно-ориентированной архитектуре можно выделить ясно очерченные слои, на каждом из которых предоставляется некий набор тесно связанных служб – с помощью четко определенных и контролируемых интерфейсов».[1] Наивная интерпретация этого утверждения может привести проектировщика к структуре наподобие изображенной на рис. 11.1. Здесь высо-

[1] [Booch96], стр. 54

коуровневый слой Policy использует слой более низкого уровня Mechanism, который, в свою очередь, пользуется слоем Utility, содержащим детали реализации. Хотя поначалу такая структура может показаться вполне приемлемой, у нее есть одна коварная особенность: слой Policy зависит от изменений во всех слоях на пути к Utility. *Эта зависимость транзитивна*. Слой Policy зависит от чего-то, что зависит от слоя Utility, то есть Policy транзитивно зависит от Utility. Это крайне нежелательно.

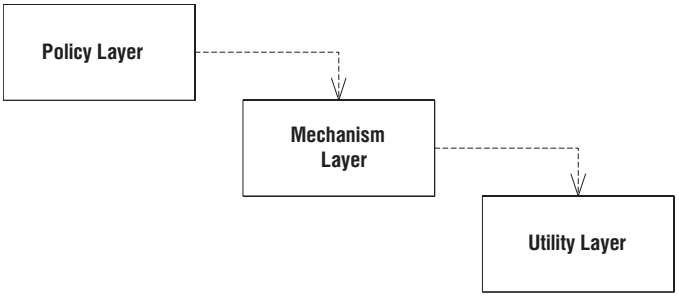


***Рис. 11.1.** Наивная схема разбиения на слои*

На рис. 11.2 изображена более подходящая модель. Слой более высокого уровня объявляет абстрактный интерфейс служб, в которых он нуждается. Затем слои нижних уровней реализуются так, чтобы удовлетворить этим интерфейсам. Любой класс, расположенный на верхнем уровне, обращается к слою соседнего снизу уровня через абстрактный интерфейс. Таким образом, верхние слои не зависят от нижних. Наоборот, нижние слои зависят от абстрактного интерфейса служб, *объявленного* на более высоком уровне. Тем самым разрывается не только транзитивная зависимость PolicyLayer от UtilityLayer, но и прямая зависимость PolicyLayer от MechanismLayer.

Инверсия владения

Отметим, что инвертируются здесь не только зависимости, но и владение интерфейсами. Мы привыкли считать, что служебные библиотеки являются владельцами своих интерфейсов. Но в случае применения принципа DIP выясняется, что именно клиентские классы владеют абстрактными интерфейсами, а серверные классы наследуют им.

Иногда это называют принципом Голливуда: «Не звоните нам, мы сами позвоним».[1] Модули нижнего уровня предоставляют реализацию интерфейсов, объявленных на более высоком уровне и оттуда же вызываемых.

[1] [Sweet85]

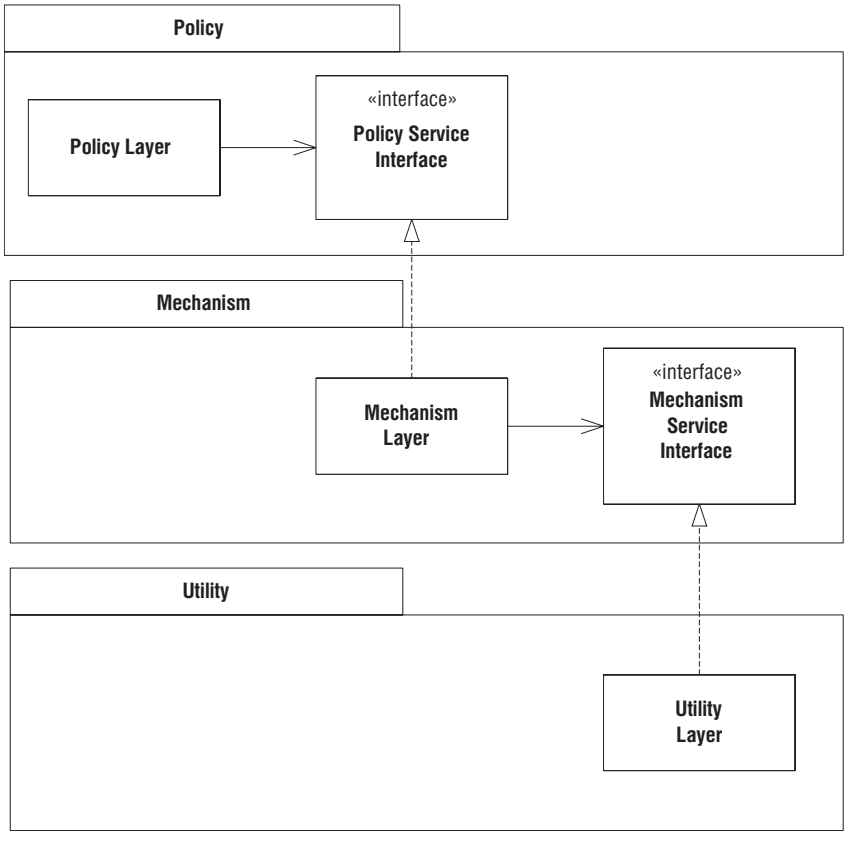


Рис. 11.2. *Инвертированные слои*

Благодаря инверсии владения слой `PolicyLayer` невосприимчив к любым изменениям слоев `MechanismLayer` и `UtilityLayer`. Более того, `PolicyLayer` можно повторно использовать в любом контексте, где определены модули нижнего уровня, согласованные с интерфейсом `PolicyServiceInterface`. Таким образом, обратив зависимости, мы создали структуру, одновременно более гибкую, прочную и подвижную.

В этом контексте под владением понимается, что интерфейсы публикуются владеющими ими клиентами, а не реализующими их серверами. Интерфейс находится в том же пакете или библиотеке, что и клиент. В результате серверная библиотека или пакет по необходимости зависит от клиентской.

Разумеется, бывают случаи, когда мы не хотим, чтобы сервер зависел от клиента. В особенности это относится к ситуации, когда есть много клиентов, но всего один сервер. Тогда клиенты должны договориться между собой об интерфейсе сервера и опубликовать его в отдельном пакете.

Зависимость от абстракций

Чуть упрощенная, но все еще весьма действенная интерпретация принципа DIP выражается простым эвристическим правилом: «Зависеть надо от абстракций». Оно гласит, что не должно быть зависимостей от конкретных классов; все связи в программе должны вести на абстрактный класс или интерфейс.

- Не должно быть переменных, в которых хранятся ссылки на конкретные классы.
- Не должно быть классов, производных от конкретных классов.
- Не должно быть методов, переопределяющих метод, реализованный в одном из базовых классов.

Конечно, эта эвристика хотя бы раз да нарушается в любой программе. Где-то ведь необходимо создавать экземпляры конкретных классов, и модуль, в котором это делается, будет от них зависеть.[1] К тому же не видно причин соблюдать это правило для конкретных, но редко изменяющихся классов. Если класс не будет часто изменяться и не предполагается создавать аналогичные ему производные классы, то зависимость от такого класса не принесет особого вреда.

Например, в большинстве систем класс, описывающий строку, конкретный. В языке C# он называется `string`. Этот класс изменяется редко, поэтому в прямой зависимости от него нет никакой беды.

Однако конкретные классы, являющиеся частью прикладной программы, которые пишем *мы сами*, в большинстве случаев изменчивы. Именно от *таких* конкретных классов мы и не хотим зависеть напрямую. Их изменчивость можно изолировать, скрыв их за абстрактным интерфейсом.

Это решение неполное. Бывает, что сам интерфейс изменчивого класса необходимо изменить, и это изменение распространяется на абстрактный интерфейс, представляющий класс. Такие изменения нарушают изолированность абстрактного интерфейса.

Потому мы и назвали это эвристическое правило упрощенным. Но, с другой стороны, если принять во внимание, что клиентские модули или слои объявляют интерфейсы необходимых им служб, то получается, что интерфейс изменяется только тогда, когда это нужно *клиенту*. Изменения же в классах, реализующих абстрактный интерфейс, не сказываются на клиенте.

---

[1] На самом деле есть способы обойти и эту трудность, если для создания классов можно использовать строки. В языке C# и в некоторых других это возможно – имена конкретных классов можно передавать в программу в виде строки в конфигурационных данных.

Простой пример принципа DIP

Инверсию зависимости можно применять всякий раз, как один класс посылает сообщение другому. Рассмотрим, например, объекты `Button` (Кнопка) и `Lamp` (Лампа).

Объект `Button` реагирует на внешние воздействия. Получив сообщение `Poll`, объект `Button` определяет, «нажал» ли пользователь на кнопку. Механизм определения не имеет значения. Это может быть кнопка в графическом интерфейсе пользователя, физическая кнопка, которую человек нажимает пальцем, или даже детектор движения в системе охраны жилища. Объект `Button` умеет распознавать, что сделал пользователь: активировал или деактивировал его.

Объект `Lamp` воздействует на окружение. Получив сообщение `TurnOn` (Включить), объект `Lamp` тем или иным образом включает освещение. Получив же сообщение `TurnOff` (Выключить), он гасит свет. Физический механизм неважен. Это может быть светодиод на консоли компьютера, ртутная лампа, освещающая парковку, или даже лазер в лазерном принтере.

Как спроектировать систему, в которой объект `Button` управляет объектом `Lamp`? На рис. 11.3 изображена наивная модель. Объект `Button` получает сообщения `Poll`, определяет, была ли нажата кнопка, а затем посылает сообщение `TurnOn` или `TurnOff` объекту `Lamp`.



***Рис. 11.3.** Наивная модель объектов Button и Lamp*

Почему наивная? Рассмотрим код на C#, вытекающий из этой модели (листинг 11.1). Отметим, что класс `Button` напрямую зависит от класса `Lamp`. Наличие такой зависимости означает, что `Button` восприимчив к изменениям в `Lamp`. И значит, повторно использовать `Button` для управления, скажем, объектом `Motor` не получится. В этой модели объекты `Button` управляют объектами `Lamp` и *только* `Lamp`.

Листинг 11.1. Button.cs

```
public class Button
{
  private Lamp lamp;
  public void Poll()
  {
    if (/* какое-то условие */)
      lamp.TurnOn();
  }
}
```

Это решение нарушает принцип DIP. Высокоуровневая стратегия приложения не отделена от низкоуровневой реализации. Абстракции не отделены от деталей. В отсутствие такого разделения стратегия верхнего уровня автоматически зависит от модулей нижнего уровня, а абстракции автоматически зависят от деталей.

Отыскание глубинных абстракций

Что такое стратегия верхнего уровня? Это абстракция, лежащая в основе приложения, та суть, которая не изменяется при изменении деталей. Это система *внутри* системы – метафора. В примере `Button/Lamp` глубинная абстракция состоит в том, чтобы распознать действие пользователя включить/выключить и сообщить о нем конечному объекту. Какой механизм применяется для распознавания действия пользователя? Несущественно! Какой объект конечный? Неважно! Все это детали, не играющие для абстракции никакой роли.

Модель на рис. 11.3 можно улучшить, инвертировав зависимость от объекта `Lamp`. На рис. 11.4 мы видим, что `Button` теперь хранит ассоциацию с неким объектом `ButtonServer`, который предоставляет интерфейс, позволяющий `Button` включить или выключить нечто. Класс `Lamp` реализует интерфейс `ButtonServer`. Следовательно, теперь не мы зависим от объекта `Lamp`, а он сам является зависимым.

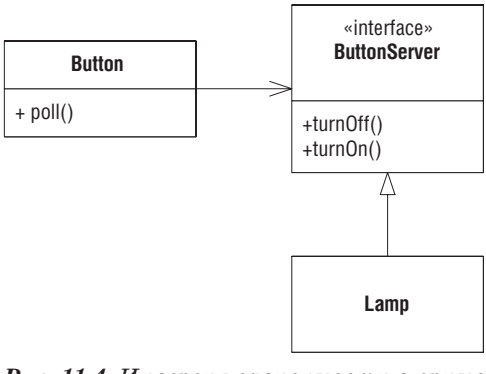


***Рис. 11.4.** Инверсия зависимости в применении к Lamp*

Дизайн, изображенный на рис. 11.4, позволяет `Button` управлять любым устройством, готовым реализовать интерфейс `ButtonServer`. Это обеспечивает высокую гибкость, а также означает, что объекты `Button` смогут управлять еще не изобретенными объектами.

Однако же это решение налагает ограничение на объект, управляемый `Button`. Такой объект *обязан* реализовать интерфейс `ButtonServer`. Это печально, потому что такие объекты, возможно, захотят управляться также объектом `Switch` или еще каким-то, помимо `Button`.

Инвертировав направление зависимости и сделав объект Lamp зависящим, мы заставили Lamp зависеть от другой детали: Button. А так ли это?

Очевидно, что Lamp зависит от ButtonServer, но ButtonServer не зависит от Button. Любой объект, знающий, как работать с интерфейсом ButtonServer, сможет управлять объектом Lamp. Следовательно, на самом деле мы зависим только от имени. И можем исправить это, переименовав ButtonServer в нечто более общее, скажем, SwitchableDevice. Мы можем также поместить Button и SwitchableDevice в разные библиотеки, чтобы использование SwitchableDevice не подразумевало использование Button.

В таком случае у этого интерфейса нет владельца. Мы получаем любопытную ситуацию, когда интерфейс может использоваться разными клиентами и реализовываться разными серверами. Стало быть, интерфейс должен существовать сам по себе, не примыкая ни к какой группе. В C# это достигается помещением его в отдельное пространство имен и в отдельную библиотеку.[1]

Задача об управлении печью

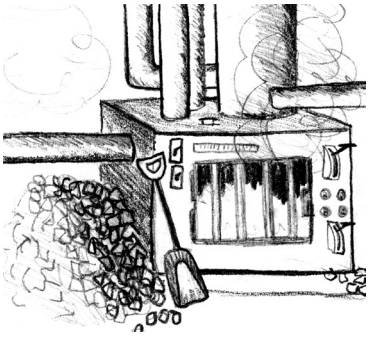
Рассмотрим более интересный пример – программу, управляющую работой печи. Она может считывать текущую температуру из канала ввода/вывода и сообщать печи о необходимости включения или выключения, посылая команды по другому каналу ввода/вывода. Алгоритм может быть построен, как показано в листинге 11.2.

Листинг 11.2. Простой алгоритм термостата

```
const byte TERMOMETER = 0x86;
const byte FURNACE = 0x87;
const byte ENGAGE = 1;
const byte 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);
  }
}
```

[1] В динамических языках типа Smalltalk, Pyrhon или Ruby этот интерфейс вообще не существовал бы в виде явно написанного кода.



Смысл этого алгоритма на верхнем уровне ясен, но код загромождён кучей низкоуровневых деталей. Его никогда не удалось бы использовать для другой контрольно-измерительной аппаратуры.

В данном случае это не такая уж большая потеря, потому что код очень короткий. Но все равно жалко было бы отказываться от повторного использования алгоритма. Давайте лучше инвертируем зависимости и придем к структуре, изображенной на рис. 11.5.

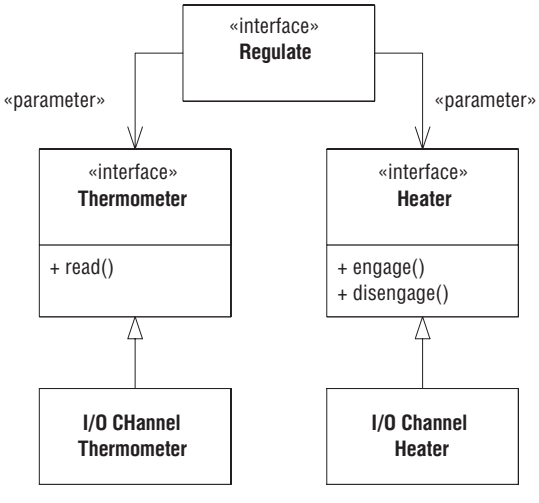


Рис. 11.5. *Регулятор общего вида*

Как видим, функция `Regulate` принимает два аргумента, оба являются интерфейсами. Интерфейс `Thermometer` позволяет считывать показания, а интерфейс `Heater` – включать и выключать нагреватель. Это все, что нужно алгоритму `Regulate`. Теперь его можно переписать, как показано в листинге 11.3.

Следовательно, мы инвертировали зависимости так, что высокоуровневая стратегия регулировки не зависит от конкретных деталей термометра или печи. Алгоритм вполне можно использовать повторно.

Листинг 11.3. Регулятор общего вида

```
void 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();
    }
  }
```

Заключение

В традиционном процедурном программировании структура зависимостей такова, что стратегия зависит от деталей. Это плачевно, потому что стратегия становится восприимчивой к изменению деталей. В объектно-ориентированном программировании структура зависимостей инвертируется, так что и детали, и стратегии зависят от абстракции, а интерфейсами служб часто владеют клиенты.

На самом деле такая инверсия зависимостей – отличительный признак объектно-ориентированного дизайна. Неважно, на каком языке написана программа. Если зависимости инвертированы, значит, мы имеем объектно-ориентированный дизайн. В противном случае дизайн процедурный.

Принцип инверсии зависимостей – это фундаментальный низкоуровневый механизм, лежащий в основе многих преимуществ, которые обещают объектно-ориентированные технологии. Его надлежащее применение необходимо для создания повторно используемых каркасов. Кроме того, он крайне важен для конструирования кода, устойчивого к изменениям. Поскольку абстракции и детали изолированы друг от друга, такой код гораздо легче сопровождать.

Библиография

[Booch96] Grady Booch «Object Solutions: Managing the Object-Oriented Project», Addison-Wesley, 1996.

[GOF95] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.[1]

[Sweet85] Richard E. Sweet, «The Mesa Programming Environment», *SIGPLAN Notices*, 20(7) July 1985: 216–229.

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – Питер, 2007.

12

Принцип разделения интерфейсов (ISP)

Этот принцип относится к недостаткам «жирных» интерфейсов. Говорят, что класс имеет «жирный» интерфейс, если функции этого интерфейса недостаточно сцепленные. Иными словами, интерфейс класса можно разбить на группы методов. Каждая группа предназначена для обслуживания разнотипных клиентов. Одним клиентам нужна одна группа методов, другим – другая.

Принцип ISP допускает, что могут существовать объекты, нуждающиеся в несцепленных интерфейсах, однако предполагает, что клиентам необязательно знать, что это единый класс. Клиенты должны лишь знать об абстрактных интерфейсах, обладающих свойством сцепленности.

Загрязнение интерфейса

Рассмотрим охранную систему, в которой объекты `Door` (Дверь) можно запирать и отпирать, а также узнавать, открыта дверь или закрыта (см. листинг 12.1). Тип `Door` закодирован в виде интерфейса, чтобы клиенты могли использовать объекты, согласованные с интерфейсом двери, не зная о деталях конкретной реализации.

Листинг 12.1. Охраняемая дверь

```
public interface Door
{
  void Lock();
  void Unlock();
  bool IsDoorOpen();
}
```

Теперь предположим, что одна такая реализация, `TimedDoor`, должна подавать звуковой сигнал, если дверь остается открытой слишком долго. Для этого `TimedDoor` взаимодействует с объектом `Timer` (листинг 12.2).

Листинг 12.2

```
public class Timer
{
  public void Register(int timeout, TimerClient client)
  {/* код */}
}

public interface TimerClient
{
  void TimeOut();
}
```

Если объект желает получать уведомление об истечении тайм-аута, он вызывает метод `Register` объекта `Timer`. Этому методу в качестве аргументов передаются величина тайм-аута и ссылка на объект `TimerClient`, метод `TimeOut` которого нужно вызвать, когда тайм-аут истечет.

Как организовать взаимодействие класса `TimerClient` с классом `TimedDoor` таким образом, чтобы некий код внутри `TimedDoor` мог получать уведомления о тайм-ауте? Есть несколько вариантов. На рис. 12.1 показано наиболее распространенное решение. Мы производим класс `Door`, а значит, и `TimedDoor` от `TimerClient`. Тем самым гарантируется, что `TimerClient` может зарегистрировать себя в объекте `Timer` и получить сообщение `TimeOut`.

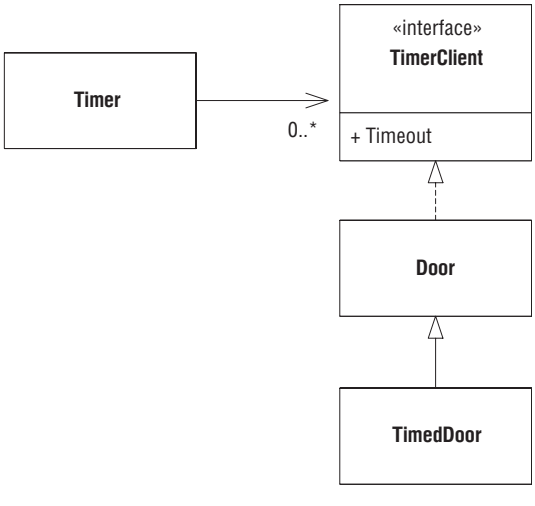


***Рис. 12.1.** Класс TimerClient на вершине иерархии*

Проблема этого решения в том, что класс `Door` теперь зависит от `TimerClient`. Но не всем разновидностям `Door` необходимо следить за временем. На самом деле первоначальная абстракция `Door` вообще ничего не знала об отсчете времени. При создании производных от `Door` классов, не интересующихся временем, нам придется включать вырожденные

реализации метода TimeOut, то есть идти на нарушение принципа LSP. К тому же приложения, в которых используются такие производные классы, должны будут импортировать определение класса TimerClient, хотя оно им и не нужно. Это попахивает ненужной сложностью и ненужным повторением.

Это пример загрязнения интерфейса – синдрома, характерного для таких статически типизированных языков, как C#, C++ и Java. Интерфейс Door загрязнен лишним методом, который включен исключительно для нужд подклассов. Если следовать такой практике, то всякий раз, как производному классу понадобится новый метод, мы должны будем добавлять его в базовый класс. От этого интерфейс базового класса загрязнится еще больше и станет «жирным».

Хуже того, при добавлении в базовый класс каждого нового метода его придется реализовывать в производных классах. На практике методы базового класса снабжают некоей вырожденной реализацией по умолчанию, чтобы не обременять клиентов их реализацией. Но, как мы выяснили ранее, такая практика нарушает принцип LSP и ведет к проблемам с сопровождением и повторным использованием.

Разделение клиентов означает разделение интерфейсов

Типы Door и TimerClient представляют интерфейсы, используемые совершенно разными клиентами. Класс Timer применяет TimerClient, а классы, манипулирующие дверьми, используют Door. Поскольку эти клиенты разделены, то должны быть разделены и интерфейсы. Почему? Потому, что клиент определяет интерфейс своего сервера.

Размышляя о необходимости изменения в программах, мы обычно думаем, как изменение интерфейса отразится на его пользователях. Например, мы задались бы вопросом о том, какие изменения придется произвести в коде пользователей TimerClient, если его интерфейс изменится. Однако существует сила, действующая в другом направлении. Иногда изменение интерфейса вызвано желанием *пользователя*.

Например, некоторые клиенты класса Timer хотели бы регистрировать более одного запроса на истечение тайм-аута. Рассмотрим класс TimedDoor. Обнаружив, что дверь открыта, он посылает сообщение Register классу Timer, запрашивая уведомление о тайм-ауте. Однако еще до истечения тайм-аута дверь закрывается, остается некоторое время закрытой, а потом открывается снова. Это заставляет нас зарегистрировать *новый* запрос уведомления о тайм-ауте еще до того, как истек предыдущий. Но в конечном итоге первый тайм-аут истекает и вызывается метод TimeOut класса TimedDoor. Класс Door подает ложную тревогу.

Эту ситуацию можно исправить, воспользовавшись соглашением, которое показано в листинге 12.3. Мы включаем в каждый запрос о реги-

страции тайм-аута уникальный код timeOutId и передаем этот код при вызове метода TimeOut. В результате объект любого класса, производного от TimerClient, знает, на какой запрос об истечении тайм-аута он отвечает.

Листинг 12.3. Класс Timer с идентификатором

```
public class Timer
{
  public void Register(int timeout,
                       int timeOutId,
                       TimerClient client)
  {/* код */}
}

public interface TimerClient
{
  void TimeOut(int timeOutID);
}
```

Понятно, что такое изменение отразится на всех пользователях интерфейса TimerClient. Мы готовы на это пойти, потому что отсутствие параметра timeOutId – упущение, которое нужно исправить. Однако представленный на рис. 12.1 дизайн влечет за собой необходимость изменить также интерфейс Door и всех его клиентов! А это уже попахивает жесткостью и вязкостью. С какой стати ошибка в TimerClient должна *хоть каким-то* образом влиять на клиентов классов, производных от Door, которым вовсе не требуется следить за временем? Это странная взаимозависимость сильно пугает заказчиков и менеджеров. Если изменение в одной части программы оказывает влияние на другие, совершенно с ней не связанные, то стоимость сопровождения становится непредсказуемой, а риск нежелательных последствий изменения резко возрастает.

> **Принцип разделения интерфейсов (Interface Segregation Principle)**
>
> *Клиенты не должны вынужденно зависеть от методов, которыми не пользуются.*

Если клиент вынужденно зависит от методов, которыми не пользуется, то он оказывается восприимчив к изменениям в этих методах. В результате возникает непреднамеренная связанность между всеми клиентами. Иначе говоря, если клиент зависит от класса, содержащего методы, которыми этот клиент не пользуется, но пользуются другие клиенты, то данный клиент становится зависим от всех изменений, вносимых в класс в связи с потребностями этих «других клиентов». Мы хотели бы по возможности избежать таких связей и потому стремимся разделять интерфейсы.

Интерфейсы классов и интерфейсы объектов

Рассмотрим еще раз класс `TimedDoor`. У него есть два раздельных интерфейса, используемые двумя разными клиентами: `Timer` и пользователями `Door`. Эти два интерфейса *должны* быть реализованы в одном и том же объекте, поскольку для их реализации используются одни и те же данные. Как в этих условиях удовлетворить принципу ISP? Как разделить интерфейсы, если они должны находиться вместе?

Ответ заключается в том, что клиенты некоторого объекта не обязаны обращаться к нему через интерфейс именно этого объекта. Им лучше воспользоваться делегированием или интерфейсом базового класса.

Разделение путем делегирования

Одно решение состоит в том, чтобы создать класс, который наследует `TimerClient` и делегирует часть своих обязанностей `TimedDoor`. Оно показано на рис. 12.2. Когда объект хочет зарегистрировать запрос на уведомление о тайм-ауте в объекте `Timer`, `TimedDoor` создает экземпляр `DoorTimerAdapter` и регистрирует его в `Timer`. Когда `Timer` посылает сообщение `TimeOut` объекту `DoorTimerAdapter`, тот передает его объекту `TimedDoor`.

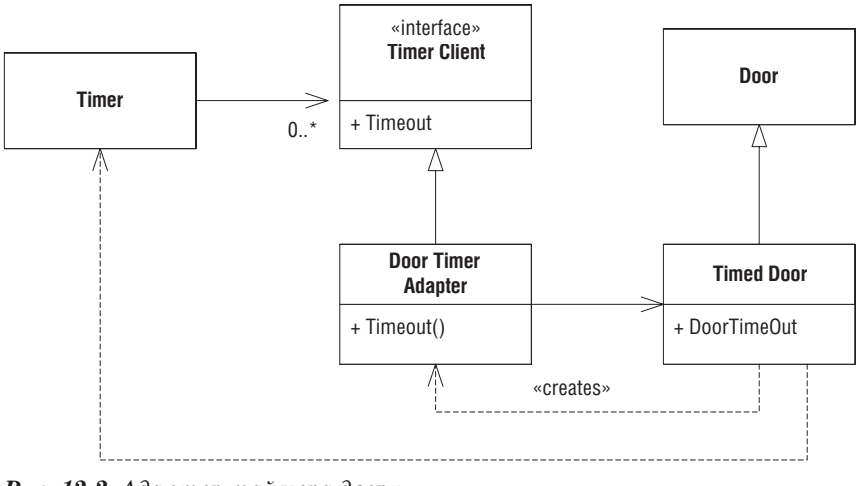


***Рис. 12.2.** Адаптер таймера двери*

Это решение согласуется с принципом ISP и не создает связей между клиентами `Door` и классом `Timer`. Даже если в класс `Timer` придется внести изменение, показанное в листинге 12.3, ни один из пользователей `Door` не пострадает. Более того, класс `TimedDoor` даже не обязан иметь в точности такой же интерфейс, как `TimerClient`. Класс `DoorTimerAdapter` может *преобразовать* интерфейс `TimerClient` в интерфейс `TimedDoor`. Поэтому такое решение обладает большой универсальностью (листинг 12.4).

Листинг 12.4. TimedDoor.cs

```
public interface TimedDoor : Door
{
  void DoorTimeOut(int timeOutId);
}

public class DoorTimerAdapter : TimerClient
{
  private TimedDoor timedDoor;

  public DoorTimerAdapter(TimedDoor theDoor)
  {
    timedDoor = theDoor;
  }

  public virtual void TimeOut(int timeOutId)
  {
    timedDoor.DoorTimeOut(timeOutId);
  }
}
```

Однако оно не слишком элегантно. Каждый раз, когда мы хотим зарегистрировать тайм-аут, приходится создавать новый объект. И делегирование требует пусть и небольшого, но ненулевого времени и памяти. В некоторых категориях приложений, например во встроенных системах управления реального времени, процессорное время и память – настолько дефицитные ресурсы, что этим недостатком пренебречь нельзя.

Разделение путем множественного наследования

На рис. 12.3 и в листинге 12.5 показано, как можно поддержать принцип ISP с помощью множественного наследования. В этой модели класс `TimedDoor` наследует одновременно `Door` и `TimerClient`. Хотя клиенты обоих базовых классов могут пользоваться объектом `TimedDoor`, ни один из них от класса `TimedDoor` не зависит. То есть они пользуются одним и тем же объектом через раздельные интерфейсы.

Листинг 12.5. TimedDoor.cs

```
public interface TimedDoor : Door, TimerClient
{
}
```

Обычно я предпочитаю именно такое решение. Подход, показанный на рис. 12.2, я выбрал бы лишь в том случае, когда преобразование интерфейсов, выполняемое объектом `DoorTimerAdapter`, необходимо или когда в разные моменты времени должны выполняться различные преобразования.

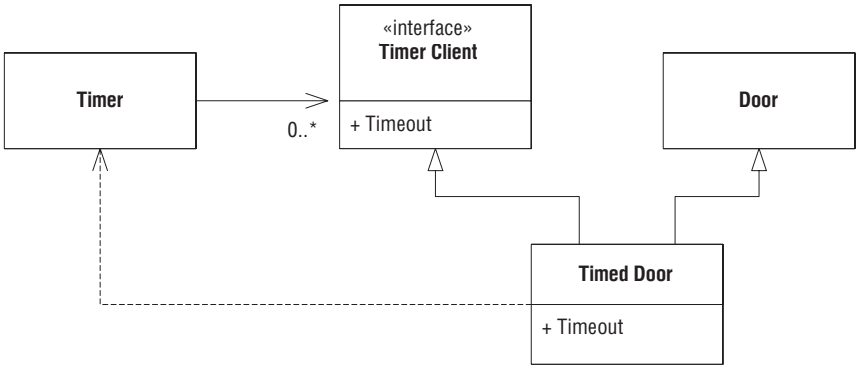


Рис. 12.3. *Класс TimedDoor с множественным наследованием*

Пользовательский интерфейс банкомата

Теперь рассмотрим чуть более практический пример: традиционную задачу о банкомате. Интерфейс пользователя (ИП) банкомата должен быть очень гибким. Может потребоваться перевод сообщений на разные языки, а выводить их можно на экран, на брайлевский планшет или с помощью синтезатора речи (рис. 12.4). Очевидно, что обеспечить такую гибкость можно путем создания абстрактного базового класса, в котором имеются абстрактные методы для всех сообщений, предусмотренных в интерфейсе.

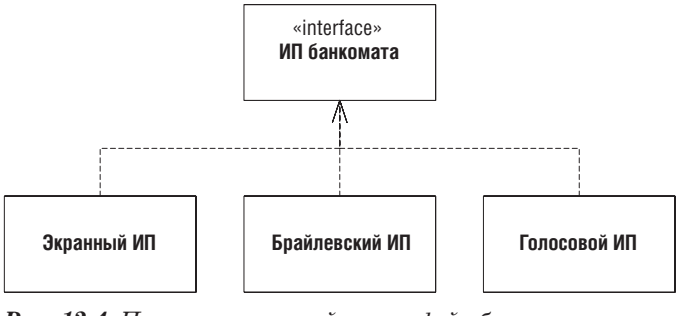


Рис. 12.4. *Пользовательский интерфейс банкомата*

Примем также во внимание, что все транзакции, которые умеет выполнять банкомат, инкапсулированы в классы, производные от `Transaction`. Следовательно, могут быть классы `DepositTransaction`, `WithdrawalTransaction`, `TransferTransaction` и т. д. Каждый из них вызывает методы ИП. Например, чтобы попросить пользователя ввести сумму депозита, объект `DepositTransaction` вызывает метод `RequestDepositAmount` класса ИП. Аналогично, чтобы спросить, какую сумму следует перевести с одного

счета на другой, объект TransferTransaction вызывает метод RequestTransferAmount ИП. Это соответствует диаграмме на рис. 12.5.

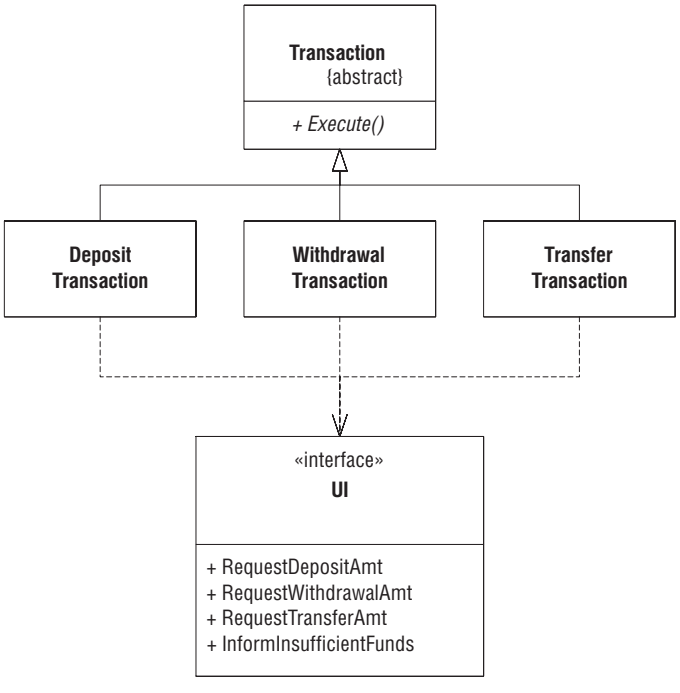


Рис. 12.5. *Иерархия транзакций банкомата*

Заметим, что это как раз та ситуация, от которой предостерегает принцип ISP. Каждая транзакция использует методы ИП, которыми не пользуется больше ни один класс. Следовательно, возникает опасность, что изменение любого подкласса Transaction вынудит вносить соответствующее изменение в ИП, повлияв тем самым на все остальные подклассы Transaction и на прочие классы, зависящие от ИП. Здесь попахивает жесткостью и хрупкостью.

Например, если бы мы захотели добавить класс PayGasBillTransaction, то пришлось бы добавлять в ИП новые методы для вывода сообщений, присущих только этой транзакции. Но так как DepositTransaction, WithdrawalTransaction и TransferTransaction зависят от ИП, то все их пришлось бы заново компилировать. Хуже того, если классы транзакций представлены компонентами в отдельных сборках, то эти сборки пришлось бы заново развертывать, хотя их логика и не изменилась. Чувствуете запах вязкости?

Такой нежелательной связанности можно избежать, разделив интерфейс ИП на части: DepositUI, WithdrawUI, TransferUI. Затем их можно объ-

единить в окончательном интерфейсе ИП с помощью множественного наследования. Такая модель показана на рис. 12.6 и в листинге 12.6.

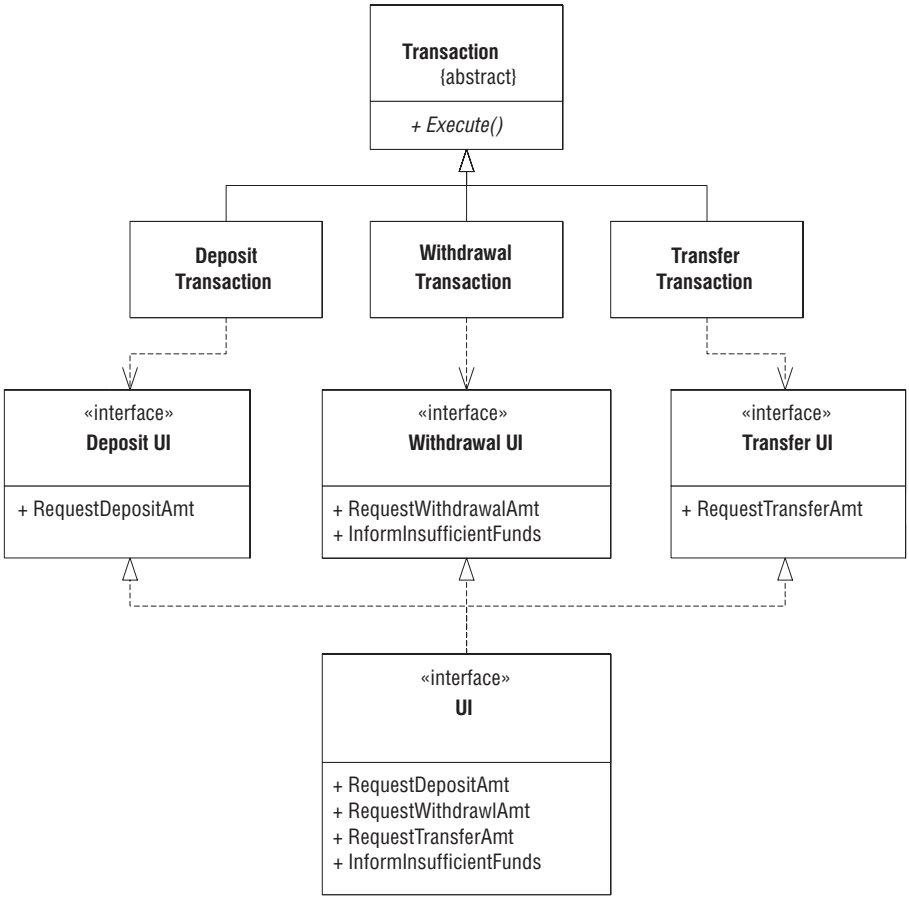


Рис. 12.6. *Разделенный интерфейс ИП банкомата*

При появлении нового класса, производного от `Transaction`, понадобится соответствующий базовый класс для абстрактного интерфейса ИП, поэтому интерфейс ИП и все производные от него классы должны измениться. Однако эти классы используются не очень широко. По существу, они нужны, пожалуй, только головной программе или процедуре, которая осуществляет начальную загрузку системы и создает экземпляр конкретного ИП. Так что добавление новых базовых классов ИП оказывает минимальное влияние.

При внимательном изучении рис. 12.6 выясняется одна проблема, связанная с принципом ISP, которая не проявлялась в примере с классом `TimedDoor`. Обратите внимание, что каждый класс транзакции должен

каким-то образом знать о своей конкретной версии ИП. DepositTransaction должен знать о DepositUI, WithdrawTransaction — о WithdrawalUI и т. д. В листинге 12.6 я решил эту проблему, передавая ссылку на конкретный ИП в конструктор класса транзакции. Это позволяет мне воспользоваться идиомой, показанной в листинге 12.7.

В общем, неплохо, но в каждом классе транзакции вынужденно появляется поле для хранения ссылки на соответствующий объект ИП. У программиста, пишущего на языке C#, может возникнуть искушение поместить все компоненты ИП в один класс. Такой подход продемонстрирован в листинге 12.8. Однако у него есть нежелательный эффект. Класс UIGlobals зависит от DepositUI, WithdrawalUI и TransferUI. Это означает, что любой модуль, который захочет использовать какой-нибудь из интерфейсов ИП, транзитивно окажется зависящим от всех, то есть мы получаем в точности ту ситуацию, от которой предостерегает принцип ISP. Если произвести изменение в любом интерфейсе ИП, то придется перекомпилировать все модули, в которых используется UIGlobals. В классе UIGlobals воссоединились все интерфейсы, которые мы так усердно разделяли!

Листинг 12.6. Разделенный интерфейс ИП банкомата

```
public interface Transaction
{
  void Execute();
}

public interface DepositUI
{
  void RequestDepositAmount();
}

public class DepositTransaction : Transaction
{
  privateDepositUI depositUI;

  public DepositTransaction(DepositUI ui)
  {
    depositUI = ui;
  }

  public virtual void Execute()
  {
    /* код */
    depositUI.RequestDepositAmount();
    /* код */
  }
}

public interface WithdrawalUI
{
```

```
  void RequestWithdrawalAmount();
}

public class WithdrawalTransaction : Transaction
{
  private WithdrawalUI withdrawalUI;

  public WithdrawalTransaction(WithdrawalUI ui)
  {
    withdrawalUI = ui;
  }

  public virtual void Execute()
  {
    /* код */
    withdrawalUI.RequestWithdrawalAmount();
    /* код */
  }
}

public interface TransferUI
{
  void RequestTransferAmount();
}

public class TransferTransaction : Transaction
{
  private TransferUI transferUI;

  public TransferTransaction(TransferUI ui)
  {
    transferUI = ui;
  }

  public virtual void Execute()
  {
    /* код */
    transferUI.RequestTransferAmount();
    /* код */
  }
}

public interface UI : DepositUI, WithdrawalUI, TransferUI
{
}
```

Листинг 12.7. Идиома инициализации интерфейса

```
UI Gui; // глобальный объект;

void f()
```

```
{
  DepositTransaction dt = new DepositTransaction(Gui);
}
```

Листинг 12.8. Обертывание Globals в класс

```
public class UIGlobals
{
  public static WithdrawalUI withdrawal;
  public static DepositUI deposit;
  public static TransferUI transfer;

  static UIGlobals()
  {
    UI Lui = new AtmUI(); // какая-то реализация ИП
    UIGlobals.deposit = Lui;
    UIGlobals.withdrawal = Lui;
    UIGlobals.transfer = Lui;
  }
}
```

Рассмотрим теперь функцию `g`[1], которой необходим доступ к `DepositUI` и `TransferUI`. Предположим также, что мы хотим передать ей эти ИП. Надо ли объявить функцию так:

```
void g(DepositUI depositUI, TransferUI transferUI)
```

Или лучше написать вот так:

```
void g(UI ui)
```

Искушение воспользоваться второй записью (одноместной) велико. В конце концов, мы же знаем, что в первом случае (многоместная функция) оба аргумента указывают на *один и тот же объект*. Более того, если бы мы остановились на многоместной форме, то вызов функции выглядел бы следующим образом:

```
g(ui, ui);
```

Извращение какое-то.

Но извращение или нет, а многоместный вариант часто предпочтительнее одноместного. Одноместная форма приводит к тому, что `g` зависит от всех интерфейсов, включенных в ИП. Поэтому при любом изменении, скажем, `WithdrawalUI` окажутся затронуты как `g`, так и все клиенты `g`. Это еще большее извращение, чем запись `g(ui,ui)`! К тому же нет никакой гарантии, что оба аргумента `g` *всегда* ссылаются на один и тот же объект! В будущем может случиться, что некоторые объекты ИП будут разделены. Ведь `g` не обязана знать о том, что все интерфейсы объеди-

[1] В языке C# нет свободных функций. По-видимому, автор имел в виду открытый статический метод. – *Прим. перев.*

нены в один объект. Поэтому я предпочитаю многоместную форму подобных функций.

Клиентов часто можно группировать вместе, исходя из методов служб, которые они вызывают. Такая группировка позволяет создавать разделенные интерфейсы для каждой группы, а не для каждого клиента. Это существенно уменьшает количество интерфейсов, которые должна реализовывать служба, и предотвращает зависимость службы от каждого типа клиента.

Иногда методы, вызываемые разными группами клиентов, перекрываются. Если перекрытие невелико, то интерфейсы групп следует оставлять разделенными. Общие методы нужно будет объявить во всех перекрывающихся интерфейсах. Класс сервера унаследует общие методы от каждого интерфейса, но реализует их только один раз.

В ходе сопровождения объектно-ориентированных приложений интерфейсы существующих классов и компонентов часто изменяются. Иногда эти изменения распространяются очень широко и вынуждают повторно компилировать и развертывать весьма значительную часть системы. Такое развитие событий можно смягчить за счет добавления новых интерфейсов в существующие объекты вместо изменения старого интерфейса. Если клиент старого интерфейса пожелает обратиться к методам нового, то он сможет запросить этот интерфейс у объекта, как показано в листинге 12.9.

Листинг 12.9

```
void Client(Service s)
{
  if(s is NewService)
  {
    NewService ns = (NewService)s;
    // использовать новый интерфейс сервера
  }
}
```

Как всегда, когда дело касается принципов, важно не переусердствовать. Призрак класса, реализующего сотни различных интерфейсов, часть которых сгруппирована по клиентам, а другие – по версиям, действительно способен напугать до полусмерти.

Заключение

Жирные классы приводят к неочевидным и вредным связям между их клиентами. Если одному клиенту требуется изменить жирный класс, то оказываются затронуты и все остальные классы. Поэтому клиенты должны зависеть только от методов, которые вызывают. Этого можно достичь путем разбиения интерфейса жирного класса на несколько интерфейсов, специально предназначенных для клиентов. В каждом

таком интерфейсе объявляются только методы, которые вызывает конкретный клиент или группа клиентов. Затем жирный класс может унаследовать всем специальным для клиентов интерфейсам и реализовать их. Это разрывает зависимость клиента от методов, к которым он не обращается, и делает клиентов независимыми друг от друга.

Библиография

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.[1]

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – Питер, 2007.

13

Обзор UML для программистов



Унифицированный язык моделирования UML (Unified Modeling Language) – это язык графического описания программных сущностей в виде диаграмм. С его помощью можно рисовать диаграммы предметной области, предполагаемого дизайна программы или уже завершенной реализации. Фаулер говорит соответственно о *концептуальном* уровне, уровне *спецификаций* и уровне *реализации*.[1] В этой книге рассматриваются только последние два уровня.

Диаграммы уровня спецификаций и реализации тесно связаны с исходным кодом. Действительно, диаграмма уровня спецификаций для того и рисуется, чтобы впоследствии преобразовать ее в исходный код. Аналогично смысл диаграммы уровня реализации заключается в том,

[1] [Fowler1999]

чтобы описать уже имеющийся исходный код. Поэтому диаграммы на этих уровнях должны следовать определенным правилам и семантическим соглашениям. Они оставляют очень мало места для неоднозначности и в высокой степени формализованы.

С другой стороны, диаграммы концептуального уровня слабо связаны с исходным кодом. Они ближе к *человеческому* языку. Это сокращенный способ описания понятий и абстракций предметной области. Поскольку они не следуют строгим семантическим соглашениям, то могут допускать различные толкования.

Рассмотрим, к примеру, следующее предложение: *собака является животным*. Его можно представить с помощью следующей концептуальной UML-диаграммы:



Рис. 13.1. *Концептуальная UML-диаграмма*

На этой диаграмме мы видим две сущности – *Животное* и *Собака*, – связанные отношением *обобщения*. *Животное* – это обобщение понятия *Собака*. *Собака* – частный случай *Животного*. Это все, что сообщается на данной диаграмме. Больше из нее никаких выводов сделать нельзя. Я мог бы утверждать, что мой песик Бобик – животное. Или что любая собака как биологический вид принадлежит животному царству. Так что диаграмма допускает интерпретацию.

Однако на уровне спецификаций или реализации та же самая диаграмма имеет гораздо более точную семантику:

```
public class Animal {}
public class Dog : Animal {}
```

В этом исходном коде определено, что `Animal` (Животное) и `Dog` (Собака) – классы, связанные отношением наследования. В то время как концептуальная модель ничего не говорит о компьютерах, обработке данных или программах, модель на уровне спецификаций *описывает часть программы*.

К сожалению, из самой диаграммы невозможно понять, к какому уровню она относится. И это источник постоянных недоразумений между программистами и аналитиками. Диаграмма концептуального уровня

не определяет и не должна определять исходный код. Диаграмма уровня спецификаций, которая описывает решение задачи, не обязана выглядеть хоть сколько-нибудь похоже на диаграмму концептуального уровня, описывающую саму задачу.

Все последующие диаграммы в этой книге относятся к уровню спецификаций или реализации и всюду, где это имеет смысл, сопровождаются исходным кодом. Больше с диаграммами концептуального уровня мы не встретимся.

Ниже мы приводим очень краткий обзор основных диаграмм, используемых в UML. Это позволит вам читать и рисовать большинство наиболее употребительных диаграмм. А в последующих главах будут рассмотрены детали и формальные соглашения, необходимые для того, чтобы стать экспертом по языку UML.

В UML есть три основных вида диаграмм. *Статические диаграммы* описывают неизменную логическую структуру программы, а именно элементы – классы, объекты, структуры данных – и отношения между ними. На *динамических диаграммах* показано, как программные сущности изменяются во время выполнения: поток выполнения или изменение состояния сущностей. На *физических диаграммах* изображается неизменная физическая структура системы: исходные файлы, библиотеки, двоичные файлы, файлы данных и прочее, а также связи между ними.

Рассмотрим код в листинге 13.1. Эта программа реализует отображение на базе простого двоичного дерева. Изучите код, прежде чем переходить к следующим за ним диаграммам.

Листинг 13.1. TreeMap.cs

```
using System;

namespace TreeMap
{
  public class TreeMap
  {
    private TreeMapNode topNode = null;

    public void Add(IComparable key, object value)
    {
      if (topNode == null)
        topNode = new TreeMapNode(key, value);
      else
        topNode.Add(key, value);
    }

    public object Get(IComparable key)
    {
      return topNode == null ? null : topNode.Find(key);
```

```
    }
  }

  internal class TreeMapNode
  {
    private static readonly int LESS = 0;
    private static readonly int GREATER = 1;
    private IComparable key;
    private object value;
    private TreeMapNode[] nodes = new TreeMapNode[2];

    public TreeMapNode(IComparable key, object value)
    {
      this.key = key;
      this.value = value;
    }

    public object Find(IComparable key)
    {
      if (key.CompareTo(this.key) == 0) return value;
      return FindSubNodeForKey(SelectSubNode(key), key);
    }

    private int SelectSubNode(IComparable key)
    {
      return (key.CompareTo(this.key) < 0) ? LESS : GREATER;
    }

    private object FindSubNodeForKey(int node, IComparable key)
    {
      return nodes[node] == null ? null : nodes[node].Find(key);
    }

    public void Add(IComparable key, object value)
    {
      if (key.CompareTo(this.key) == 0)
        this.value = value;
      else
        AddSubNode(SelectSubNode(key), key, value);
    }

    private void AddSubNode(int node, IComparable key,
      object value)
    {
      if (nodes[node] == null)
        nodes[node] = new TreeMapNode(key, value);
      else
        nodes[node].Add(key, value);
    }
  }
}
```

Диаграммы классов

На диаграмме классов (рис. 13.2) изображаются основные классы программы и отношения между ними. В классе `TreeMap` есть открытые методы `Add` и `Get` и переменная `topNode`, в которой хранится ссылка на объект `TreeMapNode`. В каждом экземпляре `TreeMapNode` хранятся ссылки на два других экземпляра `TreeMapNode`, помещенные в некий контейнер с именем `nodes`, а также ссылки `key` и `value` на некоторые другие объекты. Объект, на который ссылается переменная `key`, реализует интерфейс `IComparable`. А в переменной `value` находится ссылка на произвольный объект.

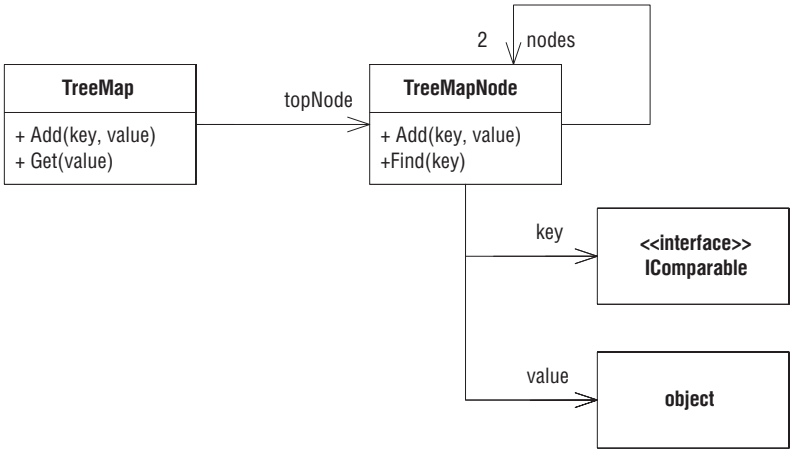


Рис. 13.2. *Диаграмма классов для TreeMap*

Различные тонкости, свойственные диаграммам классов, мы будем рассматривать в главе 19. А сейчас достаточно знать следующее.

- Классы представляются прямоугольниками, а отношения между ними – стрелками.
- На данной диаграмме все отношения являются *ассоциациями*. Ассоциацией называется простое отношение, состоящее в том, что один объект хранит ссылку на другой и вызывает методы другого объекта.
- Имя над ассоциацией соответствует имени переменной, в которой хранится ссылка.
- Число рядом с острием стрелки обычно говорит о том, сколько экземпляров может быть связано данным отношением. Если это число больше 1, то, как правило, имеется в виду массив.
- В прямоугольниках классов может быть несколько отделений. В верхнем отделении всегда записывается имя класса, а в остальных содержатся функции и переменные.

- Нотация «interface» означает, что `IComparable` — интерфейс.
- Большая часть остальных показанных обозначений необязательна.

Внимательно посмотрите на диаграмму и сопоставьте ее с кодом в листинге 13.1. Обратите внимание, как ассоциации соответствуют переменным экземпляра. Например, ассоциация, ведущая от `TreeMap` к `TreeMapNode`, называется `topNode` и соответствует переменной `topNode` в классе `TreeMap`.

Диаграммы объектов

На рис. 13.3 показана *диаграмма объектов*. На ней изображены объекты и отношения между ними в конкретный момент выполнения программы. Можно считать, что это мгновенный снимок памяти.

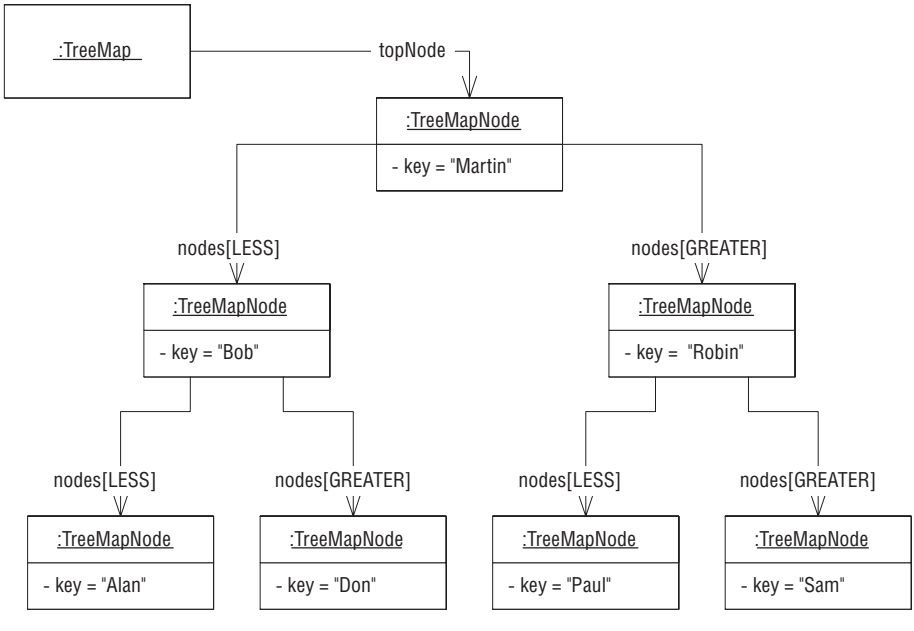


Рис. 13.3. *Диаграмма объектов для TreeMap*

На этой диаграмме прямоугольниками представлены объекты. Мы можем сказать, что это именно объекты, потому что их имена подчеркнуты. После двоеточия указывается имя класса, которому принадлежит объект. В нижнем отделении каждого прямоугольника записывается значение переменной `key` данного объекта.

Отношения между объектами, называемые связями, выводятся из ассоциаций, показанных на рис. 13.2. Обратите внимание, как поименованы связи для двух элементов массива `nodes`.

Диаграммы последовательности

На рис. 13.4 показана диаграмма последовательности. Она описывает реализацию метода `TreeMap.Add`.

Фигурка человечка обозначает неизвестную вызывающую программу. Она обращается к методу `Add` объекта `TreeMap`. Если переменная `topNode` равна `null`, то `TreeMap` отвечает созданием нового объекта `TreeMapNode` и присваиванием ссылки на него переменной `topNode`. В противном случае `TreeMap` посылает сообщение `Add` объекту `topNode`.

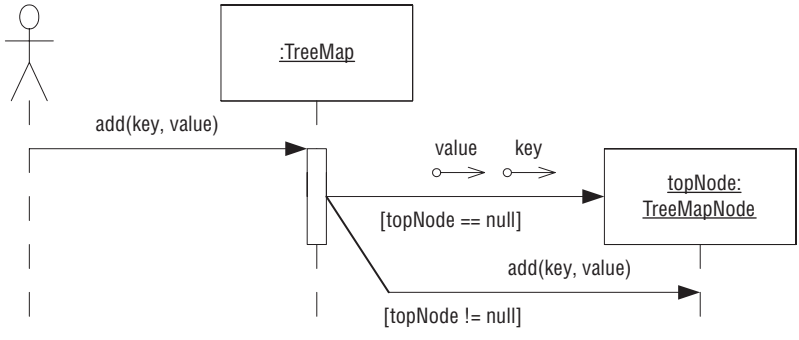


***Рис. 13.4.** Метод TreeMap.Add*

Булевские выражения в квадратных скобках называются *сторожевыми условиями*. Они показывают, по какому пути пойдет выполнение программы. Стрелка сообщения, доходящая до прямоугольника `TreeMapNode`, обозначает *конструирование*. Маленькие стрелки с кружками называются *маркерами данных*. В данном случае они описывают аргументы конструктора. Вытянутый прямоугольник под `TreeMap` называется *активацией*. Он показывает, сколько времени занимает выполнение метода `Add`.

Диаграммы кооперации

На рис. 13.5 изображена *диаграмма кооперации*, описывающая случай, когда при вызове метода `TreeMap.Add` переменная `topNode` не равна `null`. На диаграммах кооперации представлена та же информация, что на диаграммах последовательности, но если последние проясняют порядок отправки сообщений, то первые – отношения между объектами.

Объекты соединены стрелками, которые называются *связями*. Связь существует, если один объект может послать сообщение другому. Вдоль связей передаются сами сообщения. Они изображаются стрелками меньшего размера. Каждое сообщение помечается своим именем, порядковым номером и применимыми к нему сторожевыми условиями.

Точки в порядковом номере сообщения отражают иерархию вызовов. Функция `TreeMap.Add` (сообщение 1) вызывает функцию `TreeMapNode.Add`

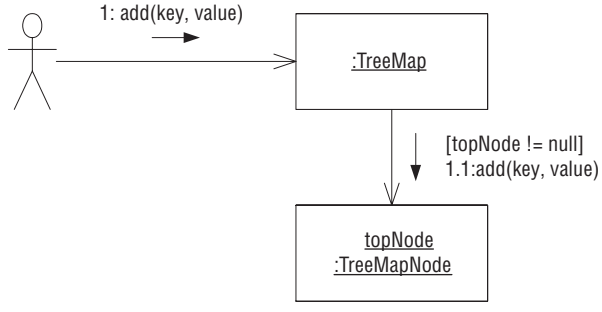


Рис. 13.5. *Диаграмма кооперации в одном из случаев выполнения метода TreeMap.Add*

(сообщение 1.1). Следовательно, сообщение 1.1 – это первое сообщение, отправленное функцией, вызванной в результате отправки сообщения 1.

Диаграммы состояний

В языке UML имеется развитая нотация для изображения конечных автоматов. На рис. 13.6 показано лишь малое ее подмножество.

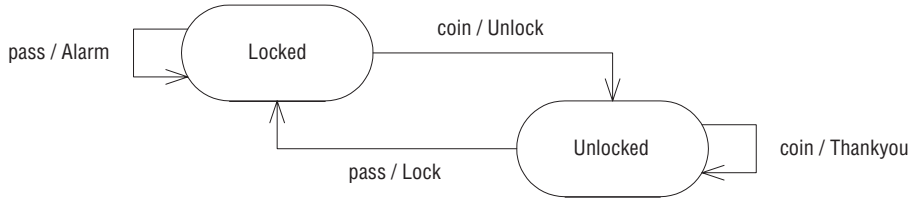


Рис. 13.6. *Конечный автомат, описывающий работу турникета в метро*

Здесь изображен конечный автомат, описывающий работу турникета в метро. Существуют два *состояния*: `Locked` (Закрыт) и `Unlocked` (Открыт). Автомату можно послать два *события*. Событие `coin` означает, что пользователь опустил в прорезь турникета монету, а событие `pass` – что пользователь прошел через турникет.

Стрелки на этой диаграмме называются *переходами*. Они помечаются парами имен: *события*, вызвавшего срабатывание перехода, и *действия*, которое выполняется во время перехода. Срабатывание перехода приводит к изменению состояния системы.

Рисунок 13.6 можно пересказать на простом человеческом языке следующим образом:

- Если мы находимся в состоянии `Locked` и получаем событие `coin`, то переходим в состояние `Unlocked` и вызываем функцию `Unlock`.

- Если мы находимся в состоянии `Unlocked` и получаем событие `pass`, то переходим в состояние `Locked` и вызываем функцию `Lock`.
- Если мы находимся в состоянии `Unlocked` и получаем событие `coin`, то остаемся в состоянии `Unlocked` и вызываем функцию `Thankyou`.
- Если мы находимся в состоянии `Locked` и получаем событие `pass`, то остаемся в состоянии `Locked` и вызываем функцию `Alarm`.

Диаграммы состояний очень полезны, когда нужно понять, как ведет себя система. Они позволяют выяснить, что должна делать система в неожиданных ситуациях, например когда пользователь опускает одну монету, а затем без всяких причин *еще одну*.

Заключение

Диаграмм, показанных в этой главе, для большинства целей достаточно. Программисту, как правило, не нужно знать о UML ничего сверх изложенного выше.

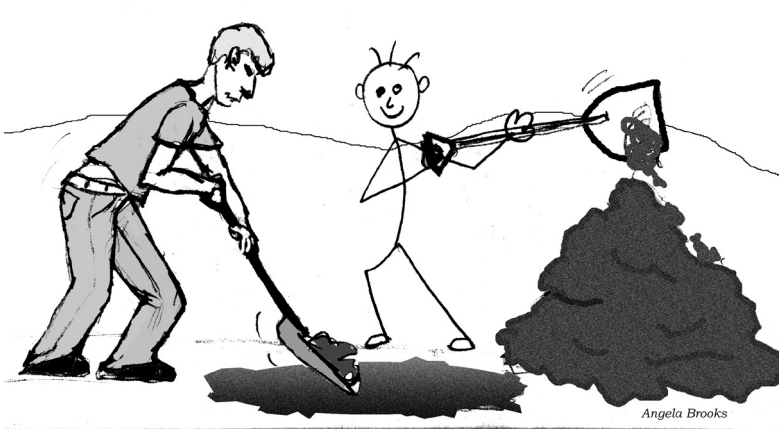
Библиография

[Fowler1999] Martin Fowler with Kendall Scott «UML Distilled: A Brief Guide to the Standard Object Modeling Language», 2-d ed., Addison-Wesley, 1999.[1]

[1] М. Фаулер «UML. Основы. Краткое руководство по стандартному языку объектного моделирования», 3-е изд. – Пер. с англ. – Символ-Плюс, 2006.

14

Работа с диаграммами



Прежде чем переходить к деталям языка UML, стоит поговорить о том, когда и почему его следует использовать. Найдется немало программных проектов, которым неправильное и чрезмерное применение UML принесло лишь вред.

Зачем нужно моделировать

Зачем вообще инженеры строят модели? Почему авиаконструкторы строят модели самолетов? А инженеры-строители – модели мостов? Для какой цели служат модели?

Инженер строит модель, чтобы выяснить, как будет работать конструкция. Авиаконструкторы строят модель самолета и помещают ее в аэродинамическую трубу, чтобы узнать, полетит ли самолет. Инженер-строитель создает модель моста, чтобы узнать, будет ли он стоять. Архитектор строит модель здания, чтобы выяснить, понравится ли оно заказчику.

Модели строятся для того, чтобы понять, будет ли некая конструкция работать.

Отсюда следует, что модели должны быть пригодны для испытаний. Не имеет смысла строить модель, если невозможно испытать ее работоспособность. Если модель нельзя оценить, то ее ценность равна нулю.

Почему бы авиаконструктору просто не построить самолет и не попытаться полететь на нем? Почему бы инженеру-строителю просто не возвести мост, а потом посмотреть, будет он стоять или нет? Да просто потому, что самолеты и мосты *гораздо* дороже моделей. *Мы изучаем проект на модели, когда модель обходится существенно дешевле, чем реальный предмет.*

Зачем строить модели программ

Можно ли испытать (протестировать) UML-диаграмму? Действительно ли создать и протестировать ее дешевле, чем описываемую ею программу? Ответ на оба вопроса совсем не так очевиден, как в случае конструирования самолетов и мостов. Не существует четких критериев для тестирования UML-диаграммы. Мы можем смотреть на нее, оценивать, применять к ней принципы и паттерны, но в конечном итоге оценка все равно остается субъективной. Рисовать UML-диаграммы дешевле, чем писать программу, но не в разы. На самом деле бывает, что исходный код изменить проще, чем диаграмму. Так когда же имеет смысл прибегать к UML?



Я бы не стал писать некоторые из этих глав, если бы использовать UML вообще не имело смысла. Однако UML очень просто применить неправильно. Мы пользуемся UML, когда имеется нечто определенное, что хотелось бы проверить, а применение UML обходится дешевле, чем написание кода. Предположим, что у меня имеется идея некоего дизайна. Я хочу понять, как ее оценят другие разработчики в моей команде. Тогда я рисую на доске UML-диаграмму и прошу коллег посмотреть на нее и высказаться.

Нужно ли проектировать систему до конца, прежде чем приступать к кодированию

Почему архитекторы, авиаконструкторы и инженеры-строители готовят чертежи? Потому, что один человек может нарисовать чертежи здания, которое будут возводить не меньше пяти человек. Потому, что несколько десятков авиаконструкторов могут создать чертежи самолета,

для строительства которого потребуется труд нескольких тысяч человек. Чтобы сделать чертеж, не нужно копать котлован под фундамент, заливать бетон или вставлять окна. Короче говоря, подготовить план здания заранее *гораздо* дешевле, чем пытаться построить его без плана. Ошибочный чертеж можно просто выкинуть, но снести неправильно построенное здание обойдется куда дороже.

Но, опять же, в области ПО все не так определенно. Вовсе не очевидно, что рисовать UML-диаграммы намного дешевле, чем писать код. И действительно, многие команды затратили на свои диаграммы *больше* усилий, чем на сам код. Вовсе не очевидно, что выбросить диаграмму намного дешевле, чем пожертвовать кодом. Поэтому вовсе не очевидно, что создание полного проекта на UML до начала кодирования – рентабельное предприятие.

Эффективное использование UML

Итак, по-видимому, архитектура, производство самолетов и проектирование сооружений не имеют очевидных аналогий в области разработки ПО. Мы не можем слепо использовать UML так, как в других дисциплинах применяются чертежи и модели (см. приложение B). Но когда и почему все же *следует* применять UML?

Диаграммы полезнее всего для общения с другими людьми и для выявления проектных проблем. Но для достижения этой цели важно использовать лишь строго необходимый уровень детализации. Перегрузить диаграмму множеством эффектных деталей можно, но нецелесообразно. Пусть ваши диаграммы будут простыми и понятными. UML-диаграммы – это не исходный код, на них не нужно изображать все без исключения методы, переменные и отношения.

Общение с другими людьми

UML необычайно удобен для распространения проектных концепций в среде разработчиков. Очень многого можно добиться, собрав небольшую группу разработчиков вокруг доски. Если вы хотите поделиться своими мыслями, то UML – как раз то, что надо.

UML очень хорош для передачи конкретных идей дизайна. Например, диаграмма на рис. 14.1 совершенно ясна. Мы видим, что класс `LoginPage` наследует классу `Page` и использует класс `UserDatabase`. Понятно, что классу `LoginPage` необходимы классы `HttpRequest` и `HttpResponse`. Легко представить себе, как группа разработчиков стоит около доски и обсуждает подобную диаграмму. На диаграмме отлично видно, как будет выглядеть структура программы.

С другой стороны, UML не особенно пригоден для передачи деталей алгоритма. Рассмотрим код простой пузырьковой сортировки в листинге 14.1. Выражение даже такого элементарного модуля на UML мало кого удовлетворит.

На рис. 14.2 приведено грубое представление структуры кода, но оно получилось громоздким и не отражает ни одной интересной детали. Рисунок 14.3 читать не проще, чем сам код, зато создать значительно труднее. В этой области UML оставляет желать лучшего.

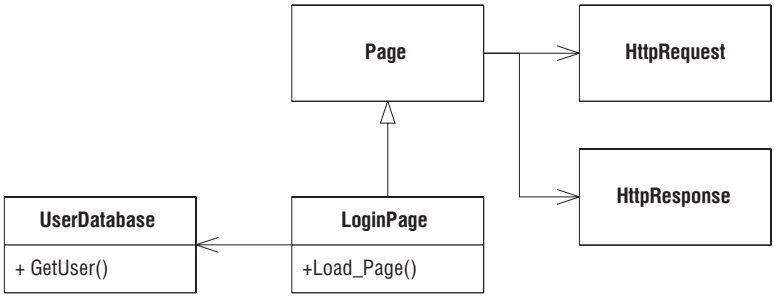


Рис. 14.1. *Класс LoginPage*

Листинг 14.1. BubbleSorter.cs

```
public class BubbleSorter
{
  private static int operations;

  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++;
  }
}
```

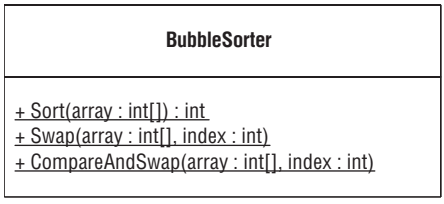


Рис. 14.2. *Класс BubbleSorter*

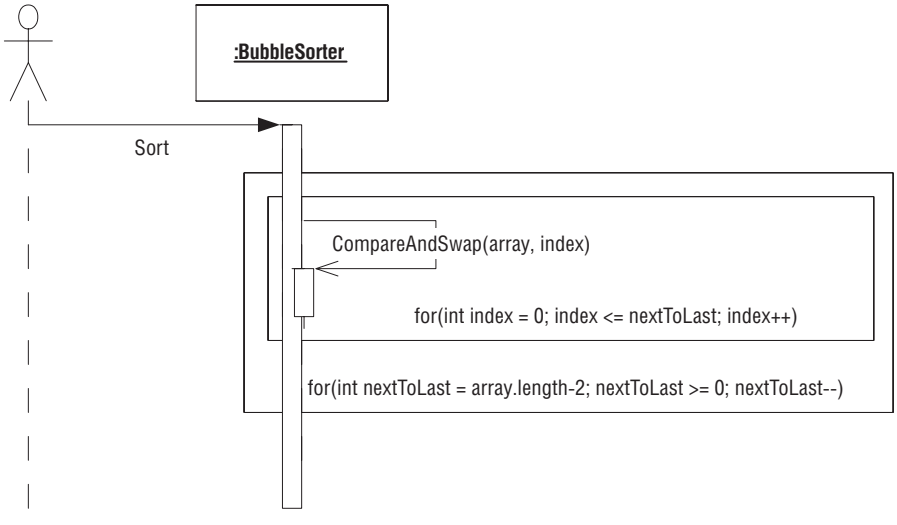
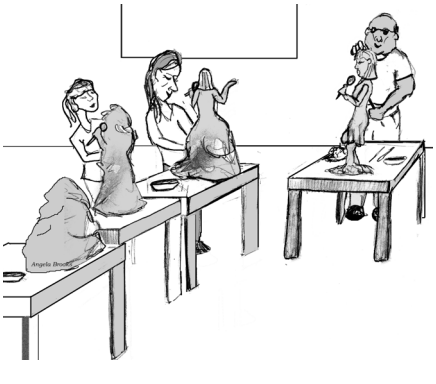


Рис. 14.3. *Диаграмма последовательности BubbleSorter*

Карты

UML может быть полезен для создания карт крупных программных структур. Такие карты позволяют разработчикам быстро определять, какие классы от каких зависят, и служат справочником по устройству системы в целом.

Например, на рис. 14.4 сразу видно, что у объектов `Space` есть атрибут `PolyLine`, составленный из нескольких объектов класса `Line`, производного от класса `LinearObject`, содержащего два объекта `Point`. На выявление этой структуры путем анализа кода ушло бы много времени, а с помощью диаграммы-карты это делается тривиально.



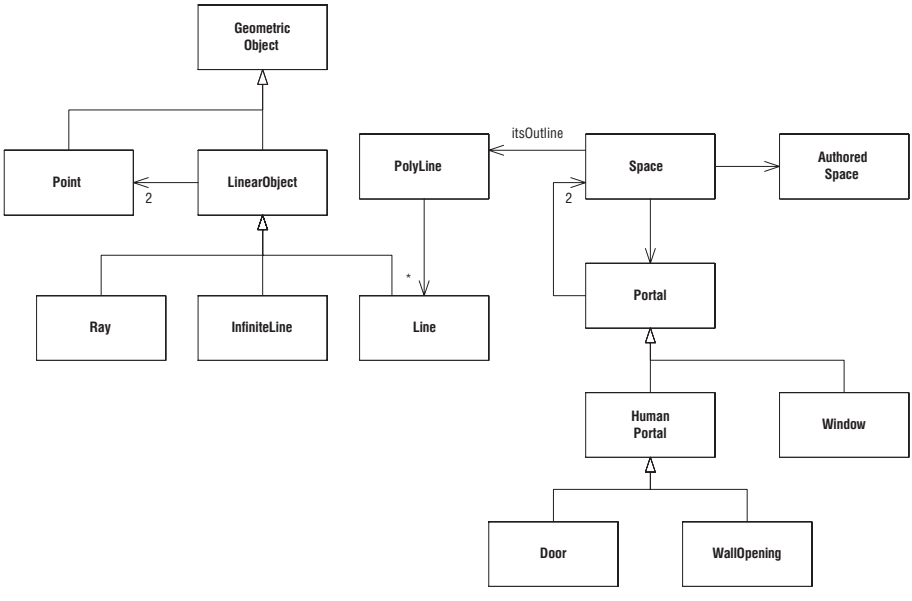


Рис. 14.4. *Диаграмма-карта*

Такие карты могут послужить полезным средством обучения. Однако каждый член команды должен быть способен нарисовать такую диаграмму на доске, даже если его разбудят посреди ночи. На самом деле я нарисовал диаграмму, показанную на рис. 14.4, по памяти, вспомнив систему, над которой работал десять лет назад. В таких диаграммах заключены знания, которые все разработчики должны держать в голове, чтобы эффективно работать. Поэтому обычно не имеет особого смысла прилагать много усилий для создания и архивирования подобных документов. Место им, как я еще раз повторяю, на доске.

Финальная документация

Самое подходящее время для создания проектной документации, которую вы хотели бы сохранить, – конец работы над проектом, последнее усилие команды. Такой документ точно отразит состояние проекта на момент прощания с ним команды и, безусловно, окажется полезен команде, которая придет на смену.

Но и тут есть свои подводные камни. UML-диаграммы следует тщательно отобрать. Нам ни к чему тысячи страниц с диаграммами последовательности! Мы хотим иметь несколько существенных диаграмм, описывающих основные особенности системы. Нет ничего хуже UML-диаграммы, загроможденной таким количеством стрелок и прямоугольников, что в этом переплетении можно потеряться (см. рис. 14.5).

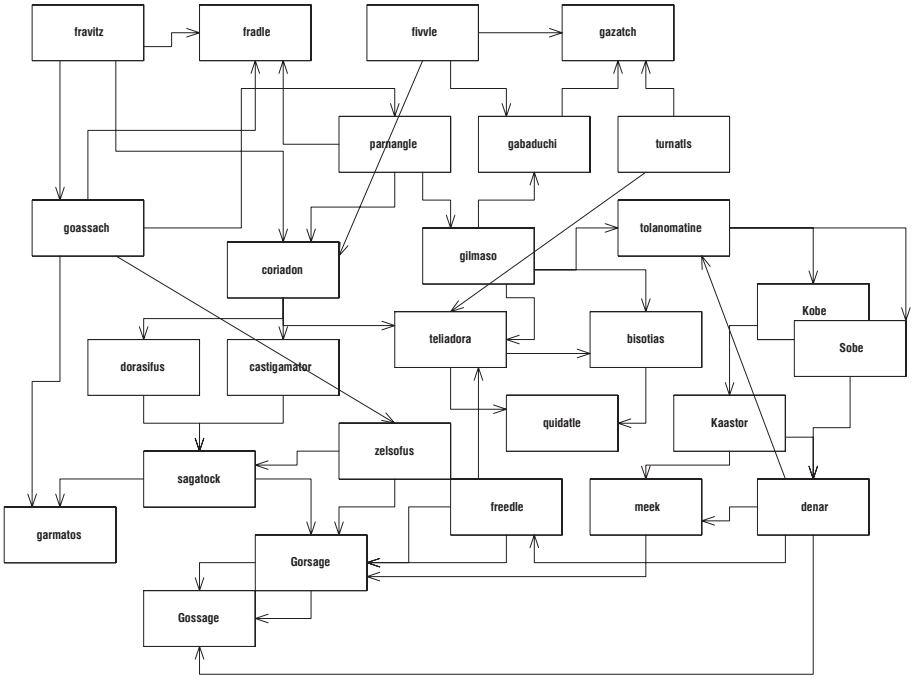


Рис. 14.5. *Плохой, но очень распространенный пример*

Что сохранить, а что выбросить

Возьмите за правило выбрасывать ненужные UML-диаграммы. А еще лучше, не создавайте их на постоянном носителе. Рисуйте на доске или на клочках бумаги. Почаще стирайте с доски и выбрасывайте эти обрывки. Не привыкайте к инструментарию CASE или к графическим редакторам. Для таких инструментов есть свое время и место, но жизнь большинства UML-диаграмм недолговечна.

Однако некоторые диаграммы полезно сохранять: те, что выражают общую идею дизайна системы. Сохраняйте диаграммы, документирующие сложные протоколы, которые трудно понять из кода. Диаграммы, являющиеся картами тех частей системы, в которые заглядывают относительно редко. Диаграммы, которые выражают намерение проектировщика лучше, чем это может сделать код.

Специально выискивать такие диаграммы не надо; достаточно их просто увидеть, и вы поймете – это оно. Не нужно пытаться создать такие диаграммы заранее. Вы будете гадать, и догадка окажется неправильной. Полезные диаграммы сами проявятся. Они будут возникать на доске или на клочках бумаги снова и снова. И в конце концов кто-то сделает постоянную копию такой диаграммы, чтобы не рисовать ее каж-

дый раз заново. Вот тогда-то и надо поместить ее в такое место, где все будут иметь к ней доступ.

Важно, чтобы эти общедоступные места были удобно организованы и не захламлены. Неплохо помещать диаграммы на веб-сервер или в сетевую базу знаний. Но следите за тем, чтобы там не скапливались сотни и тысячи диаграмм. Умейте отличать диаграммы, которые действительно полезны, от тех, что могут быть легко воспроизведены любым членом команды при необходимости. Сохраняйте лишь те, что представляют собой ценность в долгосрочной перспективе.

Итеративное уточнение

Как мы создаем UML-диаграммы? Рисуем в результате мгновенного озарения? Сначала рисуем диаграммы классов, а потом диаграммы последовательности? Набрасываем общую структуру системы, прежде чем переходим к деталям?

На все эти вопросы следует ответить решительным *нет*. Человеку удается выполнять свое дело хорошо, только продвигаясь вперед крохотными шажками, а потом оценивая сделанное. Все, что выполнено в результате нескольких гигантских прыжков, бывает сделано плохо. Мы хотим создавать полезные UML-диаграммы. Поэтому должны двигаться крохотными шажками.

Сначала поведение

Я предпочитаю начинать с поведения. Если мне кажется, что UML поможет в обдумывании задачи, то я сначала рисую простую диаграмму последовательности или кооперации. Рассмотрим, к примеру, программу управления мобильным телефоном. Как она производит телефонный вызов?

Можно было бы представить, что программа распознает нажатия кнопок и отправляет сообщения некоему объекту, который управляет набором номера. Поэтому нарисуем объекты `Button` и `Dialer` и покажем, что `Button` посылает `Dialer` много сообщений, содержащих по одной цифре (рис. 14.6). (Звездочка означает *много*.)

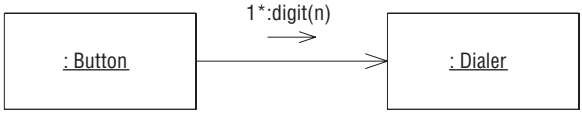


Рис. 14.6. *Простая диаграмма последовательности*

Что будет делать `Dialer`, получив очередное сообщение с цифрой? Выведет ее на экран. Так что, наверное, нужно послать сообщение `displayDigit` объекту `Screen` (рис. 14.7).

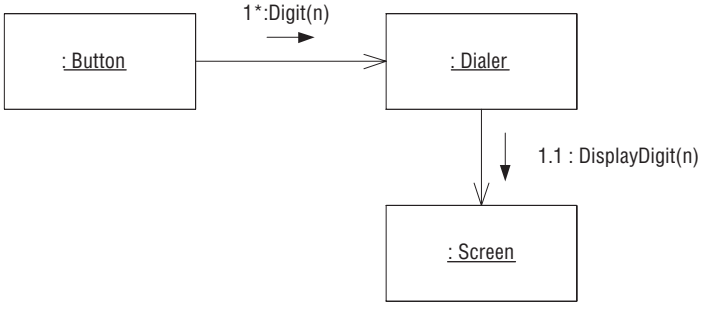


Рис. 14.7. *Продолжение рис. 14.6*

Затем `Dialer` должен подать звуковой сигнал в динамик. Следовательно, нужно отправить сообщение `tone` объекту `Speaker` (рис. 14.8).

В какой-то момент пользователь нажимает кнопку `Send`, показывая, что пора вызывать абонента. В ответ мы должны сказать сотовому радиопередатчику, чтобы он соединился с сетью и передал набранный номер (рис. 14.9).

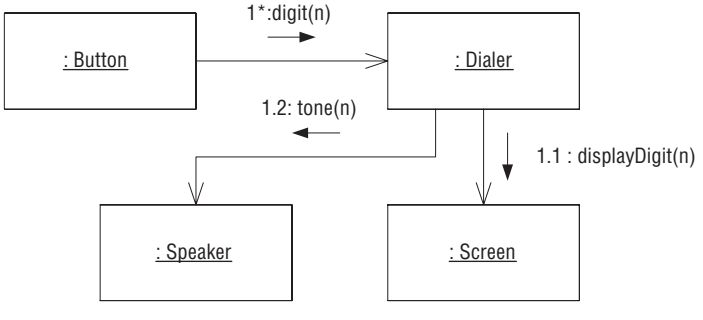


Рис. 14.8. *Продолжение рис. 14.7*

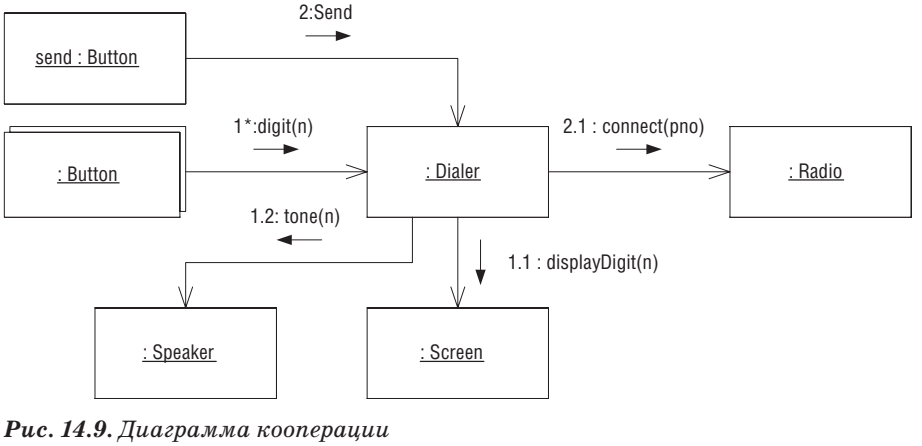


Рис. 14.9. *Диаграмма кооперации*

После того как соединение установлено, объект Radio может попросить объект Screen зажечь индикатор «работает». Это сообщение почти наверняка будет отправлено в другом потоке управления, что обозначается буквой перед порядковым номером. Окончательная диаграмма кооперации показана на рис. 14.10.

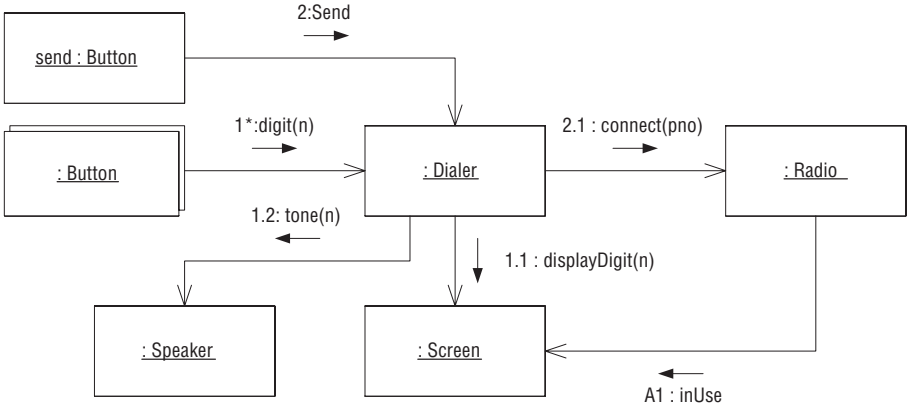


Рис. 14.10. *Диаграмма кооперации для мобильного телефона*

Проверяем структуру

Это небольшое упражнение показало, как из ничего строится диаграмма кооперации. Попутно мы ввели в обращение ряд объектов. Мы не знали заранее, понадобятся ли эти объекты; известен был лишь общий порядок действий, а объекты мы придумали для того, чтобы эти действия выполнить.

Но прежде чем продолжить, необходимо изучить описанную кооперацию и понять, что она означает с точки зрения структуры кода. Поэтому создадим диаграмму классов (рис. 14.11), поддерживающих такую кооперацию. На этой диаграмме все объекты, участвующие в кооперации, будут представлены классами, а все связи – ассоциациями.

Читатели, знакомые с UML, наверное, заметили, что мы проигнорировали агрегирование и композицию. Это неслучайно. У нас еще будет время поговорить о том, когда применяются эти отношения.

А сейчас меня интересует анализ зависимостей. Почему класс Button должен зависеть от Dialer? Поразмыслив, вы поймете, что это никуда не годится. Взгляните, как мог бы выглядеть вытекающий из этих диаграмм код:

```
public class Button
{
  private Dialer itsDialer;
  public Button(Dialer dialer)
```

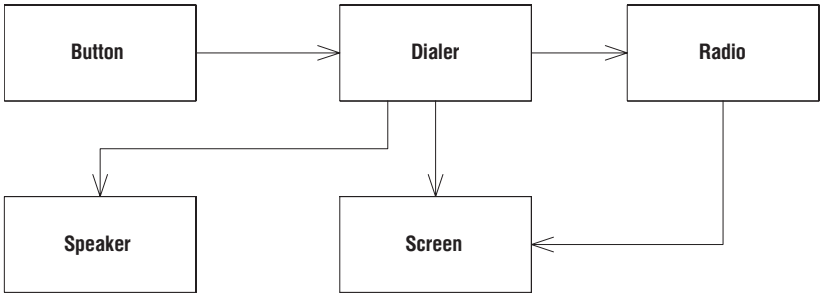


Рис. 14.11. *Диаграмма классов для мобильного телефона*

```
    {itsDialer = dialer;}
    ...
}
```

Я не хочу, чтобы в исходном коде класса `Button` упоминался класс `Dialer`. `Button` – класс, который я мог бы использовать в самых разных контекстах. Например, для управления выключателем, или для кнопки в меню, или для других управляющих кнопок на телефоне. Привязав `Button` к `Dialer`, я уже не смогу повторно использовать его для других целей.

Эту проблему можно решить, вставив интерфейс между классами `Button` и `Dialer`, как показано на рис. 14.12. Обнаружив, что кнопка нажата, класс `Button` вызовет метод `buttonPressed` интерфейса `ButtonListener`, передав ему идентификатор кнопки `token`. Тем самым мы разрываем зависимость `Button` от `Dialer` и получаем возможность использовать `Button` везде, где необходимо получать информацию о нажатии кнопок.

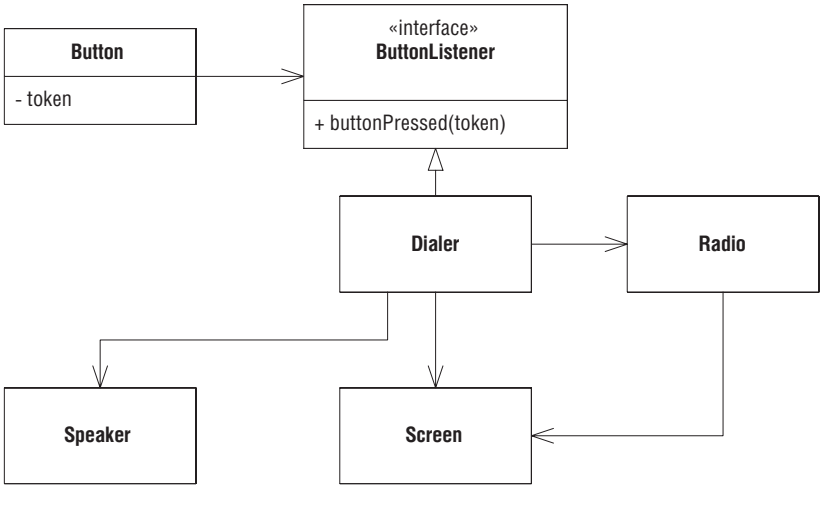
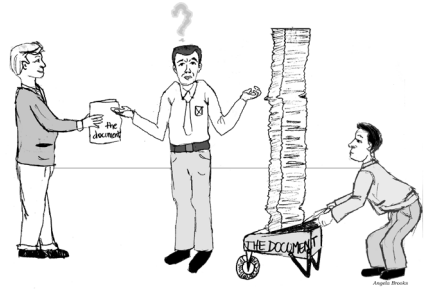


Рис. 14.12. *Изолирование Button от Dialer*



Отметим, что это изменение никак не затронуло динамическую диаграмму на рис. 14.10. Все объекты те же самые, изменились только классы.

К сожалению, теперь `Dialer` знает о `Button`. Почему класс `Dialer` должен ожидать входных данных от `ButtonListener`? Почему в нем должен быть метод с именем `buttonPressed`? Что общего между `Dialer` и `Button`?

Мы можем решить и эту проблему и избавиться от всяких идентификаторов кнопок, воспользовавшись рядом адаптеров (рис. 14.13). Класс `ButtonDialerAdapter` реализует интерфейс `ButtonListener`, получая данные от `buttonPressed` и посылая сообщение `digit(n)` классу `Dialer`. Цифра, передаваемая `Dialer`, хранится в самом адаптере.

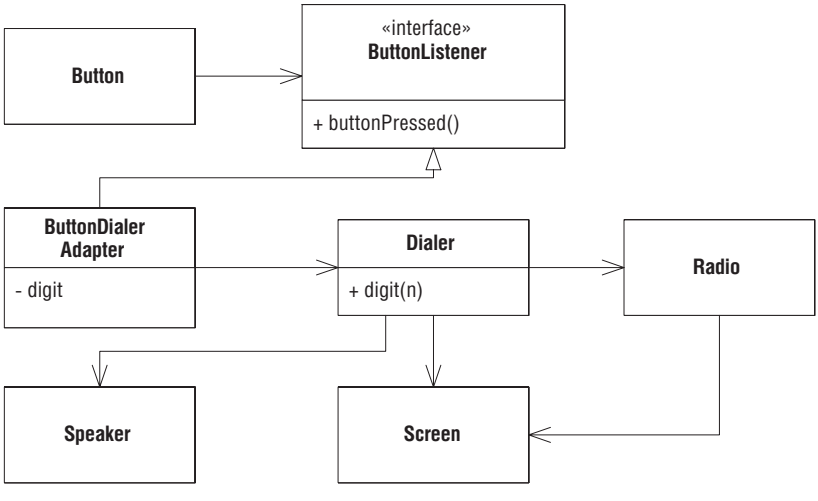


Рис. 14.13. *Адаптация класса Button к классу Dialer*

Набрасываем код

Легко представить себе, как мог бы выглядеть код класса `ButtonDialerAdapter`. Он показан в листинге 14.2. Держать в уме код *критически* важно при работе с диаграммами. Диаграммы – это схематическое представление кода, а не его замена. Рисовать диаграммы, забывая о представляемом ими коде, – все равно что строить воздушные замки. *Остановитесь и обдумайте, как перевести диаграмму на язык кода.* Не допускайте, чтобы диаграммы становились вещью в себе. Вы должны быть уверены, что знаете, какой код за ними стоит.

Листинг 14.2. ButtonDialerAdapter.cs

```
public class ButtonDialerAdapter : ButtonListener
{
  private int digit;
  private Dialer dialer;

  public ButtonDialerAdapter(int digit, Dialer dialer)
  {
    this.digit = digit;
    this.dialer = dialer;
  }

  public void ButtonPressed()
  {
    dialer.Digit(digit);
  }
}
```

Эволюция диаграмм

Отметим, что последнее произведенное на рис. 14.13 изменение сделало недействительной динамическую модель на рис. 14.10. Динамическая модель ничего не знает об адаптерах. Исправим это упущение.

На рис. 14.14 показана совместная итеративная эволюция диаграмм. Мы начинаем с динамики. Затем смотрим, как новая динамика отражается на статических отношениях. Потом изменяем статические отношения, руководствуясь принципами правильного проектирования. Затем возвращаемся и улучшаем динамические диаграммы.

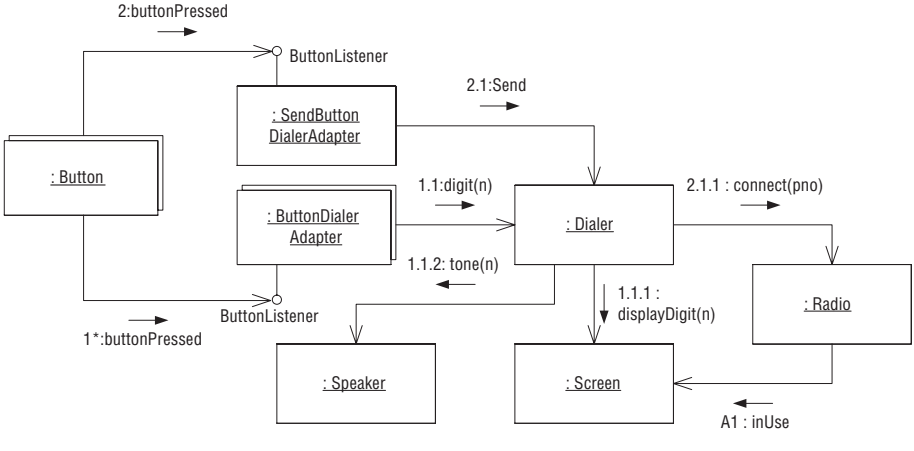


Рис. 14.14. *Добавление адаптеров в динамическую модель*

Каждый шажок был *крохотным*. Мы хотим тратить не больше *пяти минут* на динамическую диаграмму перед тем, как оценивать предпо-

лагаемую статическую структуру. И не больше пяти минут на уточнение статической структуры перед оценкой ее влияния на динамическое поведение. Обе диаграммы должны эволюционировать совместно очень короткими циклами.

Напомним, что все это лучше делать на доске, и мы не сохраняем плоды своих трудов для потомства. Мы не стремимся к особому формализму или точности. На самом деле показанные на предыдущих рисунках диаграммы даже чуть более точны и формальны, чем обычно требуется. Цель состоит не в том, чтобы правильно расставить все точки в порядковых номерах, а в том, чтобы стоящие у доски поняли смысл решения. Наша задача – отойти от доски и начать писать код.

Когда и как рисовать диаграммы

Рисование UML-диаграмм может быть очень полезным занятием. Или бесполезной тратой времени. Решение прибегнуть к UML может оказаться очень хорошим или очень плохим. Все зависит от того, как и в каком объеме вы будете его использовать.

Когда рисовать диаграммы и когда остановиться

Не берите за правило изображать на диаграммах все и вся. Такие правила более чем бесполезны. Масса времени и энергии может быть впустую растрачена на подготовку диаграмм, которые никто не будет читать.

Рисуйте диаграммы в следующих случаях.

- Несколько человек должны понимать структуру конкретной части дизайна, поскольку будут работать над ней одновременно. Остановитесь, когда общее понимание достигнуто.
- Необходимо добиться консенсуса, но хотя бы два человека не согласны с конкретным элементом дизайна. Ограничьте время обсуждения, а затем определите способ принятия решения, например голосование или мнение беспристрастного арбитра. Остановитесь, когда время истекло или решение уже можно принять. Затем сотрите диаграмму.
- Возникает потребность мысленно визуализировать идею дизайна, и диаграммы могут помочь в ее обдумывании. Остановитесь, когда понимаете, что закончить мысль уже можно в коде. Выбросьте диаграммы.
- Требуется объяснить структуру части кода самому себе или кому-то еще. Остановитесь, когда поймете, что объяснять лучше, глядя на код.
- Близится конец проекта, и заказчик попросил представить диаграммы как часть документации для третьих лиц.

Не рисуйте диаграммы:

- Только потому, что так велит процесс.
- Потому, что вы чувствуете себя виноватым в том, что не рисуете их, или думаете, что так поступают все хорошие проектировщики. Хорошие проектировщики пишут код. Диаграммы они рисуют только по мере необходимости.
- Чтобы создать полную документацию этапа проектирования перед тем, как приступить к кодированию. Такие документы почти всегда оказываются бесполезны, зато отнимают немало времени.
- Чтобы по ним могли кодировать другие люди. Настоящие архитекторы программ принимают непосредственное участие в кодировании того, что спроектировали.

CASE-средства

CASE-средства для UML могут принести пользу или стать дорогостоящими собирателями пыли. Очень тщательно подходите к решению о приобретении и развертывании CASE-средства для UML.

- *Разве CASE-средства не помогают рисовать UML-диаграммы?* Нет, они лишь существенно затрудняют это занятие. Чтобы стать квалифицированным пользователем, нужно долго учиться, и даже тогда эти средства оказываются более громоздкими, чем доска, которой пользоваться совсем просто. С этим инструментом разработчики обычно уже знакомы, а если нет, то освоят его мгновенно.
- *Разве CASE-средства не упрощают совместную работу над диаграммами в больших командах?* Иногда да. Но подавляющее большинство разработчиков в подавляющем большинстве проектов и не должны порождать диаграммы в таких количествах и такой сложности, чтобы для координации их деятельности понадобилась автоматизированная система общего пользования. В любом случае покупать систему для координации подготовки UML-диаграмм имеет смысл только тогда, когда уже внедренная ручная система становится недостаточной и единственным выбором оказывается автоматизация.
- *Разве CASE-средства не упрощают генерацию кода?* Суммарные усилия, потраченные на создание диаграмм, генерацию кода и последующее использование сгенерированного кода, вряд ли окажутся меньше, чем усилия, потраченные просто на написание кода с самого начала. Если выигрыш и есть, то не на порядок и даже не в два раза. Разработчики знают, как редактировать текстовые файлы и пользоваться IDE. Генерация кода по диаграммам может показаться неплохой идеей, но я настоятельно рекомендую оценить ее продуктивность, прежде чем тратить на нее баснословные деньги.

- *Как насчет CASE-средств, которые одновременно являются IDE и показывают код и диаграммы вместе?* Да, это солидные инструменты. Однако постоянное присутствие UML не так важно. Тот факт, что диаграмма изменяется по ходу модификации кода или код модифицируется при внесении изменений в диаграмму, мне не очень-то помогает. Честно говоря, я предпочел бы купить IDE, которая всеми силами помогает мне манипулировать программой, а не диаграммами. Повторю, оценивайте продуктивность, прежде чем идти на солидные капиталовложения.

Короче говоря, прежде чем резать, семь раз отмерьте. Или семь раз по семь. *Может быть* так, что экипировка команды дорогостоящим CASE-средством принесет пользу, но оцените эту пользу исходя из собственного опыта, прежде чем покупать нечто, вполне способное превратиться в «полочную программу».

Ну а как насчет документации?

Хорошая документация важна для любого проекта. Без нее команда не сможет ориентироваться в море кода. С другой стороны, если документации слишком много, к тому же не той, что нужно, то это только хуже, поскольку помимо моря кода у вас на руках оказывается гора отвлекающей и дезориентирующей бумаги.

Документацию создавать нужно, но делать это следует с умом. Решение о том, что *не документировать*, не менее важно, чем о том, что *документировать*. Сложные протоколы взаимодействия документировать необходимо. Как и сложные реляционные схемы. Как и сложный повторно используемый каркас. Но для всего этого не надо рисовать сотни страниц UML-диаграмм. Программная документация должна быть *краткой и по существу*. Ценность документа обратно пропорциональна его объему.

Для команды из 12 человек, работающей над проектом в миллион строк кода, я бы порекомендовал от 25 до 200 страниц долгосрочной документации, отдавая предпочтение нижней границе. В состав этой документации следует включить UML-диаграммы высокоуровневой структуры важнейших модулей, диаграммы сущность–связь (ER-диаграммы) реляционной схемы, одну-две страницы о сборке системы, инструкции по тестированию, инструкции по работе с системой управления версиями исходного кода и т. д. Я поместил бы документацию в какую-нибудь вики-систему[1] или иное средство совместной работы над документами, чтобы любой член команды мог просматривать ее на экране и при необходимости вносить изменения.

[1] Сетевое средство совместной работы над документами. См. *http://c2.com* или *http://fitnesse.org*.

Для составления небольшого по объему документа, возможно, придется как следует поработать, но это оправданные усилия. Небольшие документы люди читают. Тома на 1000 страниц – нет.

Заключение

Небольшая группа людей, собравшаяся у доски, может использовать UML для обдумывания задачи проектирования. Такие диаграммы создаются итеративно, очень короткими циклами. Сначала лучше исследовать динамику работы системы, а уже потом решать, как это отражается на ее статической структуре. Важно изменять динамические и статические диаграммы одновременно, затрачивая на каждую итерацию не более пяти минут.

CASE-средства для UML иногда могут оказаться полезны. Но для обычной команды разработчиков они будут скорее помехой, нежели подмогой. Если вам кажется, что возникла необходимость в таком средстве, пусть даже интегрированном с IDE, сначала проведите эксперименты по оценке продуктивности. Отмеряйте, прежде чем резать.

UML – это инструмент, а не самоцель. В таком качестве он может помочь при обдумывании дизайна и общении с другими людьми. Если не переусердствовать, то он станет благом. Но стоит чрезмерно увлечься, и вы только потеряете массу времени. Применяя UML, *двигайтесь потихоньку*.

15

Диаграммы состояний



UML располагает развитой нотацией для описания конечных автоматов (КА). В этой главе мы ознакомимся с ее наиболее полезными возможностями. КА – чрезвычайно удобное средство для написания самых разных программ. Я пользовался ими при реализации графических интерфейсов пользователя (ГИП), коммуникационных протоколов и других событийно-ориентированных систем. К сожалению, есть очень много разработчиков, не знакомых с идеями КА и потому упускающих возможность упростить свои программы. В этой главе я попробую внести скромный вклад в исправление такой ситуации.

Основные понятия

На рис. 15.1 показана простая *диаграмма переходов состояний* (ДПС), которая описывает КА, управляющий входом пользователя в систему. Скругленными прямоугольниками представлены *состояния*. Имя состояния записывается в верхнем отделении. В нижнем отделении записываются специальные действия, говорящие о том, что нужно сделать при входе в данное состояние и при выходе из него. Например, при входе в состояние `Приглашение к входу` вызывается действие `showLoginScreen`, и при выходе из этого состояния – действие `hideLoginScreen`.

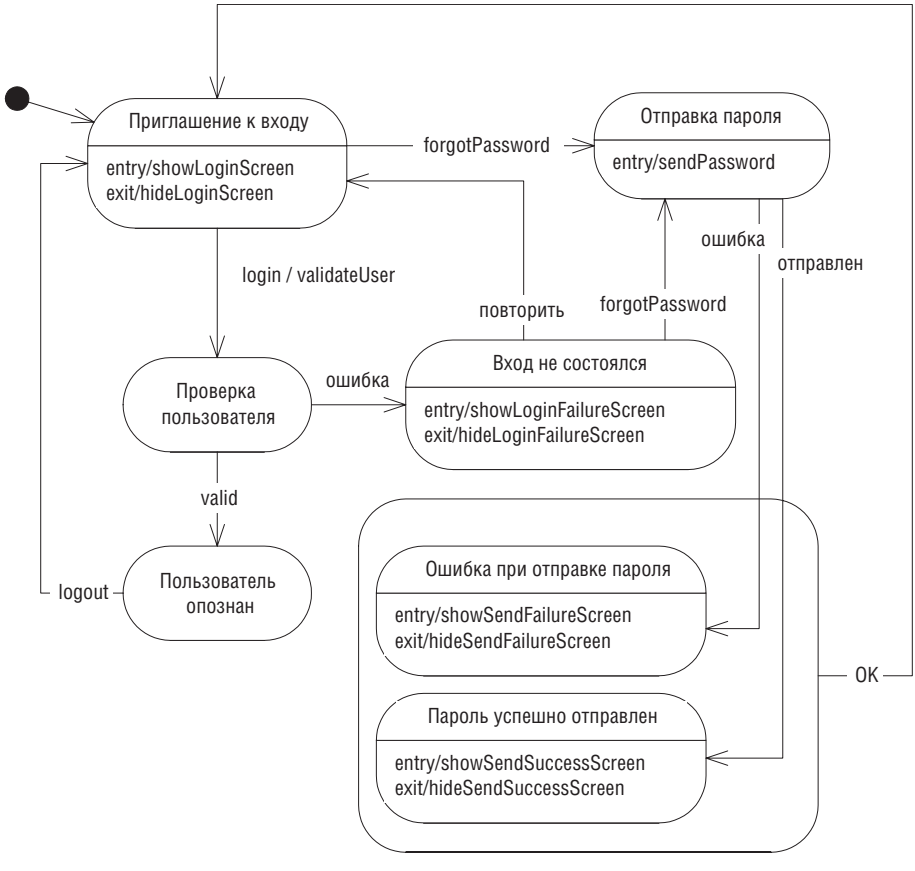


Рис. 15.1. *Простой конечный автомат, описывающий вход в систему*

Стрелки, соединяющие состояния, называются *переходами*. Каждый переход помечен именем вызвавшего его события, а некоторые – также действием, которое выполняется при срабатывании перехода. Например, если, находясь в состоянии `Приглашение к входу`, мы получаем событие `login`, то переходим в состояние `Проверка пользователя` и вызываем действие `validateUser`.



Черный кружок в левом верхнем углу диаграммы называется *начальным псевдосостоянием*. КА начинает работу переходом из этого псевдосостояния. Таким образом, в самом начале работы наш конечный автомат оказывается в состоянии `Приглашение к входу`.

Я нарисовал *суперсостояние* вокруг состояний `Ошибка при отправке пароля` и `Пароль успешно отправлен`, потому что оба эти состояния реагируют на событие `OK` переходом в состояние `Приглашение к входу`. Поскольку я не хотел рисовать две одинаковых стрелки, то воспользовался этой удобной возможностью.

Показанный КА наглядно объясняет, как работает процедура входа в систему, и разбивает ее на небольшие функции. Если реализовать все функции действий — `showLoginScreen`, `validateUser` и `sendPassword`, — а затем объединить их, следуя представленной на диаграмме логике, то есть гарантия, что процедура входа будет работать правильно.

Специальные события

В нижнем отделении прямоугольника состояния находятся пары событие/действие. События `entry` (вход) и `exit` (выход) стандартные, но, как видно из рис. 15.2, можно включать и свои собственные события. Если какое-нибудь из таких событий происходит, когда КА находится в данном состоянии, то вызывается соответствующее действие.

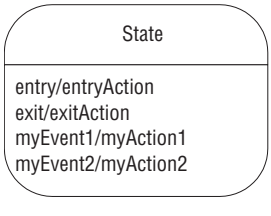


***Рис. 15.2.** Изображение состояний и специальных событий в UML*

До появления UML я обычно представлял специальное событие стрелкой перехода, которая следует в то же состояние, из которого выходит (рис. 15.3). Однако в UML у такой нотации несколько иная семантика. Любой переход, выходящий из некоторого состояния, приводит к вызову действия `exit`, если таковое задано. Аналогично любой переход, входящий в состояние, приводит к вызову действия `entry`, если таковое задано. Следовательно, в UML рефлексивный переход типа изображенного на рис. 15.3 вызовет не только действие `myAction`, но также действия `exit` и `entry`.

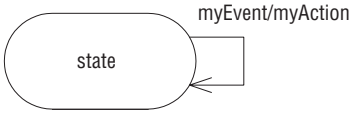


Рис. 15.3. *Рефлексивный переход*

Суперсостояния

Как мы уже видели на примере КА входа в систему (см. рис. 15.1), суперсостояния удобны, если имеется несколько состояний, одинаково реагирующих на одни и те же события. Все такие состояния можно поместить внутрь одного суперсостояния, а стрелки переходов нарисовать так, чтобы они выходили из суперсостояния, а не из отдельных состояний. Поэтому диаграммы на рис. 15.4 эквивалентны.

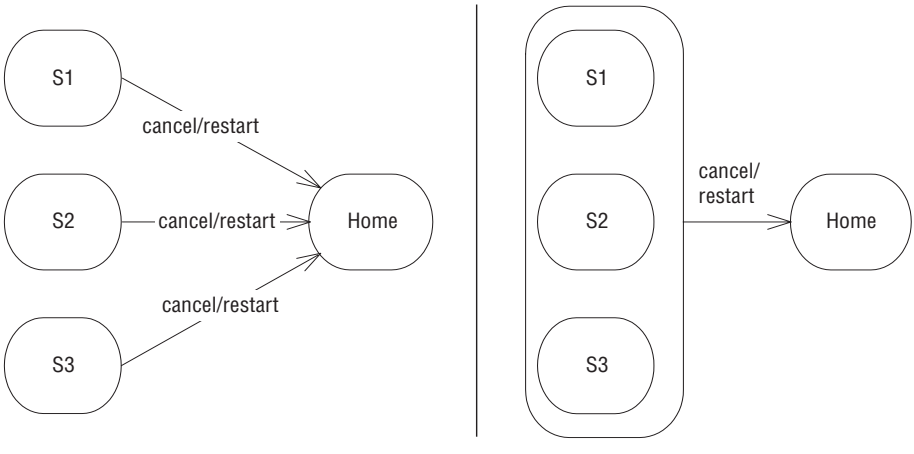
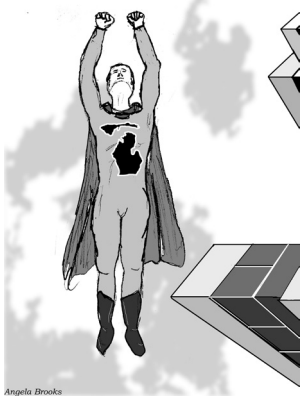


Рис. 15.4. *Переход: несколько состояний и одно суперсостояние*

Переходы из суперсостояний можно переопределять, для этого достаточно нарисовать явный переход из подсостояния. Так, на рис. 15.5 переход `pause` из состояния `S3` отменяет подразумеваемый по умолчанию переход `pause` из суперсостояния `Cancelable`. В этом смысле суперсостояние ведет себя, как базовый класс. Подсостояния могут переопределять переходы из своих суперсостояний точно так же, как производный класс может переопределять методы базового класса. Однако не рекомендуется заходить в этой метафоре слишком далеко. Отношение между суперсостоянием и подсостоянием все же не эквивалентно наследованию.



Для суперсостояний можно задавать события `entry`, `exit` и специальные события точно так же, как для обычных состояний. На рис. 15.6 показан КА, в котором действия для событий `exit` и `entry` заданы как для суперсостояния, так и для его подсостояний. При переходе из состояния `Some State` в состояние `Sub` КА сначала вызовет действие `enterSuper`, а затем действие `enterSub`. Аналогично при переходе из состояния `Sub2` в `Some State` сначала будет вызвано действие `exitSub2`, а потом `exitSuper`. Но при переходе `e2` из `Sub` в `Sub2` вызываются только действия `exitSub` и `enterSub2`, так как мы не покидали суперсостояние.

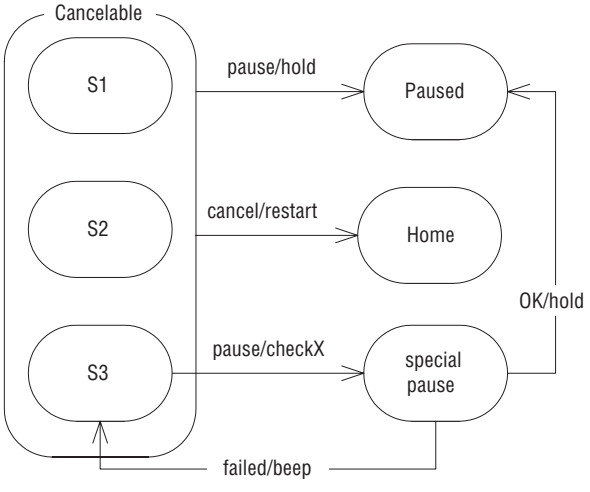


Рис. 15.5. *Переопределение переходов из суперсостояния*

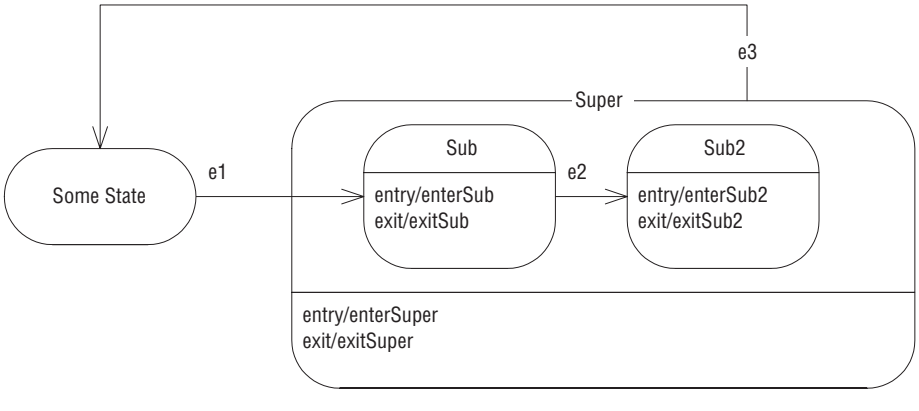


Рис. 15.6. *Иерархические вызовы действий при входе и выходе*

Начальное и конечное псевдосостояния

На рис. 15.7 показаны два часто встречающихся в UML псевдосостояния. Любой конечный автомат начинает свое существование *в процессе* выхода из начального псевдосостояния. С переходом из начального псевдосостояния не могут быть связаны события, потому что событием является само создание КА. Однако для такого перехода можно задать действие. Это будет первое действие, вызываемое после создания КА.

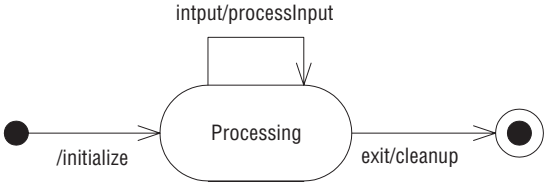


Рис. 15.7. *Начальное и конечное псевдосостояния*

Аналогично КА прекращает существование в процессе перехода в *конечное псевдосостояние*. Это псевдосостояние на самом деле никогда не достигается. Если для этого перехода определено какое-то действие, то оно будет последним действием, вызываемым КА.

Использование диаграмм состояний

Я считаю, что подобные диаграммы исключительно полезны для построения конечных автоматов подсистем, поведение которых хорошо известно. С другой стороны, поведение большинства систем, описываемых с помощью КА, заранее не известно. Оно расширяется и эволюционирует со временем. Диаграммы – не самое подходящее средство для представления часто изменяющихся подсистем. Начинают возникать проблемы с расположением отдельных элементов на рисунке и нехваткой места. Иногда такие затруднения заставляют проектировщиков отказываться от внесения необходимых изменений в дизайн. Сама мысль о том, что придется переформатировать диаграмму, выглядит настолько пугающей, что они предпочитают отказаться от добавления нужного класса или состояния и смириться с неоптимальным решением, лишь бы не трогать диаграмму.

С другой стороны, текст – очень гибкое средство, когда речь идет об изменениях. Проблем с расположением почти нет, места для добавления новых строк всегда хватает. Поэтому для описания развивающихся систем я создаю не ДПС, а таблицы переходов состояний (ТПС) в виде текстовых файлов. Взгляните на ДПС турникета в метро, изображенную на рис. 15.8. Ее можно без труда представить в виде ТПС, как показано в табл. 15.1.

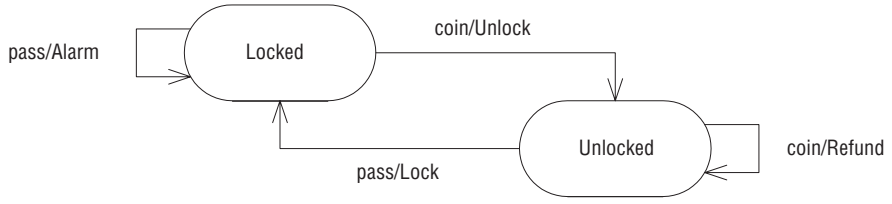


Рис. 15.8. *ДПС турникета в метро*

Таблица 15.1. ТПС турникета в метро

Текущее состояние	Событие	Новое состояние	Действие
`Locked`	`coin`	`Unlocked`	`Unlock`
`Locked`	`pass`	`Locked`	`Alarm`
`Unlocked`	`coin`	`Unlocked`	`Refund`
`Unlocked`	`pass`	`Locked`	`Lock`

Эта ТПС представляет собой простую таблицу с четырьмя столбцами. Каждая строка представляет переход. Посмотрите на стрелки переходов на диаграмме. Вы увидите, что в строке таблицы описаны оба конца стрелки, а также связанные с ней событие и действие. Читать ТПС нужно следующим образом: «Если мы находимся в состоянии `Locked` и получаем событие `coin`, то переходим в состояние `Unlocked` и вызываем функцию `Unlock`».

Таблицу очень легко преобразовать в текстовый файл:

```
Locked          coin      Unlocked       Unlock
Locked          pass      Locked         Alarm
Unlocked        coin      Unlocked       Refund
Unlocked        pass      Locked         Lock
```

Вот эти 16 слов и описывают всю логику КА.

В 1989 году я написал простую программу SMC (State Machine Compiler – компилятор конечных автоматов), которая читает ТПС и генерирует код на C++, реализующий логику КА. С тех пор SMC разрослась и теперь позволяет генерировать код на разных языках. Поближе мы ознакомимся с SMC в главе 36 при обсуждении паттерна Состояние (State). Программа SMC бесплатна, скачать ее можно из раздела ресурсов на сайте *www.objectmentor.com*.

Создавать и поддерживать КА в такой форме гораздо проще, чем рисовать диаграммы, а генерация кода экономит массу времени. Диаграммы очень полезны для обдумывания или демонстрации КА другим людям, но в процессе разработки текст гораздо удобнее.

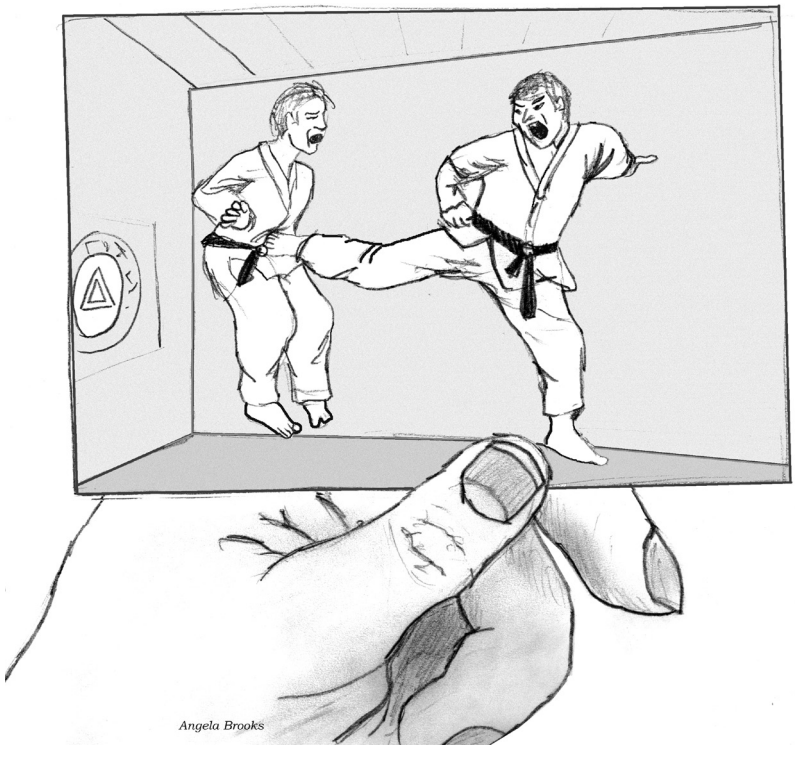
Заключение

Конечные автоматы – мощное средство структурирования программ. UML предлагает очень выразительную нотацию для визуализации КА. Но часто описывать и сопровождать КА проще не в виде диаграмм, а с помощью простого текстового языка.

Нотация диаграмм состояний в UML гораздо богаче, чем я рассказал. Существует еще несколько псевдосостояний, значков и других элементов. Но я редко находил им применение. Описанная в этой главе нотация – все, что мне когда-либо приходилось использовать.

16

Диаграммы объектов



Иногда полезно показать состояние системы в конкретный момент времени. Для этого применяются UML-диаграммы объектов, на которых представлены объекты, их связи и значения их атрибутов в данный момент.

Мгновенный снимок

Когда-то я принимал участие в разработке приложения, предоставлявшего графический интерфейс для рисования плана этажа. Программа сохраняла информацию о комнатах, дверях, окнах и стенных проемах в структуре данных, показанной на рис. 16.1. Хотя из этой диаграммы видно, какие бывают объекты вообще, она ничего не говорит об объектах и связях между ними, существующих в конкретный момент времени.

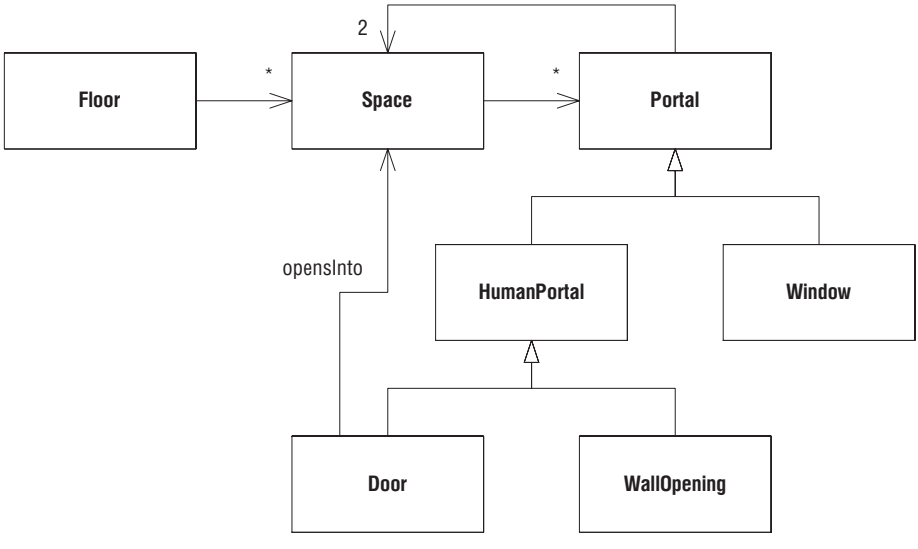


Рис. 16.1. *План этажа*

Допустим, что пользователь нарисовал две комнаты – кухню и столовую, соединенные проемом. В кухне и столовой есть окна. В столовой также есть дверь, ведущая наружу. Эта ситуация изображена на диаграмме объектов на рис. 16.2. Здесь показано, какие объекты имеются в системе и как они связаны с другими объектами. Мы видим, что `kitchen` (кухня) и `lunchRoom` (столовая) – экземпляры класса `Space`. Видно также, что эти две комнаты соединены проемом. Внешнее пространство представлено еще одним экземпляром – `Space`. Показаны также все остальные объекты и связи между ними.

Подобные диаграммы объектов полезны, когда нужно показать, как выглядит внутренняя структура системы в определенный момент времени или в определенном состоянии. Диаграмма объектов отражает намерение проектировщика. Она показывает, как предполагается использовать те или иные классы и отношения. С ее помощью можно понять, как будет изменяться система при получении различных входных данных.

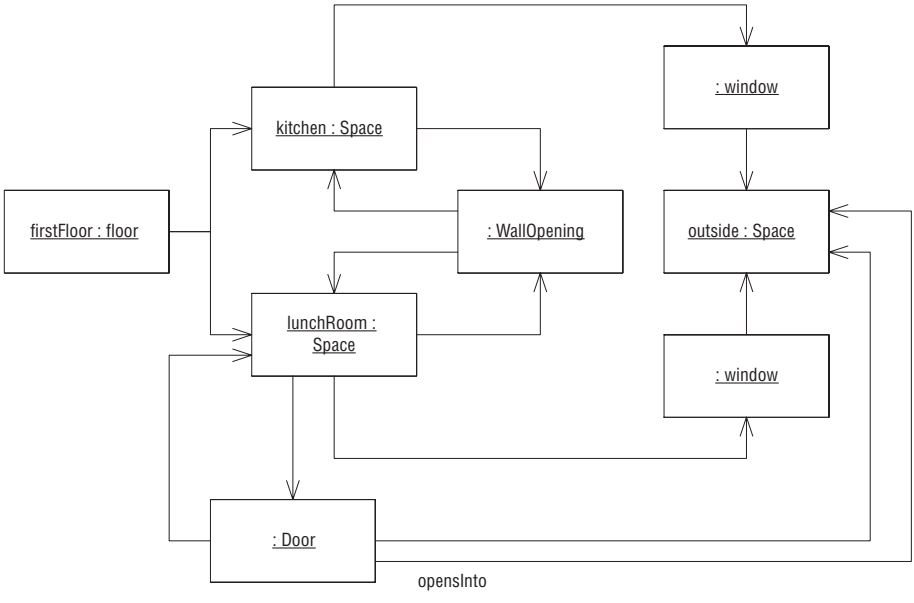
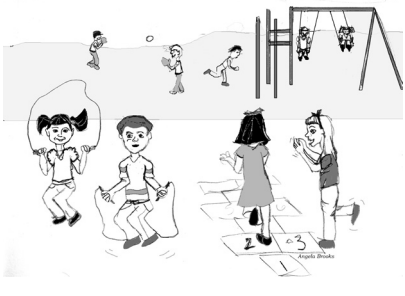


Рис. 16.2. *Столовая и кухня*

Но будьте осторожны, не слишком увлекайтесь. За последние десять лет я, пожалуй, не нарисовал и дюжины таких диаграмм. Просто нужда в них возникает нечасто. Но когда возникает, тогда они незаменимы, поэтому я и решил включить их в книгу. Однако не думайте, что их необходимо рисовать для каждого возникающего в системе сценария и во всех проектируемых вами системах.

Активные объекты



Диаграммы объектов полезны также в многопоточных системах. Рассмотрим, к примеру, код класса `SocketServer` в листинге 16.1. В нем реализован простой каркас, позволяющий писать серверы на базе сокетов, не забивая голову вопросами организации потоков и синхронизации, которые являются неизбежными спутниками сокетов.

Листинг 16.1. SocketServer.cs

```
using System.Collections;
using System.Net;
using System.Net.Sockets;
using System.Threading;
```

```
namespace SocketServer
{
  public interface SocketService
  {
    void Serve(Socket s);
  }

  public class SocketServer
  {
    private TcpListener serverSocket = null;
    private Thread serverThread = null;
    private bool running = false;
    private SocketService itsService = null;
    private ArrayList threads = new ArrayList();

    public SocketServer(int port, SocketService service)
    {
      itsService = service;
      IPAddress addr = IPAddress.Parse("127.0.0.1");
      serverSocket = new TcpListener(addr, port);
      serverSocket.Start();
      serverThread = new Thread(new ThreadStart(Server));
      serverThread.Start();
    }

    public void Close()
    {
      running = false;
      serverThread.Interrupt();
      serverSocket.Stop();
      serverThread.Join();
      WaitForServiceThreads();
    }

    private void Server()
    {
      running = true;
      while (running)
      {
        Socket s = serverSocket.AcceptSocket();
        StartServiceThread(s);
      }
    }

    private void StartServiceThread(Socket s)
    {
      Thread serviceThread =
        new Thread(new ServiceRunner(s, this).ThreadStart());
      lock (threads)
      {
        threads.Add(serviceThread);
```

```
        }
        serviceThread.Start();
      }

      private void WaitForServiceThreads()
      {
        while (threads.Count > 0)
        {
          Thread t;
          lock (threads)
          {
            t = (Thread) threads[0];
          }
          t.Join();
        }
      }

      internal class ServiceRunner
      {
        private Socket itsSocket;
        private SocketServer itsServer;

        public ServiceRunner(Socket s, SocketServer server)
        {
          itsSocket = s;
          itsServer = server;
        }

        public void Run()
        {
          itsServer.itsService.Serve(itsSocket);
          lock (itsServer.threads)
          {
            itsServer.threads.Remove(Thread.CurrentThread);
          }
          itsSocket.Close();
        }

        public ThreadStart ThreadStart()
        {
          return new ThreadStart(Run);
        }
      }
    }
  }
```

На рис. 16.3 приведена диаграмма классов для этой программы. Она мало что проясняет, из нее довольно трудно понять назначение кода. Все классы и отношения между ними присутствуют, однако общая картина как-то не складывается.

Но взгляните на диаграмму объектов на рис. 16.4. Она дает куда лучшее представление о структуре программы, нежели диаграмма классов. Мы видим, что объект `SocketServer` владеет объектом `serverThread`, а тот работает внутри делегата с именем `Server()`. Видно, что `serverThread` отвечает за создание всех экземпляров `ServiceRunner`.

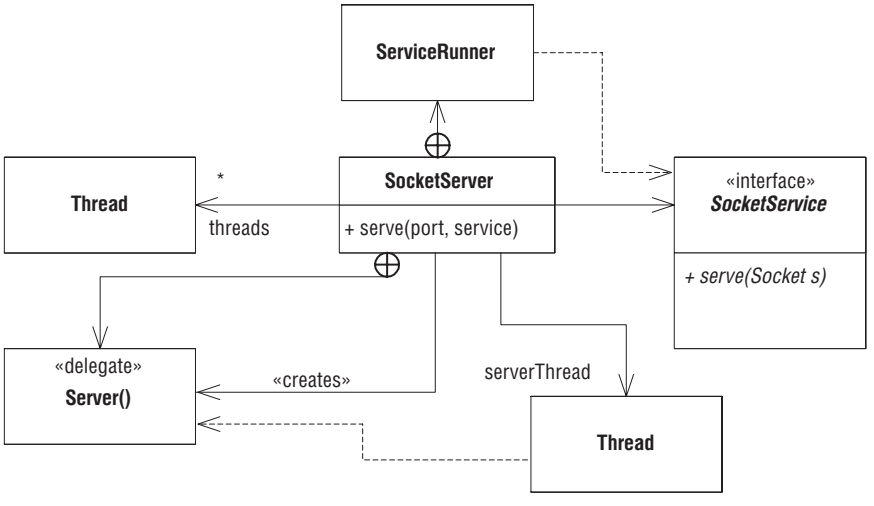


Рис. 16.3. *Диаграмма классов SocketServer*

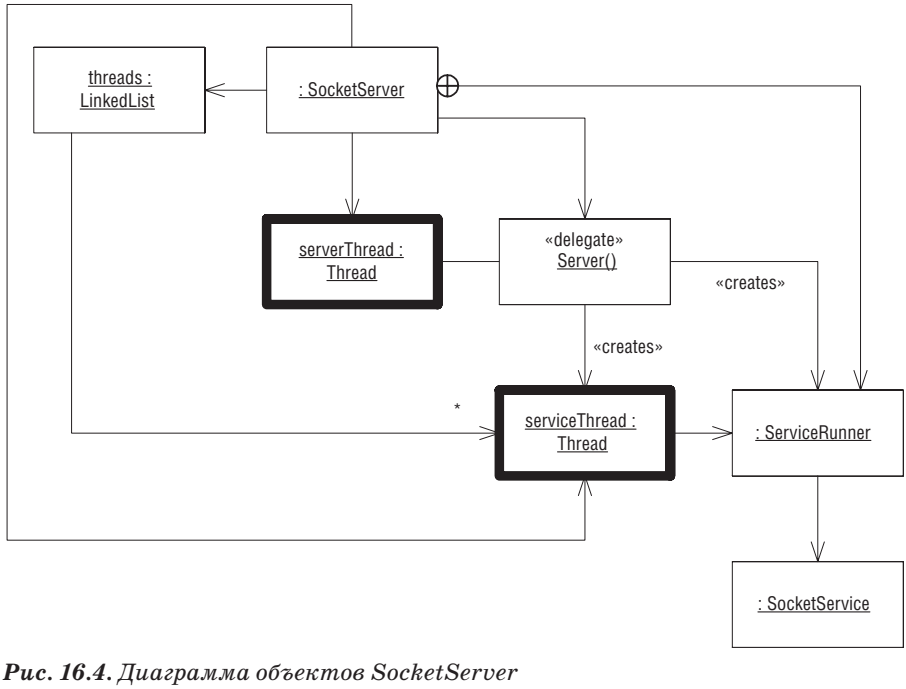


Рис. 16.4. *Диаграмма объектов SocketServer*

Обратите внимание на жирные линии, которыми обведены экземпляры `Thread`. Так обозначаются *активные объекты*, управляющие отдельными потоками. У них есть методы `Start`, `Abort`, `Sleep` и т. п. для управления потоком. На этой диаграмме все активные объекты являются экземплярами класса `Thread`, потому что вся обработка производится внутри делегатов, ссылки на которые хранятся в экземплярах `Thread`.

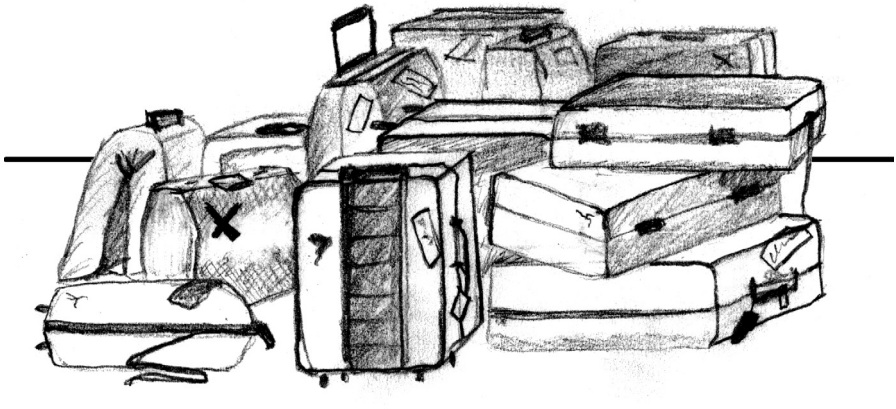
Эта диаграмма объектов выразительнее диаграммы классов в силу того, что структура данного конкретного приложения формируется во время выполнения. А в таком случае структура в большей степени связана с объектами, чем с классами.

Заключение

Диаграммы объектов дают мгновенный снимок состояния системы. Иногда это бывает полезно для описания системы, особенно если ее структура формируется динамически во время выполнения, а не диктуется статической структурой составляющих классов. Однако следует избегать рисования большого числа диаграмм объектов. Обычно их можно напрямую вывести из соответствующих диаграмм классов, и поэтому особого смысла в них нет.

17

Прецеденты



Прецеденты (Use Cases), или варианты использования, – замечательная идея, которую зачем-то чрезмерно усложнили. Сколько раз я наблюдал, как разные команды из сил выбиваются в попытках описать прецеденты. Как правило, они буксуют, решая вопросы формы, а не содержания. Яростно спорят о пред- и постусловиях, о действующих лицах, второстепенных действующих лицах и о массе других вещей, которые *вообще не имеют значения*.

Главная задача при работе с прецедентами в том, чтобы *сделать их простыми*. Откажитесь от всяких бланков описания прецедентов, записывайте их на *чистом* листе бумаги, или на *чистой* странице в простейшем текстовом редакторе, или на *чистой* учетной карточке. Не стре-

митесь выписать все подробности. Они понадобятся гораздо позже. Не пытайтесь выявить *все* прецеденты, это непосильная задача.

И главное, помните: *завтра прецеденты изменятся*. Неважно, как старательно вы их выявляли, неважно, как скрупулезно вы записывали все детали, неважно, как тщательно вы их обдумали, неважно, сколько сил вы потратили на исследование и анализ требований... *Завтра* все это изменится.

Но если завтра ситуация изменится, то не нужно фиксировать ее сегодняшние детали. Лучше пока отложить детальное изучение. Считайте, что прецеденты – это *сиюминутные требования*.

Записывание прецедентов

Обратите внимание на заголовок этого раздела. Мы *записываем* прецеденты, а не рисуем их. Прецеденты – не диаграммы, а текстовое описание требований к поведению, изложенное под некоторым углом зрения.

«Постойте, постойте! – слышу я возгласы. – Я же знаю, что в UML есть диаграммы прецедентов, сам их видел».

Да, есть. Но они ничего не говорят о *содержании* прецедентов. В них нет информации о требованиях к поведению, которую прецеденты как раз и должны зафиксировать. На диаграммах прецедентов в UML отражено нечто совсем иное.

Прецедент – это описание поведения системы. Оно записывается с точки зрения пользователя, который только что попросил систему сделать нечто конкретное. В прецеденте отражена *видимая* последовательность событий, происходящих в системе в ответ на *единственный* поступивший от пользователя приказ.

Под *видимым событием* мы понимаем нечто, что пользователь может наблюдать. Прецеденты вообще не описывают скрытое поведение. В них не обсуждаются скрытые механизмы системы. Только то, что пользователь может увидеть.

Обычно описание прецедента разбито на два раздела. Первый – это *основной поток событий*. В нем мы описываем, как система реагирует на приказ пользователя в предположении, что все идет нормально.

Вот, например, типичный прецедент для кассового терминала.

Пробить товар:

1. Кассир проводит товар над сканером; сканер считывает штрих-код.
2. Описание и цена товара вместе с текущим подытогом отображаются на дисплее, обращенном к покупателю. Описание и цена товара появляются также на экране кассового терминала.
3. Описание и цена товара печатаются в чеке.

4. Система издает звуковой сигнал, подтверждающий, что штрих-код успешно прочитан.

Это основной поток событий прецедента! Ничего более сложного и не нужно. Собственно, даже эта коротенькая последовательность, возможно, излишне детальна, если в ближайшем будущем мы не собираемся ее реализовывать. Не нужны нам такие подробности до тех пор, пока через несколько дней или недель руки не дойдут до реализации этого прецедента.

А как оценить прецедент, если не записывать его детали? Просто надо поговорить с заинтересованными сторонами о деталях, не записывая их. Это даст достаточно информации для грубой оценки. Но почему не записать детали, раз уже все равно они обсуждаются? Потому, что завтра детали могут измениться. А разве при этом не изменится оценка прецедента? Да, изменится, но если взять всю совокупность прецедентов, то сумма изменений будет близка к нулю. Записывать детали на такой ранней стадии неэффективно.

Но если мы пока не собираемся записывать детали, то что же мы все-таки записываем? Как узнать, что прецедент вообще существует, если не остается никаких письменных упоминаний о нем? Да просто записать его название. Занесите имена в электронную таблицу или в текстовый файл. А еще лучше, запишите название прецедента на учетной карточке, и храните все такие карточки в одном ящике. Детали можно будет вписать, когда подойдет время реализации.

Альтернативные потоки событий

Некоторые детали будут относиться к случаям, когда что-то идет не так, как надо. Во время бесед с заинтересованными сторонами следует обсудить и сценарии ошибок. Впоследствии, когда дело будет близиться к реализации прецедента, вы сможете более предметно подумать об альтернативных потоках событий. Они станут дополнением к основному потоку. Записывать их можно следующим образом.

Штрих-код не считался:

Если сканер не смог считать штрих-код, то система должна подать сигнал «просканировать повторно», извещающий кассира о необходимости повторить попытку. Если после трех попыток сканер так и не сумел считать штрих-код, кассир должен ввести его вручную.

Штрих-код отсутствует:

Если на товаре нет штрих-кода, то кассир должен ввести цену вручную.

Альтернативные потоки событий интересны, потому что напоминают о других прецедентах, которые не были выявлены при первоначальном исследовании. В данном случае необходимо предусмотреть возможность ручного ввода штрих-кода или цены.

Что-нибудь еще?

Как насчет действующих лиц, второстепенных действующих лиц, предусловий, постусловий и всего прочего? Не забивайте себе голову. В подавляющем большинстве систем, с которыми вам придется столкнуться, обо всем этом можно и не знать. Когда настанет момент для более подробного ознакомления с прецедентами, можете почитать исчерпывающую работу Алистера Кокберна на эту тему.[1] Сначала научитесь ходить, а потом уж бегать. Привыкните записывать простые прецеденты. Когда овладеете этим искусством – то есть научитесь успешно применять их в проекте, – тогда можете вдумчиво и не торопясь осваивать более сложные приемы. Но, главное – ничего не вымучивайте.

Представление прецедентов на диаграммах

Из всех UML-диаграмм диаграммы прецедентов наиболее невразумительные и самые бесполезные. Я советую вообще их избегать, делая исключение только для диаграмм системных границ.

На рис. 17.1 изображена диаграмма системных границ. Большой прямоугольник обозначает границу системы. Все внутри него – части разрабатываемой системы. За его пределами находятся *действующие лица* (или *акторы*), которые *воздействуют* на систему. Действующие лица находятся вне системы и отдают ей приказы (стимулы). Как правило, действующие лица – это люди. Но это могут быть и другие системы или даже аппаратные устройства, например таймеры.

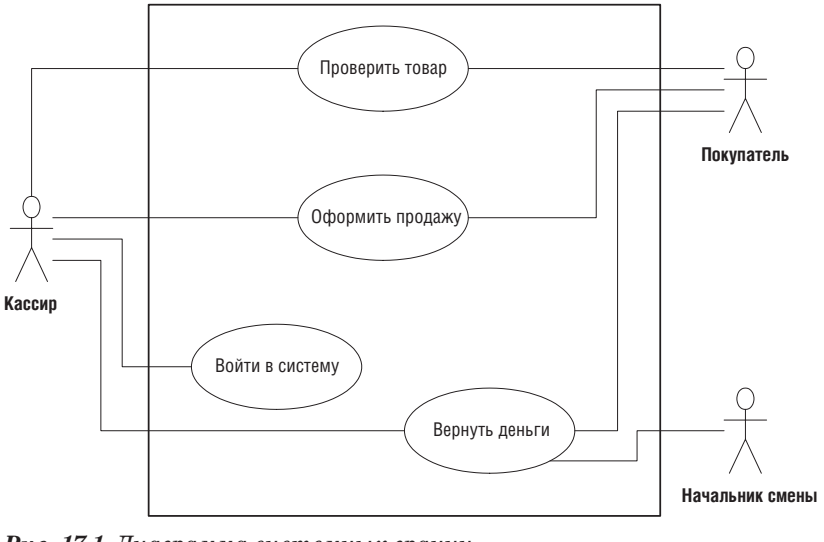


Рис. 17.1. *Диаграмма системных границ*

[1] [Cockburn2001]

Внутри ограничивающего прямоугольника располагаются прецеденты. Линии соединяют действующих лиц с инициируемыми ими прецедентами. Не пользуйтесь стрелками; никто не может объяснить, что означает в данном случае направление стрелки.

Эта диаграмма почти, хотя и не совсем, бесполезна. Она содержит очень мало информации, интересной программисту, зато неплохо выглядит в качестве первой страницы презентации заказчикам.

Отношения между прецедентами попадают в категорию явлений, которые «когда-то казались отличной идеей». Я рекомендую их всячески избегать. Они не добавляют ничего нового ни прецедентам, ни пониманию системы, а только приводят к бесконечным спорам по поводу того, что имеет место: «расширение» или «обобщение».

Заключение

Это была короткая глава. Ничего удивительного, так как сама тема простая. Именно таким и должно быть ваше отношение к прецедентам. Если вы один раз вступите на тропу усложнения прецедентов, то уже никогда с нее не сойдете. Будьте решительны в стремлении сделать прецеденты максимально простыми.

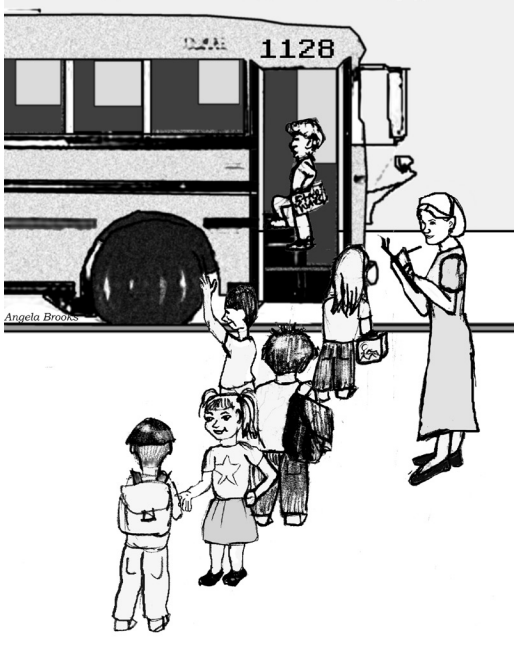
Библиография

[Cockburn2001] Alistair Cockburn «Writing Effective Use Cases», Addison-Wesley, 2001.[1]

[1] А. Коберн «Современные методы описания функциональных требований к системам». – Пер. с англ. – Лори, 2002.

18

Диаграммы последовательности



Диаграммы последовательности – самая часто рисуемая пользователями UML динамическая модель. Нетрудно догадаться, что UML предлагает множество средств, чтобы вы могли нарисовать по-настоящему непостижимую диаграмму. В этой главе я расскажу об этих средствах и постараюсь убедить вас, что применять их следует с крайней осторожностью.

Как-то раз я консультировал команду, решившую нарисовать диаграммы последовательности для всех без исключения методов всех классов. Прошу вас, не поступайте так – это пустая и бесполезная трата времени. Применяйте диаграммы последовательности, когда возникает настоятельная необходимость объяснить кому-то, какие взаимодействия существуют в некоторой группе объектов, или чтобы самому наглядно представить эти взаимодействия. Используйте их только как инструмент для эпизодической шлифовки своих аналитических навыков, но не в качестве необходимой документации.

Основные понятия

Я научился рисовать диаграммы последовательности еще в 1978 году. Джеймс Гриннинг, мой давний друг и коллега, показал их мне, когда мы работали над проектом, где присутствовали сложные коммуникационные протоколы между компьютерами, соединенными модемной связью. То, что я собираюсь продемонстрировать ниже, очень близко к простейшей нотации, которой он тогда научил меня, и этого достаточно для подавляющего большинства диаграмм последовательности, которые вам когда-либо придется рисовать.

Объекты, линии жизни, сообщения и прочее

На рис. 18.1 изображена типичная диаграмма последовательности. Сверху показаны объекты и классы, участвующие в кооперации. Имена объектов подчеркнуты, имена классов – нет. Фигурка человечка слева представляет анонимный объект. Это источник и приемник всех сообщений, входящих в данную кооперацию и выходящих из нее. Такое анонимное действующее лицо есть не на всех диаграммах последовательности, но на многих.

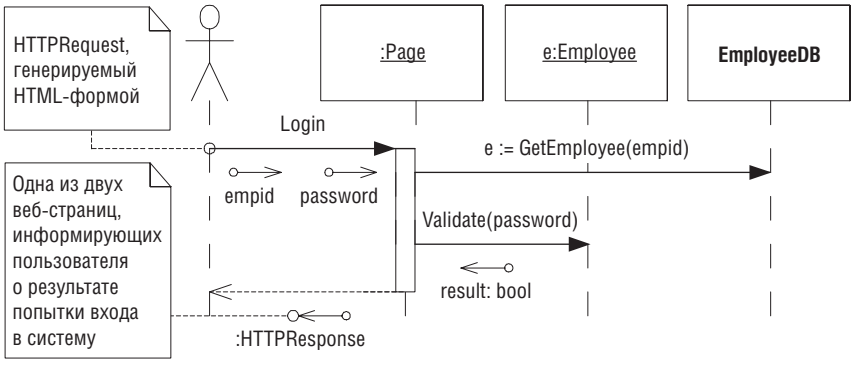


Рис. 18.1. *Типичная диаграмма последовательности*

Пунктирные линии, идущие вниз от объектов и действующего лица, называются *линиями жизни*. Сообщения, которые один объект посылает другому, показаны стрелками, соединяющими две линии жизни. Аргументы указываются либо в скобках после имени, либо рядом с *маркерами данных* (небольшие стрелки с кружком на конце). Временная ось направлена сверху вниз, то есть чем ниже расположено сообщение, тем позже оно послано.

Узкий вытянутый прямоугольник на линии жизни объекта `Page` называется *активацией*[1]. Активации необязательны, большинство диаграмм обходится без них. Активация обозначает время, в течение которого выполняется функция. В данном случае мы видим, сколько времени работает метод `Login`. Оба сообщения, исходящих из активации, отправлены методом `Login`. Непомеченная пунктирная стрелка показывает, что метод `Login` передает управление действующему лицу вместе с возвращенным значением.

Обратите внимание на использование переменной *e* в сообщении `GetEmployee`. Так изображается значение, возвращенное `GetEmployee`. Отметим еще, что объект `Employee` также назван *e*. Вы наверняка догадались, что это неслучайно. Значение, возвращенное методом `GetEmployee`, – ссылка на объект `Employee`.

Наконец, отметим, что, поскольку `EmployeeDB` – класс, его имя не подчеркнуто. Это может означать лишь одно: `GetEmployee` – статический метод. Следовательно, можно ожидать, что код `EmployeeDB` выглядит, как показано в листинге 18.1.

Листинг 18.1. EmployeeDB.cs

```
public class EmployeeDB
{
  public static Employee GetEmployee(string empid)
  {
    ...
  }
  ...
}
```

Создание и уничтожение

На диаграмме последовательности можно показать создание объекта. Для этого применяется соглашение, представленное на рис. 18.2. Непомеченная стрелка сообщения оканчивается на самом создаваемом объекте, а не на его линии жизни. Таким образом, можно предположить, что класс `ShapeFactory` реализован примерно так, как показано в листинге 18.2:

[1] Применяется также термин «фокус управления». – *Прим. перев.*

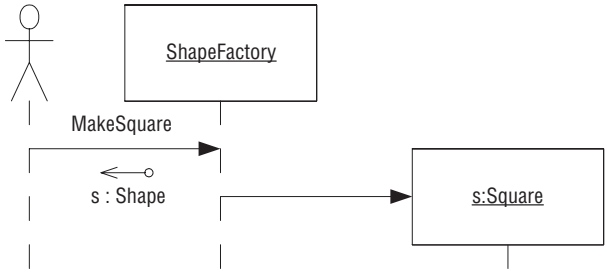


Рис. 18.2. *Создание объекта*

Листинг 18.2. ShapeFactory.cs

```
public class ShapeFactory
{
  public Shape MakeSquare()
  {
    return new Square();
  }
}
```

В языке C# объекты не уничтожаются явно. Этим занимается сборщик мусора. Но иногда бывает нужно ясно показать, что работа с объектом закончена и, значит, сборщик мусора может его убирать.

На рис. 18.3 показано, как это обозначается в UML. Линия жизни освобождаемого объекта заканчивается большим крестиком. Стрелка сообщения, ведущая к крестику, представляет акт передачи объекта сборщику мусора.

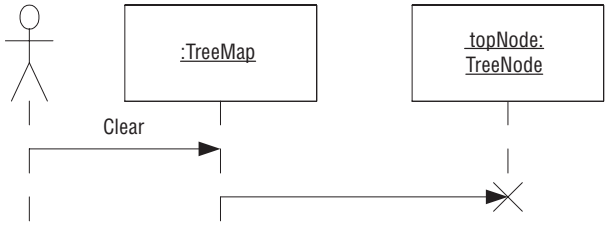


Рис. 18.3. *Передача объекта сборщику мусора*

В листинге 18.3 показано, какова могла бы быть реализация этой диаграммы. Обратите внимание, что метод `Clear` присваивает переменной `topNode` значение `null`. Поскольку ссылка на экземпляр `TreeNode` хранилась только в этой переменной, то теперь на него никто не ссылается и сборщик мусора может освободить занятую им память.

Листинг 18.3. TreeMap.cs

```
public class TreeMap
{
  private TreeNode topNode;
  public void Clear()
  {
    topNode = null;
  }
}
```

Простые циклы

На UML-диаграмме можно изобразить простой цикл, заключив повторяющиеся сообщения в прямоугольник. Условие цикла записывается в квадратных скобках и помещается где-то внутри прямоугольника, обычно в правом нижнем углу. См. рис. 18.4.

Это полезное соглашение. Однако представлять алгоритмы на диаграммах последовательностях неразумно. Эти диаграммы следует использовать для изображения связей между объектами, а не низкоуровневых деталей алгоритма.

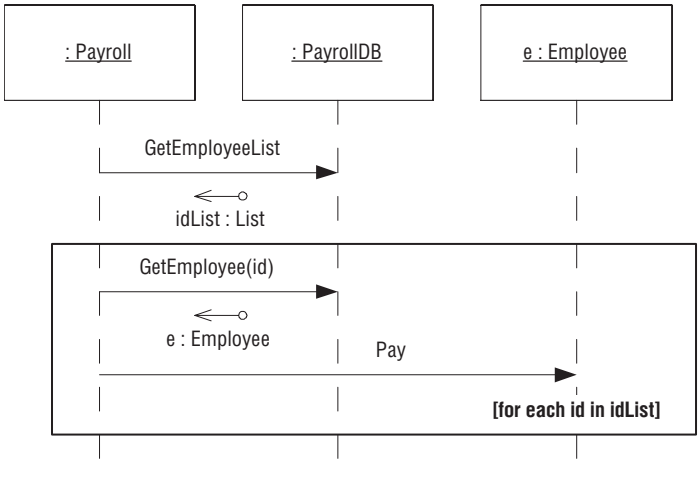


***Рис. 18.4.** Простой цикл*

Различные сценарии

Не рисуйте диаграммы последовательности, содержащие слишком много объектов и сообщений (см. рис. 18.5). Их никто не сможет прочитать. И *не будет* читать. Это только пустая трата времени. Лучше научитесь рисовать несколько небольших диаграмм, передающих *суть* того, что вы пытаетесь сделать. Любая диаграмма последовательности

должна умещаться на одной странице, причем должно еще оставаться достаточно места для пояснительного текста. И не хитрите, до предела уменьшая значки, чтобы уместить диаграмму на странице.

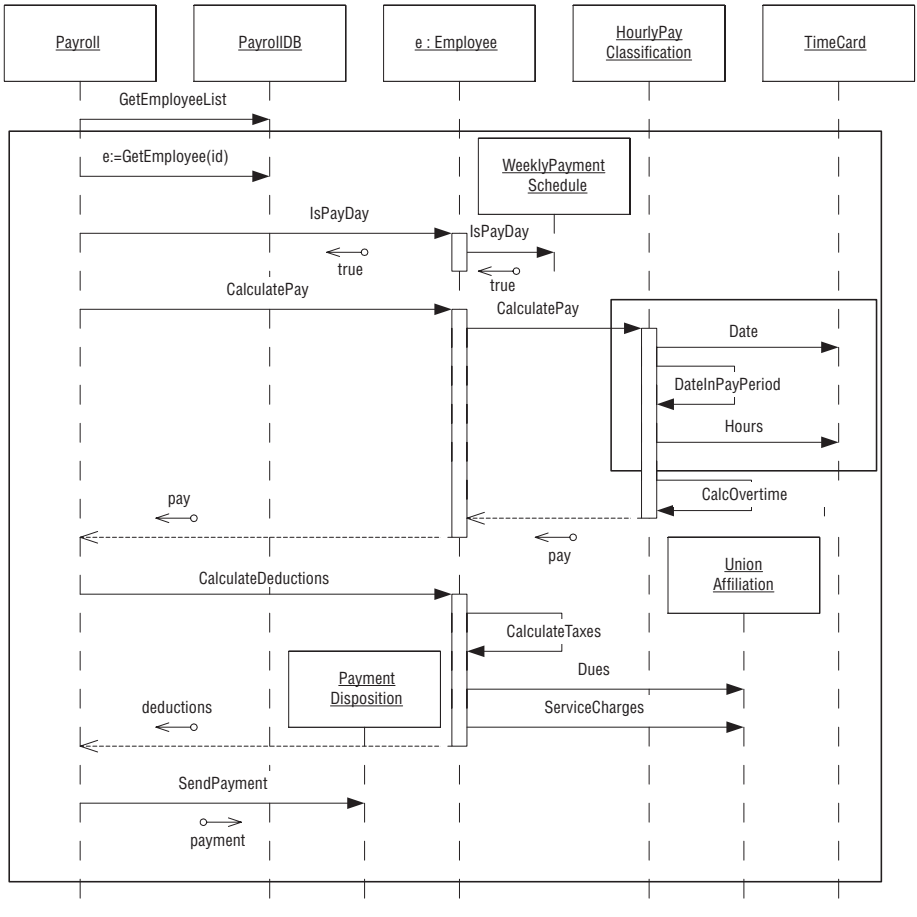


Рис. 18.5. *Чрезмерно сложная диаграмма последовательности*

Сотни диаграмм последовательности тоже рисовать не стоит. Если их слишком много, они останутся непрочитанными. Попытайтесь понять, что общего у всех сценариев, и на этом сконцентрируйтесь. *В мире UML-диаграмм сходство гораздо важнее различий*. Применяйте диаграммы для выражения общих тем и приемов. Не надо с их помощью документировать мельчайшие детали. Если ощущаете необходимость описать поток сообщений на диаграмме последовательности, делайте это кратко и рационально. Сведите их число к минимуму.

Во-первых, спросите себя, так ли уж необходима эта конкретная диаграмма. Код часто экономнее и информативнее. Например, в листинге 18.4 показано, как мог бы выглядеть класс `Payroll` (Платежная ведомость). Он настолько выразителен, что не нуждается в пояснениях. Чтобы его понять, не нужна никакая диаграмма последовательности, поэтому и рисовать ее ни к чему. Когда код самодостаточен, рисование диаграмм – излишество и расточительность.

Листинг 18.4. Payroll.cs

```
public class Payroll
{
  private PayrollDB itsPayrollDB;
  private PaymentDisposition itsDisposition;

  public void DoPayroll()
  {
    ArrayList employeeList = itsPayrollDB.GetEmployeeList();
    foreach (Employee e in employeeList)
    {
      if (e.IsPayDay())
      {
        double pay = e.CalculatePay();
        double deductions = e.CalculateDeductions();
        itsDisposition.SendPayment(pay - deductions);
      }
    }
  }
}
```

А можно ли использовать код для описания некоторой части системы? На самом деле *это и должно быть целью* разработчиков и проектировщиков. Команда должна стремиться к созданию выразительного и удобочитаемого кода. Чем лучше код описывает себя, тем меньше требуется диаграмм и тем здоровее проект.

Во-вторых, если вам кажется, что диаграмма последовательности все-таки нужна, задайте себе вопрос, нельзя ли представить ее в виде нескольких небольших сценариев. Например, гигантскую диаграмму на рис. 18.5 можно было бы разбить на ряд диаграмм гораздо меньшего размера, которые проще читать. Посмотрите, насколько проще воспринимается небольшой сценарий на рис. 18.6.

В-третьих, подумайте, что вы хотите изобразить. Детали низкоуровневой операции, как на рис. 18.6, где показан алгоритм начисления почасовой оплаты? Или высокоуровневое представление всей работы системы, как на рис. 18.7? Вообще говоря, высокоуровневые диаграммы полезнее низкоуровневых, так как помогают читателю мысленно связать все компоненты системы воедино. На высокоуровневой диаграмме показаны общие черты, а не различия.

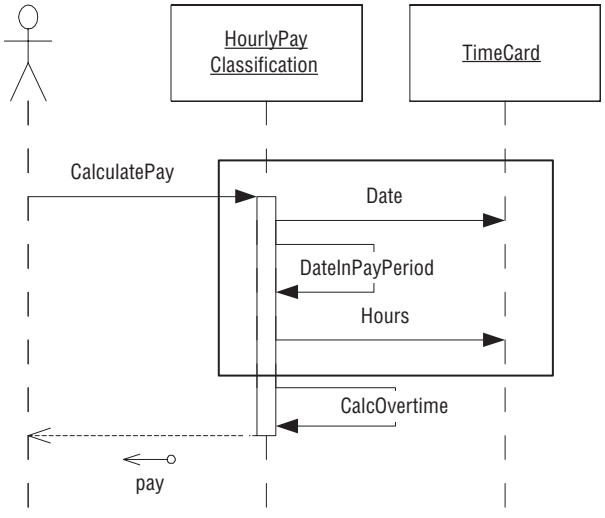


Рис. 18.6. *Один небольшой сценарий*

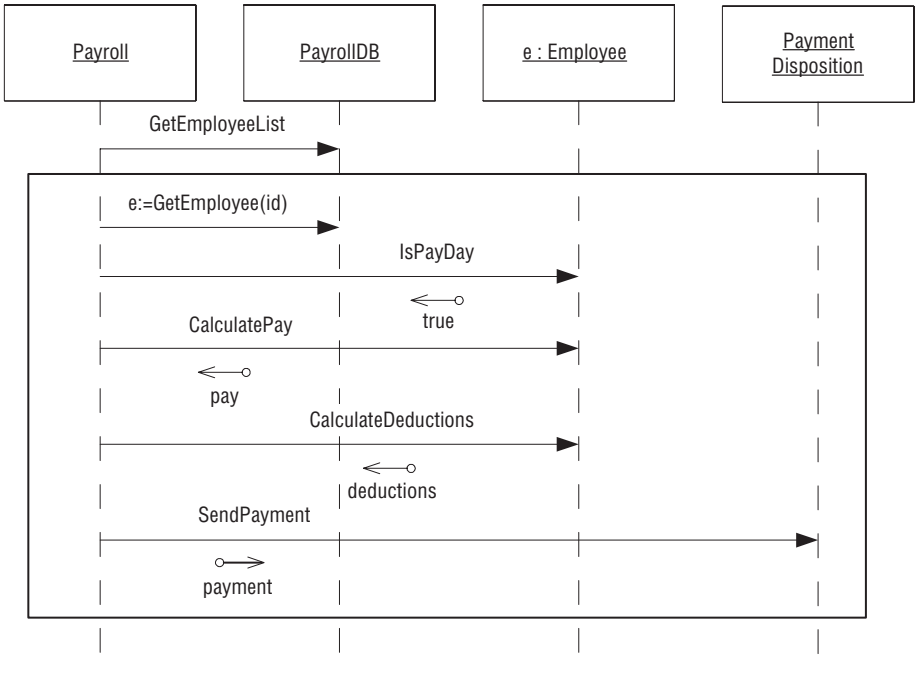


Рис. 18.7. *Высокоуровневое представление*

Более сложные конструкции

Циклы и условия

Можно нарисовать диаграмму последовательности, которая будет полностью описывать алгоритм. На рис. 18.8 представлен алгоритм расчета заработной платы со всеми циклами и ветвлениями.

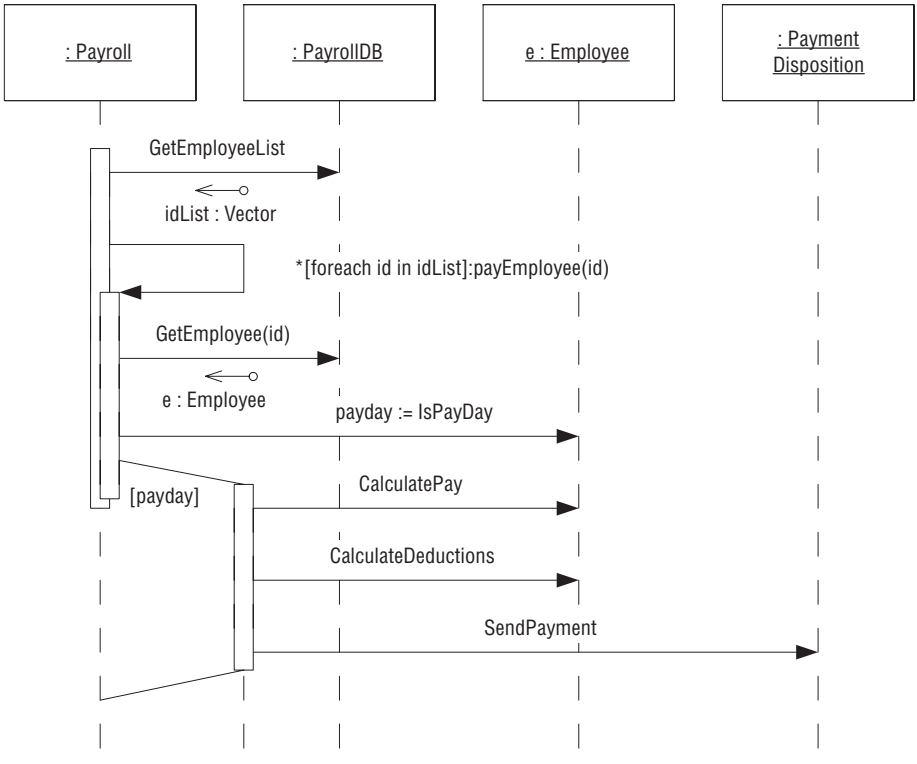


***Рис. 18.8.** Диаграмма последовательности с циклами и условиями*

Сообщению `payEmployee` предшествует выражение *повтора*:

```
*[foreach id in idList]
```

Звездочка информирует о том, что это итерация, то есть сообщение должно повторно посылаться до тех пор, пока не станет ложным *сторожевое условие* в квадратных скобках. Хотя в UML есть специальный синтаксис для сторожевых условий, мне больше нравится псевдокод, предполагающий использование итератора или предложения `foreach` (как в языке C#).

Сообщение `payEmployee` заканчивается на прямоугольнике активации, который находится в пределах первой активации, но смещен вправо.

Так обозначается, что в данный момент в одном объекте выполняются две функции. Поскольку сообщение `payEmployee` повторяется, то вторая активация тоже станет повторяться, поэтому все зависящие от нее сообщения будут частью цикла.

Обратите внимание на активацию рядом со сторожевым условием `[payday]`. Так обозначается предложение `if`. Вторая активация получает управление, только если сторожевое условие истинно. Иными словами, если `isPayDay` возвращает `true`, то методы `calculatePay`, `calculateDeductions` и `sendPayment` будут выполняться, иначе нет.

Из того, что на диаграмме последовательности *можно* передать все детали алгоритма, вовсе не следует, что все алгоритмы так и нужно документировать. Представление алгоритмов средствами UML выглядит, мягко говоря, громоздко. Код, например приведенный в листинге 18.4, *гораздо* лучше выражает алгоритм.

Сообщения, занимающие время

Обычно мы не принимаем во внимание время, потребное для отправки сообщения от одного объекта другому. В большинстве объектно-ориентированных языков это происходит практически мгновенно. Поэтому линии сообщений рисуются горизонтально – они не занимают времени. Но иногда бывает, что на отправку сообщения все-таки уходит некоторое время. Например, если сообщение отправляется по сети или вызов и выполнение метода происходят в разных потоках. В таком случае сообщение можно обозначить *наклонной* линией, как показано на рис. 18.9.

На этом рисунке показана последовательность выполнения телефонного вызова. На диаграмме последовательности есть три объекта. `caller` – это тот, кто звонит, `callee` – тот, кому звонят, `telco` – телефонная компания.

После снятия трубки объекту `telco` отправляется сообщение *трубка снята*, а объект отвечает длинным гудком готовности линии. Услышав этот сигнал, `caller` набирает номер `callee`. В ответ `telco` звонит `callee` и посылает ответный гудок `caller`. Объект `callee`, услышав звонок, берет трубку. `telco` устанавливает соединение. `callee` говорит «Алло», и телефонный вызов считается состоявшимся.

Однако существует и другая возможность, демонстрирующая полезность диаграмм такого рода. Взгляните на рис. 18.10. Отметим, что начало диаграммы точно такое же. Однако еще до того, как телефон зазвонил, `callee` снимает трубку, намереваясь позвонить сам. Теперь `caller` соединен с `callee`, но ни одна сторона об этом не знает. `caller` хочет услышать «Алло», а `callee` ждет сигнала готовности линии. В конце концов `callee` в раздражении бросает трубку, а `caller` слышит гудок.

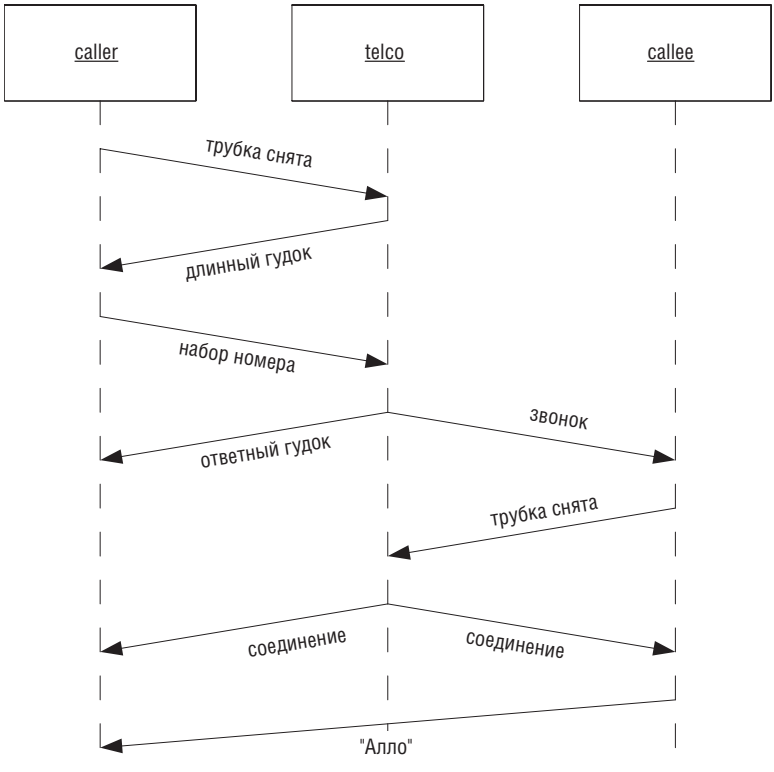


Рис. 18.9. *Обычный телефонный вызов*

Пересечение двух стрелок на рис. 18.10 называется *состязанием*. Это происходит, когда два асинхронно работающих объекта одновременно пытаются выполнить несовместимые операции. В нашем случае `telco` инициировал операцию *звонок*, а `callee` снял трубку. В этот момент стороны по-разному представляют себе состояние системы. `caller` ожидает услышать «Алло», `telco` считает свою часть работы выполненной, а `callee` ждет длинного гудка.

Состязания в программных системах очень трудно выявлять и отлаживать. Такие диаграммы могут помочь в обнаружении и диагностике причин. И главное, объяснить другим, что происходит.

Асинхронные сообщения

Посылая объекту сообщение, обычно не рассчитывают получить обратно управление до того, как вызванный объект закончит выполнение операции. Такие сообщения называются *синхронными*. Но в распределенных и многопоточных системах может случиться, что объекту-отправителю управление возвращается сразу же, а объект-получатель работает в другом потоке управления. Такие сообщения называются *асинхронными*.

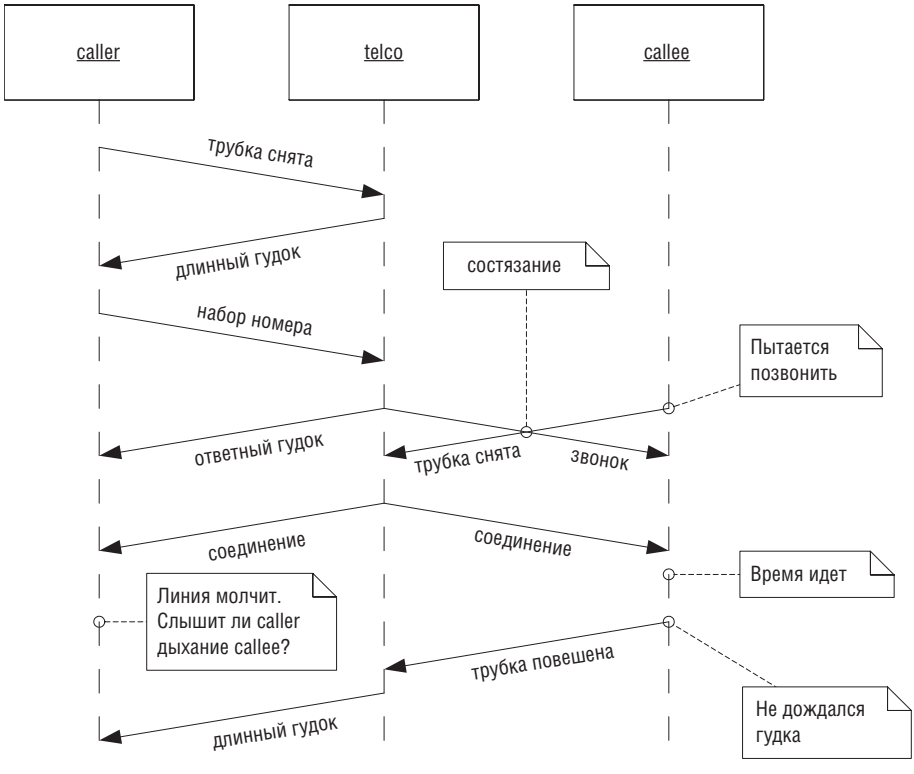


Рис. 18.10. *Несостоявшийся телефонный звонок*

На рис. 18.11 показано асинхронное сообщение. Обратите внимание, что острие стрелки не закрашено. А теперь посмотрите на все предыдущие диаграммы последовательности в этой главе. На них были нарисованы только синхронные сообщения (с закрашенными остриями стрелок). Элегантность – или, если хотите, извращенность, в зависимости от точки зрения, – UML проявляется в том, что такое тонкое отличие в изображении острия стрелки может означать глубокое различие в поведении.

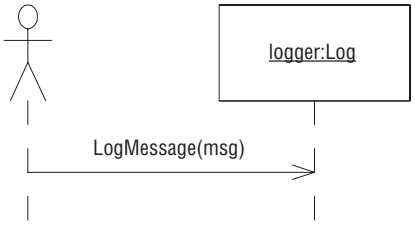


Рис. 18.11. *Асинхронное сообщение*

В предыдущих версиях UML для обозначения асинхронных сообщений применялись стрелки с половиной острия, как на рис. 18.12. Визуальное отличие гораздо явственнее – глаз читателя сразу же замечает асимметрию стрелки. Поэтому я продолжаю пользоваться этим соглашением, хотя в UML 2.0 оно отменено.

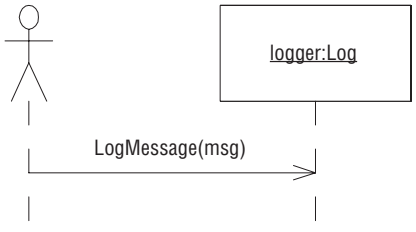


Рис. 18.12. *Прежний и лучший способ изображения асинхронных сообщений*

В листингах 18.5 и 18.6 показан код, соответствующий рис. 18.11. Листинг 18.5 – это автономный тест для класса `AsynchronousLogger` из листинга 18.6. Обратите внимание, что метод `LogMessage` возвращает управление сразу же после постановки сообщения в очередь. При этом сообщение обрабатывается в другом потоке, запущенном в конструкторе. Класс `TestLog` убеждается, что метод `LogMessage` действительно ведет себя асинхронно, для чего сначала проверяет, что сообщение помещено в очередь, но еще не обработано, потом уступает процессор другим заданиям и в конце проверяет, что сообщение обработано и удалено из очереди.

Это лишь одна из возможных реализаций асинхронного сообщения. Существуют и другие. В общем случае мы помечаем сообщение как асинхронное, если у вызывающей стороны есть основания ожидать, что она получит назад управление раньше, чем будут выполнены затребованные операции.

Листинг 18.5. TestLog.cs

```
using System;
using System.Threading;
using NUnit.Framework;

namespace AsynchronousLogger
{
  [TestFixture]
  public class TestLog
  {
    private AsynchronousLogger logger;
    private int messagesLogged;

    [SetUp]
    protected void SetUp()
```

```
{
  messagesLogged = 0;
  logger = new AsynchronousLogger(Console.Out);
  Pause();
}

[TearDown]
protected void TearDown()
{
  logger.Stop();
}

[Test]
public void OneMessage()
{
  logger.LogMessage("одно сообщение");
  CheckMessagesFlowToLog(1);
}

[Test]
public void TwoConsecutiveMessages()
{
  logger.LogMessage("другое");
  logger.LogMessage("и еще одно");
  CheckMessagesFlowToLog(2);
}

[Test]
public void ManyMessages()
{
  for (int i = 0; i < 10; i++)
  {
    logger.LogMessage(string.Format("сообщение:{0}", i));
    CheckMessagesFlowToLog(1);
  }
}

private void CheckMessagesFlowToLog(int queued)
{
  CheckQueuedAndLogged(queued, messagesLogged);
  Pause();
  messagesLogged += queued;
  CheckQueuedAndLogged(0, messagesLogged);
}

private void CheckQueuedAndLogged(int queued, int logged)
{
  Assert.AreEqual(queued, logger.MessagesInQueue(), "в очереди");
  Assert.AreEqual(logged, logger.MessagesLogged(), "зарегистрировано");
}
```

```
    private void Pause()
    {
      Thread.Sleep(50);
    }
  }
}
```

Листинг 18.6. AsynchronousLogger.cs

```
using System;
using System.Collections;
using System.IO;
using System.Threading;

namespace AsynchronousLogger
{
  public class AsynchronousLogger
  {
    private ArrayList messages =
      ArrayList.Synchronized(new ArrayList());
    private Thread t;
    private bool running;
    private int logged;
    private TextWriter logStream;

    public AsynchronousLogger(TextWriter stream)
    {
      logStream = stream;
      running = true;
      t = new Thread(new ThreadStart(MainLoggerLoop));
      t.Priority = ThreadPriority.Lowest;
      t.Start();
    }

    private void MainLoggerLoop()
    {
      while (running)
      {
        LogQueuedMessages();
        SleepTillMoreMessagesQueued();
        Thread.Sleep(10); // Напомните, чтобы я объяснил это место.
      }
    }

    private void LogQueuedMessages()
    {
      while (MessagesInQueue() > 0)
        LogOneMessage();
    }
```

```
    private void LogOneMessage()
    {
      string msg = (string) messages[0];
      messages.RemoveAt(0);
      logStream.WriteLine(msg);
      logged++;
    }

    private void SleepTillMoreMessagesQueued()
    {
      lock (messages)
      {
        Monitor.Wait(messages);
      }
    }

    public void LogMessage(String msg)
    {
      messages.Add(msg);
      WakeLoggerThread();
    }

    public int MessagesInQueue()
    {
      return messages.Count;
    }

    public int MessagesLogged()
    {
      return logged;
    }

    public void Stop()
    {
      running = false;
      WakeLoggerThread();
      t.Join();
    }

    private void WakeLoggerThread()
    {
      lock (messages)
      {
        Monitor.PulseAll(messages);
      }
    }
  }
}
```

Несколько потоков



Асинхронные сообщения подразумевают наличие нескольких потоков управления. На UML-диаграмме показать несколько потоков можно, предпослав имени сообщения идентификатор потока, как изображено на рис. 18.13.

Как видите, перед именем сообщения указывается идентификатор, например `T1`, за которым следует двоеточие. Идентификатор именует поток, из которого было отправлено сообщение. На этой диаграмме создание и вызов методов объекта `AsynchronousLogger` происходят в потоке `T1`, а собственно запись в протокол – в потоке `T2`.

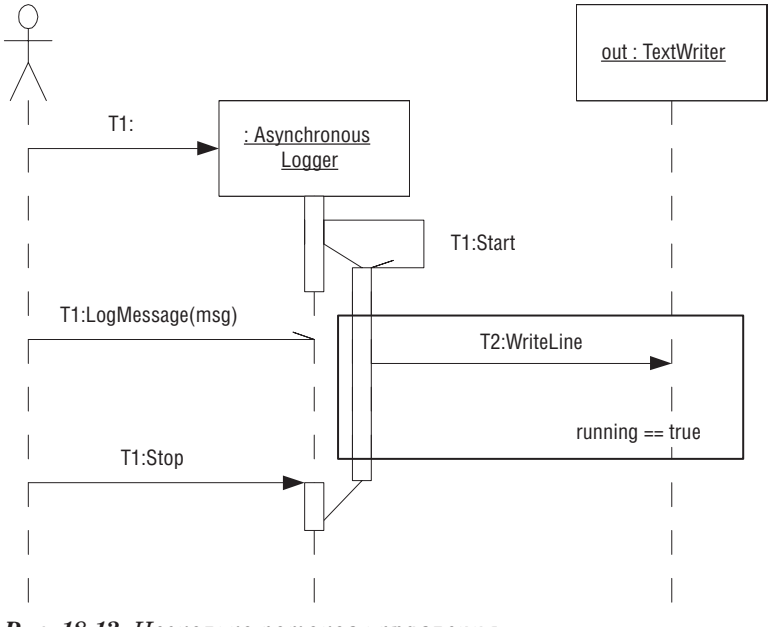


Рис. 18.13. *Несколько потоков управления*

Идентификаторы потоков не обязаны соответствовать именам в коде. Так, в листинге 18.6 поток регистрации не называется `T2`. Идентификаторы служат только для удобства представления на диаграмме.

Активные объекты

Иногда требуется обозначить объект, у которого есть отдельный внутренний поток. Такие объекты называются *активными*. Они изображаются прямоугольником с жирной рамкой, как показано на рис. 18.14.

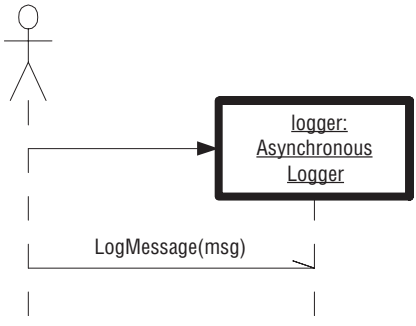
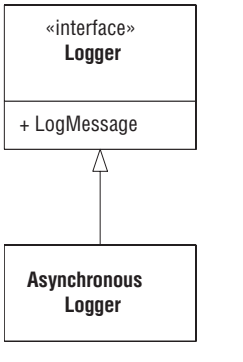


Рис. 18.14. *Активный объект*

Активные объекты сами создают свои потоки и управляют ими. Никаких ограничений на их методы не налагается. Методы могут выполняться в потоке как самого объекта, так и вызывающей программы.

Отправка сообщений интерфейсам

Наш класс `AsynchronousLogger` дает некоторый способ регистрации сообщений. А что если приложению необходимо несколько разных регистраторов? Лучше было бы создать интерфейс `Logger`, объявив в нем метод `LogMessage`, и произвести от него как класс `AsynchronousLogger`, так и все остальные реализации регистраторов (рис. 18.5).




```
interface Logger {
  void LogMessage(string msg);
}

public class AsynchronousLogger : Logger {
  ...
}
```

Рис. 18.15. *Дизайн простого регистратора*

Приложение будет посылать сообщения интерфейсу `Logger`, не зная, что за ним стоит объект `AsychronousLogger`. Как изобразить такую ситуацию на диаграмме последовательности?

Очевидный подход показан на рис. 18.16. Мы просто указываем имя интерфейса, как будто это объект, и считаем, что все нормально. На первый взгляд это нарушает правила, потому что создать экземпляр интерфейса невозможно. Однако в действительности мы лишь утверж-

даем, что объект `logger` согласован с типом `Logger`. Нигде не говорится, что каким-то образом создан экземпляр интерфейса.

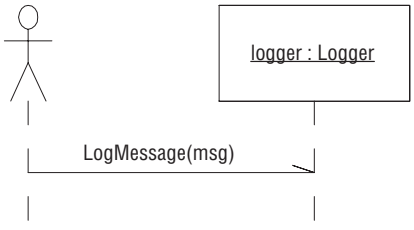


***Рис. 18.16.** Отправка сообщения интерфейсу*

Но иногда мы знаем тип объекта и все-таки хотим показать, что сообщение отправляется интерфейсу. Например, мы знаем, что создали объект типа `AsynchronousLogger`, но желаем четко заявить, что приложение работает исключительно с интерфейсом `Logger`. На рис. 18.17 показано, как изобразить такую ситуацию. На линии жизни объекта используется символ интерфейса, напоминающий леденец на палочке.

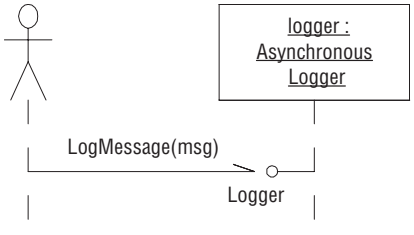


***Рис. 18.17.** Отправка производному типу через интерфейс*

Заключение

Как мы видели, диаграммы последовательности – действенный способ представления потока сообщений в объектно-ориентированном приложении. Мы также дали понять, что очень легко вовлечься в неправильное или чрезмерное употребление этих диаграмм.

Своевременная диаграмма последовательности на доске может оказаться неоценимой вещью. Клочок бумаги с пятью-шестью диаграммами последовательности, описывающими самые распространенные в системе взаимодействия, может цениться на вес золота. С другой стороны, документ с тысячами диаграмм последовательности, скорее всего, не будет стоить и той бумаги, на которой напечатан.

Одним из величайших заблуждений теории разработки ПО в 1990 годах была идея о том, что разработчики должны рисовать диаграммы после-

довательности для всех методов еще *до* написания кода. Это всегда оказывается крайне расточительной тратой времени. Не поступайте так.

Лучше используйте диаграммы последовательности как инструмент, каким они и задумывались. Рисуйте их на доске во время общения с коллегами. Включайте в краткий документ, предназначенный для описания основных взаимодействий в системе.

Пусть лучше диаграмм последовательности будет меньше, чем нужно, но не больше. При необходимости недостающую диаграмму всегда можно будет нарисовать позже.

19

Диаграммы классов



UML-диаграммы классов позволяют описывать статический состав классов и отношения между ними. На диаграмме классов можно показать переменные-члены и методы, а также обозначить, что один класс наследует другому или хранит ссылку на другой. Короче говоря, можно изобразить все *зависимости между классами, существующие в исходном коде*.

Это может быть полезно. Структуру зависимостей в системе гораздо проще воспринимать, глядя на диаграмму, а не на исходный код. Диаграмма позволяет точно передать структуру зависимостей. На ней сразу видны образуемые зависимостями циклы и оптимальные способы их разрыва. Видно, когда абстрактные классы зависят от конкретных, и можно выработать стратегию переориентации таких зависимостей.

Основные понятия

Классы

На рис. 19.1 показана простейшая диаграмма классов. Простым прямоугольником представлен класс `Dialer`. Диаграмме соответствует код, приведенный справа от нее.

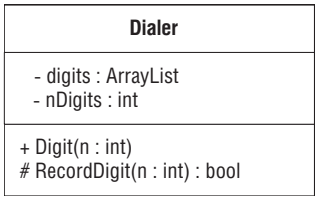



```
public class Dialer
{
}
```

***Рис. 19.1.** Значок класса*

Это наиболее распространенный способ изображения класса. На большинстве диаграмм ничего, кроме имени класса, и не нужно, чтобы понять происходящее.

В прямоугольнике класса может быть несколько отделений. В верхнем записывается имя класса, в среднем – переменные-члены, в нижнем – методы. На рис. 19.2 показаны отделения и соответствующий такому представлению код.




```
public class Dialer
{
  private ArrayList digits;
  private int nDigits;
  public void Digit(int n);
  protected bool RecordDigit(int n);
}
```

***Рис. 19.2.** Значок класса с отделениями и соответствующий код*

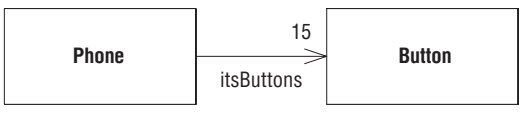
Обратите внимание на символ, предшествующий именам переменных и методов в прямоугольнике класса. Минусом (–) обозначаются закрытые (`private`) члены, решеткой (#) – защищенные (`protected`), плюсом (+) – открытые (`public`).

Тип переменной или аргумента метода указывается после двоеточия, стоящего за их именами. А вслед за именем метода после двоеточия указывается тип возвращаемого значения.

Иногда подобные детали полезны, но злоупотреблять ими не стоит. UML-диаграммы – не место для объявления переменных и методов. Это лучше делать в исходном коде. Использовать описанные дополнения следует лишь в том случае, когда они существенны с точки зрения назначения диаграммы.

Ассоциация

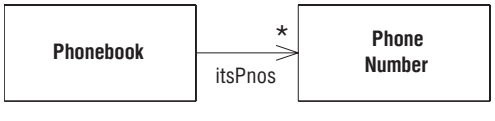
Ассоциации между классами чаще всего представляют переменные экземпляра, в которых хранятся ссылки на другие объекты. Например, на рис. 19.3 показана ассоциация между классами `Phone` и `Button`. Направление стрелки говорит о том, что в `Phone` хранится ссылка на `Button`. Рядом со стрелкой указывается имя переменной экземпляра, а число говорит о том, сколько в этой переменной может храниться ссылок.




```
public class Phone
{
  private Button
itsButtons[15];
}
```

Рис. 19.3. *Ассоциация*

На рис. 19.3 с объектом `Phone` связано 15 объектов `Button`. На рис. 19.4 показана ситуация, когда ограничения на количество нет. Объект `Phonebook` (Телефонная книга) может быть связан со *многими* объектами `PhoneNumber`. (Звездочкой обозначается понятие *много*.) В C# такое отношение чаще всего реализуется с помощью класса `ArrayList` или иных классов коллекций.




```
public class Phonebook
{
  private ArrayList itsPnos;
}
```

Рис. 19.4. *Ассоциация один-ко-многим*

Можно было бы сказать: «`Phonebook` *имеет* много `PhoneNumber`». Но я стараюсь избегать слова *имеет*. И не без причины. Общеупотребительные в ООП глаголы ИМЕЕТ и ЯВЛЯЕТСЯ не раз приводили к печальным недоразумениям. Поэтому не ждите от меня употребления стандартных терминов. Я буду пользоваться словами, описывающими, что именно происходит в программе, например *связан с*.

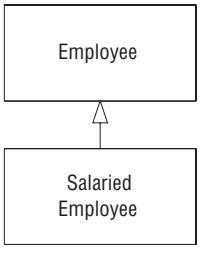
Наследование

В UML нужно очень внимательно относиться к остриям стрелок. На рис. 19.5 показано, почему. Стрелка, указывающая на `Employee`, обо-

значает *наследование*.[1] Если небрежно отнестись к рисованию стрелок, то будет непонятно, имеется ли в виду наследование или ассоциация. Чтобы не возникало путаницы, я часто располагаю классы, связанные отношением наследования, по вертикали, а связанные отношением ассоциации – по горизонтали.



В UML направления всех стрелок определяются *зависимостью в исходном коде*. Так как имя `Employee` упомянуто в классе `SalariedEmployee`, а не наоборот, то стрелка указывает на `Employee`. Таким образом, в UML стрелка наследования всегда направлена в сторону базового класса.




```
public class Employee
{
  ...
}

public class SalariedEmployee : Employee
{
  ...
}
```

***Рис. 19.5.** Наследование*

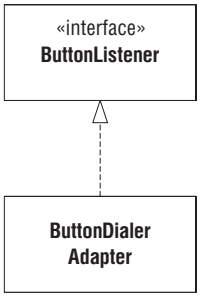
В UML есть специальная нотация для того вида наследования, который существует в C# между классом и интерфейсом. Это пунктирная стрелка наследования, показанная на рис. 19.6.[2] Возможно, впоследствии вы поймаете меня на том, что я забываю рисовать пунктиром стрелки, ведущие на интерфейс. Предлагаю и вам игнорировать пунктир, когда вы рисуете на доске. Жизнь слишком коротка, чтобы рисовать стрелки пунктиром.

На рис. 19.7 показан еще один способ донесения той же информации. Интерфейсы можно рисовать в виде «леденцов на палочке», соединенных с классами, которые их реализуют. Такую нотацию мы часто встречаем в проектах, реализованных с помощью технологии COM.

---

[1] На самом деле она обозначает обобщение, но, с точки зрения программиста на C#, различие чисто теоретическое.

[2] Такое отношение называется *реализует*. За ним стоит нечто большее, чем просто наследование интерфейса, но обсуждение данного различия выходит за рамки этой книги, да, пожалуй, и за рамки интересов всех тех, кто зарабатывает на хлеб написанием кода.




```
interface ButtonListener
{
  ...
}

public class ButtonDialerAdapter
  : ButtonListener
{
  ...
}
```

Рис. 19.6. *Отношение «реализует»*

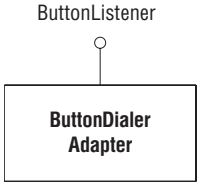
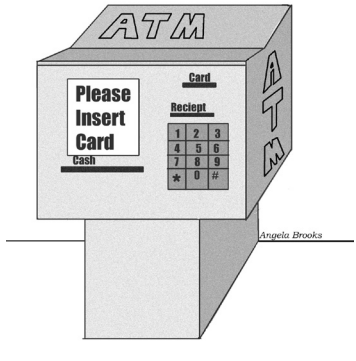


Рис. 19.7. *Обозначение интерфейса «леденцом на палочке»*

Пример диаграммы классов

На рис. 19.8 приведена простая диаграмма классов, входящих в систему банкомата. Эта диаграмма интересна не только тем, что на ней показано, но и тем, что не показано. Обратите внимание, что я дал себе труд пометить все интерфейсы. Я считаю крайне важным, чтобы читатель знал, какие классы я хочу сделать интерфейсами, а какие реализовать. Например, из диаграммы сразу видно, что класс `WithdrawalTransaction` общается с интерфейсом `CashDispenser`. Понятно, что в системе должен быть класс, реализующий `CashDispenser`, но здесь нас не интересует его конкретное воплощение.

Заметьте также, что я не стал подробно документировать методы различных интерфейсов ИП. Конечно, в интерфейсе `WithdrawalUI` будут и еще методы, кроме двух показанных. Как насчет методов `PromptForAccount` или `InformCashDispenserEmpty`? Но помещать их на эту диаграмму означало бы только загромоздить ее. Приведя репрезентативную выборку методов, я сообщил читателю идею. И это все, что необходимо.



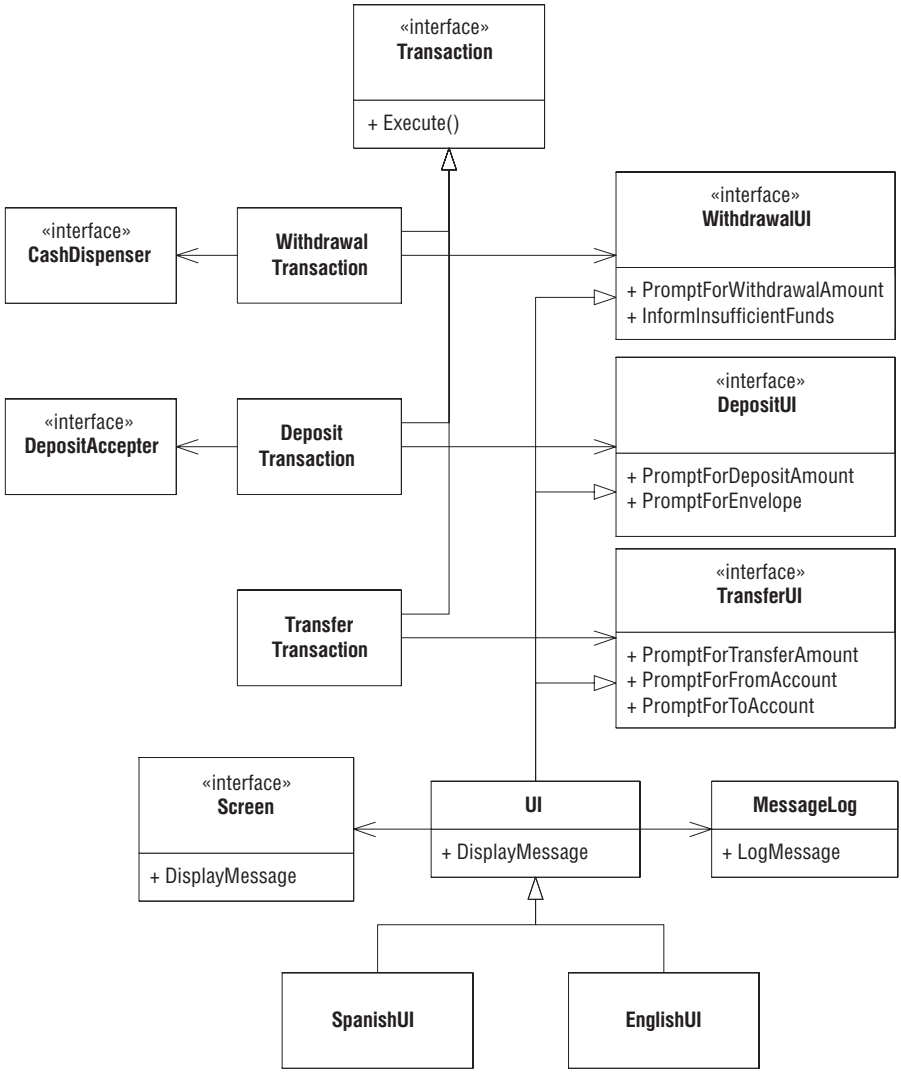


Рис. 19.8. *Диаграмма классов банкомата*

Еще отметим соглашение о расположении ассоциаций по горизонтали, а наследования – по вертикали. Это очень помогает не путать два принципиально различных вида отношений. Без такого соглашения было бы очень трудно извлечь смысл из переплетения стрелок.

Посмотрите, как я выделил в диаграмме три разных зоны. Транзакции и их действия находятся слева, различные интерфейсы ИП – справа, а реализация ИП – внизу. Обратите также внимание, что соединений

между группами немного и они расположены регулярно. В одном случае это три ассоциации, причем все направлены в одну сторону. В другом – три отношения наследования, объединенные в одну линию. Такие группы и способ их соединения помогают читателю выделить взаимосвязанные части диаграммы.

Глядя на эту диаграмму, вы должны быть в состоянии *увидеть* код. Насколько код в листинге 19.1 близок к тому, как вы реализовали бы ИП?

Листинг 19.1. UI.cs

```
public abstract class UI :
  WithdrawalUI, DepositUI, TransferUI
{
  private Screen itsScreen;
  private MessageLog itsMessageLog;

  public abstract void PromptForDepositAmount();
  public abstract void PromptForWithdrawalAmount();
  public abstract void InformInsufficientFunds();
  public abstract void PromptForEnvelope();
  public abstract void PromptForTransferAmount();
  public abstract void PromptForFromAccount();
  public abstract void PromptForToAccount();

  public void DisplayMessage(string message)
  {
    itsMessageLog.LogMessage(message);
    itsScreen.DisplayMessage(message);
  }
}
```

Детали

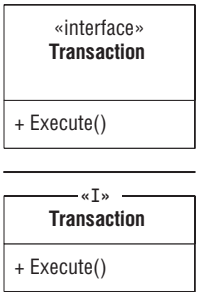
UML-диаграмму классов можно снабдить многочисленными деталями и дополнениями. Как правило, делать этого не нужно. Но в некоторых случаях они бывают полезны.

Стереотипы классов

Стереотипы классов указываются в кавычках-«елочках»[1], обычно над именем класса. Мы уже встречались с ними раньше. Обозначение «interface» на рис. 19.8 – это стереотип класса. Программисты, пишущие на C#, могут использовать два стандартных стереотипа: «interface» и «utility».

[1] Так называются кавычки, которые выглядят, как двойные угловые скобки. Именно такие. Если вы решите, что парой знаков меньше и больше роли не сыграет, то познакомитесь с полицейским управлением UML.

«Interface». Все методы классов, помеченных этим стереотипом, являются абстрактными. Ни один из них нельзя реализовывать. Кроме того, в таких классах не может быть переменных экземпляра. Разрешено включать лишь статические переменные. Эта концепция в точности соответствует интерфейсам в языке C#.[1] См. рис. 19.9.

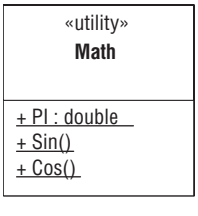



```
interface Transaction
{
  public void Execute();
}
```

Рис. 19.9. *Стереотип класса «interface»*

Я так часто рисую интерфейс, что каждый раз выписывать на доске стереотип целиком утомительно. Поэтому я нередко использую сокращенную запись, показанную в нижней части рис. 19.9. Это расходится со стандартом UML, но гораздо удобнее.

«Utility». Все методы и переменные класса со стереотипом «utility» статические. Буч называет такие классы *служебными* (Class utilities)[2] – рис. 19.10.




```
public class Math
{
  public static readonly double PI =
    3.14159265358979323;
  public static double Sin(double theta){...}
  public static double Cos(double theta){...}
}
```

Рис. 19.10. *Стереотип класса «utility»*

Если хотите, можете определить собственные стереотипы. Я часто употребляю стереотипы «persistent», «C-API», «struct» и «function». Нужно лишь, чтобы читатели диаграммы понимали, что означает стереотип.

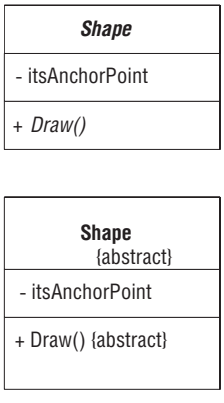
---

[1] В языке C# интерфейс не может содержать статических членов – ни переменных, ни методов (в отличие от Java). – *Прим. перев.*

[2] [Booch94], стр. 186

Абстрактные классы

В UML есть два способа показать, что класс или метод абстрактный. Можно либо записать его имя курсивом, либо воспользоваться свойством {abstract}. Оба варианта изображены на рис. 19.11.




```
public abstract class Shape
{
  private Point itsAnchorPoint;
  public abstract void Draw();
}
```

Рис. 19.11. *Абстрактные классы*

На доске писать курсивом затруднительно, а свойство {abstract} слишком длинное. Поэтому, рисуя на доске, я применяю соглашение, показанное на рис. 19.12. Опять же, это не стандартный UML, но для доски гораздо удобнее.[1]

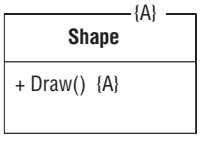


Рис. 19.12. *Неофициальное обозначение абстрактных классов*

Свойства

Свойства, например {abstract}, можно включить в любой класс. Они сообщают дополнительную информацию, которая обычно не является частью класса. Вы и сами можете создавать любые свойства.

Свойства записываются в виде списка пар имя/значение, разделенных запятыми:

```
{author=Martin, date=20020429, file=shape.cs, private}
```

[1] Кто-то из читателей, возможно, помнит нотацию Буча. Одним из ее несомненных плюсов было удобство. Она была прямо-таки идеальна для записи на доске.

Показанные в этом примере свойства не являются частью UML. Кроме того, свойства необязательно должны быть как-то связаны с кодом, а могут содержать произвольные метаданные. {abstract} – единственное определенное в UML свойство, полезное для программистов.

Если для свойства не указано значение, предполагается булевское значение true. Следовательно, {abstract} и {abstract = true} – одно и то же. Свойства записываются справа внизу под именем класса, как показано на рис. 19.13.

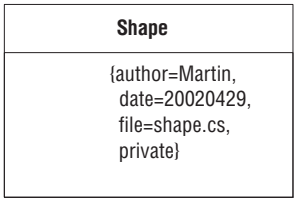
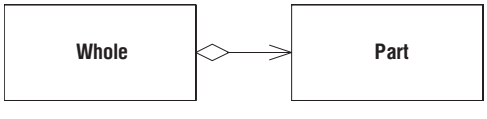


***Рис. 19.13.** Свойства*

Если не считать свойства {abstract}, не знаю, для чего может быть полезно это средство. Лично я за многие годы рисования UML-диаграмм ни разу не нашел свойствам применения.

Агрегирование

Агрегирование – это частный случай ассоциации, подразумевающий отношение часть/целое. На рис. 19.14 показано, как оно изображается и реализуется. Отметим, что реализация неотличима от ассоциации общего вида. Это просто подсказка.




```
public class Whole
{
  private Part itsPart;
}
```

***Рис. 19.14.** Агрегирование*

К сожалению, в UML нет строгого определения этого отношения. Это ведет к недоразумениям, потому что разные программисты и аналитики понимают под ним что-то свое. Поэтому я вообще не употребляю это отношение и вам не рекомендую. Вообще, в версии UML 2.0 оно практически исчезло.

В UML есть одно твердое правило, касающееся агрегирования: целое не может быть собственной частью. Поэтому *экземпляры* не могут образовывать циклы агрегирований. Один объект не может агрегировать себя самого, два объекта не могут агрегировать друг друга, три объекта не могут образовывать кольцо агрегирований и т. д. См. рис. 19.15.

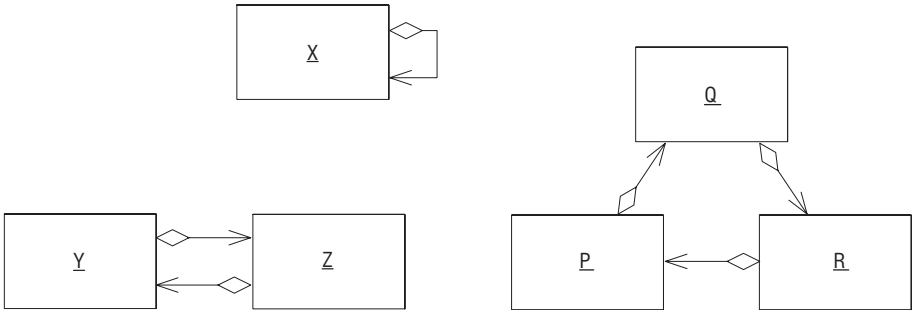
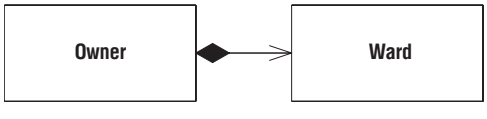


Рис. 19.15. *Недопустимые циклы агрегирований экземпляров*

Не думаю, что это определение сколько-нибудь полезно. Как часто меня заботит вопрос о том, чтобы ориентированный граф экземпляров был ациклическим? Не очень часто. Поэтому в тех диаграммах, что я обычно рисую, это отношение бесполезно.

Композиция

Композиция – это частный случай агрегирования, который показан на рис. 19.16. Снова отмечу, что реализация неотличима от ассоциации общего вида. Но на сей раз причина в том, что это отношение находит мало применений в программах на языке C#. Напротив, программисты, пишущие на C++, используют его сплошь и рядом.




```
public class Owner
{
  private Ward itsWard;
}
```

Рис. 19.16. *Композиция*

К композиции применимо то же правило, что и к агрегированию. Не должно быть циклов экземпляров. Владелец (Owner) не может быть одновременно владеемым (Ward). Однако в UML определение композиции более точное.

- Ни один владеемый не может принадлежать одновременно двум владельцам. Диаграмма объектов на рис. 19.17 некорректна. Однако соответствующая диаграмма классов вполне допустима. Владелец может передавать право владения другому владельцу.
- Владелец несет ответственность за время жизни владеемого. Если владелец уничтожается, то и владеемый должен быть уничтожен вместе с ним. Если владелец копируется, то и владеемый должен быть скопирован вместе с ним.

В C# уничтожение объектов производится неявно сборщиком мусора, поэтому редко возникает необходимость управлять временем жизни

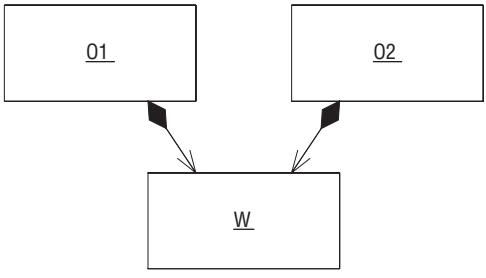
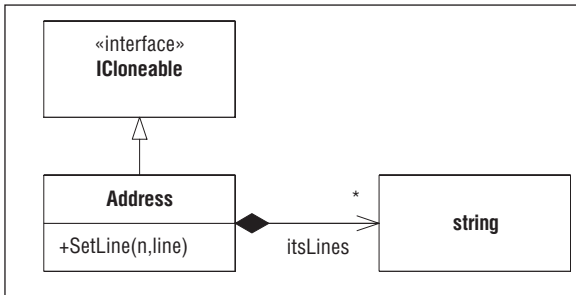


Рис. 19.17. *Недопустимая композиция*

объекта. Идея глубокого копирования не чужда C#, но отображать семантику глубокого копирования на диаграммах случается нечасто. Поэтому если мне и приходилось применять отношение композиции для описания программ на C#, то довольно редко.

На рис. 19.18 показано, как композиция используется для обозначения глубокого копирования. Имеется класс `Address`, содержащий много строк (объектов `string`). В каждой строке хранится один компонент




```
public class Address : ICloneable
{
  private ArrayList itsLines = new ArrayList();

  public void SetLine(int n, string line)
  {
    itsLines[n] = line;
  }

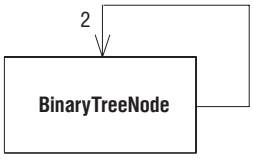
  public object Clone()
  {
    Address clone = (Address) this.MemberwiseClone();
    clone.itsLines = (ArrayList) itsLines.Clone();
    return clone;
  }
}
```

Рис. 19.18. *Глубокое копирование, подразумеваемое композицией*

адреса. Ясно, что при копировании адреса хотелось бы, чтобы копию можно было изменять независимо от оригинала. Поэтому мы выполняем глубокое копирование. Отношение композиции между классами `Address` и `String` означает, что копирование должно быть глубоким.[1]

Кратность

В объектах могут храниться массивы или коллекции других объектов либо несколько ссылок на однородные объекты в различных переменных экземпляра. В UML это может быть выражено путем размещения выражения *кратности* на дальнем конце ассоциации. В качестве выражения кратности могут выступать одиночные числа, диапазоны или сочетания того и другого. Например, на рис. 19.19 показан класс `BinaryTreeNode`, в котором кратность равна 2.




```
public class BinaryTreeNode
{
  private BinaryTreeNode leftNode;
  private BinaryTreeNode rightNode;
}
```

Рис. 19.19. *Простой пример кратности*

Вот еще несколько допустимых форм записи кратности:

- Цифра — точное число элементов
- * или 0..* — 0 или более
- 0..1 — 0 или 1; в Java часто реализуется ссылкой, которая может быть равна `null`
- 1..* — 1 или более
- 3..5 — от трех до пяти
- 0,2..5,9..* — странно, но допустимо

Стереотипы ассоциаций

Ассоциации можно помечать стереотипами, которые изменяют их интерпретацию. На рис. 19.20 показаны стереотипы, которыми я часто пользуюсь.

Стереотип «create» означает, что экземпляр класса на конце ассоциации создается экземпляром класса в ее начале. Иначе говоря, объект-источник создает объект-цель и затем передает его другим частям системы. В примере я показал типичную фабрику.

---

[1] *Упражнение*: почему достаточно клонировать коллекцию `itsLines`? Почему нет необходимости клонировать фактические экземпляры `string`?

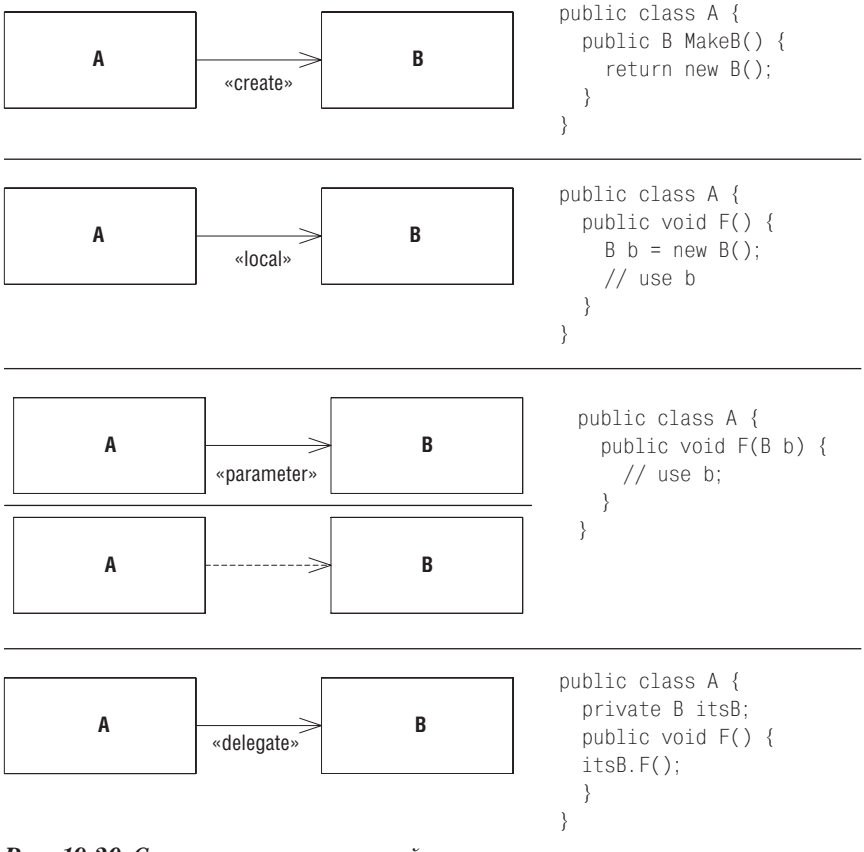


Рис. 19.20. *Стереотипы ассоциаций*

Стереотип «local» применяется, когда класс-источник создает экземпляр класса-цели и сохраняет его в локальной переменной. Отсюда следует, что созданный экземпляр живет не дольше, чем создавший его метод, то есть он не хранится ни в какой переменной экземпляра и не передается другим частям системы.

Стереотип «parameter» показывает, что класс-источник получает доступ к объекту-цели через параметр одного из своих методов. И снова это означает, что источник забывает об этом объекте, как только метод возвращает управление. Объект-цель не сохраняется в переменной экземпляра.

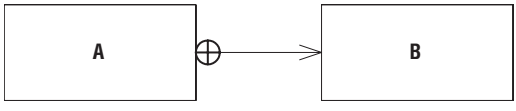
Использование пунктирной стрелки зависимости – общеупотребимый и удобный способ обозначения параметров. Обычно я предпочитаю ее стереотипу «parameter».

Стереотип «delegate» используется, когда класс-источник переадресует вызов своего метода объекту-цели. Этот прием встречается в различных паттернах проектирования: Заместитель (Proxy), Декоратор

(Decorator) и Компоновщик (Composite)[1]. Поскольку я очень часто прибегаю к этим паттернам, то нахожу этот стереотип полезным.

Вложенные классы

Вложенные классы представляются в UML ассоциацией, дополненной перечеркнутым кружком, как показано на рис. 19.21.




```
public class A {
  private class B {
    ...
  }
}
```

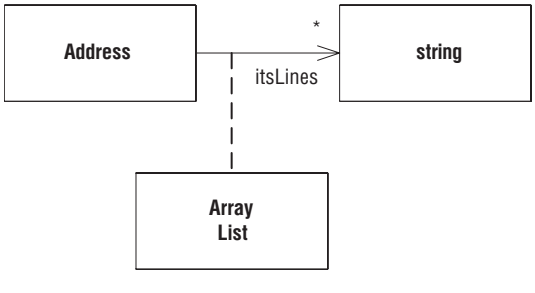
Рис. 19.21. *Вложенный класс*

Классы ассоциаций

Ассоциации с кратностью сообщают, что источник соединен с несколькими экземплярами цели, но диаграмма ничего не говорит о том, какой именно контейнер используется. Эту информацию можно представить с помощью класса ассоциации, как показано на рис. 19.22.

Класс ассоциации показывает, как реализована эта конкретная ассоциация. На диаграмме он выглядит как обычный класс, соединенный с ассоциацией пунктирной линией. В языке C# это интерпретируется следующим образом: класс-источник содержит ссылку на класс ассоциации, который, в свою очередь, содержит ссылку на объект-цель.

Классами ассоциаций могут быть также классы, которые пишутся для хранения ссылок на экземпляры какого-то другого класса. Иногда эти классы навязывают бизнес-правила. Например, на рис. 19.23 класс `Company` содержит несколько экземпляров класса `Employee`, хранящихся в объекте `EmployeeContracts`. Честно говоря, эта нотация никогда не казалась мне особенно удобной.




```
public class Address {
  private ArrayList itsLines;
};
```

Рис. 19.22. *Класс ассоциации*

[1] [GOF95], стр. 163, 175, 207

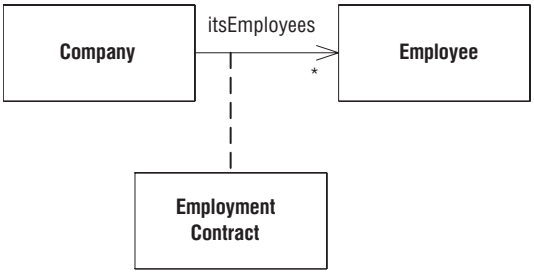


Рис. 19.23. *Контракт с работником*

```
public class Company {
  private EmploymentContract[] itsEmployees;
};
```

Квалификаторы ассоциаций

Квалификаторы ассоциаций используются, когда ассоциация реализована с помощью некоего ключа, а не обычной ссылки C#. В примере на рис. 19.24 показан класс `LoginTransaction`, ассоциированный с `Employee`. Ассоциация опосредована переменной-членом `empid`, в которой хранится ключ объекта `Employee` в базе данных.

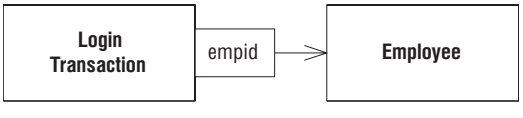


Рис. 19.24. *Квалификатор ассоциации*

```
public class LoginTransaction {
  private string empid;
  public string Name() {
    get {
      Employee e=DB.GetEmp(empid);
      return e.GetName();
    }
  }
}
```

Я редко нахожу применение этой нотации. Иногда удобно показать, что один объект ассоциирован с другим с помощью ключа базы данных или словаря. Однако важно, чтобы все читатели диаграммы знали, как квалификатор используется для доступа к объекту. А вот это как раз из самой нотации неочевидно.

Заключение

В UML есть множество различных приспособлений, дополнений и черт его знает чего еще. Их так много, что можно потратить кучу времени на то, чтобы стать знатоком языка UML и впоследствии заниматься тем, чем занимаются все знатоки: читать документы, которые больше никто понять не может.

В этой главе я не описывал большинство загадочных и запутанных средств UML, а показал лишь те из них, которыми пользуюсь сам. Надеюсь, что попутно я исподволь внушил вам мысль о ценности минимализма. Лучше применять UML в малых дозах, чем страдать от передозировки.

Библиография

[Booch94] Grady Booch «Object-Oriented Analysis and Design with Applications», 2-d ed. Addison-Wesley, 1994.[1]

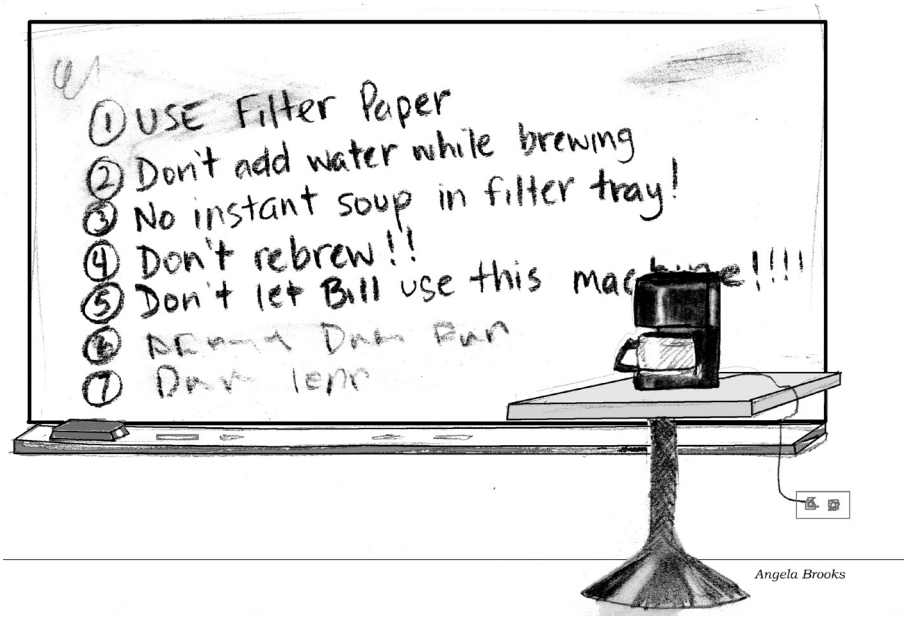
[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.[2]

[1] Гради Буч и др. «Объектно-ориентированный анализ и проектирование с примерами приложений», 3-е изд. – Пер. с англ. – Вильямс, 2010.

[2] Эрик Гамма «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – Питер, 2007.

20

Эвристика и кофе



Angela Brooks

Последние лет десять я преподавал объектно-ориентированное проектирование профессиональным разработчикам ПО. Все курсы были построены по единому образцу: до обеда – лекция, после обеда – упражнения. Для работы над упражнениями я разбивал группу на отдельные команды и каждой поручал решить ту или иную задачу проектирования с использованием UML. На следующее утро мы просили одну-две команды представить свои решения на доске и высказывали критические замечания.

Я провел сотни таких курсов и выделил ряд типичных ошибок проектирования, допускаемых учащимися. В этой главе я расскажу о нескольких самых распространенных ошибках, объясню, почему это именно ошибки, и покажу, как их исправить. А затем мы решим задачу таким образом, который, на мой взгляд, элегантно справляется со всеми проблемами проектирования.

Кофеварка Mark IV Special

На первом занятии по курсу объектно-ориентированного проектирования я определяю основные понятия: классы, объекты, отношения, методы, полиморфизм и т. д. По ходу дела я рассказываю об основах UML. Таким образом, аудитория знакомится с базовыми понятиями, терминологией и инструментами объектно-ориентированного проектирования.

После обеда я предлагаю группе следующее упражнение: спроектировать программу для управления простой кофеваркой. Вот какая предлагается спецификация.[1]

Спецификация

Кофеварка Mark IV Special может за раз приготовить до 12 чашек кофе. Пользователь вставляет фильтр в фильтродержатель, засыпает в фильтр молотый кофе и устанавливает фильтродержатель в приемник. Затем он заливает в контейнер до 12 чашек воды и наживает кнопку «Сварить». Вода нагревается до кипения. Под давлением пара кипяток подается на засыпанный кофе, и готовый напиток льется через фильтр в кофейник. Кофейник подогревается нагревательной подставкой, которая включается, только если в кофейнике есть кофе. Если снять кофейник с подставки в то время, когда кипяток еще поступает из кипятильника, то подача воды прекращается, чтобы кофе не лился на подставку. Имеется следующее оборудование, которым необходимо управлять:

- Нагревательный элемент для кипятильника. Может быть включен или выключен.
- Нагревательный элемент для подставки. Может быть включен или выключен.
- Датчик нагревательной подставки. Может находиться в одном из трех состояний: `warmerEmpty` (на подставке нет кофейника), `potEmpty` (кофейник пуст), `potNotEmpty` (кофейник не пуст).
- Датчик кипятильника, определяющий, есть ли в нем вода. Может находиться в одном из двух состояний: `boilerEmpty` (кипятильник пуст) или `boilerNotEmpty` (кипятильник не пуст).

[1] Эта задача появилась в моей первой книге [Martin1995], стр. 60.

- Кнопка «Сварить». Ее нажатие запускает процесс приготовления кофе. На кнопке есть индикатор, который зажигается, когда процесс закончился и кофе готов.
- Клапан сброса давления, который открывается, чтобы понизить давление в кипятильнике. При понижении давления подача воды в фильтр прекращается. Клапан может быть открыт или закрыт.

Аппаратная часть Mark IV уже спроектирована, и сейчас идет подготовка к производству. Конструкторы даже предоставили низкоуровневый API, чтобы избавить нас от необходимости писать драйвер ввода/вывода, работающий на уровне битов. Описание интерфейсных функций показано в листинге 20.1. Если оно кажется вам странным, вспомните, что писали его конструкторы оборудования.

Листинг 20.1. CoffeeMakerAPI.cs

```
namespace CoffeeMaker
{
  public enum WarmerPlateStatus
  {
    WARMER_EMPTY,
    POT_EMPTY,
    POT_NOT_EMPTY
  };

  public enum BoilerStatus
  {
    EMPTY,NOT_EMPTY
  };

  public enum BrewButtonStatus
  {
  PUSHED,NOT_PUSHED
  };

  public enum BoilerState
  {
    ON,OFF
  };

  public enum WarmerState
  {
    ON,OFF
  };

  public enum IndicatorState
  {
    ON,OFF
  };
```

```
public enum ReliefValveState
{
  OPEN, CLOSED
};

public interface CoffeeMakerAPI
{
  /*
   * Эта функция возвращает состояние датчика нагревательной
   * подставки. Датчик определяет, есть ли на подставке кофейник
   * и есть ли в нем кофе.
   */

  WarmerPlateStatus GetWarmerPlateStatus();

  /*
   * Эта функция возвращает состояние переключателя кипятильника.
   * Это поплавковый переключатель, который определяет, осталось
   * ли в кипятильнике воды больше, чем на половину чашки.
   */

  BoilerStatus GetBoilerStatus();

  /*
   * Эта функция возвращает состояние кнопки «Сварить».
   * Это кнопка без фиксации, которая запоминает свое состояние.
   * При каждом обращении данная функция возвращает
   * запомненное состояние,после чего сбрасывает его
   * в NOT_PUSHED (не нажата).
   *
   * Таким образом, даже если интервалы между вызовами
   * функции очень велики, она все равно распознает,
   * что кнопка «Сварить» нажата.
   */

  BrewButtonStatus GetBrewButtonStatus();

  /*
   * Эта функция включает или выключает
   * нагревательный элемент кипятильника.
   */

  void SetBoilerState(BoilerState s);

  /*
   * Эта функция включает или выключает
   * нагревательный элемент подставки.
   */

  void SetWarmerState(WarmerState s);
```

```
    /*
     * Эта функция включает или выключает световой индикатор.
     * Индикатор должен загораться по завершении процесса варки
     * и гаснуть, когда пользователь нажимает кнопку «Сварить».
     */

    void SetIndicatorState(IndicatorState s);

    /*
     * Эта функция открывает или закрывает клапан сброса давления.
     * Если клапан закрыт, то давление пара в кипятильнике подает
     * горячую воду в кофейный фильтр. Если клапан открыт, то пар
     * выходит из кипятильника наружу, так что вода не подается
     * в фильтр.
     */

    void SetReliefValveState(ReliefValveState s);
  }
}
```

Если хотите принять вызов, то отложите книгу и попробуйте спроектировать эту программу самостоятельно. Не забывайте, что вы проектируете ПО для простой встроенной системы реального времени. От своих студентов я ожидаю получить набор диаграмм классов, последовательности и состояний.

Типичное, но никуда не годное решение



Чаще всего студенты предлагают решение, показанное на рис. 20.1. В нем центральный класс `CoffeeMaker` окружен спутниками, которые управляют различными устройствами. `CoffeeMaker` содержит ссылки на классы `Boiler` (Кипятильник), `War merPlate` (Нагревательная подставка), `Button` (Кнопка) и `Light` (Лампочка). Класс `Boiler` содержит `BoilerSensor` (Датчик кипятильника) и `BoilerHeater` (Нагреватель кипятильника). Класс `WarmerPlate` содержит `PlateSensor` (Датчик подставки) и `PlateHeater` (Нагреватель подставки). Наконец, два базовых класса `Sensor` и `Heater` выступают в роли родителей соответствующих элементов для `Boiler` и `WarmerPlate`.

Начинающему трудно понять, насколько плоха такая структура. В этой диаграмме скрыто несколько довольно серьезных ошибок. Многие из них останутся незамеченными, пока вы не начнете воплощать этот дизайн в коде и не обнаружите, что код получается нелепым.

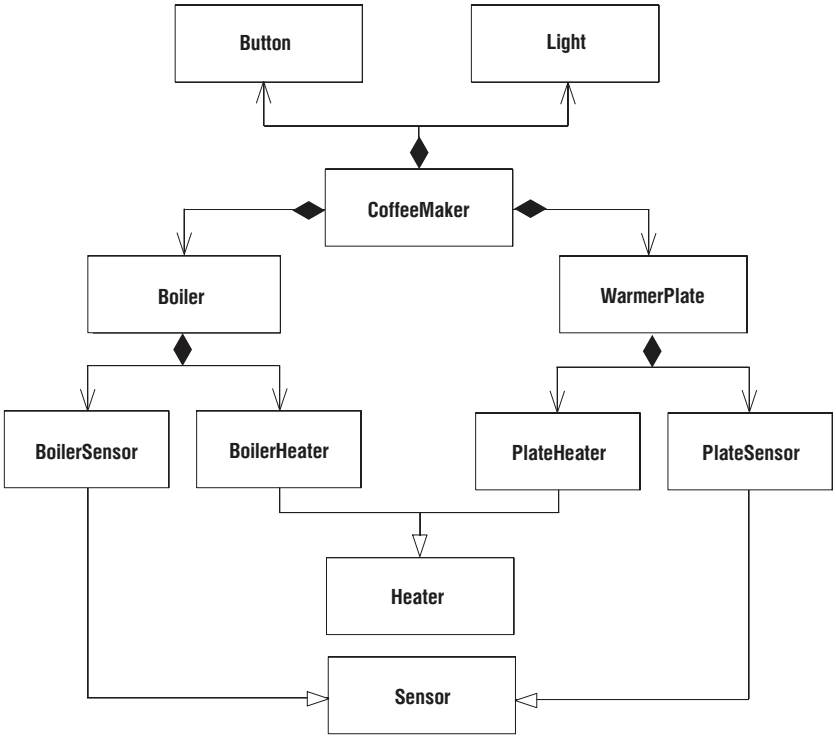


Рис. 20.1. *Гиперконкретная кофеварка*

Но прежде чем переходить к ошибкам дизайна, рассмотрим недостатки самой UML-диаграммы.

Отсутствующие методы. Самая вопиющая проблема на рис. 20.1 – полное отсутствие методов. Мы же собираемся писать *программу*, а программа – это прежде всего поведение! Ну и где на этой диаграмме поведение?

Рисуя диаграммы без методов, проектировщик, вероятно, разбивает программу на части по какому-то признаку, отличному от поведения. Но такие разбиения почти всегда в корне ошибочны. Именно поведение дает ключ к тому, как следует структурировать программу.

Классы-миражи. Если задуматься о том, какие методы следовало бы поместить в класс `Light`, то начинают проступать недостатки данного дизайна. Очевидно, что индикатор можно включить или выключить. Следовательно, в класс `Light` следовало бы добавить методы `On()` и `Off()`. Но как могла бы выглядеть их реализация? Взгляните на листинг 20.2.

Листинг 20.2. Light.cs

```
public class Light {
  public void On() {
```

```
    CoffeeMakerAPI.SetIndicatorState(IndicatorState.ON);
  }

  public void Off() {
    CoffeeMakerAPI.SetIndicatorState(IndicatorState.OFF);
  }
}
```

Какой-то странный класс. Прежде всего, в нем нет никаких переменных-членов. Это удивительно, так как объект обычно обладает состоянием, которым манипулирует. Более того, методы `On()` и `Off()` просто делегируют работу методу `SetIndicatorState` класса `CoffeeMakerAPI`. Получается, что класс `Light` – всего лишь транслятор вызовов и ничего полезного не делает.

То же самое можно сказать о классах `Button`, `Boiler` и `WarmerPlate`. Это просто адаптеры, транслирующие вызов функции из одного вида в другой. Их вообще можно убрать из дизайна, и логика класса `CoffeeMaker` при этом не изменится. Только он будет напрямую обращаться к `CoffeeMakerAPI` в обход адаптеров.

Рассмотрев потенциально возможные методы, а затем их предполагаемый код, мы лишили эти классы главенствующего положения, которое они занимали на рис. 20.1, и понизили до простых заглушек, в общем-то не имеющих прав на существование. Поэтому я назвал их *классами-миражами*.

Иллюзорная абстракция

Обратимся теперь к базовым классам `Sensor` и `Heater` на рис. 20.1. Прочитав предыдущий раздел, вы, наверное, убедились в том, что производные от них классы обратились в дым, но что сказать о базовых? На первый взгляд они кажутся вполне осмысленными. Только вот не видно, куда приспособить их подклассы.

Абстракции – вещь хитрая. Человек встречается с ними повсюду, но немногие подходят на роль классов. Вот и для этих абстракций не нашлось места в данном дизайне. И в этом легко убедиться, задав себе следующий вопрос: *«Кто их использует?»*

Ни один класс в системе не использует классы `Sensor` и `Heater`. Но раз их никто не использует, то зачем они нужны? Иногда с базовым классом, которым никто не пользуется, можно смириться, потому что в нем находится какой-то код, общий для всех его подклассов, но в данном случае базовые классы никакого кода не содержат. В лучшем случае их методы абстрактны. Взгляните, к примеру, на интерфейс класса `Heater` в листинге 20.3. Класс, содержащий только абстрактные методы, которым не пользуются никакие другие классы, официально признается бесполезным.

Листинг 20.3. Heater.cs

```
public interface Heater {
  void TurnOn();
  void TurnOff();
}
```

Класс `Sensor` (листинг 20.4) еще хуже! Как и `Heater`, он содержит только абстрактные методы и не имеет ни одного пользователя. Но при этом еще и непонятно, что возвращает его единственный метод! В классе `BoilerSensor` он может возвращать два различных значения, а в классе `WarmerPlateSensor` – три. Короче говоря, не удается специфицировать контракт класса `Sensor` в его интерфейсе. Можно лишь сказать, что он возвращает значение типа `int`. Маловато будет.

Листинг 20.4. Sensor.cs

```
public interface Sensor {
  int Sense();
}
```

А произошло вот что. Мы прочитали спецификацию, нашли в ней ряд многообещающих существительных, сделали какие-то выводы о том, как они соотносятся друг с другом, и на основе этого умозрительного рассуждения нарисовали UML-диаграмму. Если бы мы положили эти решения в основу архитектуры и реализовали их буквально, то получился бы всемогущий класс `CoffeeMaker`, окруженный спутниками-миражами. Тогда с тем же успехом можно программировать сразу на C!

Классы-боги. Все знают, что классы-боги – порочная идея. Мы не хотим втискивать всю систему в единственный объект или функцию. Одна из целей объектно-ориентированного проектирования – рассредоточить поведение системы по разным классам или функциям. Однако выясняется, что многие объектные модели, которые представляются рассредоточенными, на деле оказываются местом обитания богов. Рисунок 20.1 дает наглядный пример. Вроде бы мы видим множество классов с интересным поведением. Но стоит углубиться в код, реализующий эти классы, как обнаруживается, что лишь один из них – `CoffeeMaker` – действительно обладает содержательным поведением, а все остальные – иллюзорные абстракции, или классы-миражи.



Улучшенное решение

Решение задачи о кофеварке – интересное упражнение на абстрагирование. Большинству разработчиков, начинающих изучать ООП, результат кажется поразительным.

Ключ к решению этой (да и любой другой) задачи состоит в том, чтобы вернуться на шаг назад и отделить детали от сути. Забудем о кипятильниках, клапанах, нагревателях, датчиках и прочих мелких деталях и сосредоточимся на самой проблеме. В чем заключается задача? Как сварить кофе.

Ну и как же его сварить? Простейшее, самое очевидное решение задачи состоит в том, чтобы залить молотый кофе кипятком и собрать получившийся настой в какой-то сосуд. Откуда берется кипяток? Назовем его источник `HotWaterSource`. Куда наливается кофе? Назовем эту посудину `ContainmentVessel`[1].

Подходят ли эти две абстракции на роль классов? Обладает ли `HotWaterSource` поведением, которое можно воплотить в коде? Есть ли у `ContainmentVessel` нечто, чем могла бы управлять программа? Если обратиться к устройству Mark IV, то можно представить себе, что кипятильник, клапан и датчик кипятильника выступают в роли `HotWaterSource`. Класс `HotWaterSource` будет отвечать за нагрев воды и подачу ее к молотому кофе, проходя через который она будет стекать в `ContainmentVessel`. Можно также предположить, что нагревательная подставка и ее датчик будут играть роль класса `ContainmentVessel`. Его задача – следить за тем, чтобы кофе был горячим, и уведомлять нас о том, есть ли в сосуде напиток.

Как отразить эти соображения на UML-диаграмме? На рис. 20.2 показана одна из возможных схем. `HotWaterSource` и `ContainmentVessel` представлены в виде классов, связанных ассоциацией «поток кофе».

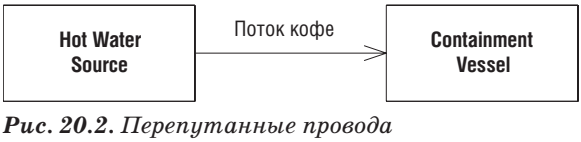


Рис. 20.2. *Перепутанные провода*

Эта ассоциация демонстрирует ошибку, которую часто допускают начинающие проектировщики. Она отражает некий физический аспект задачи, а не поведение программы. Тот факт, что кофе течет из источника `HotWaterSource` в сосуд `ContainmentVessel`, не имеет никакого отношения к ассоциации между классами.

---

[1] Это имя особенно подходит для кофе, который люблю готовить *я* сам. (Под Vessel подразумевается большой сосуд. – *Прим. перев.*)

Например, что если часть программы внутри ContainmentVessel будет сообщать HotWaterSource, когда открывать и закрывать поток кипятка в сосуд? Это можно изобразить, как показано на рис. 20.3. Обратите внимание, что ContainmentVessel отправляет сообщение HotWaterSource. Это означает, что направление ассоциации на рис. 20.2 прямо противоположно реальному. HotWaterSource вообще не зависит от ContainmentVessel. Наоборот, ContainmentVessel зависит от HotWaterSource.

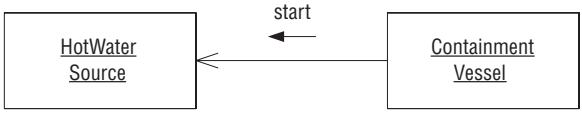


***Рис. 20.3.** Открыть поток кипятка*

Урок простой: ассоциации – это тракты прохождения сообщений между объектами. Они не имеют ничего общего с физическими объектами. Из того, что поток кипятка течет из кипятильника в кофейник, вовсе не следует существование ассоциации, направленной от HotWaterSource к ContainmentVessel.

Я называю ошибки такого вида *перепутанными проводами*, потому что соединяющие классы линии проведены неверно из-за путаницы между логическим и физическим представлениями задачи.

Пользовательский интерфейс кофеварки. Вам должно быть понятно, что в нашей модели кофеварки чего-то не хватает. Имеются классы HotWaterSource и ContainmentVessel, но не видно, как взаимодействует с системой пользователь. В системе должно быть место, где принимаются команды от человека. И еще она должна как-то сообщать человеку о своем состоянии. Разумеется, в Mark IV предусмотрено оборудование специально для этой цели. Это кнопка и лампочка-индикатор, которые играют роль интерфейса с пользователем.

Поэтому добавим в модель кофеварки класс UserInterface. Тем самым мы получаем тройку классов для приготовления кофе по указаниям пользователя.

Прецедент 1: пользователь нажимает кнопку «Сварить». Ну и как в этом случае взаимодействуют три наших класса? Рассмотрим несколько случаев и попытаемся извлечь из них некоторое поведение.

Какой из наших объектов обнаруживает нажатие на кнопку «Сварить»? Понятно, что это должен быть объект UserInterface. Что должен сделать этот объект, когда кнопка нажата?

Наша цель – пустить поток кипятка. Но прежде чем это делать, неплохо бы убедиться, что ContainmentVessel готов к приему кофе. Еще стоило бы проверить готовность HotWaterSource. В случае Mark IV надо удостовериться, что кипятильник полон, а кофейник пуст и стоит на подставке.

Таким образом, объект `UserInterface` сначала посылает сообщения `HotWaterSource` и `ContainmentVessel`, опрашивая их готовность. Это показано на рис. 20.4

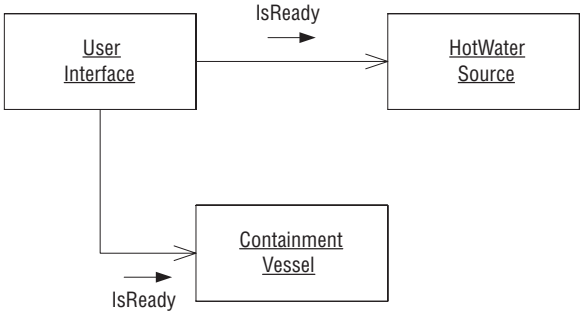


***Рис. 20.4.** Нажата кнопка «Сварить», проверяем готовность*

Если хотя бы в одном случае возвращается `false`, мы отказываемся варить кофе. Объект `UserInterface` может уведомить пользователя о невозможности выполнить запрос. В случае Mark IV можно было бы заставить лампочку несколько раз мигнуть.

Если оба запроса возвращают `true`, то нужно открыть поток кипятка. По-видимому, объект `UserInterface` должен послать сообщение `Start` объекту `HotWaterSource`. В ответ последний сделает то, что необходимо для пуска кипятка. В случае Mark IV он закроет клапан и включит кипятильник. На рис. 20.5 этот сценарий изображен целиком.

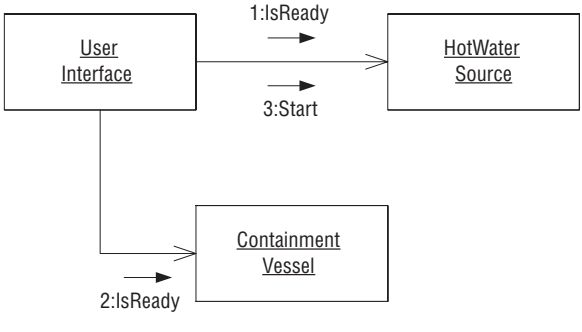


***Рис. 20.5.** Нажата кнопка «Сварить», полный сценарий*

Прецедент 2: приемный сосуд не готов. В случае Mark IV мы знаем, что пользователь может снять кофейник с подставки, когда кофе еще варится. Какой из наших объектов узнает, что кофейник снят? Конечно же, `ContainmentVessel`. Требования к Mark IV говорят, что в этом случае нужно прекратить налив кофе. Следовательно, `ContainmentVessel` должен уметь извещать `HotWaterSource` о необходимости прекратить подачу

воды. А также о необходимости возобновить подачу, когда кофейник поставят на место. Новые методы добавлены на рис. 20.6.

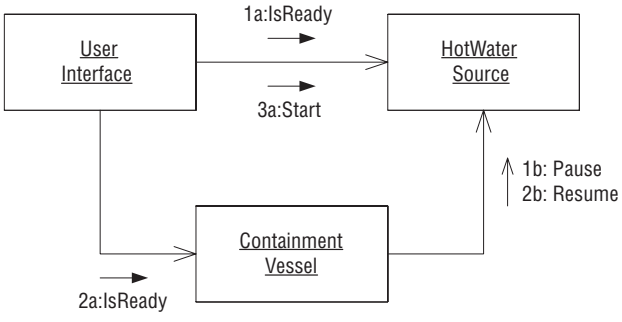


Рис. 20.6. *Приостановка и возобновление подачи кипятка*

Прецедент 3: кофе сварен. Но вот кофе сварен, и подачу кипятка следует прекратить. Какой из наших объектов знает, что процесс приготовления кофе завершен? В случае Mark IV о том, что кипятильник пуст, сообщает датчик кипятильника, поэтому распознает эту ситуацию объект `HotWaterSource`. Однако нетрудно представить себе кофеварку, в которой завершение варки кофе распознает `ContainmentVessel`. Например, что если кофеварка подключена к водопроводу и потому располагает бесконечным запасом воды? Что если мощный генератор СВЧ-излучения нагревает воду по мере ее поступления по трубам в теплоизолированный сосуд?[1] Что если в этом сосуде есть кран, открыв который пользователь наливает себе кофе? В таком случае о том, что сосуд полон и подачу кипятка следует прекратить, должен узнавать датчик в сосуде.

Я хочу всем этим сказать, что в абстрактной предметной области, где есть объекты `HotWaterSource` и `ContainmentVessel`, нельзя однозначно сказать, кто должен обнаруживать, что процесс приготовления кофе завершен. Поэтому я решаю проигнорировать этот вопрос и буду предполагать, что любой объект может сообщить остальным, что все готово.

Какие объекты в нашей модели должны знать, что приготовление кофе завершено? Очевидно, что `UserInterface`, потому что в случае Mark IV он должен зажечь лампочку. Кроме того, должно быть ясно, что и объекту `HotWaterSource` необходимо знать о завершении приготовления кофе, поскольку он должен перекрыть подачу кипятка. В Mark IV он выключит кипятильник и откроет клапан. Должен ли `ContainmentVessel` знать, что кофе готов? Должен ли `ContainmentVessel` что-то сделать или сохранить какую-то пометку, когда приготовление завершится? В Mark IV требуется распознавать, не помещен ли на подставку пустой кофейник; это означает, что пользователь вылил весь имеющийся кофе. В результате

[1] Ладно, шучу. Но все-таки, а вдруг?

Mark IV гасит лампочку. Стало быть, да, `ContainmentVessel` должен знать о завершении процесса. И то же самое рассуждение показывает, что `UserInterface` должен послать сообщение `Start` объекту `ContainmentVessel`, когда процесс варки кофе начинается. На рис. 20.7 показаны эти новые сообщения. Обратите внимание, что сообщение `Done` может посылать как `HotWaterSource`, так и `ContainmentVesslel`.

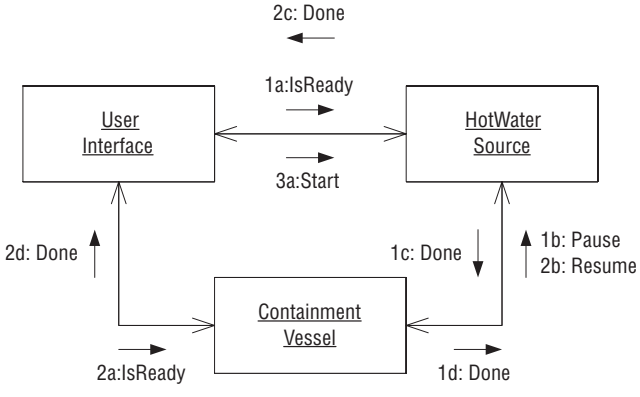


Рис. 20.7. *Обнаружение завершения процесса приготовления кофе*

Прецедент 4: кофе кончился. Mark IV гасит лампочку, когда процесс варки завершен и на подставку помещен пустой кофейник. Очевидно, что в нашей объектной модели эту ситуацию должен распознавать `ContainmentVessel`. Его задача – послать сообщение `Complete` объекту `UserInterface`. На рис. 20.8 изображена полная диаграмма кооперации.

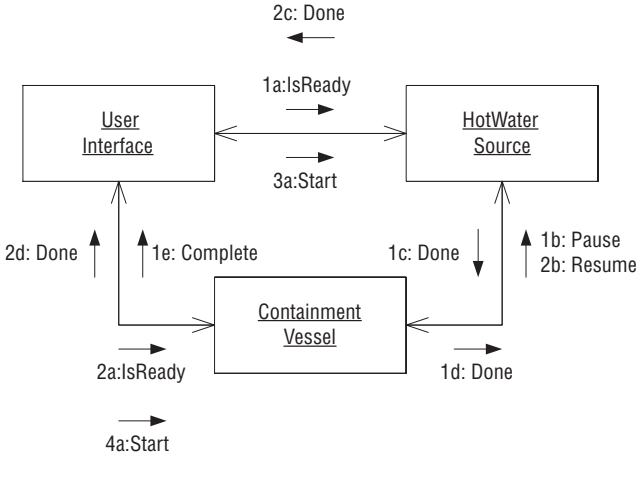


Рис. 20.8. *Кофе кончился*

Основываясь на этой диаграмме, мы можем нарисовать диаграмму классов с теми же самыми ассоциациями (рис. 20.9). Никаких сюрпризов она не таит.

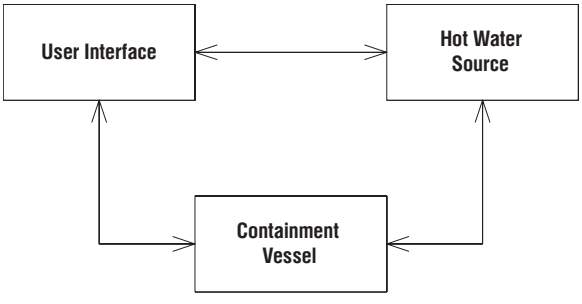


***Рис. 20.9.** Диаграмма классов*

Реализация абстрактной модели

Наша объектная модель довольно хорошо разбита на части. Имеются три четко очерченных зоны ответственности, и потоки входных и выходных сообщений между ними, похоже, сбалансированы. Нигде не видно следов объекта-бога. И классов-миражей тоже не наблюдается. Но как все-таки мы собираемся реализовать Mark IV при такой структуре? Просто напишем методы всех трех классов, так, чтобы они обращались к `CoffeeMakerAPI`? Да это же стыд и позор! Мы только что выявили самую суть процесса приготовления кофе. И дизайн, который привязывал бы эту суть к конкретной кофеварке Mark IV, стал бы огромным шагом назад.

На самом деле я собираюсь прямо сейчас объявить новое правило: ни один из только что созданных классов не должен знать *ничего* о кофеварке Mark IV. Это принцип инверсии зависимости (DIP). Мы не позволим высокоуровневой стратегии приготовления кофе зависеть от деталей низкоуровневой реализации.

Ладно, а как тогда реализовать Mark IV? Давайте снова рассмотрим все наши прецеденты. Но на этот раз – с точки зрения Mark IV.

Прецедент 1: пользователь нажимает кнопку «Сварить». Откуда объект `UserInterface` узнает, что кнопка «Сварить» нажата? Очевидно, он должен вызвать метод `CoffeeMakerAPI.GetBrewButtonStatus()`. А где он должен его вызвать? Мы уже определили, что сам класс `UserInterface` ничего не знает о `CoffeeMakerAPI`. Ну и где в таком случае поместить вызов?

Применим принцип DIP и разместим вызов в классе, производном от `UserInterface`. Детали показаны на рис. 20.10.

Мы произвели класс `M4UserInterface` от `UserInterface` и включили в `M4UserInterface` метод `CheckButton()`. А он уже вызовет метод `CoffeeMakerAPI.GetBrewButtonStatus()`. И если кнопка нажата, то метод `CheckButton()` вы-

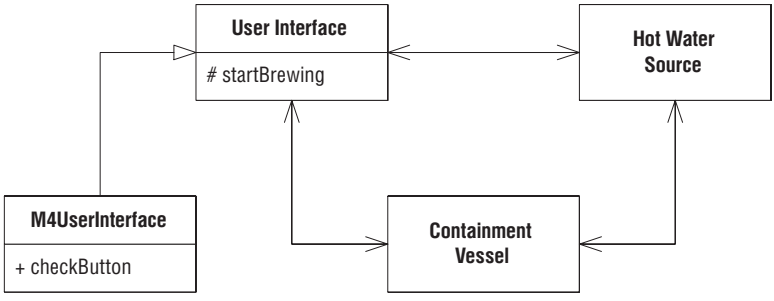


Рис. 20.10. *Обнаружение нажатия кнопки «Сварить»*

зовет защищенный метод `StartBrewing()` класса `UserInterface`. В листингах 20.5 и 20.6 показано, как можно написать такой код.

Листинг 20.5. M4UserInterface.cs

```
public class M4UserInterface : UserInterface
{
  private void CheckButton()
  {
    BrewButtonStatus status =
      CoffeeMaker.api.GetBrewButtonStatus();
    if (status == BrewButtonStatus.PUSHED)
    {
      StartBrewing();
    }
  }
}
```

Листинг 20.6. UserInterface.cs

```
public class UserInterface
{
  private HotWaterSource hws;
  private ContainmentVessel cv;

  public void Done() {}
  public void Complete() {}
  protected void StartBrewing()
  {
    if (hws.IsReady() && cv.IsReady())
    {
      hws.Start();
      cv.Start();
    }
  }
}
```

Возможно, вам интересно, зачем я вообще создал защищенный метод `StartBrewing()`. Почему просто не вызывать оба метода `Start()` из `M4User-`

Interface? Причина простая, но важная. Проверки IsReady() и следующие за ними вызовы методов Start() объектов HotWaterSource и ContainmentVessel – это высокоуровневая стратегия, место которой в классе UserInterface. Этот код остается в силе вне зависимости от того, реализуем мы Mark IV или какую-то другую кофеварку. Поэтому он не должен быть связан с производным классом, специфичным для Mark IV. Это очередной пример принципа единственной обязанности (SRP). Мы еще не раз встретимся с таким разнесением при рассмотрении данного примера. Я стараюсь помещать как можно больше кода в классы верхнего уровня. А в производных остается код, который прямо и неразрывно связан с кофеваркой Mark IV.

Реализация методов IsReady(). Как следует реализовать методы IsReady() в классах HotWaterSource и ContainmentVessel? Должно быть понятно, что в действительности эти методы абстрактны, а потому и сами классы абстрактные. Реализацию предоставляют производные классы M4HotWaterSource и M4ContainmentVessel, где вызываются соответствующие функции CoffeeMakerAPI. На рис. 20.11 изображена новая структура, а в листингах 20.7 и 20.8 показаны реализации производных классов.

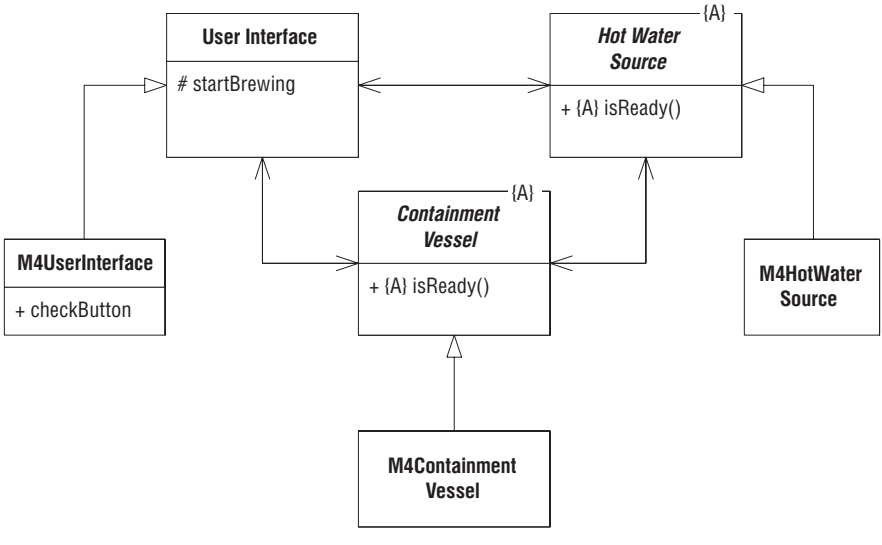


Рис. 20.11. *Реализация методов IsReady()*

Листинг 20.7. M4HotWaterSource.cs

```
public class M4HotWaterSource : HotWaterSource
{
  public override bool IsReady()
  {
    BoilerStatus status =
```

```
        CoffeeMaker.api.GetBoilerStatus();
      return status == BoilerStatus.NOT_EMPTY;
    }
  }
```

Листинг 20.8. M4ContainmentVessel.cs

```
public class M4ContainmentVessel : ContainmentVessel
{
  public override bool IsReady()
  {
    WarmerPlateStatus status =
      CoffeeMaker.api.GetWarmerPlateStatus();
    return status == WarmerPlateStatus.POT_EMPTY;
  }
}
```

Реализация методов Start(). Метод `Start()` в классе `HotWaterSource` — это абстрактный метод, реализованный в классе `M4HotWaterSource` так, что он вызывает функции `CoffeeMakerAPI`, которые закрывают клапан и включают кипятильник. Когда я писал эти функции, мне надоело каждый раз повторять конструкцию CoffeeMakerAPI.XXX, поэтому я попутно провел небольшой рефакторинг. Результат показан в листинге 20.9.

Листинг 20.9. M4HotWaterSource.cs

```
public class M4HotWaterSource : HotWaterSource
{
  private CoffeeMakerAPI api;
  public M4HotWaterSource(CoffeeMakerAPI api)
  {
    this.api = api;
  }

  public override bool IsReady()
  {
    BoilerStatus status = api.GetBoilerStatus();
    return status == BoilerStatus.NOT_EMPTY;
  }

  public override void Start()
  {
    api.SetReliefValveState(ReliefValveState.CLOSED);
    api.SetBoilerState(BoilerState.ON);
  }
}
```

Метод `Start()` в классе `ContainmentVessel` немного интереснее. Единственное, что должен сделать объект `M4ContainmentVessel`, — запомнить состояние процесса приготовления кофе. Позже мы увидим, что это позволит правильно реагировать на помещение и снятие кофейника с подставки. Код показан в листинге 20.10.

Листинг 20.10. M4ContainmentVessell.cs

```
public class M4ContainmentVessel : ContainmentVessel
{
  private CoffeeMakerAPI api;
  private bool isBrewing = false;

  public M4ContainmentVessel(CoffeeMakerAPI api)
  {
    this.api = api;
  }

  public override bool IsReady()
  {
    WarmerPlateStatus status = api.GetWarmerPlateStatus();
    return status == WarmerPlateStatus.POT_EMPTY;
  }

  public override void Start()
  {
    isBrewing = true;
  }
}
```

Вызов M4UserInterface.CheckButton. Как программа попадает в место, откуда можно вызвать метод `CoffeeMakerAPI.GetBrewButtonStatus()`? И вообще, как мы попадаем туда, где можно получить состояние какого-нибудь датчика?

Многие команды студентов затруднялись ответить на этот вопрос. Одни не хотели предполагать, что в кофеварке установлена многопоточная операционная система, поэтому предпочитали опрашивать датчики. Другие, наоборот, хотели ввести многопоточность и не беспокоиться об опросах. Я неоднократно наблюдал, как команды дебатировали по этому поводу целый час, а то и больше.

Ошибка – на которую я в конце концов указывал, дав им предварительно попотеть, – заключалась в том, что выбор между многопоточностью и опросом совершенно несущественный. Это решение можно принять в самый последний момент, безо всякого ущерба для дизайна. Поэтому всегда лучше предполагать, что сообщения можно посылать асинхронно, как будто имеются независимые потоки, а в последнюю минуту включить либо опрос, либо потоки.

До сих пор мы предполагали, что поток управления каким-то образом асинхронно попадает в объект `M4UserInterface`, который сможет вызвать метод `CoffeeMakerAPI.GetBrewButtonStatus()`. Ну а теперь предположим, что мы работаем на минимальной платформе, которая не поддерживает многопоточность. Значит, нам придется опрашивать датчики. Как это сделать?

Рассмотрим интерфейс `Pollable`, показанный в листинге 20.11. В нем нет ничего, кроме метода `Poll()`. Что если класс `M4UserInterface` реализу-

ет этот интерфейс? Что если метод `Main()` будет работать в цикле, снова и снова вызывая этот метод? Тогда поток управления будет раз за разом попадать в объект `M4UserInterface` и мы сможем обнаружить нажатие кнопки «Сварить».

Листинг 20.11. Pollable.cs

```
public interface Pollable
{
  void Poll();
}
```

Точно такую же конструкцию можно повторить во всех трех производных классах для Mark IV. В каждом из них есть датчик, который следует проверять. Поэтому, как показано на рис. 20.12, все классы `M4` можно произвести от `Pollable` и вызывать их из `Main()`.

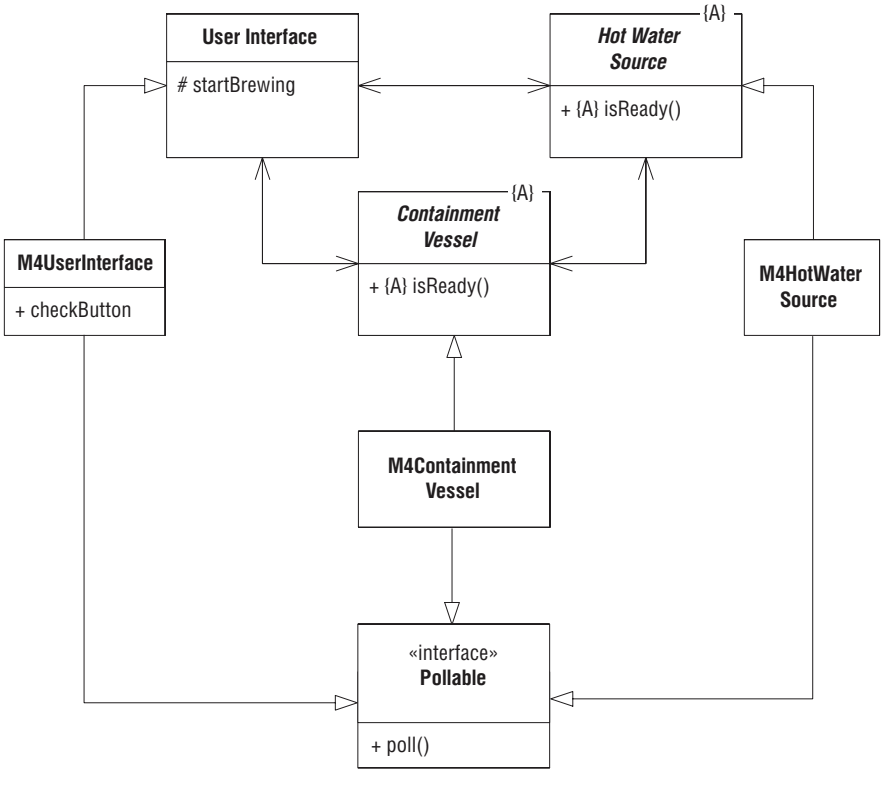


Рис. 20.12. *Кофеварка с опросом*

В листинге 20.12 показано, как может выглядеть метод `Main`. Он находится в классе `M4CoffeeMaker`. Этот метод создает объект `api`, а затем все три компонента `M4`. Он вызывает их методы `Init()`, чтобы связать компо-

ненты друг с другом, после чего входит в бесконечный цикл, поочередно вызывая метод `Poll()` каждого компонента.

Листинг 20.12. M4CoffeeMaker.cs

```
public static void Main(string[] args)
{
  CoffeeMakerAPI api = new M4CoffeeMakerAPI();

  M4UserInterface ui = new M4UserInterface(api);
  M4HotWaterSource hws = new M4HotWaterSource(api);
  M4ContainmentVessel cv = new M4ContainmentVessel(api);

  ui.Init(hws,cv);
  hws.Init(ui, cv);
  cv.Init(hws,ui);

  while (true)
  {
    ui.Poll();
    hws.Poll();
    cv.Poll();
  }
}
```

Вот теперь должно быть понятно, как вызывается метод `M4UserInterface.CheckButton()`. Более того, теперь ясно, что этот метод следовало назвать не `CheckButton()`, а `Poll()`. В листинге 20.13 показано, как сейчас выглядит класс `M4UserInterface`.

Листинг 20.13. M4UserInterface.cs

```
public class M4UserInterface : UserInterface
                             , Pollable
{
  private CoffeeMakerAPI api;

  public M4UserInterface(CoffeeMakerAPI api)
  {
    this.api = api;
  }

  public void Poll()
  {
    BrewButtonStatus status = api.GetBrewButtonStatus();
    if (status == BrewButtonStatus.PUSHED)
    {
      StartBrewing();
    }
  }
}
```

Конец задачи о кофеварке. Рассуждения, изложенные в предыдущих разделах, можно повторить для всех остальных компонентов кофеварки. Результаты показаны в листингах с 20.14 по 20.21.

Достоинства описанного дизайна

Хотя задача и тривиальна, у рассмотренного выше дизайна есть ряд весьма примечательных характеристик. На рис. 20.13 изображена структура в целом. Я обвел линией три абстрактных класса. В них инкапсулирована высокоуровневая стратегия кофеварки. Обратите внимание, что все зависимости, пересекающие линию, направлены внутрь. Ничто внутри линии не зависит от внешних по отношению к ней классов. Таким образом, абстракции полностью отделены от деталей реализации.

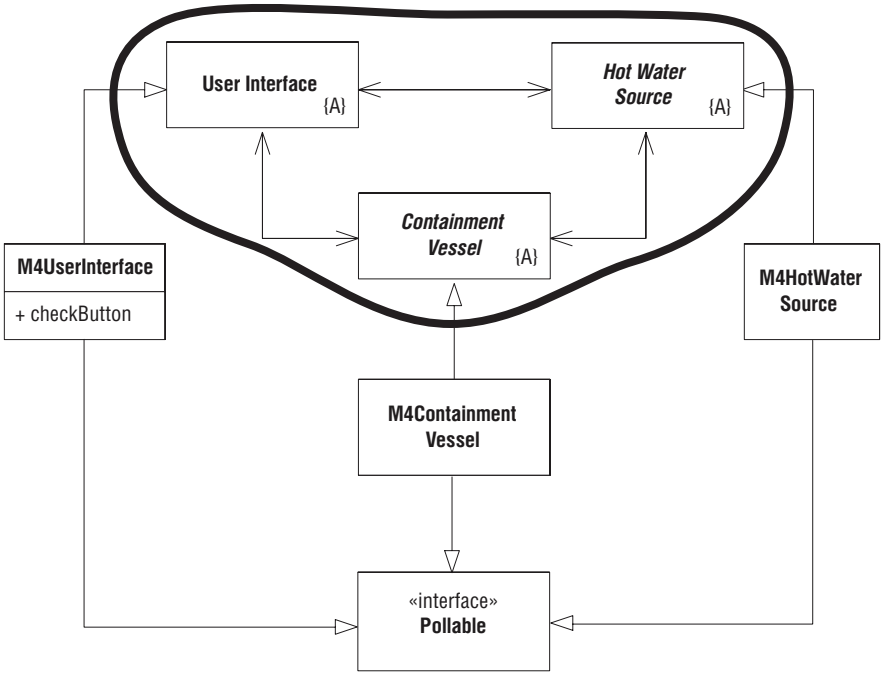


Рис. 20.13. *Компоненты кофеварки*

Абстрактные классы ничего не знают о кнопках, лампочках, клапанах, датчиках и прочих элементах конструкции кофеварки. А производные классы, напротив, очень даже осведомлены о таких деталях.

Отметим, что три абстрактных класса можно было бы использовать для самых разных кофеварок, например для подключенной к водопроводу и снабженной баком с краном. Вполне вероятно, что они применимы и к кофейному автомату. Думается мне, что они подойдут также для автоматического заваривания чая и даже приготовления куриного супа.

Такое отделение высокоуровневой стратегии от деталей и составляет смысл объектно-ориентированного проектирования.

Откуда взялся этот дизайн. Не могу сказать, что я как-то раз сел и сразу придумал этот дизайн. На самом деле первый вариант решения задачи о кофеварке, который я разработал в 1993 году, очень походил на тот, что показан на рис. 20.1. Однако я часто писал об этой задаче и раз за разом использовал ее в качестве упражнения на занятиях. Так что со временем дизайн совершенствовался.



При написании кода применялась методика разработки через тестирование, автономные тесты приведены в листинге 20.22. Я писал код, исходя из структуры на рис. 20.13, но делал это постепенно, добавляя на каждом шаге по одному успешно выполненному тесту.[1]

Не убежден, что комплект тестов полон. Если бы это была не учебная программа, то я проанализировал бы тесты более тщательно. Но для данной книги это, пожалуй, чересчур.

Объектно-ориентированный перебор

У рассмотренного примера есть определенные дидактические достоинства. При всей своей краткости и простоте он показывает, как можно использовать принципы ООП для управления зависимостями и разделения обязанностей.

С другой стороны, в данном случае преимущества такого разделения слишком малы, чтобы перевешивать затраты на его реализацию.

Если бы мы реализовали кофеварку Mark IV в виде конечного автомата, то обнаружили бы 7 состояний и 18 переходов.[2] Для их кодирования потребовалось бы 18 строк на языке программы SMC. Еще примерно десять строк занял бы простой главный цикл, в котором опрашиваются датчики, и еще десятка два – функции действий, вызываемые конечным автоматом. Короче говоря, весь код программы уместился бы на одной странице.

А код объектно-ориентированного решения даже без тестов занимает пять страниц. И оправдать такую диспропорцию нечем. В крупных приложениях выгоды, которые приносит управление зависимостями и разделение обязанностей, определенно перевешивают затраты на ООП. В данном же примере верно, скорее, обратное.

---

[1] [Beck2002]

[2] [Martin1995], стр. 65

Листинг 20.14. UserInterface.cs

```
using System;

namespace CoffeeMaker
{
  public abstract class UserInterface
  {
    private HotWaterSource hws;
    private ContainmentVessel cv;
    protected bool isComplete;

    public UserInterface()
    {
      isComplete = true;
    }

    public void Init(HotWaterSource hws, ContainmentVessel cv)
    {
      this.hws = hws;
      this.cv = cv;
    }

    public void Complete()
    {
      isComplete = true;
      CompleteCycle();
    }

    protected void StartBrewing()
    {
      if (hws.IsReady() && cv.IsReady())
      {
        isComplete = false;
        hws.Start();
        cv.Start();
      }
    }

    public abstract void Done();
    public abstract void CompleteCycle();
  }
}
```

Листинг 20.15. M4UserInterface.cs

```
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4UserInterface : UserInterface
                               , Pollable
```

```
  {
    private CoffeeMakerAPI api;

    public M4UserInterface(CoffeeMakerAPI api)
    {
      this.api = api;
    }

    public void Poll()
    {
      BrewButtonStatus buttonStatus = api.GetBrewButtonStatus();
      if (buttonStatus == BrewButtonStatus.PUSHED)
      {
        StartBrewing();
      }
    }

    public override void Done()
    {
      api.SetIndicatorState(IndicatorState.ON);
    }

    public override void CompleteCycle()
    {
      api.SetIndicatorState(IndicatorState.OFF);
    }
  }
}
```

Листинг 20.16. HotWaterSource.cs

```
namespace CoffeeMaker
{
  public abstract class HotWaterSource
  {
    private UserInterface ui;
    private ContainmentVessel cv;
    protected bool isBrewing;

    public HotWaterSource()
    {
      isBrewing = false;
    }

    public void Init(UserInterface ui, ContainmentVessel cv)
    {
      this.ui = ui;
      this.cv = cv;
    }

    public void Start()
    {
```

```
      isBrewing = true;
      StartBrewing();
    }

    public void Done()
    {
      isBrewing = false;
    }

    protected void DeclareDone()
    {
      ui.Done();
      cv.Done();
      isBrewing = false;
    }

    public abstract bool IsReady();
    public abstract void StartBrewing();
    public abstract void Pause();
    public abstract void Resume();
  }
}
```

Листинг 20.17. M4HotWaterSource.cs

```
using System;
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4HotWaterSource : HotWaterSource
                                , Pollable
  {
    private CoffeeMakerAPI api;

    public M4HotWaterSource(CoffeeMakerAPI api)
    {
      this.api = api;
    }

    public override bool IsReady()
    {
      BoilerStatus boilerStatus = api.GetBoilerStatus();
      return boilerStatus == BoilerStatus.NOT_EMPTY;
    }

    public override void StartBrewing()
    {
      api.SetReliefValveState(ReliefValveState.CLOSED);
      api.SetBoilerState(BoilerState.ON);
    }
```

```
    public void Poll()
    {
      BoilerStatus boilerStatus = api.GetBoilerStatus();
      if (isBrewing)
      {
        if (boilerStatus == BoilerStatus.EMPTY)
        {
          api.SetBoilerState(BoilerState.OFF);
          api.SetReliefValveState(ReliefValveState.CLOSED);
          DeclareDone();
        }
      }
    }

    public override void Pause()
    {
      api.SetBoilerState(BoilerState.OFF);
      api.SetReliefValveState(ReliefValveState.OPEN);
    }

    public override void Resume()
    {
      api.SetBoilerState(BoilerState.ON);
      api.SetReliefValveState(ReliefValveState.CLOSED);
    }
  }
}
```

Листинг 20.18. ContainmentVessel.cs

```
using System;

namespace CoffeeMaker
{
  public abstract class ContainmentVessel
  {
    private UserInterface ui;
    private HotWaterSource hws;
    protected bool isBrewing;
    protected bool isComplete;

    public ContainmentVessel()
    {
      isBrewing = false;
      isComplete = true;
    }

    public void Init(UserInterface ui, HotWaterSource hws)
    {
      this.ui = ui;
      this.hws = hws;
    }
```

```
    public void Start()
    {
      isBrewing = true;
      isComplete = false;
    }

    public void Done()
    {
      isBrewing = false;
    }

    protected void DeclareComplete()
    {
      isComplete = true;
      ui.Complete();
    }

    protected void ContainerAvailable()
    {
      hws.Resume();
    }

    protected void ContainerUnavailable()
    {
      hws.Pause();
    }

    public abstract bool IsReady();
  }
}
```

Листинг 20.19. M4ContainmentVessel.cs

```
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4ContainmentVessel : ContainmentVessel
                                   , Pollable
  {
    private CoffeeMakerAPI api;
    private WarmerPlateStatus lastPotStatus;

    public M4ContainmentVessel(CoffeeMakerAPI api)
    {
      this.api = api;
      lastPotStatus = WarmerPlateStatus.POT_EMPTY;
    }

    public override bool IsReady()
    {
      WarmerPlateStatus plateStatus =
```

```
  api.GetWarmerPlateStatus();
  return plateStatus == WarmerPlateStatus.POT_EMPTY;
}

public void Poll()
{
  WarmerPlateStatus potStatus = api.GetWarmerPlateStatus();
  if (potStatus != lastPotStatus)
  {
    if (isBrewing)
    {
      HandleBrewingEvent(potStatus);
    }
    else if (isComplete == false)
    {
      HandleIncompleteEvent(potStatus);
    }
    lastPotStatus = potStatus;
  }
}

private void
HandleBrewingEvent(WarmerPlateStatus potStatus)
{
  if (potStatus == WarmerPlateStatus.POT_NOT_EMPTY)
  {
    ContainerAvailable();
    api.SetWarmerState(WarmerState.ON);
  }
  else if (potStatus == WarmerPlateStatus.WARMER_EMPTY)
  {
    ContainerUnavailable();
    api.SetWarmerState(WarmerState.OFF);
  }
  else
  { // potStatus == POT_EMPTY
    ContainerAvailable();
    api.SetWarmerState(WarmerState.OFF);
  }
}

private void
HandleIncompleteEvent(WarmerPlateStatus potStatus)
{
  if (potStatus == WarmerPlateStatus.POT_NOT_EMPTY)
  {
    api.SetWarmerState(WarmerState.ON);
  }
  else if (potStatus == WarmerPlateStatus.WARMER_EMPTY)
  {
    api.SetWarmerState(WarmerState.OFF);
```

```
        }
        else
        { // potStatus == POT_EMPTY
          api.SetWarmerState(WarmerState.OFF);
          DeclareComplete();
        }
      }
    }
  }
```

Листинг 20.20. Pollable.cs

```
using System;

namespace M4CoffeeMaker
{
  public interface Pollable
  {
    void Poll();
  }
}
```

Листинг 20.21. CoffeeMaker.cs

```
using CoffeeMaker;

namespace M4CoffeeMaker
{
  public class M4CoffeeMaker
  {
    public static void Main(string[] args)
    {
      CoffeeMakerAPI api = new M4CoffeeMakerAPI();
      M4UserInterface ui = new M4UserInterface(api);
      M4HotWaterSource hws = new M4HotWaterSource(api);
      M4ContainmentVessel cv = new M4ContainmentVessel(api);
      ui.Init(hws, cv);
      hws.Init(ui, cv);
      cv.Init(ui, hws);
      while (true)
      {
        ui.Poll();
        hws.Poll();
        cv.Poll();
      }
    }
  }
}
```

Листинг 20.22. TestCoffeeMaker.cs

```
using M4CoffeeMaker;
using NUnit.Framework;
```

```
namespace CoffeeMaker.Test
{
  internal class CoffeeMakerStub : CoffeeMakerAPI
  {
    public bool buttonPressed;
    public bool lightOn;
    public bool boilerOn;
    public bool valveClosed;
    public bool plateOn;
    public bool boilerEmpty;
    public bool potPresent;
    public bool potNotEmpty;

    public CoffeeMakerStub()
    {
      buttonPressed = false;
      lightOn = false;
      boilerOn = false;
      valveClosed = true;
      plateOn = false;
      boilerEmpty = true;
      potPresent = true;
      potNotEmpty = false;
    }

    public WarmerPlateStatus GetWarmerPlateStatus()
    {
      if (!potPresent)
        return WarmerPlateStatus.WARMER_EMPTY;
      else if (potNotEmpty)
        return WarmerPlateStatus.POT_NOT_EMPTY;
      else
        return WarmerPlateStatus.POT_EMPTY;
    }

    public BoilerStatus GetBoilerStatus()
    {
      return boilerEmpty ?
             BoilerStatus.EMPTY : BoilerStatus.NOT_EMPTY;
    }

    public BrewButtonStatus GetBrewButtonStatus()
    {
      if (buttonPressed)
      {
        buttonPressed = false;
        return BrewButtonStatus.PUSHED;
      }
      else
      {
```

```
        return BrewButtonStatus.NOT_PUSHED;
      }
    }

    public void SetBoilerState(BoilerState boilerState)
    {
      boilerOn = boilerState == BoilerState.ON;
    }

    public void SetWarmerState(WarmerState warmerState)
    {
      plateOn = warmerState == WarmerState.ON;
    }

    public void
    SetIndicatorState(IndicatorState indicatorState)
    {
      lightOn = indicatorState == IndicatorState.ON;
    }

    public void
    SetReliefValveState(ReliefValveState reliefValveState)
    {
      valveClosed = reliefValveState == ReliefValveState.CLOSED;
    }
  }

  [TestFixture]
  public class TestCoffeeMaker
  {
    private M4UserInterface ui;
    private M4HotWaterSource hws;
    private M4ContainmentVessel cv;
    private CoffeeMakerStub api;

    [SetUp]
    public void SetUp()
    {
      api = new CoffeeMakerStub();
      ui = new M4UserInterface(api);
      hws = new M4HotWaterSource(api);
      cv = new M4ContainmentVessel(api);
      ui.Init(hws, cv);
      hws.Init(ui, cv);
      cv.Init(ui, hws);
    }

    private void Poll()
    {
      ui.Poll();
```

```
  hws.Poll();
  cv.Poll();
}

[Test]
public void InitialConditions()
{
  Poll();
  Assert.IsFalse(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

[Test]
public void StartNoPot()
{
  Poll();
  api.buttonPressed = true;
  api.potPresent = false;
  Poll();
  Assert.IsFalse(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

[Test]
public void StartNoWater()
{
  Poll();
  api.buttonPressed = true;
  api.boilerEmpty = true;
  Poll();
  Assert.IsFalse(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

[Test]
public void GoodStart()
{
  NormalStart();
  Assert.IsTrue(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}
```

```
private void NormalStart()
{
  Poll();
  api.boilerEmpty = false;
  api.buttonPressed = true;
  Poll();
}

[Test]
public void StartedPotNotEmpty()
{
  NormalStart();
  api.potNotEmpty = true;
  Poll();
  Assert.IsTrue(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsTrue(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

[Test]
public void PotRemovedAndReplacedWhileEmpty()
{
  NormalStart();
  api.potPresent = false;
  Poll();
  Assert.IsFalse(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsFalse(api.valveClosed);
  api.potPresent = true;
  Poll();
  Assert.IsTrue(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

[Test]
public void PotRemovedWhileNotEmptyAndReplacedEmpty()
{
  NormalFill();
  api.potPresent = false;
  Poll();
  Assert.IsFalse(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsFalse(api.valveClosed);
  api.potPresent = true;
  api.potNotEmpty = false;
```

```
  Poll();
  Assert.IsTrue(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsFalse(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

private void NormalFill()
{
  NormalStart();
  api.potNotEmpty = true;
  Poll();
}

[Test]
public void PotRemovedWhileNotEmptyAndReplacedNotEmpty()
{
  NormalFill();
  api.potPresent = false;
  Poll();
  api.potPresent = true;
  Poll();
  Assert.IsTrue(api.boilerOn);
  Assert.IsFalse(api.lightOn);
  Assert.IsTrue(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

[Test]
public void BoilerEmptyPotNotEmpty()
{
  NormalBrew();
  Assert.IsFalse(api.boilerOn);
  Assert.IsTrue(api.lightOn);
  Assert.IsTrue(api.plateOn);
  Assert.IsTrue(api.valveClosed);
}

private void NormalBrew()
{
  NormalFill();
  api.boilerEmpty = true;
  Poll();
}

[Test]
public void BoilerEmptiesWhilePotRemoved()
{
  NormalFill();
  api.potPresent = false;
```

```
        Poll();
        api.boilerEmpty = true;
        Poll();
        Assert.IsFalse(api.boilerOn);
        Assert.IsTrue(api.lightOn);
        Assert.IsFalse(api.plateOn);
        Assert.IsTrue(api.valveClosed);
        api.potPresent = true;
        Poll();
        Assert.IsFalse(api.boilerOn);
        Assert.IsTrue(api.lightOn);
        Assert.IsTrue(api.plateOn);
        Assert.IsTrue(api.valveClosed);
      }

      [Test]
      public void EmptyPotReturnedAfter()
      {
        NormalBrew ();
        api.potNotEmpty = false;
        Poll ();
        Assert.IsFalse(api.boilerOn);
        Assert.IsFalse(api.lightOn);
        Assert.IsFalse(api.plateOn);
        Assert.IsTrue(api.valveClosed);
      }
    }
  }
```

Библиография

[Beck2002] Kent Beck «Test-Driven Development», Addison-Wesley, 2002.[1]

[Martin1995] Robert C. Martin «Designing Object-Oriented C++ Applications Using the Booch Method», Prentice Hall, 1995.

[1] Кент Бек «Экстремальное программирование: разработка через тестирование». – Пер. с англ. – Питер, 2003.

III

Задача о расчете заработной платы



Вот и пришло время заняться нашим первым большим примером. Мы изучили методики и принципы. Обсудили, в чем состоит смысл проектирования. Поговорили о тестировании и планировании. Пора сделать что-то реальное.

В последующих нескольких главах мы будем заниматься проектированием и реализацией пакетной системы расчета заработной платы, краткая спецификация которой будет приведена ниже. При решении этой задачи мы будем пользоваться несколькими паттернами проектирования: Команда (Command), Шаблонный метод (Template Method), Стра-

тегия (Strategy), Одиночка (Singleton), Null-объект (Null Object), Фабрика (Factory) и Фасад (Facade). Эти паттерны составляют содержание нескольких следующих глав. В главе 26 мы приступим непосредственно к проектированию и реализации задачи о расчете заработной платы.

Читать этот раздел можно несколькими способами.

- Подряд, то есть сначала изучить паттерны проектирования, а потом посмотреть, как они применяются к задаче о расчете заработной платы.
- Если вы уже знакомы с паттернами и не хотите еще раз читать о них, переходите прямо к главе 26.
- Сначала прочитать главу 26, а потом вернуться к главам об использованных в ней паттернах.
- Читать главу 26 по частям. Когда речь зайдет о незнакомом паттерне, прочитать главу, в которой он описывается, а потом вернуться к главе 26.

Но вообще-то никаких жестких правил нет. Выбирайте одну из предложенных стратегий или придумайте свою собственную – лишь бы вам было удобно.

Краткая спецификация системы расчета заработной платы

Ниже приведены некоторые заметки, сделанные во время беседы с заказчиком. (Они повторены также в главе 26.)

Система состоит из базы данных о работниках компании, в которой хранятся, в частности, карточки табельного учета. Система должна в оговоренное время начислить всем работникам зарплату в соответствии с условиями найма и выплатить ее тем способом, который указал работник. Кроме того, из зарплаты должны быть произведены различные вычеты.

- Часть работников работает на условиях почасовой оплаты. Почасовая ставка хранится в одном из полей записи о работнике. Ежедневно такой работник заполняет карточку табельного учета, проставляя дату и количество отработанных часов. Если работник в какой-то день отработал более 8 часов, то дополнительные часы оплачиваются с коэффициентом 1,5. Выплаты производятся каждую пятницу.
- Части работников начисляется твердый оклад. Им зарплата выплачивается в последний рабочий день месяца. Величина месячного оклада хранится в одном из полей записи о работнике.
- Части работников на окладе выплачиваются также комиссионные, рассчитываемые из объема произведенных ими продаж. Они представляют справки, в которых указаны дата и сумма продажи. Ко-

миссионная ставка хранится в одном из полей записи о работнике. Выплаты производятся каждую вторую пятницу.

- Работник может сам выбрать способ платежа. Чек может быть отправлен на указанный работником почтовый адрес, храниться у кассира до востребования, или же сумма может быть переведена на указанный банковский счет.
- Некоторые работники являются членами профсоюза. Для них в записи о работнике хранится ставка еженедельных членских взносов. Величина членских взносов должна быть вычтена из зарплаты. Кроме того, профсоюз может иногда выставлять своим членам счет за оказанные дополнительные услуги. Такие счета подаются еженедельно, и предъявленная к оплате сумма должна вычитаться из очередной зарплаты работника.
- Программа расчета заработной платы запускается каждый рабочий день и начисляет зарплату тем работникам, с которыми надлежит рассчитаться в этот день. Системе сообщается, по какую дату должен быть произведен расчет с работниками, поэтому она рассчитывает платежи по документам, поступившим с даты последнего расчета по указанную дату.

Упражнение

Прежде чем читать дальше, вы можете попробовать спроектировать систему расчета заработной платы самостоятельно. Быть может, вы захотите набросать начальные UML-диаграммы. А еще лучше – написать несколько тестов. Применяйте изученные ранее принципы и методики и старайтесь создать сбалансированный и жизнеспособный дизайн. Помните о кофеварке!

Если вы решитесь на подобное предприятие, то ознакомьтесь с приведенными ниже прецедентами. В противном случае можете их пропустить, они будут повторены в главе 26.

Прецедент 1: добавление нового работника

Новый работник добавляется при получении входной записи `AddEmp`, которая содержит имя и адрес работника, а также присвоенный ему табельный номер. Запись может быть представлена в одном из трех форматов:

1. `AddEmp <EmpID> “<name>” “<address>” H <hrly-rate>`
2. `AddEmp <EmpID> “<name>” “<address>” S <mtly-slry>`
3. `AddEmp <EmpID> “<name>” “<address>” C <mtly-slry> <comm-rate>`

В результате создается запись о работнике, в которой заполнены те или иные поля.

Альтернатива: ошибка в структуре входной записи. Если входная запись имеет неправильную структуру, то печатается сообщение об ошибке и больше никаких действий не выполняется.

Прецедент 2: удаление работника

Работник удаляется при получении входной записи `DelEmp`, имеющей следующий формат:

```
DelEmp <EmpID>
```

В результате удаляется запись о соответствующем работнике.

Альтернатива: недопустимое или неизвестное значение `EmpID`. Если формат поля `<EmpID>` неправилен или не существует записи о работнике с таким табельным номером, то печатается сообщение об ошибке и больше никаких действий не выполняется.

Прецедент 3: регистрация карточки табельного учета

При получении входной записи `TimeCard` система создает карточку табельного учета и ассоциирует ее с записью о соответствующем работнике:

```
TimeCard <empid> <date> <hours>
```

Альтернатива 1: указанному работнику не начисляется почасовая оплата. Система печатает сообщение об ошибке и больше никаких действий не выполняет.

Альтернатива 2: ошибка в структуре входной записи. Система печатает сообщение об ошибке и больше никаких действий не выполняет.

Прецедент 4: регистрация справки о продажах

При получении входной записи `SalesReceipt` система создает новую запись о справке о продажах и ассоциирует ее с записью о соответствующем работнике:

```
SalesReceipt <EmpID> <date> <amount>
```

Альтернатива 1: указанному работнику не начисляются комиссионные. Система печатает сообщение об ошибке и больше никаких действий не выполняет.

Альтернатива 2: ошибка в структуре входной записи. Система печатает сообщение об ошибке и больше никаких действий не выполняет.

Прецедент 5: регистрация платежного требования от профсоюза

При получении такой входной записи система создает запись о платежном требовании и ассоциирует ее с записью о соответствующем члене профсоюза:

```
ServiceCharge <memberID> <amount>
```

Альтернатива: ошибка в структуре входной записи. Если структура записи некорректна или номер <memberID> не принадлежит ни одному члену профсоюза, то печатается сообщение об ошибке.

Прецедент 6: изменение сведений о работнике

При получении такой входной записи система изменяет данные в записи о соответствующем работнике. Запись может быть представлена в одном из следующих форматов:

`ChgEmp <EmpID> Name <name>`	Изменить имя работника
`ChgEmp <EmpID> Address <address>`	Изменить адрес работника
`ChgEmp <EmpID> Hourly <hourlyRate>`	Перевести на почасовую оплату
`ChgEmp <EmpID> Salaried <salary>`	Перевести на оклад
`ChgEmp <EmpID> Commissioned <salary> <rate>`	Перевести на комиссионную оплату
`ChgEmp <EmpID> Hold`	Оставлять чек у кассира
`ChgEmp <EmpID> Direct <bank> <account>`	Перевод на банковский счет
`ChgEmp <EmpID> Mail <address>`	Отправлять чек почтой
`ChgEmp <EmpID> Member <memberID> Dues <rate>`	Сделать членом профсоюза
`ChgEmp <EmpID> NoMember`	Исключить из членов профсоюза

Альтернатива: ошибки во входной записи. Если структура записи некорректна, или работника с табельным номером <EmpID> не существует, или номер <memberID> не принадлежит ни одному члену профсоюза, то печатается сообщение об ошибке и больше никаких действий не выполняется.

Прецедент 7: расчет заработной платы на сегодня

При получении такой входной записи система находит всех работников, которым следует начислить зарплату на указанную дату. Затем система рассчитывает для них величину зарплаты и производит выплату в соответствии с указанным методом платежа. Распечатывается контрольный протокол, в котором отражаются действия, произведенные для каждого работника:

```
Payday <date>
```

21

Команда и Активный объект: многогранность и многозадачность



Никто от природы не наделен правом повелевать другими людьми.

Дени Дидро (1713–1784)

Из всех паттернов проектирования, о которых рассказывалось на протяжении последних лет, паттерн Команда кажется мне одним из самых простых и элегантных. Но, как мы вскоре увидим, его простота обманчива. Область применения этого паттерна практически безгранична.

Паттерн Команда прост до смешного (рис. 21.1). И листинг 21.1 не умаляет его несерьезности. Трудно поверить в полезность паттерна, который не содержит ничего, кроме интерфейса с единственным методом.

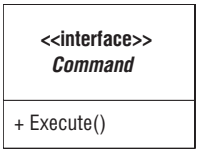


Рис. 21.1. *Паттерн Команда*

Листинг 21.1. Command.cs

```
public interface Command
{
  void Execute();
}
```

Но на самом деле этот паттерн пересекает одну весьма любопытную границу. И именно этим обусловлена его интересность и сложность. В большинстве классов имеется набор методов, ассоциированный с соответствующими переменными-членами. Паттерн Команда ничего такого не подразумевает. Он инкапсулирует единственную функцию, свободную от каких бы то ни было переменных.

С точки зрения строгой объектной ориентированности его следовало бы предать анафеме, так как от него за версту разит функциональной декомпозицией. Он возвышает функцию до уровня класса. Богохульство! И тем не менее на границе, где эти парадигмы сталкиваются, происходят очень интересные вещи.

Простые команды

Несколько лет назад я консультировал крупную компанию, производящую фотокопировальные устройства. Я помогал командам разработчиков проектировать и реализовывать встроенные системы реального времени, управляющие работой устройства. Нам пришла в голову идея воспользоваться паттерном Команда для управления аппаратурой. Была создана иерархия, подобная той, что изображена на рис. 21.2.

Роль этих классов не вызывает вопросов. Метод `Execute()` в классе `RelayOnCommand` включает реле, а в классе `MotorOffCommand` – выключает электродвигатель. Адреса электродвигателя или реле передаются объекту в качестве аргумента конструктора.

При такой структуре объекты `Command` можно передавать между разными компонентами системы, которые будут вызывать метод `Execute()`, ничего не зная о том, какую именно команду они представляют. Это привело к некоторым интересным упрощениям.

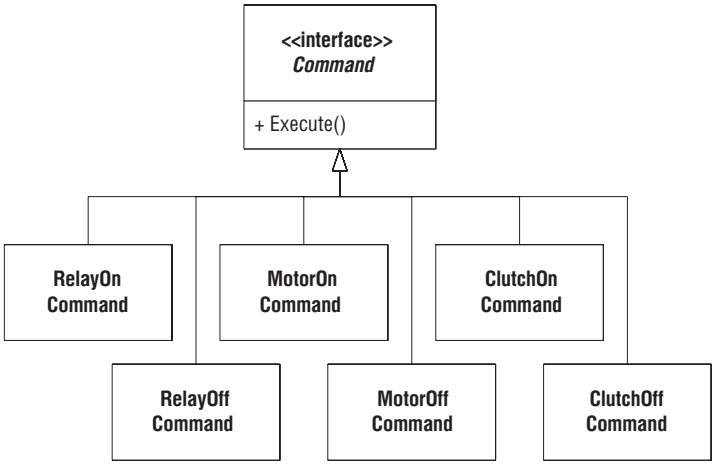


Рис. 21.2. *Несколько простых команд для программы управления копировальным устройством*

Система управлялась событиями. Реле замыкаются и размыкаются, двигатели запускаются и останавливаются, муфты включаются и выключаются в зависимости от происходящих в системе событий. Многие такие события обнаруживаются датчиками. Например, когда оптический датчик обнаруживает, что лист бумаги дошел до определенного места на тракте, необходимо включить определенную муфту. Нам удалось реализовать это, просто связав подходящий объект `ClutchOnCommand` с объектом, управляющим данным оптическим датчиком. См. рис. 21.3.



Рис. 21.3. *Команда, управляемая датчиком*

У этой простой структуры есть одно огромное преимущество. Класс `Sensor` понятия не имеет о том, что делает. Обнаружив событие, он просто вызывает метод `Execute()` связанного с ним объекта `Command`. Это означает, что датчики ничего не знают ни о муфтах, ни о реле. Равно как и о механическом устройстве тракта прохождения бумаги. Их функционирование становится восхитительно простым.

Вся сложность определения того, какие реле замыкать при поступлении событий от определенных датчиков, ложится на функцию инициализации. В какой-то момент на этапе инициализации системы каждый объект `Sensor` связывается с соответствующим ему объектом `Command`. В результате все логические связи между датчиками и командами –

монтажная схема – оказываются в одном месте и выносятся из основного кода системы. Можно было бы даже создать простой текстовый файл, в котором описано, какие датчики с какими командами связаны. Программа инициализации прочитала бы этот файл и соответственно сконфигурировала систему. Таким образом, монтажная схема системы оказалась бы целиком вне самой программы и ее можно было бы изменять без повторной компиляции.

Инкапсулировав понятие команды, этот паттерн позволил нам отделить логические связи внутри системы от связываемых устройств. Это нам очень помогло.

А куда делось I?

В сообществе .NET принято ставить в начале имени интерфейса прописную букву I. Так, в примере выше интерфейс `Command` следовало бы назвать `ICommand`. Но хотя многие соглашения .NET хороши и мы стараемся следовать им в этой книге, именно это, по скромному мнению авторов, не очень удачно.

Вообще говоря, не стоит включать в имя некую ортогональную ему концепцию, особенно если эта концепция может измениться. Например, что если мы захотим сделать `ICommand` не интерфейсом, а абстрактным классом? Будем отыскивать все упоминания `ICommand` и заменять их на `Command`? Будем заново компилировать и развертывать все затронутые этим изменением сборки?

На дворе двадцать первый век. У нас есть умные среды IDE, в которых достаточно задержать указатель мыши над именем, чтобы узнать, является ли оно именем интерфейса. Настало время предать забвению эти последние рудименты венгерской нотации.

Транзакции

У паттерна Команда есть еще одно расхожее применение, которое окажется полезным в задаче о расчете заработной платы: создание и выполнение транзакций. Представим, к примеру, что мы пишем программу для поддержки базы данных о работниках (см. рис. 21.4). Пользователь может выполнять в этой базе ряд операций, в том числе добавление новых и удаление старых работников или изменение атрибутов работников.

Пользователь, желающий добавить нового работника, должен задать всю информацию, необходимую для создания записи о нем в базе данных. Но прежде чем приступать к выполнению операции, система должна проверить синтаксическую и семантическую правильность данных. Паттерн Команда может в этом помочь. Объект-команда играет роль

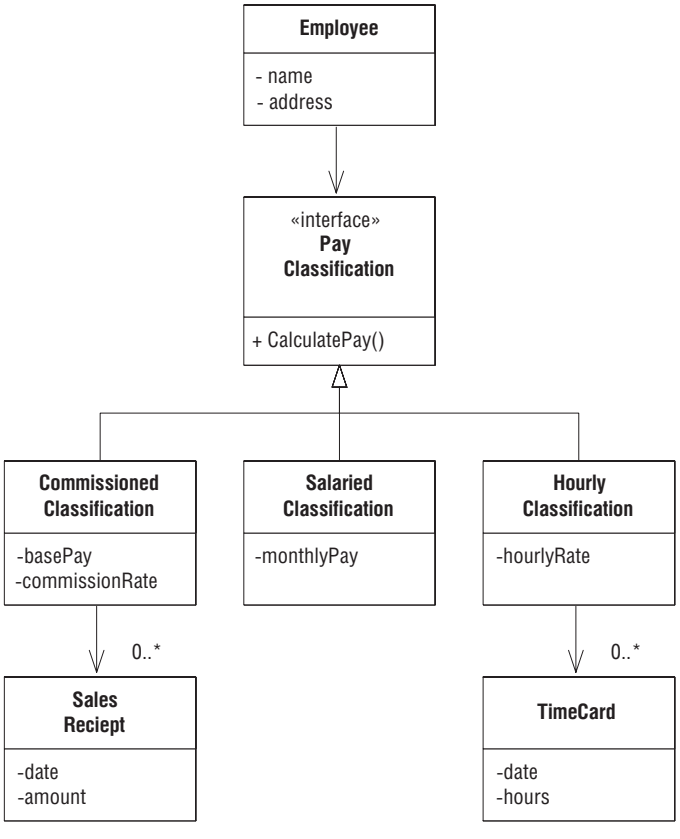


***Рис. 21.4.** База данных о работниках*

контейнера непроверенных данных, реализует методы контроля и методы выполнения транзакции.

Взгляните на рис. 21.5. Объект `AddEmployeeTransaction` содержит те же поля, что и объект `Employee`, и дополнительно указатель на объект `PayClassification`. И поля, и этот объект создаются на основе данных, которые ввел пользователь, давший системе указание добавить нового работника.

Метод `Validate` исследует все данные и убеждается, что они осмысленны. Он проверяет синтаксическую и семантическую корректность. Он может даже проверить, что данные, участвующие в транзакции, не противоречат текущему состоянию базы данных, например удостовериться в том, что работника с указанным табельным номером не существует.

Метод `Execute` использует проверенные данные для обновления базы. В нашем простом примере будет создан объект `Employee`, после чего его поля будут инициализированы значениями, взятыми из объекта `AddEm-`

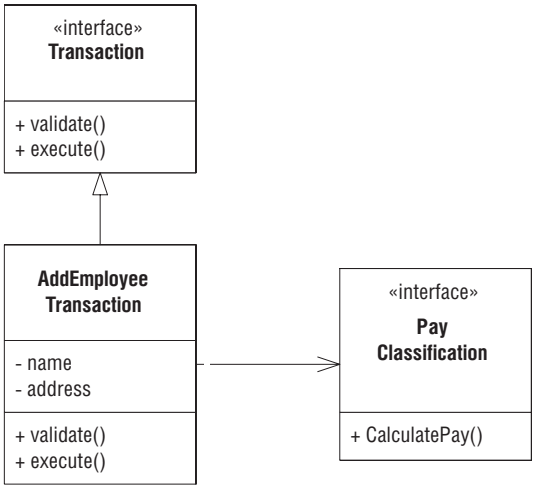


Рис. 21.5. *Транзакция AddEmployee*

ployeeTransaction. В объект Employee будет также скопирован объект PayClassification (целиком или по ссылке).

Разрыв физических и темпоральных связей

Что это нам дает? Полный разрыв связей между частью программы, которая получает данные от пользователя, частью, которая проверяет и обрабатывает их, и самими бизнес-объектами. Например, можно предположить, что данные для добавления нового пользователя вводятся в диалоговом окне программы с графическим интерфейсом. Было бы совершенно неправильно включать в ГИП код для проверки данных и их последующей обработки. Наличие такой связанности не позволило бы использовать код проверки и обработки в других интерфейсах. Поместив этот код в класс AddEmployeeTransaction, мы физически отделили его от интерфейса получения данных. И, что еще важнее, отделили код, знающий о том, как манипулировать базой данных, от самих бизнес-объектов.

Разрыв темпоральных связей

Мы отделили код проверки и обработки еще и в другом отношении. Получив тем или иным способом данные, мы вовсе не обязаны сразу же вызывать для них методы проверки и обработки. Объекты транзакций можно пока сохранить в списке, а проверить и обработать гораздо позже.

Предположим, что в течение дня база данных не должна изменяться. Все изменения следует вносить только между полуночью и часом ночи. Нелепо было бы ждать до полуночи, а потом быстро-быстро набирать все команды, стараясь уложиться до часу. Гораздо удобнее ввести ко-

манды в рабочее время, сразу же их проверить, а выполнение отложить до полуночи. Паттерн Команда дает нам такую возможность.

Метод Undo

На рис. 21.6 в паттерн Команда добавлен метод `Undo()`. Разумно предположить, что если реализация метода `Execute()` в классе, производном от `Command`, позволяет запомнить детали выполняемой операции, то реализация метода `Undo()` дает возможность откатить эту операцию и привести систему в исходное состояние.

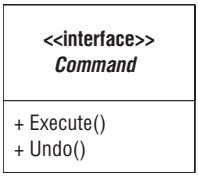
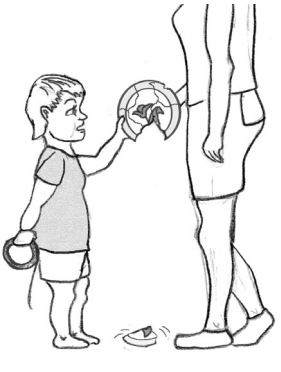


Рис. 21.6. *Вариант паттерна Команда с методом Undo*

Представим, к примеру, приложение, которое позволяет пользователю рисовать на экране геометрические фигуры. На панели инструментов есть кнопки для рисования кругов, квадратов, прямоугольников и т. д. Предположим, что пользователь нажал кнопку **Нарисовать круг**. Система создает объект `DrawCircleCommand` и вызывает его метод `Execute()`. Объект `DrawCircleCommand` отслеживает состояние мыши, ожидая, когда пользователь щелкнет в окне рисования. В момент щелчка он делает точку, где находится указатель мыши, центром круга и начинает рисовать анимированный круг, следя за положением указателя. Кроме того, он сохраняет идентификатор нового круга в своей закрытой переменной. В конечном итоге метод `Execute()` возвращает управление и система помещается отработавший объект `DrawCirlceCommand` в стек выполненных команд.



Позже пользователь нажимает кнопку **Отменить** на панели инструментов. Система извлекает из стека объекта `Command` и вызывает его метод `Undo()`. Получив сообщение `Undo()`, объект `DrawCircleCommand` удаляет круг с тем идентификатором, который в нем хранится, из списка объектов, нарисованных на холсте.

Подобная техника позволяет без труда реализовать команду отмены практически в любом приложении. Код, знающий, как отменить команду, почти всегда находится по соседству с кодом, знающим, как ее выполнить.

Активный объект

Одно из моих излюбленных применений паттерна Команда – паттерн Активный объект[1]. Это довольно старая техника реализации нескольких потоков управления использовалась в том или ином виде в тысячах промышленных систем для организации простого многопоточного ядра.

Идея очень проста. Взгляните на листинги 21.2 и 21.3. Объект `ActiveObjectEngine` хранит связанный список объектов `Command`. Пользователь может добавить в него новые команды или вызвать метод `Run()`. Этот метод просто проходит по списку, выполняя и затем удаляя каждую встретившуюся команду.

Листинг 21.2. ActiveObjectEngine.cs

```
using System.Collections;

public class ActiveObjectEngine
{
  ArrayList itsCommands = new ArrayList();

  public void AddCommand(Command c)
  {
    itsCommands.Add(c);
  }

  public void Run()
  {
    while (itsCommands.Count > 0)
    {
      Command c = (Command) itsCommands[0];
      itsCommands.RemoveAt(0);
      c.Execute();
    }
  }
}
```

Листинг 21.3. Command.cs

```
public interface Command
{
  void Execute();
}
```

На первый взгляд не слишком впечатляет. Но подумайте, что произойдет, если какой-то из находившихся в списке объектов `Command` поместит себя обратно в список. Тогда список никогда не опустеет и метод `Run()` будет работать вечно.

[1] [Lavender96]

Рассмотрим тест в листинге 21.4. Он создает объект SleepCommand, передавая его конструктору среди прочего и величину задержки 1000 мс. Затем объект SleepCommand помещается в ActiveObjectEngine. Тест ожидает, что после вызова Run() должно пройти не менее 1000 мс.

Листинг 21.4. TestSleepCommand.cs

```
using System;
using NUnit.Framework;

[TestFixture]
public class TestSleepCommand
{
  private class WakeUpCommand : Command
  {
    public bool executed = false;
    public void Execute()
    {
      executed = true;
    }
  }

  [Test]
  public void TestSleep()
  {
    WakeUpCommand wakeup = new WakeUpCommand();
    ActiveObjectEngine e = new ActiveObjectEngine();
    SleepCommand c = new SleepCommand(1000, e, wakeup);
    e.AddCommand(c);
    DateTime start = DateTime.Now;
    e.Run();
    DateTime stop = DateTime.Now;
    double sleepTime = (stop-start).TotalMilliseconds;
    Assert.IsTrue(sleepTime >= 1000,
       "SleepTime " + sleepTime + " expected > 1000");
    Assert.IsTrue(sleepTime <= 1100,
       "SleepTime " + sleepTime + " expected < 1100");
    Assert.IsTrue(wakeup.executed, "Command Executed");
  }
}
```

Взглянем на этот тест внимательнее. У конструктора SleepCommand есть три аргумента. Первый – время задержки в миллисекундах. Второй – объект ActiveObjectEngine, внутри которого будет работать команда. А третий, wakeup, – еще одна команда, которую следует вызвать при возобновлении работы, то есть по прошествии указанного числа миллисекунд.

В листинге 21.5 показана реализация SleepCommand. При выполнении объект проверяет, исполнялся ли он раньше, и если нет, то запоминает время начала работы. Если задержка еще не истекла, то объект поме-

щает себя обратно в ActiveObjectEngine. В противном случае в ActiveObjectEngine помещается команда wakeup.

Листинг 21.5. SleepCommand.cs

```
using System;

public class SleepCommand : Command
{
  private Command wakeupCommand = null;
  private ActiveObjectEngine engine = null;
  private long sleepTime = 0;
  private DateTime startTime;
  private bool started = false;

  public SleepCommand(long milliseconds, ActiveObjectEngine e,
                      Command wakeupCommand)
  {
    sleepTime = milliseconds;
    engine = e;
    this.wakeupCommand = wakeupCommand;
  }

  public void Execute()
  {
    DateTime currentTime = DateTime.Now;
    if (!started)
    {
      started = true;
      startTime = currentTime;
      engine.AddCommand(this);
    }
    else
    {
      TimeSpan elapsedTime = currentTime - startTime;
      if (elapsedTime.TotalMilliseconds < sleepTime)
      {
        engine.AddCommand(this);
      }
      else
      {
        engine.AddCommand(wakeupCommand);
      }
    }
  }
}
```

Мы можем провести аналогию между этой программой и многопоточной программой, ожидающей события. Когда поток в многопоточной программе ждет события, он обычно делает вызов операционной системы, который блокирует поток, пока событие не произойдет. Программа

в листинге 21.5 не блокируется. Но если ожидаемое событие не произошло в течение времени `elapsedTime.TotalMilliseconds`, то поток просто помещает себя назад в объект `ActiveObjectEngine`.

Построение многопоточных систем с использованием этой техники было и остается весьма распространенной практикой. Такие потоки называются *исполняемыми до завершения* (Run-to-Completion – RTC); каждый экземпляр `Command` отрабатывает до конца, прежде чем запускается следующий экземпляр. Аббревиатура RTC подразумевает, что экземпляры `Command` не блокируют программу.

Факт отработки экземпляров `Command` до конца дает RTC-потокам то преимущество, что все они пользуются одним и тем же машинным стеком. В отличие от традиционных многопоточных систем, нет необходимости выделять каждому потоку отдельный стек. В системах с ограниченной памятью и большим количеством потоков это может оказаться важным достоинством.

В продолжение этого примера в листинге 21.6 приведена простая программа, которая демонстрирует многопоточное поведение, применяя объект `SleepCommand`. Эта программа называется `DelayedTyper`.

Листинг 21.6. DelayedTyper.cs

```
using System;

public class DelayedTyper : Command
{
  private long itsDelay;
  private char itsChar;
  private static bool stop = false;
  private static ActiveObjectEngine engine =
    new ActiveObjectEngine();

  private class StopCommand : Command
  {
    public void Execute()
    {
      DelayedTyper.stop = true;
    }
  }

  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 StopCommand();
    engine.AddCommand(
      new SleepCommand(20000, engine, stopCommand));
      engine.Run();
```

```
  }

  public DelayedTyper(long delay, char c)
  {
    itsDelay = delay;
    itsChar = c;
  }

  public void Execute()
  {
    Console.Write(itsChar);
    if (!stop)
      DelayAndRepeat();
  }

  private void DelayAndRepeat()
  {
    engine.AddCommand(
    new SleepCommand(itsDelay, engine, this));
  }
}
```

Отметим, что `DelayedTyper` реализует интерфейс `Command`. Метод `Execute` просто печатает символ, переданный конструктору, проверяет флаг `stop` и, если тот не установлен, вызывает метод `DelayAndRepeat`. Метод `DelayAndRepeat` конструирует объект `SleepCommand` с задержкой, величина которой передана конструктору, и вставляет этот объект в `ActiveObjectEngine`.

Поведение этого объекта `Command` легко предсказать. Он работает в цикле, печатая заданный символ и ожидая истечения задержки. Выход из цикла происходит, когда установлен флаг `stop`.

Метод `Main` создает несколько экземпляров `DelayedTyper`, каждый со своим символом и задержкой, помещает их в `ActiveObjectEngine`, а затем добавляет туда же команду `SleepCommand`, которая по истечении некоторого времени установит флаг `stop`. Если запустить эту программу, то будет напечатана строка, содержащая символы 1, 3, 5 и 7. При повторном запуске будет напечатана другая строка, состоящая из тех же символов. Вот два типичных прогона:

```
13571131151137111315113171513111315173111135111371153111135 7...
13571113151317113151131117135111311517311131511317113511131 17...
```

Строки различаются, потому что таймер процессора и таймер реального времени синхронизированы не идеально. Такое недетерминированное поведение является отличительным признаком многопоточных систем.

Но недетерминированное поведение – это еще и источник неприятностей, разочарований и огорчений. Всякий, кому доводилось работать над встроенными системами реального времени, знает, как трудно отлаживать недетерминированную программу.

Заключение

Кажущаяся простота паттерна Команда создает ложное впечатление о его возможностях. Этот паттерн можно использовать для самых разных целей: для реализации транзакций базы данных, управления устройствами, имитации многопоточного ядра, выполнения и отмены операций в графическом интерфейсе пользователя.

Высказывалось мнение, что паттерн Команда идет вразрез с объектно-ориентированной парадигмой, выдвигая на передний план не классы, а функции. Возможно, это и так, но разработчик реального ПО отдает предпочтение полезности, а не теории. А паттерн Команда может быть весьма полезен.

Библиография

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

[Lavender96] R. G. Lavender and D. C. Schmidt «Active Object: An Object Behavioral Pattern for Concurrent Programming», в сборнике под ред. J. O. Coplien, J. Vlissides, N. Kerth «Pattern Languages of Program Design», Addison-Wesley, 1996.

22

Шаблонный метод и Стратегия: наследование или делегирование



Лучшая жизненная стратегия – усердие.

Китайская пословица

В начале 1990-х годов, когда объектно-ориентированные технологии только зарождались, всех нас захватила идея наследования. Это отношение сулило грандиозные перспективы. С помощью наследования можно было *программировать только различия*!

То есть имея класс, делающий нечто полезное, мы могли создать его подкласс и изменить лишь те части, которые нас не устраивали. Мы могли повторно использовать код, просто унаследовав его! Мы могли организовывать целые иерархии программных конструкций, в которых на каждом уровне использовался код с предыдущих уровней. Нам открылся прекрасный новый мир.

Но, как большинство прекрасных новых миров, этот на поверку тоже оказался не вполне пригодным к обитанию. К 1995 году стало ясно, что наследованием очень просто злоупотребить, а обходится такое злоупотребление крайне дорого. Гамма, Хелм, Джонсон и Влиссидес даже сочли уместным подчеркнуть: «*Отдавайте предпочтение композиции объектов, а не наследованию классов*»[1]. Поэтому мы стали реже применять наследование, частенько заменяя его композицией или делегированием.

В этой главе мы расскажем о двух паттернах, наглядно иллюстрирующих различия между наследованием и делегированием. Паттерны Шаблонный метод и Стратегия предназначены для решения сходных задач и нередко взаимозаменяемы. Но в Шаблонном методе применяется наследование, а в Стратегии – делегирование.

И Шаблонный метод, и Стратегия решают задачу отделения общего алгоритма от конкретного контекста. Такая необходимость возникает при проектировании ПО сплошь и рядом. Имеется алгоритм общего вида, применимый к разным ситуациям. В соответствии с принципом инверсии зависимости мы хотели бы, чтобы этот алгоритм не зависел от деталей реализации. Желательно, чтобы как алгоритм, так и конкретная реализация зависели только от абстракций.

Шаблонный метод

Вспомните обо всех программах, которые вы когда-либо писали. Наверное, во многих встречался основной цикл такого вида:

```
Initialize();
while (!Done()) // основной цикл
{
  Idle(); // сделать нечто полезное.
}
Cleanup();
```

Сначала мы инициализируем приложение, а потом входим в главный цикл, где программа делает то, для чего написана. Это может быть, например, обработка событий ГИП или записей базы данных. Когда все сделано, мы выходим из главного цикла и подчищаем за собой.

[1] [GOF95], стр. 20

Эта структура настолько распространена, что ее можно инкапсулировать в класс Application и использовать его в каждой новой программе. Только подумайте, вам больше никогда не придется писать этот цикл![1]

Рассмотрим, к примеру, листинг 22.1. В нем присутствуют все элементы стандартной программы. Объекты TextReader и TextWriter уже инициализированы. Цикл в методе Main читает из объекта Console.In данные о температуре по шкале Фаренгейта и выводит их эквиваленты по шкале Цельсия. В конце печатается сообщение о выходе.

Листинг 22.1. FtoCRaw.cs

```
using System;
using System.IO;

public class FtoCRaw
{
  public static void Main(string[] args)
  {
    bool done = false;
    while (!done)
    {
      string fahrString = Console.In.ReadLine();
      if (fahrString == null || fahrString.Length == 0)
        done = true;
      else
      {
        double fahr = Double.Parse(fahrString);
        double celcius = 5.0/9.0*(fahr - 32);
        Console.Out.WriteLine("F={0}, C={1}",fahr,celcius);
      }
    }
    Console.Out.WriteLine("ftoc exit");
  }
}
```

В этой программе имеются все элементы рассмотренного выше главного цикла. Небольшая инициализация, содержательная работа в цикле, затем очистка и выход.

Отделить базовую структуру от конкретной программы ftoc позволяет паттерн Шаблонный метод. В соответствии с этим паттерном код, описывающий общую структуру алгоритма, находится в имеющем реализацию методе абстрактного базового класса, а детали вынесены в абстрактные методы.

Так, структуру главного цикла можно инкапсулировать в абстрактном базовом классе Application, как показано в листинге 22.2.

[1] А еще я могу продать вам участок на Луне.

Листинг 22.2. Application.cs

```
public abstract class Application
{
  private bool isDone = false;

  protected abstract void Init();
  protected abstract void Idle();
  protected abstract void Cleanup();
  protected void SetDone();

  {
    isDone = true;
  }

  protected bool Done()
  {
    return isDone;
  }

  public void Run()
  {
    Init();
    while (!Done())
      Idle();
    Cleanup();
  }
}
```

Здесь представлена общая структура приложения с главным циклом. Сам цикл находится в реализованном методе `Run`. А содержательная работа вынесена в абстрактные методы `Init`, `Idle` и `Cleanup`. Метод `Init` берет на себя инициализацию. Метод `Idle` выполняет основную работу программы и вызывается до тех пор, пока `Done()` возвращает `false`. Ну а метод `Cleanup` отвечает за очистку перед выходом.

Класс `FtoCRaw` можно переписать, унаследовав его от `Application` и реализовав абстрактные методы. Результат показан в листинге 22.3.

Листинг 22.3. FtoCTemplateMethod.cs

```
using System;
using System.IO;

public class FtoCTemplateMethod : Application
{
  private TextReader input;
  private TextWriter output;

  public static void Main(string[] args)
  {
    new FtoCTemplateMethod().Run();
  }

  protected override void Init()
```

```
    {
      input = Console.In;
      output = Console.Out;
    }

    protected override void Idle()
    {
      string fahrString = input.ReadLine();
      if (fahrString == null || fahrString.Length == 0)
        SetDone();
      else
      {
        double fahr = Double.Parse(fahrString);
        double celcius = 5.0/9.0*(fahr - 32);
        output.WriteLine("F={0}, C={1}", fahr, celcius);
      }
    }

    protected override void Cleanup()
    {
      output.WriteLine("ftoc exit");
    }
  }
```

Легко видеть, что приложение `ftoc` отлично ложится на паттерн Шаблонный метод.

Злоупотребление паттерном

Но сейчас вы, наверное, думаете: «*Он что, серьезно думает, что я буду использовать класс Application во всех новых приложениях? Я же ничего не выиграю, а только усложню задачу*».

Э-э... да... :^(

Я выбрал этот пример только потому, что он простой и в то же время позволяет продемонстрировать механизм применения паттерна Шаблонный метод. Однако писать реальную программу `ftoc` именно так я бы не рекомендовал.

Это хороший пример злоупотребления паттерном. Применять Шаблонный метод для данного конкретного приложения глупо. Он лишь усложняет и увеличивает программу. Инкапсуляция главного цикла всех мыслимых приложений поначалу казалась прекрасной идеей, но ее практическое воплощение в данном случае не принесло никаких осязаемых плодов.

Паттерны проектирования – чудесная вещь. Они способны помочь в решении многих задач проектирования. Но из того, что они существуют, вовсе не следует, что их нужно употреблять к месту и не к месту. Хотя к данному случаю Шаблонный метод и применим, использовать его не стоит. Издержки превышают выгоду.

Пузырьковая сортировка



А сейчас рассмотрим чуть более полезный пример (листинг 22.4). Алгоритм Bubble Sort столь же легко понять, как и класс `Application`, поэтому он удобен для дидактических целей. Однако ни один человек в здравом уме не станет применять пузырьковую сортировку, когда нужно отсортировать сколько-нибудь большой массив данных. Есть *гораздо* более эффективные алгоритмы.

Листинг 22.4. BubbleSorter.cs

```
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++;
  }
}
```

Класс `BubbleSorter` знает, как сортировать массив целых чисел, применяя алгоритм пузырьковой сортировки. Метод `Sort` содержит сам алгоритм пузырьковой сортировки, а два вспомогательных метода – `Swap`

и CompareAndSwap — посвящены деталям, связанным с целыми числами и массивами.

С помощью паттерна Шаблонный метод мы можем выделить алгоритм пузырьковой сортировки в абстрактный базовый класс BubbleSorter. Он содержит реализацию метода Sort, который вызывает абстрактные методы OutOfOrder и Swap. Метод OutOfOrder сравнивает два соседних элемента массива и возвращает true, если они расположены не по порядку. Метод Swap переставляет местами два соседних элемента массива.

Метод Sort ничего не знает о массиве, ему все равно, какие в нем хранятся объекты. Он просто вызывает OutOfOrder, передавая ему индекс элемента в массиве, и узнает, нужно ли переставить соседние элементы (листинг 22.5).

Листинг 22.5. BubbleSorter.cs

```
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 bool OutOfOrder(int index);
}
```

Имея класс BubbleSorter, мы можем создать производные от него классы для сортировки объектов других видов. Например, класс IntBubbleSorter будет сортировать массивы целых чисел, а DoubleBubbleSorter — массивы чисел с двойной точностью. См. рис. 22.1 и листинги 22.6 и 22.7.

На примере паттерна Шаблонный метод мы видим одну из классических форм повторного использования в объектно-ориентированном программировании. Обобщенный алгоритм помещается в базовый

класс, который наследуется в различных конкретных контекстах. Но с этой техникой сопряжены издержки. Наследование – очень сильное отношение. Подклассы оказываются неразрывно связаны со своими базовыми классами.

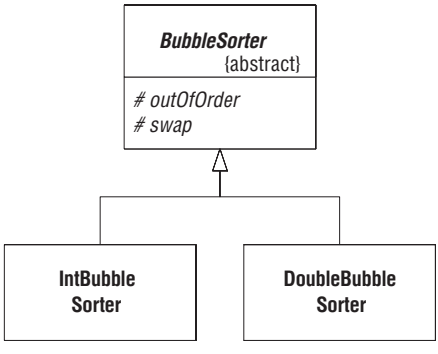


***Рис. 22.1.** Структура иерархии классов BubbleSorter*

Листинг 22.6. IntBubbleSorter.cs

```
public class IntBubbleSorter : BubbleSorter
{
  private int[] array = null;

  public int Sort(int[] theArray)
  {
    array = theArray;
    length = array.Length;
    return DoSort();
  }

  protected override void Swap(int index)
  {
    int temp = array[index];
    array[index] = array[index + 1];
    array[index + 1] = temp;
  }

  protected override bool OutOfOrder(int index)
  {
    return (array[index] > array[index + 1]);
  }
}
```

Листинг 22.7. DoubleBubbleSorter.cs

```
public class DoubleBubbleSorter : BubbleSorter
{
  private double[] array = null;
```

```
    public int Sort(double[] theArray)
    {
      array = theArray;
      length = array.Length;
      return DoSort();
    }

    protected override void Swap(int index)
    {
      double temp = array[index];
      array[index] = array[index + 1];
      array[index + 1] = temp;
    }

    protected override bool OutOfOrder(int index)
    {
      return (array[index] > array[index + 1]);
    }
  }
```

Например, методы `OutOfOrder` и `Swap`, реализованные в классе `IntBubbleSorter`, — это как раз то, что необходимо и другим алгоритмам сортировки. Однако использовать их в других алгоритмах мы уже не можем. Унаследовав класс `IntBubbleSorter` от `BubbleSorter`, мы навечно привязали один к другому. Паттерн Стратегия предлагает иной подход.

Стратегия

Паттерн Стратегия решает проблему инверсии зависимости между общим алгоритмом и деталями реализации совершенно по-другому. Давайте снова рассмотрим злоупотребивший паттернами класс `Application`.

Вместо того чтобы помещать общий алгоритм работы приложения в абстрактный базовый класс, мы поместим его в *конкретный* класс `ApplicationRunner`. Абстрактные методы, которые может вызывать общий алгоритм, мы определим в интерфейсе `Application`. Затем создадим производный от `Application` класс `FtoCStrategy` и будем передавать его в `ApplicationRunner`. Таким образом, `ApplicationRunner` делегирует содержательную работу этому интерфейсу. См. рис. 22.2 и листинги 22.8–22.10.

Должно быть понятно, что у этой структуры по сравнению с Шаблонным методом есть свои плюсы и минусы. В паттерне Стратегия больше классов и выше уровень косвенности, чем в Шаблонном методе. Делегирование по указателю в `ApplicationRunner` обходится с точки зрения времени и памяти чуть дороже, чем наследование. С другой стороны, если нам нужно запускать много приложений, то можно было бы использовать *один* экземпляр `ApplicationRunner` и передавать ему различные реализации `Application`, что позволит сэкономить память.

Но ни потери, ни приобретения сами по себе не являются определяющими. В большинстве случаев они пренебрежимо малы. Обычно наибольшее беспокойство вызывает дополнительный класс, необходимый в паттерне Стратегия. Однако это еще не все.

Рассмотрим реализацию пузырьковой сортировки на основе паттерна Стратегия (листинги 22.11–22.13).

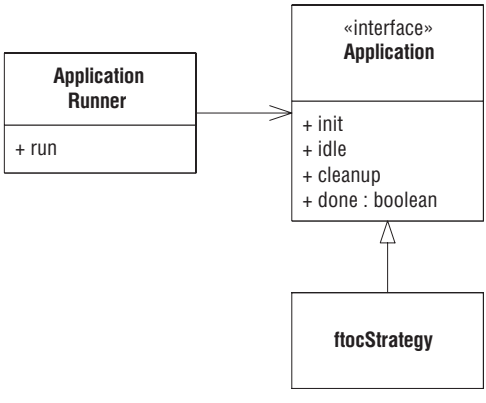


***Рис. 22.2.** Структура алгоритма Application с паттерном Стратегия*

Листинг 22.8. ApplicationRunner.cs

```
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 22.9. Application.cs

```
public interface Application
{
  void Init();
  void Idle();
  void Cleanup();
```

```
  bool Done();
}
```

Листинг 22.10. FtoCStrategy.cs

```
using System;
using System.IO;

public class FtoCStrategy : Application
{
  private TextReader input;
  private TextWriter output;
  private bool isDone = false;

  public static void Main(string[] args)
  {
     (new ApplicationRunner(new FtoCStrategy())).run();
  }

  public void Init()
  {
    input = Console.In;
    output = Console.Out;
  }

  public void Idle()
  {
    string fahrString = input.ReadLine();
    if (fahrString == null || fahrString.Length == 0)
      isDone = true;
    else
    {
      double fahr = Double.Parse(fahrString);
      double celcius = 5.0/9.0*(fahr - 32);
      output.WriteLine("F={0}, C={1}", fahr, celcius);
    }
  }

  public void Cleanup()
  {
    output.WriteLine("ftoc exit");
  }

  public bool Done()
  {
    return isDone;
  }
}
```

Листинг 22.11. BubbleSorter.cs

```
public class BubbleSorter
{
```

```
  private int operations = 0;
  private int length = 0;
  private SortHandler itsSortHandler = null;

  public BubbleSorter(SortHandler handler)
  {
    itsSortHandler = handler;
  }

  public int Sort(object array)
  {
    itsSortHandler.SetArray(array);
    length = itsSortHandler.Length();
    operations = 0;
    if (length <= 1)
      return operations;
    for (int nextToLast = length - 2;
      nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
      {
        if (itsSortHandler.OutOfOrder(index))
        itsSortHandler.Swap(index);
        operations++;
      }

    return operations;
  }
}
```

Листинг 22.12. SortHandler.cs

```
public interface SortHandler
{
  void Swap(int index);
  bool OutOfOrder(int index);
  int Length();
  void SetArray(object array);
}
```

Листинг 22.13. Int SortHandler.cs

```
public class IntSortHandler : SortHandler
{
  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 bool OutOfOrder(int index)
    {
      return (array[index] > array[index + 1]);
    }
  }
```

Отметим, что класс `IntSortHandler` ничего не знает о `BubbleSorter` и никак не зависит от реализации пузырьковой сортировки. В случае Шаблонного метода дело обстоит иначе. Посмотрите еще раз на листинг 22.6 – вы увидите, что `IntBubbleSorter` напрямую зависит от класса `BubbleSorter`, содержащего алгоритм пузырьковой сортировки.

Паттерн Шаблонный метод отчасти нарушает принцип инверсии зависимости. Реализация методов `Swap` и `OutOfOrder` зависит от алгоритма пузырьковой сортировки. В паттерне Стратегия такой зависимости нет – класс `IntSortHandler` можно использовать и с другими реализациями сортировщика, а не только с `BubbleSorter`.

Например, можно написать вариант пузырьковой сортировки, который прекращал бы работу, как только на очередном проходе по массиву выясняется, что он уже отсортирован (см. листинг 22.14). Такой класс `QuickBubbleSorter` мог бы воспользоваться классом `IntSortHandler` или любым другим, производным от `SortHandler`.

Листинг 22.14. QuickBubbleSorter.cs

```
public class QuickBubbleSorter
{
  private int operations = 0;
  private int length = 0;
  private SortHandler itsSortHandler = null;

  public QuickBubbleSorter(SortHandler handler)
  {
    itsSortHandler = handler;
  }

  public int Sort(object array)
  {
    itsSortHandler.SetArray(array);
    length = itsSortHandler.Length();
    operations = 0;
    if (length <= 1)
```

```
      return operations;
    bool thisPassInOrder = false;
    for (int nextToLast = length-2;
      nextToLast >= 0 && !thisPassInOrder; nextToLast--)
    {
      thisPassInOrder = true; //potenially.
      for (int index = 0; index <= nextToLast; index++)
      {
        if (itsSortHandler.OutOfOrder(index))
        {
          itsSortHandler.Swap(index);
          thisPassInOrder = false;
        }
        operations++;
      }
    }

    return operations;
  }
}
```

Итак, у паттерна Стратегия есть одно преимущество по сравнению с Шаблонным методом. Если Шаблонный метод позволяет подставлять в общий алгоритм различные детальные реализации, то Стратегия в полном соответствии с принципом DIP еще и разрешает использовать любую детальную реализацию в различных общих алгоритмах.

Заключение

Паттерн Шаблонный метод легко использовать на практике, но он недостаточно гибок. Паттерн Стратегия обладает нужной гибкостью, но приходится вводить дополнительный класс, создавать дополнительный объект и инкорпорировать его в систему. Поэтому выбор между этими паттернами зависит от того, нужна ли вам гибкость Стратегии или вы готовы удовольствоваться простотой Шаблонного метода. Лично я неоднократно останавливался на Шаблонном методе просто потому, что его проще реализовать и использовать. Например, я бы применил его в задаче о пузырьковой сортировке, если бы не был абсолютно уверен, что потребуются и другие алгоритмы сортировки.

Библиография

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

[PLOPD3] Robert C. Martin, Dirk Riehle, Frank Buschmann, eds. «Pattern Languages of Program Design 3», Addison-Wesley, 1998.

23

Фасад и Посредник



Символизм воздвигает респектабельный фасад,
чтобы скрыть за ним низменные фантазии.

Мейсон Кули

Оба паттерна, рассматриваемые в этой главе, преследуют одну цель: наложить какую-то политику на группу объектов. Фасад (Facade) накладывает политику сверху, а Посредник (Mediator) – снизу. Фасад виден и вводит ограничения, Посредник не виден и ни в чем не ограничивает.

Фасад

Паттерн Фасад применяется, когда нужно предоставить простой специализированный интерфейс к группе объектов, имеющих сложный общий интерфейс. Рассмотрим, к примеру, файл `DB.cs` в листинге 34.9.

Этот класс накладывает очень простой интерфейс, специфичный для ProductData, на сложные общие интерфейсы классов из пространства имен System.Data. Его структура изображена на рис. 23.1.

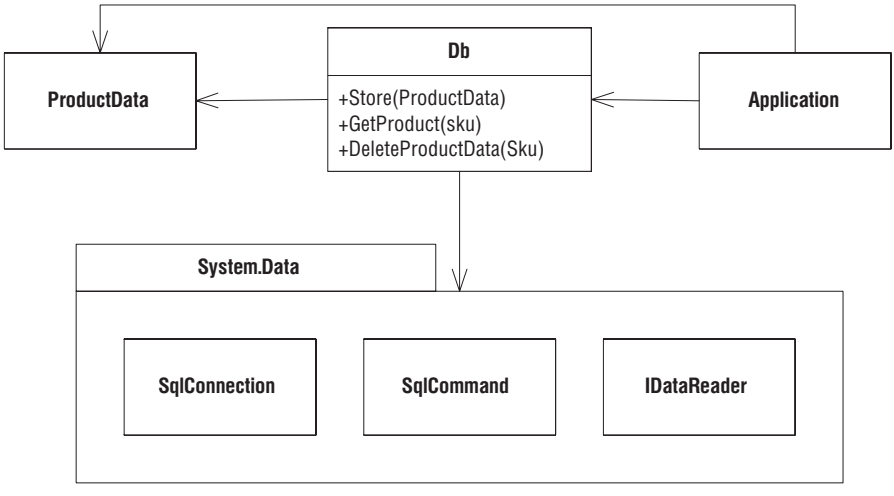


***Рис. 23.1.** Фасад DB*

Отметим, что класс DB избавляет Application от необходимости вникать в тонкости пространства имен System.Data. Он скрывает общность и сложность System.Data за простым специализированным интерфейсом.

Класс DB, являющийся частным случаем Фасада, определяет политику использования System.Data; он знает, как открыть и закрыть соединение с базой данных, как установить соответствие между переменными-членами ProductData и полями базы данных, как строить запросы для манипулирования данными. Вся эта сложность скрыта от пользователя. С точки зрения Application пространства имен System.Data вообще не существует, оно скрыто за Фасадом.

Использование паттерна Фасад подразумевает следующее: разработчики согласны с тем, что все обращения к базе данных должны производиться только через класс DB. Если где-то в коде Application имеются прямые обращения к классам из System.Data в обход Фасада, то это соглашение нарушается. Таким образом, Фасад навязывает приложению свою политику. По соглашению класс DB становится единственным уполномоченным представителем System.Data.

Фасад можно использовать для сокрытия любого аспекта программы. Однако применение его для сокрытия деталей работы с базой данных настолько распространено, что даже выделилось в отдельный паттерн Шлюз к табличным данным (Table Data Gateway).

Посредник

Паттерн Посредник также накладывает политику. Но если политика, налагаемая Фасадом, видимая и ограничительная, то Посредник работает скрытно и не вводит никаких ограничений. Например, класс `QuickEntryMediator` в листинге 23.1 тихонько сидит за кулисами и привязывает текстовое поле ввода к списку. Когда вы вводите текст в поле, первый элемент списка, начинающийся с введенной строки, подсвечивается. Это позволяет набирать только начало текста и затем производить быстрый выбор из списка.

Листинг 23.1. QuickEntryMediator.cs

```
using System;
using System.Windows.Forms;

/// <summary>
/// QuickEntryMediator. Этот класс принимает объекты TextBox
/// и ListBox. Предполагается, что пользователь будет вводить
/// в TextBox префиксы строк, находящихся в ListBox. Класс
/// автоматически выбирает первый элемент ListBox, который
/// начинается с префикса, введенного в TextBox.
///
/// Если значение в поле TextBox равно null или префикс
/// не соответствует никакому элементу ListBox, то выделение
/// в ListBox снимается.
///
/// В этом классе нет открытых методов. Вы просто создаете
/// объект класса и забываете о его существовании. (Но следите
/// за тем, чтобы он не был передан сборщику мусора...)
///
/// Пример:
///
/// TextBox t = new TextBox();
/// ListBox l = new ListBox();
///
/// QuickEntryMediator qem = new QuickEntryMediator(t,l);
/// // и больше ничего не надо.
///
/// Первоначально написан на Java
/// авторы Роберт К. Мартин, Роберт С. Косс
/// 30 Jun, 1999 2113 (SLAC)
/// Перевел на C# Мик Мартин
/// May 23, 2005 (в поезде)
/// </summary>
public class QuickEntryMediator
{
  private TextBox itsTextBox;
  private ListBox itsList;
```

```
  public QuickEntryMediator(TextBox t, ListBox l)
  {
    itsTextBox = t;
    itsList = l;
    itsTextBox.TextChanged += new EventHandler(TextFieldChanged);
  }

  private void
    TextFieldChanged(object source, EventArgs args)
  {
    string prefix = itsTextBox.Text;
    if (prefix.Length == 0)
    {
      itsList.ClearSelected();
      return;
    }

    ListBox.ObjectCollection listItems = itsList.Items;
    bool found = false;
    for (int i = 0; found == false &&
             i < listItems.Count; i++)
    {
      Object o = listItems[i];
      String s = o.ToString();
      if (s.StartsWith(prefix))
      {
        itsList.SetSelected(i, true);
        found = true;
      }
    }
    if (!found)
    {
      itsList.ClearSelected();
    }
  }
}
```

Структура класса `QuickEntryMediator` показана на рис. 23.2. Конструктору экземпляра `QuickEntryMediator` передаются ссылки на `ListBox` и `TextBox`. `QuickEntryMediator` регистрирует обработчик события `TextChanged` от `TextBox`. Этот обработчик при любом изменении текста вызывает метод `TextFieldChanged`, который ищет в списке `ListBox` элемент, начинающийся с текущего значения текстового поля, и выделяет его.

Пользователи классов `ListBox` и `TextField` понятия не имеют о существовании этого Посредника. Он сидит в сторонке и незаметно накладывает свою политику на объекты, не спрашивая у них разрешения и даже не ставя их в известность.

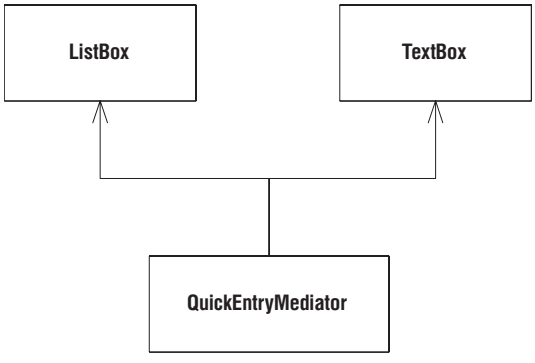


Рис. 23.2. *QuickEntryMediator*

Заключение

Накладывать политику можно сверху, используя паттерн Фасад, если эта политика должна быть явной. С другой стороны, если необходимы скромность и деликатность, то больше подойдет паттерн Посредник. Фасады обычно служат предметом соглашения. Все должны быть готовы использовать Фасад вместо скрывающихся за ним объектов. Посредник, напротив, скрыт от пользователей. Его политика – это свершившийся факт, а не предмет договоренностей.

Библиография

[Fowler03] Martin Fowler «Patterns of Enterprise Application Architecture», Addison-Wesley, 2003.[1]

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

[1] Мартин Фаулер «Архитектура корпоративных программных приложений». – Пер. с англ. – Вильямс, 2007.

24

Одиночка и Моносостояние



Неисчерпаемая красота бытия! Она!
Она есть Она и только Она, ничего, кроме Нее.

Эдвин Эббот, Флатландия (1884)

Обычно между классами и их экземплярами существует отношение один-ко-многим, то есть можно создавать много экземпляров одного класса. Экземпляры создаются, когда в них возникает нужда, и уничтожаются, когда перестают быть необходимыми. Их приносит и уносит поток выделения и освобождения памяти.

Но у некоторых классов должен быть только один экземпляр. Этот экземпляр должен быть создан в начале работы программы и уничтожен вместе с ее завершением. Иногда такие объекты являются корневыми

объектами приложения. Следуя от корня, можно добраться до многих других объектов системы. А иногда они служат фабриками, порождающими другие объекты. А бывают и менеджерами, которые следят за другими объектами и сопровождают их на жизненном пути.

Для чего бы такие объекты ни использовались, наличие нескольких их экземпляров было бы серьезной логической ошибкой. Если существует более одного корня, то доступ к объектам приложения может зависеть от выбранного корня. Программисты, не знающие о наличии нескольких корней, могут, не сознавая того, видеть лишь подмножество всех объектов приложения. Если существует несколько фабрик, то может быть потерян контроль над созданными объектами. При наличии нескольких менеджеров операции, предполагавшиеся последовательными, могут оказаться параллельными.

Может показаться, что вводить специальные механизмы обеспечения единственности таких объектов – излишество. В конце концов, на этапе инициализации приложения можно просто создать по одному экземпляру каждого, и дело с концом.[1] На самом деле это обычно самый лучший подход. Подобных механизмов следует избегать, если в них нет очевидной и безусловной необходимости. Но ведь мы еще хотим, чтобы код выражал наши намерения. Если механизм обеспечения единственности тривиален, то выгода может превысить затраты.

В этой главе рассказывается о двух паттернах, гарантирующих единственность. Соотношение достоинств и недостатков в них существенно различается. В большинстве контекстов издержки, сопряженные с использованием этих паттернов, достаточно низки и с лихвой окупаются достигаемой выразительностью.

Одиночка

Одиночка (Singleton) – очень простой паттерн.[2] Тесты в листинге 24.1 показывают, как он должен работать. Из первого теста видно, что к экземпляру `Singleton` обращаются с помощью открытого статического метода `Instance`, и если вызвать `Instance` несколько раз, то мы неизменно будем получать ссылку на один и тот же экземпляр. Из второго теста видно, что класс `Singleton` не имеет открытых конструкторов, поэтому невозможно создать его экземпляр в обход метода `Instance`.

Листинг 24.1. Тесты для класса Singleton

```
using System;
using System.Reflection;
using NUnit.Framework;
```

[1] Такой шаблон я называю «Просто создай одного».

[2] [GOF95], стр. 127

```
[TestFixture]
public class TestSimpleSingleton
{
  [Test]
  public void TestCreateSingleton()
  {
    Singleton s = Singleton.Instance;
    Singleton s2 = Singleton.Instance;
    Assert.AreSame(s, s2);
  }

  [Test]
  public void TestNoPublicConstructors()
  {
    Type singleton = typeof(Singleton);
    ConstructorInfo[] ctrs = singleton.GetConstructors();
    bool hasPublicConstructor = false;
    foreach(ConstructorInfo c in ctrs)
    {
      if(c.IsPublic)
      {
        hasPublicConstructor = true;
        break;
      }
    }
    Assert.IsFalse(hasPublicConstructor);
  }
}
```

Этот набор тестов служит спецификацией паттерна Одиночка и непосредственно подводит к коду, показанному в листинге 24.2. Из него с очевидностью следует, что в области видимости статической переменной `Singleton.theInstance` не может быть более одного экземпляра класса `Singleton`.

Листинг 24.2. Реализация класса Singleton

```
public class Singleton
{
  private static Singleton theInstance = null;
  private Singleton() {}

  public static Singleton Instance
  {
    get
    {
      if (theInstance == null)
        theInstance = new Singleton();
      return theInstance;
    }
  }
}
```

Достоинства

- *Нелокальность*. При использовании подходящего ПО промежуточного уровня (например, технологии Remoting) паттерн Одиночка можно обобщить так, что единственность будет обеспечиваться в нескольких экземплярах CLR (общеязыковой среды выполнения) и на нескольких компьютерах.
- *Применимость к любому классу*. Любой класс можно преобразовать в Одиночку, если сделать его конструкторы закрытыми и добавить соответствующие статические методы и переменную-член.
- *Может быть создан путем наследования*. Имея некоторый класс, можно создать его подкласс, который будет Одиночкой.
- *Отложенное вычисление*. Если Одиночка не используется, то он и не создается.

Недостатки

- *Уничтожение не определено*. Не существует приемлемого способа уничтожить или «списать» Одиночку. Даже если добавить метод, обнуляющий переменную `theInstance`, другие модули могут хранить у себя ссылку на Одиночку. При последующих обращениях к `Instance` будет создан новый экземпляр, что приведет к образованию двух одновременно существующих экземпляров. Эта проблема особенно остро стоит в языке C++, где экземпляр *может быть уничтожен*, что приведет к разыменованию уже не существующего объекта.
- *Не наследуется*. Класс, производный от Одиночки, сам не является Одиночкой. Если необходимо, чтобы он был Одиночкой, придется добавить статический метод и переменную-член.
- *Эффективность*. Каждое обращение к свойству `Instance` приводит к выполнению предложения `if`. Для большинства обращений это предложение бесполезно.
- *Непрозрачность*. Пользователи Одиночки знают, с чем имеют дело, потому что вынуждены обращаться к свойству `Instance`.

Одиночка в действии

Предположим, что имеется веб-приложение, позволяющее пользователям входить в защищенные области сервера. В таком приложении будет некая база данных, содержащая имена, пароли и другие атрибуты пользователей. Предположим далее, что доступ к базе данных осуществляется с помощью стороннего API. Можно было бы в каждом модуле, которому необходимо читать и изменять данные о пользователях, обращаться к базе напрямую. Но тогда вызовы стороннего API оказались бы разбросаны по всему коду, что лишило бы нас возможности навязать какие-то соглашения о доступе и структуре программы.

Лучше воспользоваться паттерном Фасад и создать класс `UserDatabase`, предоставляющий методы для чтения и изменения объектов `User`.[1] Эти методы обращаются к стороннему API доступа к базе данных, осуществляя отображение между объектами `User` и таблицами базы. Внутри класса `UserDatabase` можно обеспечить соглашения о структуре и порядке доступа. Например, можно гарантировать, что не будет добавлена запись `User`, в которой поле `username` пусто. Или сериализовать обращения к записи `User`, так, чтобы никакие два модуля не могли одновременно читать и изменять ее.

Решение на основе паттерна Одиночка показано в листингах 24.3 и 24.4. Соответствующий класс называется `UserDatabaseSource` и реализует интерфейс `UserDatabase`. Отметим, что в коде свойства `Instance` нет традиционного предложения `if`, защищающего от многократного создания. Вместо этого используется механизм статической инициализации, имеющийся в .NET.

Листинг 24.3. Интерфейс UserDatabase

```
public interface UserDatabase
{
  User ReadUser(string userName);
  void WriteUser(User user);
}
```

Листинг 24.4. Класс-одиночка UserDatabase

```
public class UserDatabaseSource : UserDatabase
{
  private static UserDatabase theInstance =
    new UserDatabaseSource();

  public static UserDatabase Instance
  {
    get
    {
      return theInstance;
    }
  }

  private UserDatabaseSource()
  {}

  public User ReadUser(string userName)
  {
    // Реализация
  }
```

[1] Этот частный случай шаблона Фасад известен под названием Шлюз (Gateway). Подробное обсуждение шлюзов см. в книге [Fowler03].

```
    public void WriteUser(User user)
    {
      // Реализация
    }
  }
```

Такое использование паттерна Одиночка распространено чрезвычайно широко. Гарантируется, что весь доступ к базе данных производится через единственный экземпляр `UserDatabaseSource`. При этом в `UserDatabaseSource` очень легко вставлять различные проверки, счетчики и блокировки, обеспечивающие выполнение вышеупомянутых соглашений о порядке доступа и структуре кода.

Моносостояние

Паттерн Моносостояние (Monostate) предлагает иной способ обеспечения единственности с использованием совершенно другого механизма. Как этот механизм работает, видно из тестов в листинге 24.5.

В первом тесте просто описан объект, имеющий свойство `x`, которое можно читать и устанавливать. Однако из второго теста следует, что два экземпляра одного и того же класса ведут себя так, *будто это единственный экземпляр*. Записав в свойство `x` одного экземпляра некоторое значение, мы получаем это же значение, прочитав свойство `x` другого экземпляра. Создается впечатление, что эти два экземпляра являются разными именами одного объекта.

Листинг 24.5. Тестовая фикстура для паттерна Моносостояние

```
using NUnit.Framework;

[TestFixture]
public class TestMonostate
{
  [Test]
  public void TestInstance()
  {
    Monostate m = new Monostate();
    for (int x = 0; x < 10; x++)
    {
      m.X = x;
      Assert.AreEqual(x, m.X);
    }
  }

  [Test]
  public void TestInstancesBehaveAsOne()
  {
    Monostate m1 = new Monostate();
    Monostate m2 = new Monostate();
```

```
    for (int x = 0; x < 10; x++)
    {
      m1.X = x;
      Assert.AreEqual(x, m2.X);
    }
  }
}
```

Если бы мы заменили в этих тестах все предложения `new Monostate` вызовами `Singleton.Instance`, то тесты все равно прошли бы успешно. Таким образом, тесты описывают поведение Одиночки, не налагая ограничения на единственность экземпляра!

Каким образом два экземпляра могут вести себя так, будто это единственный объект? Да просто это означает, что у них одни и те же переменные-члены. А добиться этого можно, сделав все переменные-члены статическими. В листинге 24.6 приведена реализация класса `Monostate`, которая проходит все тесты. Отметим, что переменная `itsX` статическая, но *ни один метод статическим не является*. Ниже мы увидим, что это важно.

Листинг 24.6. Реализация класса Monostate

```
public class Monostate
{
  private static int itsX;

  public int X
  {
    get { return itsX; }
    set { itsX = value; }
  }
}
```

Я нахожу этот паттерн восхитительным в своей причудливости. Сколько бы экземпляров класса `Monostate` ни создать, все они ведут себя так, как будто являются одним и тем же объектом. Можно даже уничтожить все текущие экземпляры, не потеряв при этом данных.

Отметим, что различие между двумя описанными паттернами – это различие между поведением и структурой. Паттерн Одиночка навязывает структуру единственности, не позволяя создать более одного экземпляра. Моносостояние, напротив, навязывает поведение единственности, не налагая структурных ограничений. Это различие станет понятным, если заметить, что тесты для паттерна Моносостояние проходят и для класса `Singleton`, однако у класса `Monostate` нет ни малейшей надежды пройти тесты для Одиночки.

Достоинства

- *Прозрачность*. Пользователь работает точно так же, как с обычным объектом, ничего не зная о том, что это «моносостояние».

- *Допускает наследование*. Подклассы моносостояния также обладают этим свойством. Более того, все его подклассы являются частями *одного и того же* моносостояния, так как разделяют одни и те же статические переменные-члены.
- *Полиморфизм*. Поскольку методы моносостояния не являются статическими, их можно переопределять в производных классах. Это означает, что подклассы могут реализовывать различное поведение при одном и том же наборе статических переменных-членов.
- *Точно определенные моменты создания и уничтожения*. Поскольку переменные-члены моносостояния статические, то моменты их создания и уничтожения точно определены.

Недостатки

- *Невозможность преобразования*. Класс, не являющийся моносостоянием, невозможно превратить в моносостояние с помощью наследования.
- *Эффективность*. Будучи настоящим объектом, моносостояние может многократно создаваться и уничтожаться. Иногда это обходится дорого.
- *Постоянное присутствие*. Переменные-члены моносостояния занимают место в памяти, даже если объект никогда не используется.
- *Локальность*. Паттерн Моносостояние не может гарантировать единственность в нескольких экземплярах CLR или на нескольких компьютерах.

Моносостояние в действии

Рассмотрим реализацию простого конечного автомата (КА), описывающего работу турникета в метро (рис. 24.1). Первоначально турникет находится в состоянии `Locked` (Закрыт). Если опустить монету, турникет перейдет в состояние `Unlocked`, откроет дверцы, сбросит сигнал тревоги (если он был включен) и поместит монету в монетоприемник. Если в этот момент пользователь пройдет через турникет, тот вернется в состояние `Locked` и закроет дверцы.

Существуют два аномальных условия. Если пользователь опускает несколько монет, прежде чем пройти, то лишние монеты возвращаются, а дверцы остаются открытыми. Если пользователь пытается пройти, не заплатив, то раздается сигнал тревоги и дверцы остаются закрытыми.

Тестовая программа, описывающая работу турникета, приведена в листинге 24.7. Заметьте, во всех методах предполагается, что объект `Turnstile` (Турникет) – моносостояние, поэтому предназначенные ему события можно посылать через различные экземпляры. Это разумно, если считать, что турникет может быть только один.

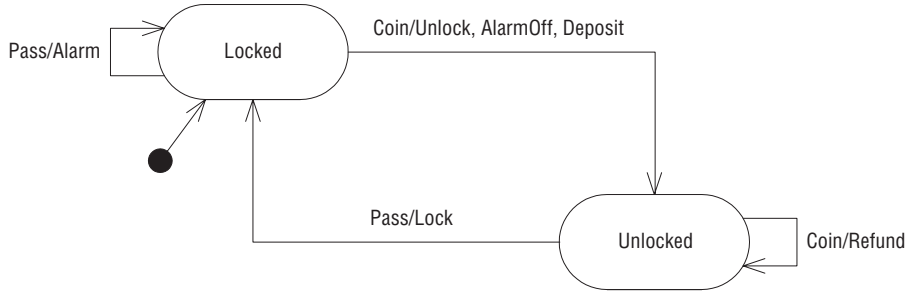


***Рис. 24.1.** Конечный автомат турникета*

Реализация моносостояния `Turnstile` показана в листинге 24.8. Базовый класс `Turnstile` делегирует методы, описывающие события `Coin` (Опущена монета) и `Pass` (Попытка пройти), подклассам `Locked` и `Unlocked`, которым представляют состояния конечного автомата.

Листинг 24.7. TurnstileTest

```
using NUnit.Framework;

[TestFixture]
public class TurnstileTest
{
  [SetUp]
  public void SetUp()
  {
    Turnstile t = new Turnstile();
    t.reset();
  }

  [Test]
  public void TestInit()
  {
    Turnstile t = new Turnstile();
    Assert.IsTrue(t.Locked());
    Assert.IsFalse(t.Alarm());
  }

  [Test]
  public void TestCoin()
  {
    Turnstile t = new Turnstile();
    t.Coin();
    Turnstile t1 = new Turnstile();
    Assert.IsFalse(t1.Locked());
    Assert.IsFalse(t1.Alarm());
    Assert.AreEqual(1, t1.Coins);
}
```

```
[Test]
public void TestCoinAndPass()
{
  Turnstile t = new Turnstile();
  t.Coin();
  t.Pass();

  Turnstile t1 = new Turnstile();
  Assert.IsTrue(t1.Locked());
  Assert.IsFalse(t1.Alarm());
  Assert.AreEqual(1, t1.Coins, “coins”);
}

[Test]
public void TestTwoCoins()
{
  Turnstile t = new Turnstile();
  t.Coin();
  t.Coin();

  Turnstile t1 = new Turnstile();
  Assert.IsFalse(t1.Locked(), “unlocked”);
  Assert.AreEqual(1, t1.Coins, “coins”);
  Assert.AreEqual(1, t1.Refunds, «refunds»);
  Assert.IsFalse(t1.Alarm());
}

[Test]
public void TestPass()
{
  Turnstile t = new Turnstile();
  t.Pass();
  Turnstile t1 = new Turnstile();
  Assert.IsTrue(t1.Alarm(), «alarm»);
  Assert.IsTrue(t1.Locked(), «locked»);
}

[Test]
public void TestCancelAlarm()
{
  Turnstile t = new Turnstile();
  t.Pass();
  t.Coin();
  Turnstile t1 = new Turnstile();
  Assert.IsFalse(t1.Alarm(), “alarm”);
  Assert.IsFalse(t1.Locked(), «locked»);
  Assert.AreEqual(1, t1.Coins, “coin”);
  Assert.AreEqual(0, t1.Refunds, «refund»);
}
```

```
  [Test]
  public void TestTwoOperations()
  {
    Turnstile t = new Turnstile();
    t.Coin();
    t.Pass();
    t.Coin();
    Assert.IsFalse(t.Locked(), «unlocked»);
    Assert.AreEqual(2, t.Coins, «coins»);
    t.Pass();
    Assert.IsTrue(t.Locked(), «locked»);
  }
}
```

Листинг 24.8. Класс Turnstile

```
public class Turnstile
{
  private static bool isLocked = true;
  private static bool isAlarming = false;
  private static int itsCoins = 0;
  private static int itsRefunds = 0;
  protected static readonly
  Turnstile LOCKED = new Locked();
  protected static readonly
    Turnstile UNLOCKED = new Unlocked();
  protected static Turnstile itsState = LOCKED;

  public void reset()
  {
    Lock(true);
    Alarm(false);
    itsCoins = 0;
    itsRefunds = 0;
    itsState = LOCKED;
  }

  public bool Locked()
  {
    return isLocked;
  }

  public bool Alarm()
  {
    return isAlarming;
  }

  public virtual void Coin()
  {
    itsState.Coin();
  }
```

```
  public virtual void Pass()
  {
    itsState.Pass();
  }

  protected void Lock(bool shouldLock)
  {
    isLocked = shouldLock;
  }

  protected void Alarm(bool shouldAlarm)
  {
    isAlarming = shouldAlarm;
  }

  public int Coins
  {
    get { return itsCoins; }
  }

  public int Refunds
  {
    get { return itsRefunds; }
  }

  public void Deposit()
  {
    itsCoins++;
  }

  public void Refund()
  {
    itsRefunds++;
  }
}

internal class Locked : Turnstile
{
  public override void Coin()
  {
    itsState = UNLOCKED;
    Lock(false);
    Alarm(false);
    Deposit();
  }

  public override void Pass()
  {
    Alarm(true);
  }
}
```

```
internal class Unlocked : Turnstile
{
  public override void Coin()
  {
    Refund();
  }

  public override void Pass()
  {
    Lock(true);
    itsState = LOCKED;
  }
}
```

В этом примере продемонстрированы полезные особенности паттерна Моносостояние. Мы воспользовались возможностью создавать полиморфные подклассы и тем фактом, что подклассы сами являются моносостояниями. Кроме того, видно, насколько трудно бывает превратить объект-моносостояние в объект, таковым не являющийся. Структура решения существенно опирается на то, что `Turnstile` – моносостояние. Если бы мы захотели применить этот КА к управлению несколькими турникетами, код пришлось бы сильно переработать.

У вас мог возникнуть вопрос в связи с необычным использованием наследования в этом примере. Тот факт, что классы `Unlocked` и `Locked` сделаны производными от `Turnstile`, представляется нарушением принципов ООП. Но поскольку `Turnstile` – моносостояние, то не существует его отдельных экземпляров. Поэтому `Unlocked` и `Locked` – это не самостоятельные классы, а части абстракции `Turnstile`. Они имеют доступ к тем же переменным и методам, что и `Turnstile`.

Заключение

Часто бывает необходимо обеспечить единственность объекта некоторого класса. В этой главе мы ознакомились с двумя принципиально различными способами решения этой задачи. Паттерн Одиночка опирается на использование закрытых конструкторов, статической переменной-члена и статического метода, которые в совокупности ограничивают количество создаваемых экземпляров. В паттерне Моносостояние все переменные-члены просто сделаны статическими.

Одиночку лучше применять, когда уже есть некоторый класс; тогда обеспечить единственность экземпляра можно, создав его подкласс, если, конечно, вы ничего не имеете против обращения к свойству `Instance` для получения доступа к этому экземпляру. Моносостояние удобнее, когда единичную природу класса желательно сделать прозрачной для пользователей или когда необходимо полиморфное поведение единственного объекта.

Библиография

[Fowler03] Martin Fowler «Patterns of Enterprise Application Architecture», Addison-Wesley, 2003.

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

[PLOPD3] Robert C. Martin, Dirk Riehle, and Frank Buschmann, eds. «Pattern Languages of Program Design 3», Addison-Wesley, 1998.

25

Null-объект



Неправильно безупречные, холодно правильные,
роскошно невыразительные. Мертвое совершенство.

Лорд Альфред Теннисон (1809–1892)

Описание

Рассмотрим следующий код:

```
Employee e = DB.GetEmployee("Bob");
if (e != null && e.IsTimeToPay(today))
  e.Pay();
```

Мы запрашиваем у базы данных объект `Employee`, представляющий работника по имени Bob. Если такого работника не существует, то класс

DB вернет null, в противном случае – интересующий нас объект Employee. Если работник существует и ему должна быть начислена зарплата, то мы вызываем метод Pay.

Все мы писали такой код. Эта идиома стала привычной, потому что в языках, ведущих происхождение от C, сначала вычисляется первый член выражения &&, а второй – только в случае, когда первый равен true. Многие программисты обжигались, забыв включить проверку на null. Но какой бы распространенной ни была эта идиома, она некрасива и провоцирует ошибки.

Снизить шансы на ошибку можно, заставив метод DB.GetEmployee возбуждать исключение вместо того, чтобы возвращать null. Однако блоки try/catch могут выглядеть еще уродливее, чем проверка на null.

Проблему можно решить с помощью паттерна Null-объект (Null Object).[1] Он устраняет необходимость проверки на null и способствует упрощению кода.

Структура паттерна показана на рис. 25.1. Employee становится интерфейсом, у которого есть две реализации. Класс EmployeeImplementation, регулярная реализация, содержит все методы и переменные, которые можно ожидать в классе, описывающем работника. Если запись о работнике есть в базе данных, то DB.GetEmployee возвращает экземпляр EmployeeImplementation. Объект NullEmployee возвращается в том случае, когда метод DB.GetEmployee не нашел работника.

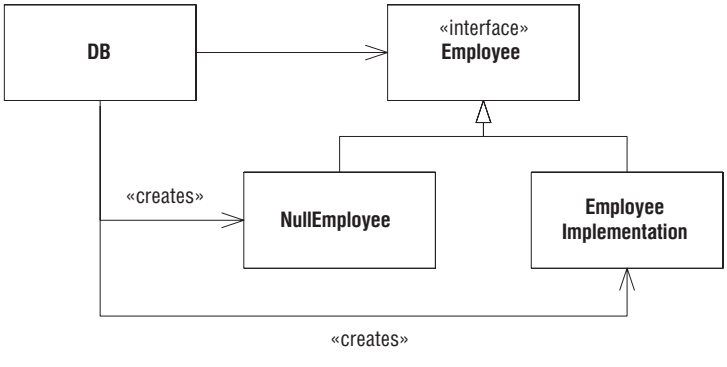


Рис. 25.1. *Паттерн Null-объект*

В классе NullEmployee методы Employee реализованы, но «ничего не делают». Что именно означает слово «ничего», зависит от конкретного метода. Например, разумно предположить, что метод IsTimeToPay (Пора платить) будет возвращать false, поскольку время платить несуществующему работнику NullEmployee не настанет никогда.

[1] [PLOPD3], стр. 5. Блестящая статья, полная юмора, иронии и вполне практических советов.

Таким образом, воспользовавшись этим паттерном, мы сможем переписать код в следующем виде:

```
Employee e = DB.GetEmployee("Bob");
if (e.IsTimeToPay(today))
  e.Pay();
```

Этот вариант и не уродлив, и не подвержен ошибкам. Ему свойственны элегантность и согласованность. Метод `DB.GetEmployee` *всегда* возвращает объект `Employee`. Гарантируется, что этот объект ведет себя ожидаемым образом вне зависимости от того, найден работник или нет.

Разумеется, во многих случаях нам хотелось бы знать, найден работник или нет. Этого можно добиться, создав в классе `Employee` переменную-член с атрибутами `static readonly`, в которой будет храниться единственно возможный экземпляр `NullEmployee`.

В листинге 25.1 приведен тест для класса `NullEmployee`. В данном случае работника Bob не существует. Заметьте, тест ожидает, что метод `IsTimeToPay` вернет `false`, а объект, возвращенный методом `DB.GetEmployee`, совпадает с `Employee.NULL`.

Листинг 25.1. EmployeeTest.cs (неполный)

```
[Test]
public void TestNull()
{
  Employee e = DB.GetEmployee("Bob");
  if (e.IsTimeToPay(new DateTime()))
   Assert.Fail();
  Assert.AreSame(Employee.NULL, e);
}
```

Сам класс `DB` показан в листинге 25.2. Для целей тестирования метод `GetEmployee` просто возвращает `Employee.NULL`.

Листинг 25.2. DB.cs

```
public class DB
{
  public static Employee GetEmployee(string s)
  {
    return Employee.NULL;
  }
}
```

Класс `Employee` показан в листинге 25.3. Обратите внимание на статическую переменную `NULL`. В ней хранится единственный экземпляр закрытого вложенного класса `NullEmployee`, в котором метод `IsTimeToPay` возвращает `false`, а метод `Pay` не делает ничего.

Листинг 25.3. Employee.cs

```
using System;

public abstract class Employee
{
  public abstract bool IsTimeToPay(DateTime time);
  public abstract void Pay();
  public static readonly Employee NULL =
    new NullEmployee();

  private class NullEmployee : Employee
  {
    public override bool IsTimeToPay(DateTime time)
    {
      return false;
    }

    public override void Pay()
    {
    }
  }
}
```

Сделав `NullEmployee` закрытым вложенным классом, мы гарантируем единственность экземпляра. Никто просто не сможет создать еще один экземпляр. И это хорошо, потому что нам хотелось иметь возможность написать такое предложение:

```
if (e == Employee.NULL)
```

Если бы экземпляров null-работника могло быть несколько, то такое сравнение было бы некорректным.

Заключение

Программисты, уже давно пишущие на языках, произошедших от C, привыкли к функциям, которые возвращают `null` или `0` в качестве признака ошибки. Предполагается, что вызывающая программа должна проверить возвращенное значение. Паттерн Null-объект меняет положение дел. Он позволяет всегда возвращать допустимый объект, даже если произошла ошибка. Просто объекты, возвращаемые в случае ошибки, «ничего не делают».

Библиография

[PLOPD3] Robert C. Martin, Dirk Riehle, Frank Buschmann, eds. «Pattern Languages of Program Design 3», Addison-Wesley, 1998.

26

Система расчета заработной платы: первая итерация



Все сколько-нибудь прекрасное прекрасно само по себе и в себе завершено, не включая в себя похвалу.

Марк Аврелий (ок. 170 до н. э.)

В этой главе мы опишем первую итерацию разработки простой пакетной системы расчета заработной платы. Приведенные ниже пользовательские истории несколько упрощены. Например, налоги не упомянуты вовсе. Это типично для первой итерации. Она предоставляет лишь малую часть функциональности, необходимой заказчику.

В этой главе мы проведем только предварительный анализ и первое совещание по проектированию, как часто бывает в начале обычной итерации. Заказчик выбрал, какие истории рассматривать на данной итерации, и теперь мы должны придумать, как их реализовать. Подобные

совещания обычно оказываются такими же краткими и поверхностными, как и эта глава. UML-диаграммы, которые вы увидите, – не более чем наброски на доске, сделанные наспех. Настоящее проектирование начнется в следующей главе, когда мы займемся автономными тестами и реализацией.

Краткая спецификация

Ниже приведены некоторые заметки, сделанные во время беседы с заказчиком по поводу историй, отобранных для первой итерации.

- Часть работников работает на условиях почасовой оплаты. Почасовая ставка хранится в одном из полей записи о работнике. Ежедневно такой работник заполняет карточку табельного учета, проставляя дату и количество отработанных часов. Если работник в какой-то день отработал более 8 часов, то дополнительные часы оплачиваются с коэффициентом 1,5. Выплаты производятся каждую пятницу.
- Части работников начисляется твердый оклад. Им зарплата выплачивается в последний рабочий день месяца. Величина месячного оклада хранится в одном из полей записи о работнике.
- Части работников на окладе выплачиваются также комиссионные, рассчитываемые из объема произведенных ими продаж. Они представляют справки, в которых указаны дата и сумма продажи. Комиссионная ставка хранится в одном из полей записи о работнике. Выплаты производятся каждую вторую пятницу.
- Работник может сам выбрать способ платежа. Чек может быть отправлен на указанный работником почтовый адрес, храниться у кассира до востребования, или же сумма может быть переведена на указанный банковский счет.
- Некоторые работники являются членами профсоюза. Для них в записи о работнике хранится ставка еженедельных членских взносов. Величина членских взносов должна быть вычтена из зарплаты. Кроме того, профсоюз может иногда выставлять своим членам счет за оказанные дополнительные услуги. Такие счета подаются еженедельно, и предъявленная к оплате сумма должна вычитаться из очередной зарплаты работника.
- Программа расчета заработной платы запускается каждый рабочий день и начисляет зарплату тем работникам, с которыми надлежит рассчитаться в этот день. Системе сообщается, по какую дату должен быть произведен расчет с работниками, поэтому она рассчитывает платежи по документам, поступившим с даты последнего расчета по указанную дату.

Можно было бы начать со схемы базы данных. Очевидно, что для этой задачи потребуется какая-то реляционная база данных, и на основе требований можно составить отчетливое представление о таблицах

и полях. Было бы несложно спроектировать работоспособную схему и приступить к составлению некоторых запросов. Однако при таком подходе мы построили бы приложение, в центре которого находится база данных.

Базы данных – это деталь реализации! К вопросу о базе данных следует переходить как можно позже. Нет числа приложениям, которые были спроектированы в расчете на конкретные СУБД и в результате оказались неразрывно с ними связаны. Вспомните, что такое абстрагирование: «выделение важного и исключение несущественного». На этой стадии проекта база данных несущественна; это всего лишь способ хранения и доступа к данным, и ничего более.

Анализ по прецедентам

Вместо того чтобы начинать с анализа циркулирующих в системе данных, займемся лучше рассмотрением ее поведения. Ведь именно за реализацию нужного поведения системы нам и платят.

Один из способов сбора и анализа сведений о поведении системы – создание *прецедентов*. В том виде, в каком прецеденты были первоначально описаны Джекобсоном, они очень похожи на пользовательские истории в экстремальном программировании[1], разве что чуть более детализированы. Более тщательная проработка уместна, если данная история была выбрана для реализации на текущей итерации.

В ходе анализа прецедентов мы изучаем пользовательские истории и приемочные тесты, ставя целью выявить действия со стороны пользователей системы (внешние стимулы). Затем мы стараемся понять, как система отвечает на эти действия. Вот, например, истории, которые заказчик выбрал для очередной итерации:

1. Добавить нового работника.
2. Удалить работника.
3. Зарегистрировать карточку табельного учета.
4. Зарегистрировать справку о продажах.
5. Зарегистрировать платежное требование, выставленное профсоюзом.
6. Изменить сведения о работнике (например, почасовую ставку, ставку членских взносов и т. д.).
7. Рассчитать зарплату на сегодняшний день.

Давайте представим эти пользовательские истории в виде проработанных прецедентов. Излишняя детализация ни к чему: нам требуется лишь понять структуру кода для реализации каждой истории.

[1] [Jacobson92]

Добавление работников

Прецедент 1: добавление нового работника

Новый работник добавляется при получении входной записи `AddEmp`, которая содержит имя и адрес работника, а также присвоенный ему табельный номер. Запись может быть представлена в одном из трех форматов:

1. `AddEmp <EmpID> "<name>" "<address>" H <hrly-rate>`
2. `AddEmp <EmpID> "<name>" "<address>" S <mtly-slry>`
3. `AddEmp <EmpID> "<name>" "<address>" C <mtly-slry> <comm-rate>`

В результате создается запись о работнике, в которой заполнены те или иные поля.

Альтернатива 1: ошибка в структуре входной записи

Если входная запись имеет неправильную структуру, то печатается сообщение об ошибке и больше никаких действий не выполняется.

Прецедент 1 наводит на мысль об абстракции. У входной записи `AddEmp` есть три формата, содержащих общие поля `<EmpID>`, `<name>` и `<address>`. Воспользовавшись паттерном Команда, мы можем создать абстрактный класс `AddEmployeeTransaction` с тремя подклассами: `AddHourlyEmployeeTransaction`, `AddSalariedEmployeeTransaction`, `AddCommissionedEmployeeTransaction` (рис. 26.1).

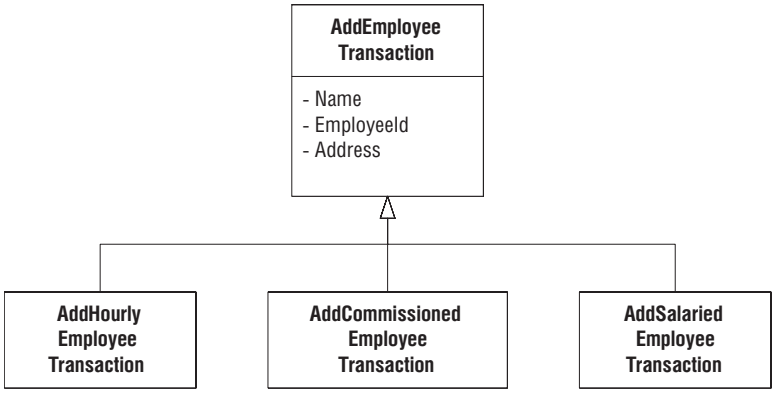


Рис. 26.1. *Иерархия классов, производных от AddEmployeeTransaction*

Эта структура хорошо согласуется с принципом единственной обязанности (SRP), поскольку под каждую задачу отведен свой класс. Можно было бы вместо этого поместить все задачи в один модуль. Да, при та-

ком подходе общее количество классов в системе сократилось бы и, следовательно, система оказалась бы проще, но это означает, что весь код обработки входной записи будет находиться в одном месте, тем самым увеличивая размер модуля и вероятность ошибок.

В прецеденте 1 есть слова «запись о работнике», подразумевающие наличие какой-то базы данных. Из-за предрасположенности к базам данных может возникнуть искушение перейти к проектированию структуры таблиц и записей в реляционной базе, однако ему следует всячески противиться. В действительности в этом прецеденте нас всего лишь просят создать работника. Какова объектная модель работника? Или лучше задать вопрос иначе: что именно создается в результате обработки трех входных записей? На мой взгляд, создаются три разновидности объекта работника. Возможная структура показана на рис. 26.2.

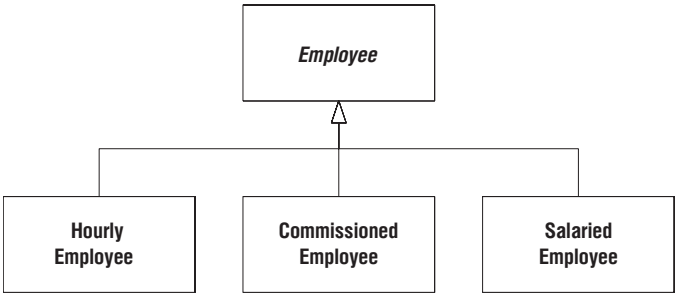


Рис. 26.2. *Возможная иерархия классов, производных от Employee*

Удаление работников

Прецедент 2: удаление работника

Работник удаляется при получении входной записи `DelEmp`, имеющей следующий формат:

```
DelEmp <EmpID>
```

В результате удаляется запись о соответствующем работнике.

Альтернатива 1: недопустимое или неизвестное значение EmpID

Если формат поля `<EmpID>` неправилен или не существует записи о работнике с таким табельным номером, то печатается сообщение об ошибке и больше никаких действий не выполняется.

Если не считать очевидного класса `DeleteEmployeeTransaction`, ни на какие другие мысли этот прецедент меня не наводит. Пойдем дальше.

Регистрация карточки табельного учета

Прецедент 3: регистрация карточки табельного учета

При получении входной записи `TimeCard` система создает карточку табельного учета и ассоциирует ее с записью о соответствующем работнике:

```
TimeCard <empid> <date> <hours>
```

Альтернатива 1: указанному работнику не начисляется почасовая оплата

Система печатает сообщение об ошибке и больше никаких действий не выполняет.

Альтернатива 2: ошибка в структуре входной записи

Система печатает сообщение об ошибке и больше никаких действий не выполняет.

Этот прецедент показывает, что существуют входные записи, применяемые только к работникам определенного вида. Это утверждает нас в мысли о том, что каждый вид следует представлять отдельным классом. В данном случае просматривается также ассоциация между карточками табельного учета и работниками с почасовой оплатой. На рис. 26.3 представлена возможная статическая модель этой ассоциации.

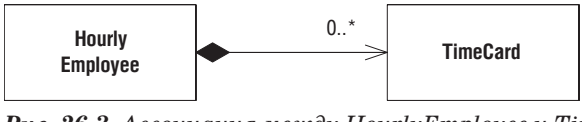


Рис. 26.3. *Ассоциация между HourlyEmployee и TimeCard*

Регистрация справки о продажах

Прецедент 4: регистрация справки о продажах

При получении входной записи `SalesReceipt` система создает новую запись о справке о продажах и ассоциирует ее с записью о соответствующем работнике:

```
SalesReceipt <EmpID> <date> <amount>
```

Альтернатива 1: указанному работнику не начисляются комиссионные

Система печатает сообщение об ошибке и больше никаких действий не выполняет.

Альтернатива 2: ошибка в структуре входной записи

Система печатает сообщение об ошибке и больше никаких действий не выполняет.

Этот прецедент очень похож на прецедент 3 и подразумевает структуру, показанную на рис. 26.4.

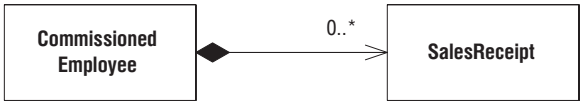


Рис. 26.4. *Работники, получающие комиссионные, и справки о продажах*

Регистрация платежного требования от профсоюза

Прецедент 5: регистрация платежного требования от профсоюза

При получении такой входной записи система создает запись о платежном требовании и ассоциирует ее с записью о соответствующем члене профсоюза:

```
ServiceCharge <memberID> <amount>
```

Альтернатива 1: ошибка в структуре входной записи

Если структура записи некорректна или номер `<memberID>` не принадлежит ни одному члену профсоюза, то печатается сообщение об ошибке.

Этот прецедент показывает, что доступ к информации о членах профсоюза производится не по табельному номеру. В профсоюзе принята своя схема идентификации членов. Поэтому система должна уметь сопоставлять членов профсоюза и работников. Существует много способов реализовать такую ассоциацию, поэтому не будем принимать произвольное решение, а отложим его на потом. Возможно, ограничения, налагаемые другими частями системы, вынудят нас выбрать вполне определенный способ.

Ясно одно. Имеется прямая ассоциация между членами профсоюза и платой за услуги. На рис. 26.5 показана возможная статическая модель такой ассоциации.

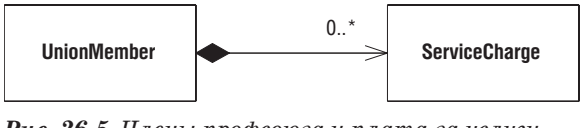


Рис. 26.5. *Члены профсоюза и плата за услуги*

Изменение сведений о работнике

Прецедент 6: изменение сведений о работнике

При получении такой входной записи система изменяет данные в записи о соответствующем работнике. Запись может быть представлена в одном из следующих форматов:

`ChgEmp <EmpID> Name <name>`	Изменить имя работника
`ChgEmp <EmpID> Address <address>`	Изменить адрес работника
`ChgEmp <EmpID> Hourly <hourlyRate>`	Перевести на почасовую оплату
`ChgEmp <EmpID> Salaried <salary>`	Перевести на оклад
`ChgEmp <EmpID> Commissioned <salary> <rate>`	Перевести на комиссионную оплату
`ChgEmp <EmpID> Hold`	Оставлять чек у кассира
`ChgEmp <EmpID> Direct <bank> <account>`	Перевод на банковский счет
`ChgEmp <EmpID> Mail <address>`	Отправлять чек почтой
`ChgEmp <EmpID> Member <memberID> Dues <rate>`	Сделать членом профсоюза
`ChgEmp <EmpID> NoMember`	Исключить из членов профсоюза

Альтернатива 1: ошибки во входной записи

Если структура записи некорректна, или работника с табельным номером `<EmpID>` не существует, или номер `<memberID>` не принадлежит ни одному члену профсоюза, то печатается сообщение об ошибке и больше никаких действий не выполняется.

Этот прецедент многое проясняет. В нем перечислены все характеристики работника, которые можно изменять. Возможность перевести работника с почасовой оплаты на оклад означает, что диаграмма на рис. 26.2 откровенно ошибочна. Пожалуй, для расчета зарплаты было бы лучше применить паттерн Стратегия. В классе `Employee` могла бы храниться ссылка на класс стратегии `PaymentClassification`, как показано на рис. 26.6. Это плюс, потому что мы сможем изменять объект `PaymentClassification`, не затрагивая других частей объекта `Employee`. Чтобы перевести работника с почасовой оплаты на оклад, достаточно заменить ссылку на объект `HourlyClassification` в объекте `Employee` ссылкой на `SalariedClassification`.

Объекты PaymentClassification бывают трех видов. В объекте HourlyClassification хранится почасовая ставка и список объектов TimeCard. В объекте SalariedClassification хранится величина месячного оклада, а в объекте CommissionedClassification — месячный оклад, ставка комиссионного вознаграждения и список объектов SalesReceipt.

Способ платежа также должен быть изменяемым. На рис. 26.6 эта идея реализована с использованием паттерна Стратегия и трех производных от PaymentMethod классов. Если объект Employee содержит ссылку на объект MailMethod, то работнику будет высылаться чек по почте на адрес, хранящийся в объекте MailMethod. Если же Employee ссылается на объект DirectMethod, то причитающаяся работнику сумма будет перечислена в банк, указанный в объекте DirectMethod. Ну а если Employee ссылается на HoldMethod, то чеки будут храниться у кассира до востребования.

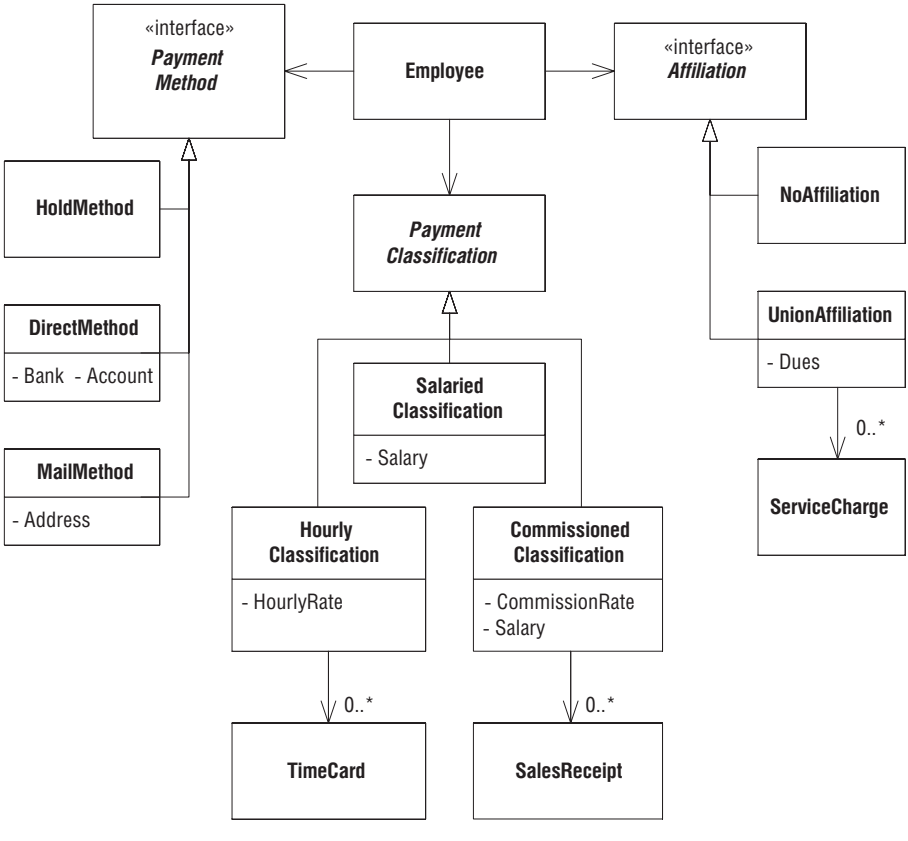


Рис. 26.6. *Пересмотренная диаграмма классов для системы расчета зарплаты: принципиальная модель*

Наконец, на рис. 26.6 показано применение паттерна Null-объект для представления членства в профсоюзе. В каждом объекте `Employee` хранится ссылка на объект `Affiliation` одного из двух видов. Если это объект типа `NoAffiliation`, то из суммы, начисленной работнику, не делаются вычеты в пользу каких-то иных организаций. Если же `Employee` содержит ссылку на объект типа `UnionAffiliation`, то работник должен оплачивать членские взносы, а также счета за услуги, перечисленные в объекте `UnionAffiliation`.

Применение паттернов позволило спроектировать систему, согласованную с принципом открытости/закрытости (OCP). Класс `Employee` закрыт от изменений в способе платежа, тарификации и членства в профсоюзе. Новые способы платежа, тарификации и членства в различных организациях можно добавлять, не изменяя класса `Employee`.

На рис. 26.6 изображено то, что станет нашей *принципиальной моделью*, или архитектурой. Это основа всего, что делает система расчета заработной платы. В приложении появится еще много других классов и проектных решений, но все они будут вторичны по отношению к этой фундаментальной структуре. Разумеется, сама она не высечена в камне. Как и все остальное, мы будем ее модифицировать.

Расчетный день

Прецедент 7: расчет заработной платы на сегодня

При получении такой входной записи система находит всех работников, которым следует начислить зарплату на указанную дату. Затем система рассчитывает для них величину зарплаты и производит выплату в соответствии с указанным способом платежа. Распечатывается контрольный протокол, в котором отражаются действия, произведенные для каждого работника:

```
Payday <date>
```

Хотя понять намерение этого прецедента легко, определить, как он повлияет на статическую структуру, показанную на рис. 26.6, уже не так просто. Нам предстоит ответить на несколько вопросов.

Во-первых, откуда объект `Employee` знает, как вычислить свою зарплату? Понятно, что для работника с почасовой оплатой система должна сложить величины в карточках табельного учета и умножить на почасовую ставку. Аналогично для работников с комиссионной оплатой суммируются величины продаж, указанные в справках, результат умножается на ставку комиссионных и прибавляется базовый оклад. Но где все это делается? Идеальным местом представляются классы, про-

изводные от PaymentClassification. В этих объектах хранятся записи, необходимые для расчета зарплаты, так почему бы не включить в них и способы расчета? На рис. 26.7 показана диаграмма кооперации, описывающая, как эта идея могла бы работать.

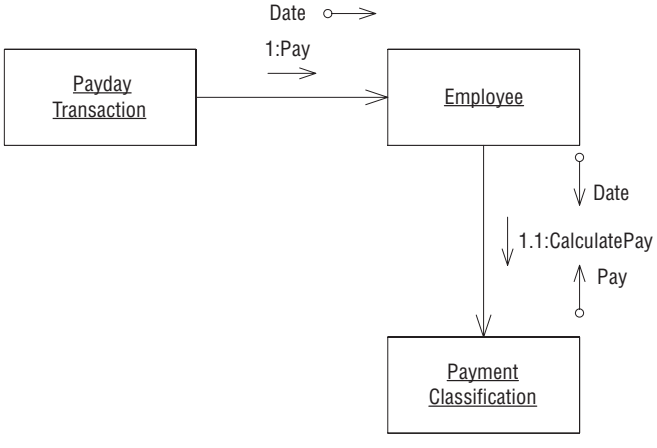


Рис. 26.7. *Вычисление зарплаты работника*

Когда объект Employee просят вычислить зарплату, он переадресует запрос своему объекту PaymentClassification. Используемый алгоритм зависит от подкласса PaymentClassification, на который указывает ссылка в объекте Employee. На рис. 26.8–26.10 показаны все три возможных сценария.

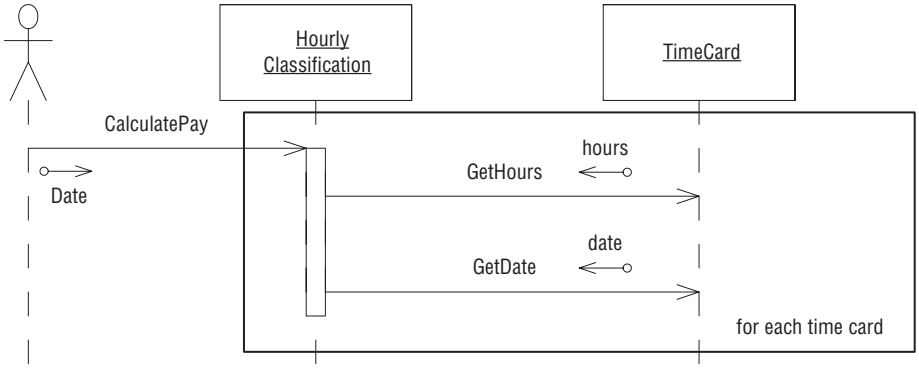


Рис. 26.8. *Вычисление зарплаты работника с почасовой оплатой*

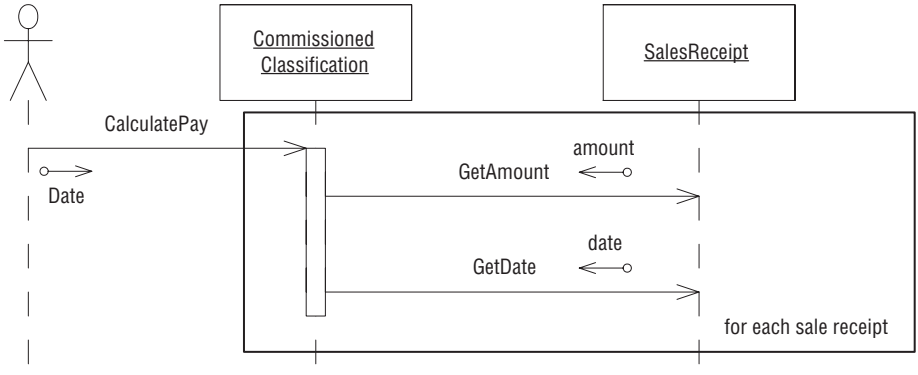


Рис. 26.9. *Вычисление зарплаты работника с комиссионной оплатой*

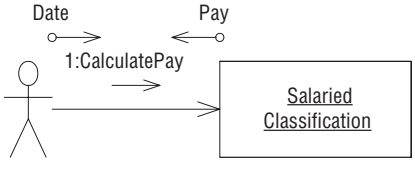
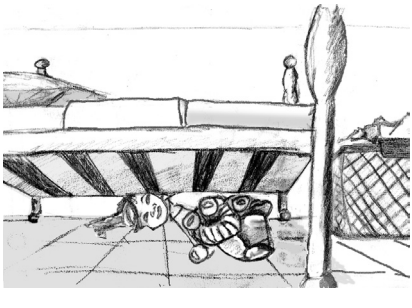


Рис. 26.10. *Вычисление зарплаты работника с твердым окладом*

Осмысление: поиск основополагающих абстракций



Пока что мы выяснили, что даже простой анализ прецедентов может дать массу полезной информации и помочь разобраться в связях системы. Рисунки 26.6–26.10 появились в результате обдумывания прецедентов, то есть размышлений о поведении системы.

Для эффективного применения принципа OCP мы должны активно искать и находить абстракции, которые лежат в основе приложения. Часто эти абстракции не сформулированы явно и даже намеком не упоминаются ни в требованиях, ни в прецедентах. И те и другие могут быть загромождены деталями, мешающими осознать общность основополагающих абстракций.

Тарификация работника

Давайте еще раз взглянем на требования. Мы встречаем такие фразы: «Часть работников работает на условиях почасовой оплаты», «Части работников начисляется твердый оклад», «Части работников... выплачиваются комиссионные». Из этих подсказок можно извлечь следующее обобщение: труд всех работников оплачивается, но по разным схемам. Абстракция здесь состоит в том, что *труд всех работников оплачивается*. Наша модель иерархии `PaymentClassification` на рис. 26.7–26.10 прекрасно выражает эту абстракцию. Следовательно, она уже была найдена среди пользовательских историй в ходе простого анализа прецедентов.

График выплат

В поисках других абстракций мы натыкаемся на фразы «Выплаты производятся каждую пятницу», «Зарплата выплачивается в последний рабочий день месяца» и «Выплаты производятся каждую вторую пятницу». Это подводит нас еще к одному обобщению: *зарплата выплачивается всем работникам по определенному графику*. В данном случае абстракцией является понятие *«график»*. Должна быть возможность спросить у объекта `Employee`, настал ли день выплаты. В прецедентах об этом сказано вскользь. Требования ассоциируют график выплат работнику с тарификацией. Точнее, работникам с почасовой оплатой выплаты производятся еженедельно, работникам на окладе – раз в месяц, а работникам с комиссионной оплатой – раз в две недели. Но так ли важна эта ассоциация? Не может ли политика выплат в один прекрасный день измениться так, что работник сам сможет выбирать удобный для себя график, или так, что для работников из разных отделов или подразделений будут действовать различные графики? А может ли политика выплат стать независимой от тарификации? Все это кажется вполне вероятным.

Если, как следует из требований, мы делегируем решение вопроса о графике выплат классу `PaymentClassification`, то он может оказаться не защищенным от изменения графика. При изменении тарификации нам пришлось бы тестировать также график выплат, а при изменении графика – тарификацию. Нарушенными оказались бы принципы OCP и SRP.

Ассоциация между графиком и тарификацией могла бы стать причиной ошибок, в результате которых изменение тарификации привело бы к неправильному графику выплат для некоторых работников. Такие ошибки привычны программистам, но вселяют страх в сердца руководителей и пользователей. Они опасаются, и не без основания, что раз изменение тарификации может привести к нарушению графика выплат, то *любое* изменение в *любом* месте может вызвать проблемы в *любой* не связанной части системы. Они опасаются непредсказуемости послед-

ствий изменения. А если нельзя предсказать последствия, то теряется всякая уверенность, и программа в мыслях руководителей и пользователей переходит в разряд «опасных и нестабильных».

Несмотря на принципиальную важность абстракции графика, наш анализ прецедентов не обнаружил, что она существует. Для ее выявления потребовалось внимательно рассмотреть требования и заглянуть в закоулки сознания пользователей. Чрезмерное доверие инструментам и процедурам вкупе с недостаточным доверием к здравому смыслу и опыту – прямой путь к провалу.

На рис. 26.11 и 26.12 показаны статическая и динамическая модели абстракции графика. Как видите, мы еще раз воспользовались паттерном Стратегия. Класс `Employee` содержит абстрактный класс `PaymentSchedule`. Три подкласса `PaymentSchedule` соответствуют трем известным графикам выплат.

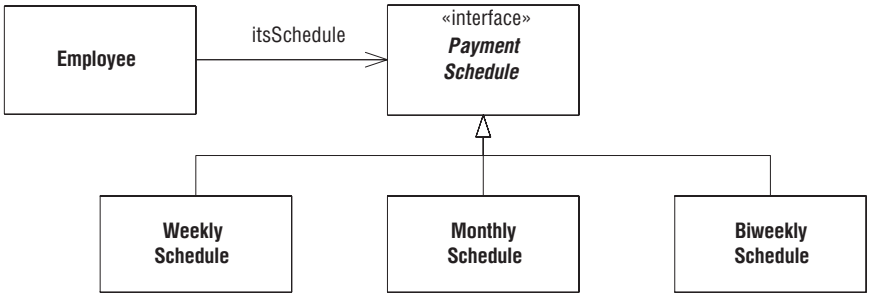


Рис. 26.11. *Статическая модель абстракции Schedule*

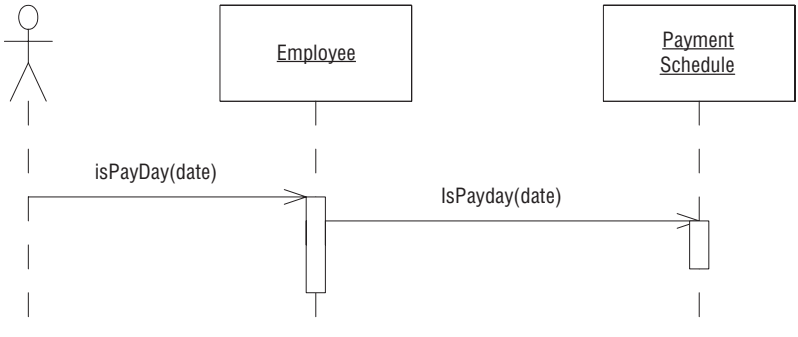


Рис. 26.12. *Динамическая модель абстракции Schedule*

Способы платежа

Еще одно обобщение, следующее из анализа требований, – тот факт, что *все работники получают зарплату определенным способом*. В этом

случае абстракцией будет класс PaymentMethod. Интересно, однако, что эта абстракция уже присутствует на рис. 26.6.

Принадлежность к другим организациям

Из требований вытекает, что работники могут быть членами профсоюза, однако помимо профсоюзов могут существовать и другие организации, претендующие на часть зарплаты работника. Работник может попросить автоматически перечислять определенные суммы в указанные благотворительные фонды или вычитать из зарплаты взносы в профессиональные ассоциации. Поэтому мы можем сформулировать такое обобщение: «*Работник может быть членом нескольких организаций, которым следует автоматически перечислять часть зарплаты*».

Соответствующая абстракция выражается классом Affiliation, который показан на рис. 26.6. Однако из рисунка не следует, что объект Employee может содержать более одного объекта Affiliation, зато включен класс NoAffiliation. Такой дизайн не полностью соответствует предполагаемой абстракции. На рис. 26.13 и 26.14 показаны статическая и динамическая модели, представляющие абстракцию Affiliation.

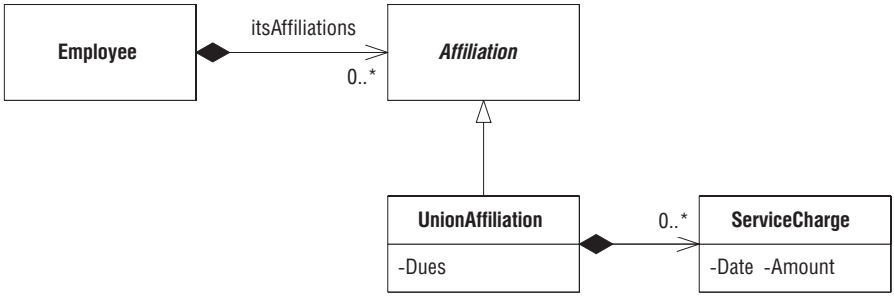


Рис. 26.13. *Статическая структура абстракции Affiliation*

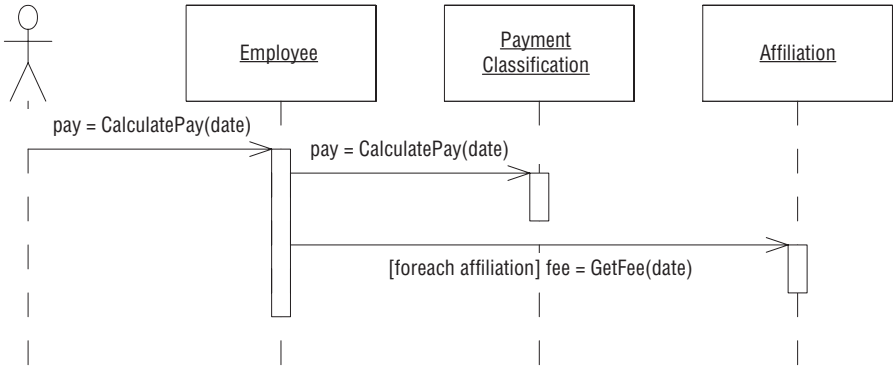


Рис. 26.14. *Динамическая структура абстракции Affiliation*

Наличие списка объектов `Affiliation` делает излишним применение паттерна `Null-объект` к работникам, не входящим ни в одну организацию. Просто список организаций для таких работников будет пуст.

Заключение

Итак, начало дизайну положено, и начало неплохое. Мы преобразовали пользовательские истории в прецеденты и проанализировали их на предмет поиска абстракций. В результате система приняла определенные *очертания*. Начинает вырисовываться архитектура. Однако сразу отметим, что эта архитектура получена в результате рассмотрения лишь самых первых пользовательских историй. Мы еще не подвергли скрупулезному анализу все требования. И не требовали идеальной точности от каждой истории и прецедента. Также мы не стали доводить дизайн системы до такого состояния, где для каждой мыслимой мелочи имеются диаграммы классов и последовательности.

Размышлять о дизайне важно. Но *критически* важно продвигаться вперед мелкими шажками. Сделать слишком много хуже, чем слишком мало. В этой главе мы сделали как раз в меру. Она оставляет ощущение незаконченности, но сделанного достаточно для понимания и дальнейшего развития.

Библиография

[Jacobson92] Ivar Jacobson «Object-Oriented Software Engineering: A Use Case Driven Approach», Addison-Wesley, 1992.

27

Система расчета заработной платы: реализация



Уже давно мы согласились, что следует писать код, который поддерживает и верифицирует разрабатываемый дизайн. Я создавал код очень небольшими шагами, но привожу его в тех местах текста, где это наиболее уместно. Однако, глядя на завершенные фрагменты кода, не думайте, что я его сразу в таком виде и написал. На самом деле один фрагмент от другого отделяли десятки правок, компиляций и тестов и всякий раз вносились крохотные эволюционные изменения.

Вы увидите много UML-диаграмм. Рассматривайте их как наброски на доске, с помощью которых я демонстрирую вам, моему партнеру по парному программированию, что собираюсь сделать. UML будет для нас удобным средством выражения мыслей.

Операции

Начнем с осмысления входных записей, или операций, соответствующих прецедентам. На рис. 27.1 показано, что операции представлены интерфейсом `Transaction`, в котором объявлен метод `Execute()`. Разумеется, это ни что иное, как паттерн Команда. Код интерфейса `Transaction` приведен в листинге 27.1.

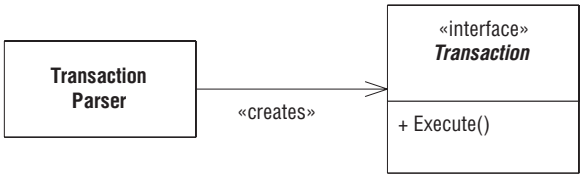


***Рис. 27.1.** Интерфейс Transaction*

Листинг 27.1. Transaction.cs

```
namespace Payroll
{
  public interface Transaction
  {
    void Execute();
  }
}
```

Добавление работников

На рис. 27.2 показана потенциальная структура операций добавления работников. Именно в них график выплат работнику ассоциируется с тарификацией. Это разумно, потому что операции – лишь вспомогательные приспособления, не являющиеся частью принципиальной модели. Например, принципиальная модель ничего не знает о том, что работникам с почасовой оплатой выдача денег производится каждую неделю. Ассоциация между тарификацией и графиком выплат находится на периферии системы, ее можно изменить в любой момент. Например, ничто не мешает добавить операцию, позволяющую изменить график выплат для работника.

Это решение вполне согласуется с принципами OCP и SRP. Определять ассоциацию между тарификацией и графиком выплат – обязанность операций, а не принципиальной модели. Причем эту ассоциацию можно изменить, не затрагивая принципиальную модель.

Заметьте также, по умолчанию подразумевается, что чек остается у кассира. Если работник предпочитает другой способ платежа, то необходимо будет выполнить операцию `ChgEmp`.

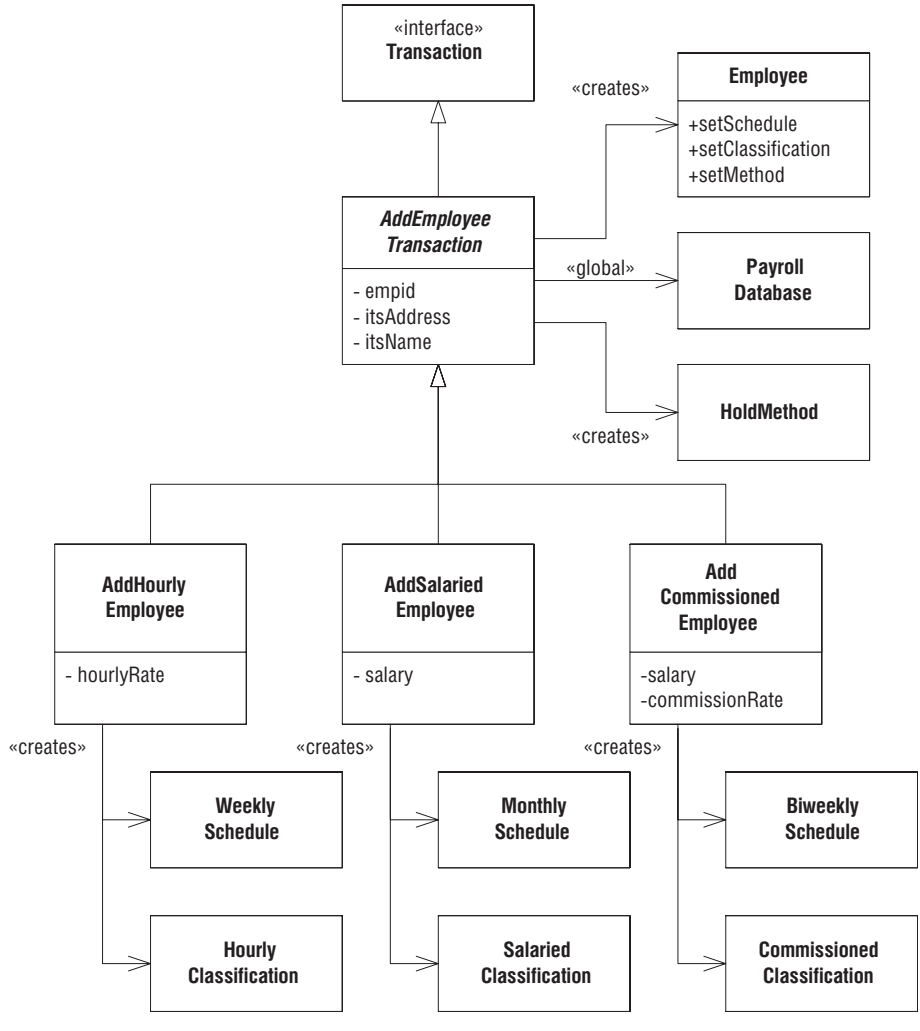


Рис. 27.2. *Статическая модель AddEmployeeTransaction*

Как обычно, начинаем с тестов. Тесты в листинге 27.2 доказывают, что класс `AddSalariedTransaction` работает правильно. А следующий за ними код написан так, чтобы эти тесты прошли успешно.

Листинг 27.2. PayrollTest.TestAddSalariedEmployee

```
[Test]
public void TestAddSalariedEmployee()
{
  int empId = 1;
  AddSalariedEmployee t =
    new AddSalariedEmployee(empId, "Bob", "Home", 1000.00);
  t.Execute();
```

```
    Employee e = PayrollDatabase.GetEmployee(empId);
    Assert.AreEqual("Bob", e.Name);

    PaymentClassification pc = e.Classification;
    Assert.IsTrue(pc is SalariedClassification);
    SalariedClassification sc = pc as SalariedClassification;
    Assert.AreEqual(1000.00, sc.Salary, .001);
    PaymentSchedule ps = e.Schedule;
    Assert.IsTrue(ps is MonthlySchedule);

    PaymentMethod pm = e.Method;
    Assert.IsTrue(pm is HoldMethod);
  }
```

База данных о работниках. Класс `AddEmployeeTransaction` пользуется классом `PayrollDatabase`. Пока в нем хранятся все существующие объекты `Employee`, помещенные в хэш-таблицу `Hashtable` с ключом `empID`. Кроме того, здесь же имеется таблица `Hashtable`, отображающая `memberID` на `empID`. О том, как хранить эти данные долговременно, мы подумаем позже. Структура этого класса представлена на рис. 27.3. `PayrollDatabase` — это пример паттерна Фасад.

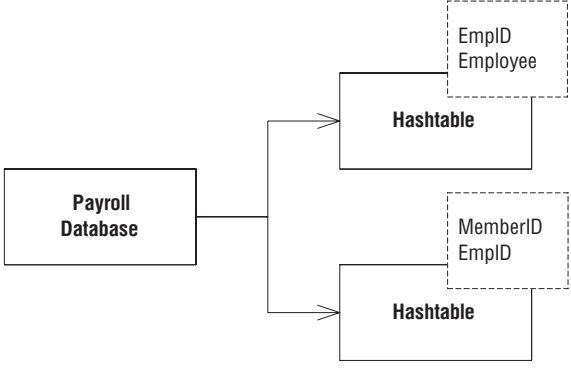


***Рис. 27.3.** Статическая структура класса PayrollDatabase*

В листинге 27.3 показана рудиментарная реализация `PayrollDatabase`. У нее лишь одна цель — помочь при написании и выполнении начальных тестов. В ней еще нет даже хэш-таблицы, отображающей идентификаторы членов профсоюза на табельные номера.

Листинг 27.3. PayrollDatabase.cs

```
using System.Collections;

namespace Payroll
{
  public class PayrollDatabase
  {
```

```
    private static Hashtable employees = new Hashtable();

    public static void AddEmployee(int id, Employee employee)
    {
      employees[id] = employee;
    }

    public static Employee GetEmployee(int id)
    {
      return employees[id] as Employee;
    }
  }
}
```

Вообще говоря, я считаю вопрос о выборе базы данных деталью реализации. Решения по таким вопросам следует откладывать как можно дольше. Будет ли эта конкретная база данных реализована в виде реляционной СУБД, плоского файла или объектно-ориентированной СУБД, в данный момент неважно. Сейчас я просто хочу спроектировать API, который будет предоставлять услуги базы данных остальным частям приложения. А подходящую реализацию я уж как-нибудь найду.

Откладывание вопроса о базе данных – не слишком распространенная, но очень полезная практика. Обычно с его решением можно подождать до тех пор, пока не будет собрано гораздо больше информации о разрабатываемой системе. И таким образом, мы не столкнемся с проблемой перемещения в базу чрезмерно большой части инфраструктуры. Лучше оставить в базе данных только то, что необходимо для удовлетворения текущих требований к приложению.

Использование паттерна Шаблонный метод для добавления работников. На рис. 27.4 показана динамическая модель добавления работника. Обратите внимание, что объект `AddEmployeeTransaction` посылает сообщения *самому себе*, чтобы получить нужные объекты `PaymentClassification` и `PaymentSchedule`. Соответствующие методы реализованы в подклассах `AddEmployeeTransaction`. Это и есть применение паттерна Шаблонный метод.

В листинге 27.4 приведена реализация паттерна Шаблонный метод в классе `AddEmployeeTransaction`. Из метода `Execute()` вызываются два виртуальных метода, реализованных в подклассах: `MakeSchedule()` и `MakeClassification()`, возвращающие соответственно объекты типа `PaymentSchedule` и `PaymentClassification`, которые необходимы объекту `Employee`. Затем метод `Execute()` связывает эти объекты с `Employee` и сохраняет `Employee` в `PayrollDatabase`.

Тут стоит отметить два момента. Во-первых, когда паттерн Шаблонный метод применяется, как в данном случае, с единственной целью создания объектов, он называется Фабричным методом (Factory Method). Во-вторых, методы создания в паттерне Фабричный метод принято назы-

вать MakeXXX(). И то и другое я осознал уже в процессе написания кода, поэтому имена методов на диаграмме и в коде различаются.

Надо ли было вернуться и исправить диаграмму? Я не видел в этом необходимости. Я не собираюсь оставлять эту диаграмму в качестве справочного материала для других. В реальном проекте она, скорее всего, была бы нарисована на доске, чтобы через минуту быть стертой.

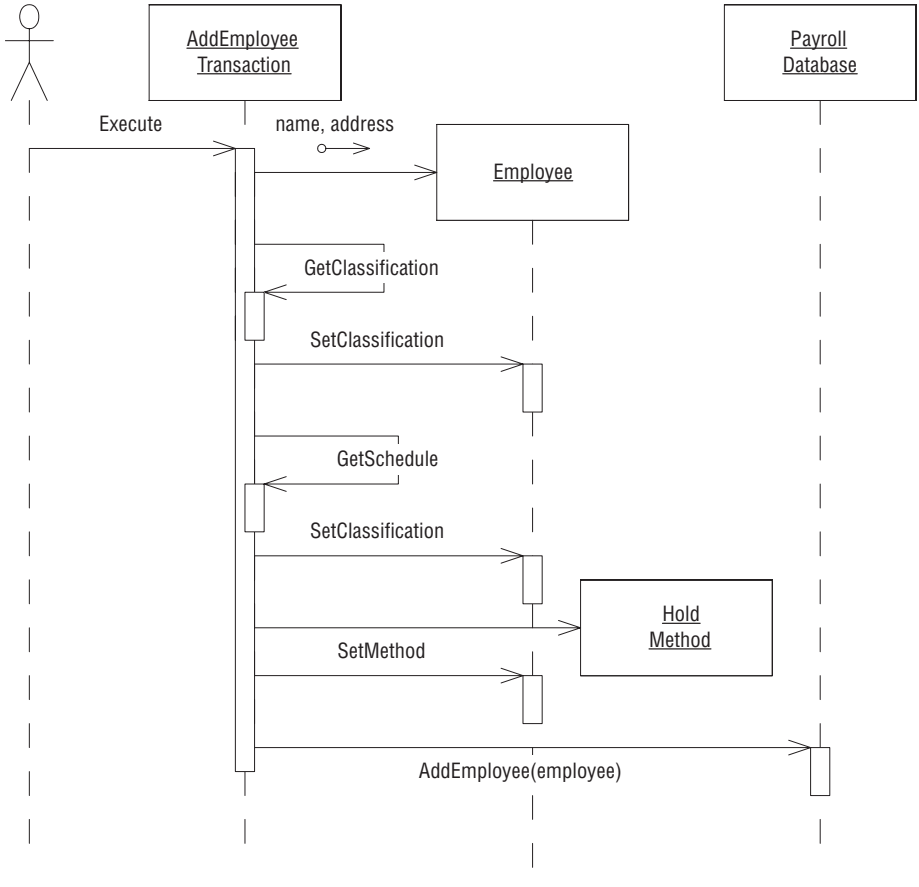


Рис. 27.4. *Динамическая модель добавления работника*

Листинг 27.4. AddEmployeeTransaction.cs

```
namespace Payroll
{
  public abstract class AddEmployeeTransaction : Transaction
  {
    private readonly int empid;
    private readonly string name;
    private readonly string address;
```

```
    public AddEmployeeTransaction(int empid,
      string name, string address)
    {
      this.empid = empid;
      this.name = name;
      this.address = address;
    }

    protected abstract
      PaymentClassification MakeClassification();
    protected abstract
      PaymentSchedule MakeSchedule();

    public void Execute()
    {
      PaymentClassification pc = MakeClassification();
      PaymentSchedule ps = MakeSchedule();
      PaymentMethod pm = new HoldMethod();
      Employee e = new Employee(empid, name, address);
      e.Classification = pc;
      e.Schedule = ps;
      e.Method = pm;
      PayrollDatabase.AddEmployee(empid, e);
    }
  }
}
```

В листинге 27.5 показана реализация класса `AddSalariedEmployee`**. Он является производным от** `AddEmployeeTransaction` **и реализует методы** `MakeSchedule()` **и** `MakeClassification()` **таким образом, что они возвращают** `AddEmployeeTransaction.Execute()` **объекты подходящего типа.**

Листинг 27.5. AddSalariedEmployee.cs

```
namespace Payroll
{
  public class AddSalariedEmployee : AddEmployeeTransaction
  {
    private readonly double salary;
    public AddSalariedEmployee(int id, string name,
      string address, double salary)
      : base(id, name, address)
    {
      this.salary = salary;
    }

    protected override
      PaymentClassification MakeClassification()
    {
      return new SalariedClassification(salary);
    }
```

```
        protected override PaymentSchedule MakeSchedule()
        {
          return new MonthlySchedule();
        }
      }
    }
```

Реализацию классов `AddHourlyEmployee` и `AddCommissionedEmployee` оставляю вам в качестве упражнения. Не забудьте сначала написать тесты.

Удаление работников

На рис. 27.5 и 27.6 представлены статическая и динамическая модели операции удаления работника. В листинге 27.6 показан тест для этого случая, а в листинге 27.7 – реализация класса `DeleteEmployeeTransaction`. Это типичный пример применения паттерна Команда. Конструктор сохраняет данные, которыми впоследствии будет оперировать метод `Execute()`.

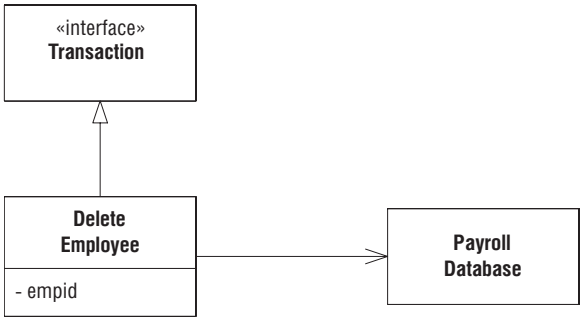


Рис. 27.5. *Статическая модель операции DeleteEmployee*

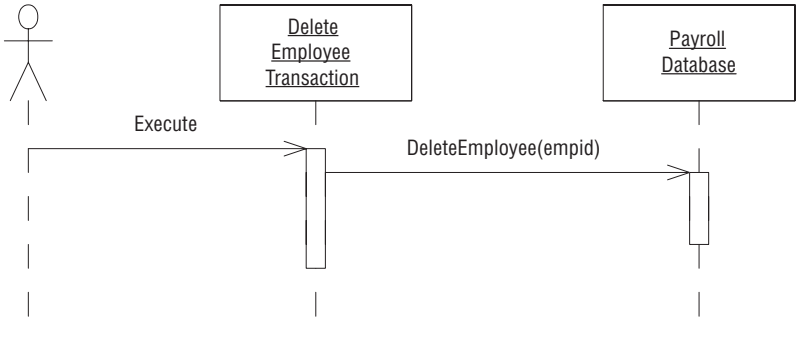


Рис. 27.6. *Динамическая модель операции DeleteEmployee*

Листинг 27.6. PayrollTest.DeleteEmployee

```
[Test]
public void DeleteEmployee()
{
  int empId = 4;
  AddCommissionedEmployee t =
    new AddCommissionedEmployee(
      empId, "Bill", "Home", 2500, 3.2);
  t.Execute();

  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);
  DeleteEmployeeTransaction dt =
    new DeleteEmployeeTransaction(empId);
  dt.Execute();

  e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNull(e);
}
```

Листинг 27.7. DeleteEmployeeTransaction.cs

```
namespace Payroll
{
  public class DeleteEmployeeTransaction : Transaction
  {
    private readonly int id;

    public DeleteEmployeeTransaction(int id)
    {
      this.id = id;
    }

    public void Execute()
    {
      PayrollDatabase.DeleteEmployee(id);
    }
  }
}
```

Вы уже, наверное, заметили, что класс `PayrollDatabase` предоставляет статический доступ к своим полям. По существу, `PayrollDatabase.employees` – глобальная переменная. Но уже на протяжении десятилетий учебники и преподаватели предостерегают нас от использования глобальных переменных – и не без причины. Впрочем, ничего дурного или вредного в глобальных переменных как таковых нет. И в данной ситуации такая переменная – идеальный выбор. Существует один и только один экземпляр класса `PayrollDatabase` со всеми его методами и переменными, и он должен быть известен повсеместно.

Возможно, вам кажется, что лучше было бы применить паттерн Одиночка или Моносостояние. Да, это решило бы задачу. Но дело в том, что они и сами используют глобальные переменные. По определению, Одиночка и Моносостояние являются глобальными сущностями. Мне кажется, что в данном случае эти паттерны лишь внесли бы ненужную сложность. Проще оставить базу данных глобальным объектом.

Карточки табельного учета, справки о продажах и плата за услуги

На рис. 27.7 показана статическая структура операции, в которой регистрируются карточки табельного учета, а на рис. 27.8 – ее динамическая модель. Основная идея в том, что эта операция извлекает объект `Employee` из базы `PayrollDatabase`, запрашивает у него объект `PaymentClassification`, после чего создает объект `TimeCard` и добавляет его в `PaymentClassification`.

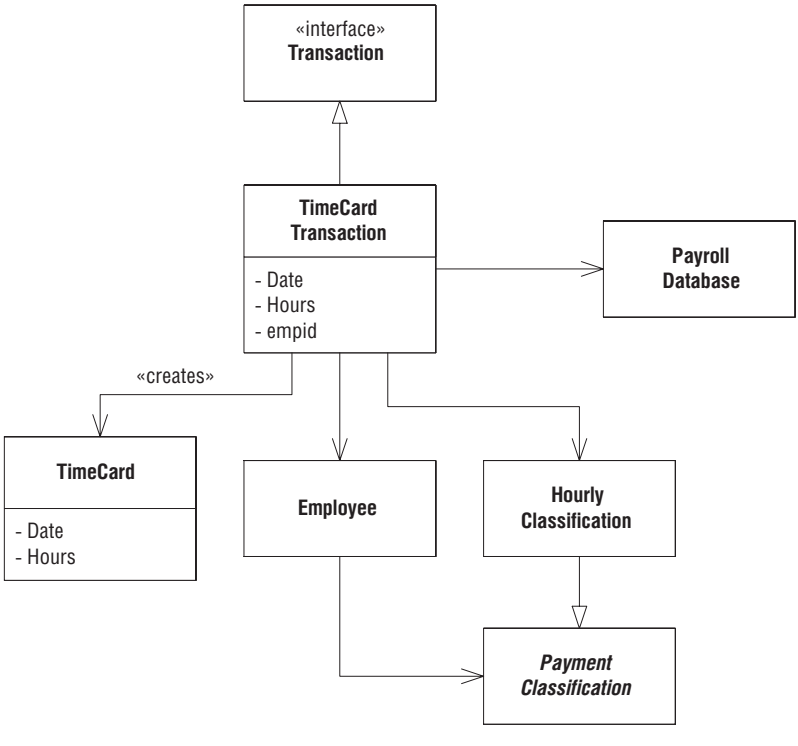


Рис. 27.7. *Статическая структура класса TimeCardTransaction*

Отметим, что мы не можем добавлять объекты `TimeCard` в объект `PaymentClassification` общего вида, их допустимо добавлять только в объект `HourlyClassification`. Отсюда следует, что сначала необходимо привести

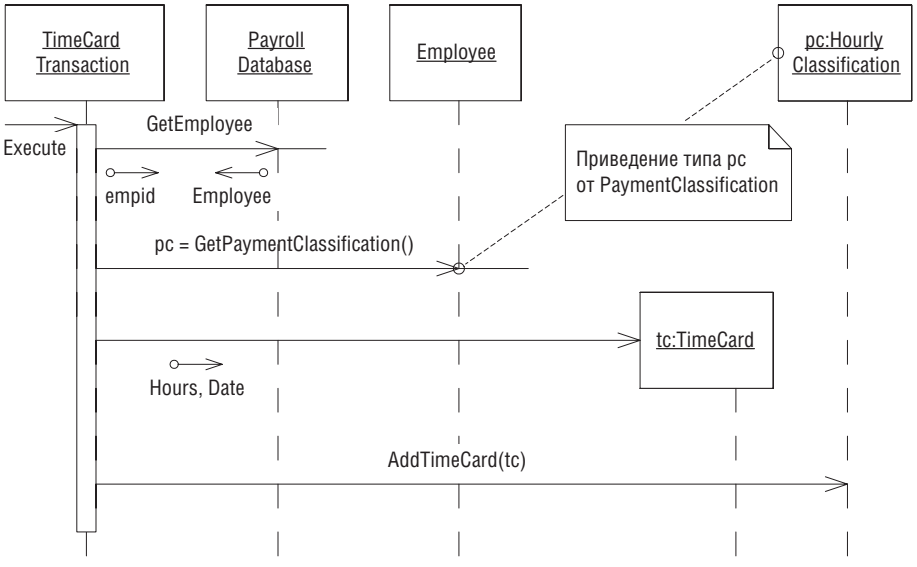


Рис. 27.8. *Динамическая модель регистрации TimeCard*

объект `PaymentClassification`, полученный от `Employee`, к типу `HourlyClassification`. Самое время воспользоваться оператором `as`, имеющимся в языке C# (см. листинг 27.10).

В листинге 27.8 показан тест, который проверяет, что карточки табельного учета можно добавлять для работников с почасовой оплатой. Мы просто создаем такого работника и добавляем его в базу данных. Затем в тесте создается объект `TimeCardTransaction`, вызывается его метод `Execute()` и проверяется, что объект `HourlyClassification`, принадлежащий данному работнику, действительно содержит помещенную в него карточку.

Листинг 27.8. PayrollTest.TestTimeCardTransaction

```
[Test]
public void TestTimeCardTransaction()
{
  int empId = 5;
  AddHourlyEmployee t =
    new AddHourlyEmployee(empId, "Bill", "Home", 15.25);
  t.Execute();
  TimeCardTransaction tct =
    new TimeCardTransaction(
      new DateTime(2005, 7, 31), 8.0, empId);
  tct.Execute();

  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);
```

```
    PaymentClassification pc = e.Classification;
    Assert.IsTrue(pc is HourlyClassification);
    HourlyClassification hc = pc as HourlyClassification;

    TimeCard tc = hc.GetTimeCard(new DateTime(2005, 7, 31));
    Assert.IsNotNull(tc);
    Assert.AreEqual(8.0, tc.Hours);
  }
```

В листинге 27.9 приведена реализация класса `TimeCard`. Пока что он ничего не делает, а только содержит данные.

Листинг 27.9. TimeCard.cs

```
using System;

namespace Payroll
{
  public class TimeCard
  {
    private readonly DateTime date;
    private readonly double hours;

    public TimeCard(DateTime date, double hours)
    {
      this.date = date;
      this.hours = hours;
    }

    public double Hours
    {
      get { return hours; }
    }

    public DateTime Date
    {
      get { return date; }
    }
  }
}
```

В листинге 27.10 приведена реализация класса `TimeCardTransaction`. Обратите внимание на использование исключений `InvalidOperationExceptions`. Если говорить о длительной перспективе, то такой подход не особенно хорош, но на данной стадии разработки это нормально. Позже, когда мы поймем, какие могут быть исключения, можно будет вернуться и создать подходящие классы исключений.

Листинг 27.10. TimeCardTransaction.cs

```
using System;

namespace Payroll
```

```
{
  public class TimeCardTransaction : Transaction
  {
    private readonly DateTime date;
    private readonly double hours;
    private readonly int empId;

    public TimeCardTransaction(
      DateTime date, double hours, int empId)
    {
      this.date = date;
      this.hours = hours;
      this.empId = empId;
    }

    public void Execute()
    {
      Employee e = PayrollDatabase.GetEmployee(empId);
      if (e != null)
      {
        HourlyClassification hc =
          e.Classification as HourlyClassification;
        if (hc != null)
          hc.AddTimeCard(new TimeCard(date, hours));
        else
          throw new InvalidOperationException(
            "Попытка добавить карточку табельного учета " +
            "для работника не на почасовой оплате");
      }
      else
        throw new InvalidOperationException(
          "Работник не найден.");
    }
  }
}
```

На рис. 27.9 и 27.10 показан аналогичный дизайн для операции, которая регистрирует справки о продажах для работника с комиссионной оплатой. Реализацию класса оставляю вам в качестве упражнения.

На рис. 27.11 и 27.12 показан дизайн операции, которая регистрирует счета за услуги, выставленные членам профсоюза. Здесь обнаруживается несоответствие между моделью операции и ранее созданной принципиальной моделью. До сих пор мы считали, что объект `Employee` может принадлежать нескольким организациям, однако в модели этой операции предполагается, что любое членство – это членство в профсоюзе. Иными словами, предложенная модель не дает возможности указать конкретный тип членства. Считается, что раз мы регистрируем счет за услуги, то работник обязательно является членом профсоюза.

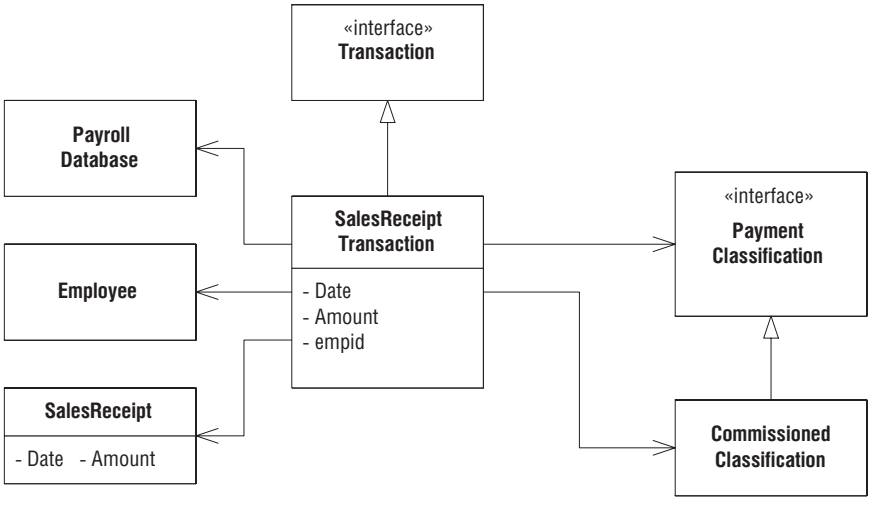


Рис. 27.9. *Статическая модель операции SalesReceiptTransaction*

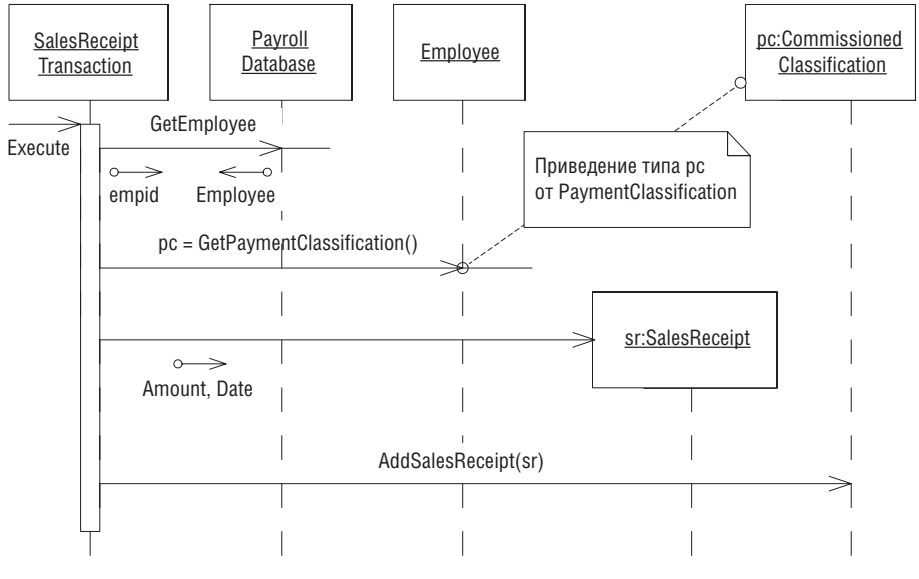


Рис. 27.10. *Динамическая модель операции SalesReceiptTransaction*

В динамической модели эта проблема разрешается путем поиска в наборе объектов `Affiliation`, хранящемся в объекте `Employee`, объекта типа `UnionAffiliation`. И в него мы и добавляем объект `ServiceCharge`.

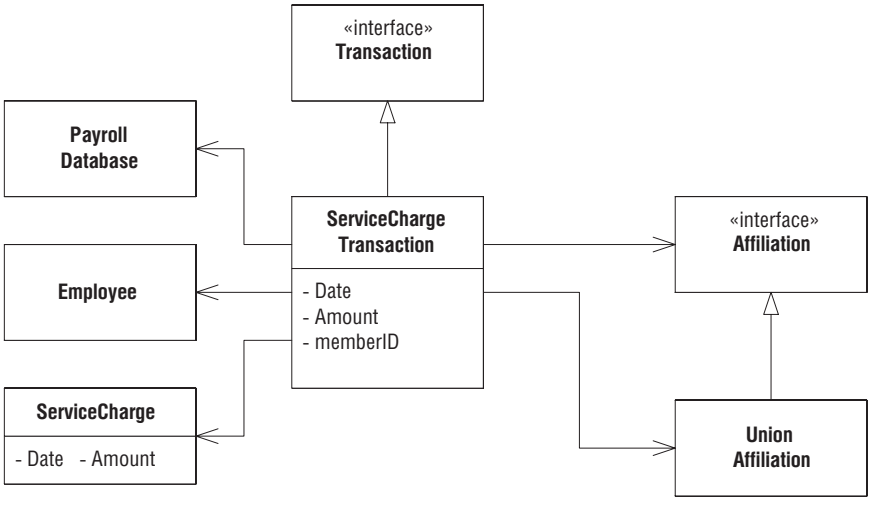


Рис. 27.11. *Статическая модель операции ServiceChargeTransaction*

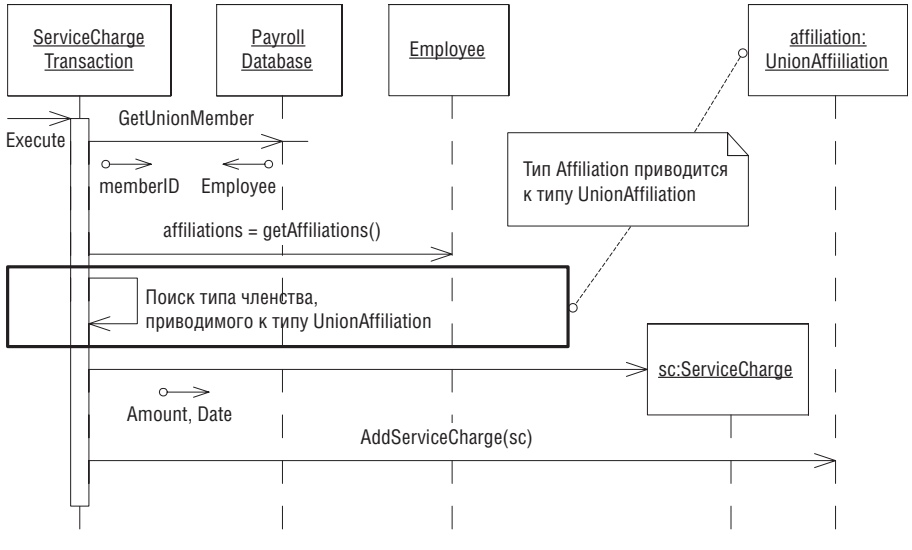


Рис. 27.12. *Динамическая модель операции ServiceChargeTransaction*

В листинге 27.11 показан тест для класса `ServiceChargeTransaction`. В нем мы создаем работника с почасовой оплатой, добавляем в него объект `UnionAffiliation`, проверяем, что соответствующий идентификатор члена профсоюза зарегистрирован в `PayrollDatabase`, создаем объект `ServiceChargeTransaction` и вызываем его метод `Execute()`, после чего проверяем, что объект `ServiceCharge` действительно добавлен в объект `UnionAffiliation` для данного `Employee`.

Листинг 27.11. PayrollTest.AddServiceCharge

```
[Test]
public void AddServiceCharge()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Bill", "Home", 15.25);
  t.Execute();
  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);
  UnionAffiliation af = new UnionAffiliation();
  e.Affiliation = af;
  int memberId = 86; // Maxwell Smart
  PayrollDatabase.AddUnionMember(memberId, e);
  ServiceChargeTransaction sct =
    new ServiceChargeTransaction(
      memberId, new DateTime(2005, 8, 8), 12.95);
  sct.Execute();
  ServiceCharge sc =
    af.GetServiceCharge(new DateTime(2005, 8, 8));
  Assert.IsNotNull(sc);
  Assert.AreEqual(12.95, sc.Amount, .001);
}
```

Рисуя UML-диаграмму, изображенную на рис. 27.12, я думал, что замена `NoAffiliation` списком объектов `Affiliation` – удачное решение. Оно казалось мне более гибким и простым. Ведь я мог добавлять новые объекты, представляющие принадлежность к организации, в любой момент и обошелся бы без класса `NoAffiliation`. Однако начав писать тест, показанный в листинге 27.11, я понял, что установка свойства `Affiliation` объекта `Employee` лучше, чем вызов метода `AddAffiliation`. В конце концов, в требованиях не сказано, что работник может быть членом нескольких организаций, поэтому нет необходимости выполнять приведение типа для выбора одной из нескольких возможностей. Это только вызвало бы излишнее усложнение.

На этом примере мы убеждаемся, что рисовать слишком много UML-диаграмм без проверки их кодом опасно. Код помогает оценить разрабатываемый дизайн лучше, чем UML. В данном случае я включил в диаграмму ненужные структуры. Быть может, в один прекрасный день они и пригодятся, но сопровождать-то их придется уже сейчас. А результат может и не окупить затраты на сопровождение.

В этом примере сопровождать приведение типа относительно дешево, но раз я все равно не собираюсь этим пользоваться, то гораздо проще обойтись без списка объектов `Affiliation`. Поэтому я возвращаю на место паттерн Null-объект в форме класса `NoAffiliation`.

В листинге 27.12 показана реализация класса `ServiceChargeTransaction`. Действительно, без цикла поиска объектов `UnionAffiliation` он стал на-

много проще. Нужно лишь извлечь объект `Employee` **из базы данных, привести его объект** `Affillation` **к типу** `UnionAffilliation` **и добавить в него** `ServiceCharge`**.**

Листинг 27.12. ServiceChargeTransaction.cs

```
using System;

namespace Payroll
{
  public class ServiceChargeTransaction : Transaction
  {
    private readonly int memberId;
    private readonly DateTime time;
    private readonly double charge;
    public ServiceChargeTransaction(
        int id, DateTime time, double charge)
    {
      this.memberId = id;
      this.time = time;
      this.charge = charge;
    }

    public void Execute()
    {
      Employee e = PayrollDatabase.GetUnionMember(memberId);
      if (e != null)
      {
        UnionAffiliation ua = null;
        if(e.Affiliation is UnionAffiliation)
          ua = e.Affiliation as UnionAffiliation;
        if (ua != null)
          ua.AddServiceCharge(
            new ServiceCharge(time, charge));
        else
          throw new InvalidOperationException(
            "Попытка добавить плату за услуги для члена "
            + "профсоюза с незарегистрированным членством");
      }
      else
        throw new InvalidOperationException(
          "Член профсоюза не найден.");
    }
  }
}
```

Изменение сведений о работнике

На рис. 27.13 изображена статическая структура операций, изменяющих сведения о работнике. Она легко выводится из прецедента 6. Все операции принимают в качестве аргумента `EmpID`, поэтому мы можем

создать на верхнем уровне базовый класс `ChangeEmployeeTransaction`. Под ним находятся классы для изменения отдельных атрибутов, например `ChangeNameTransaction` и `ChangeAddressTransaction`. У операций, изменяющих порядок оплаты, есть общая черта: все они модифицируют одно и то же поле объекта `Employee`.



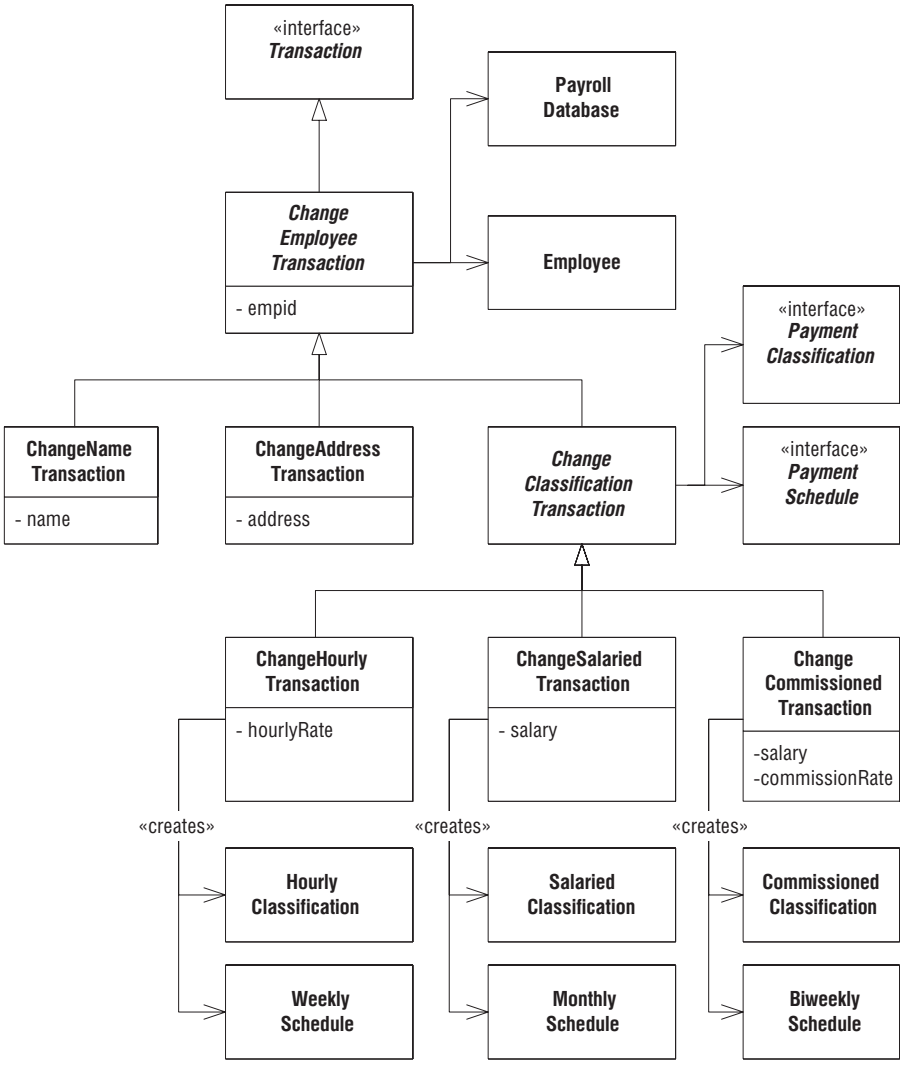


Рис. 27.13. *Статическая модель операции ChangeEmployeeTransaction*

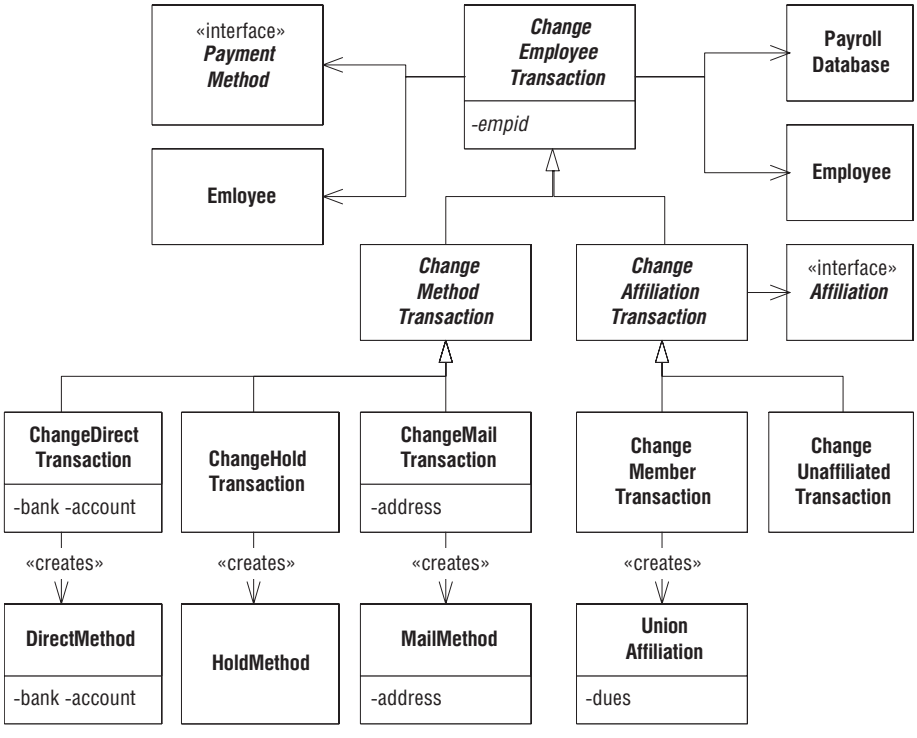


***Рис. 27.13** (продолжение)*

Поэтому их можно объединить в абстрактный базовый класс `ChangeClassificationTransaction`. То же самое относится к операциям, изменяющим способ платежа и членство в организациях. Им соответствуют базовые классы `ChangeMethodTransaction` и `ChangeAffiliationTransaction`.

На рис. 27.14 изображена динамическая модель всех операций изменения. И снова мы встречаемся с паттерном Шаблонный метод. Во всех

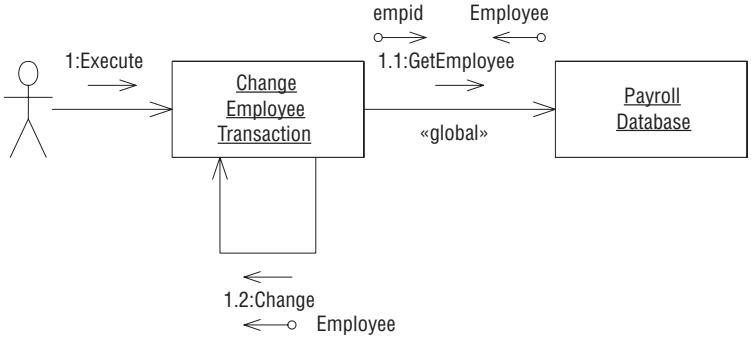


***Рис. 27.14.** Динамическая модель операции ChangeEmployeeTransaction*

случаях из базы данных PayrollDatabase нужно извлекать объект Employee с идентификатором EmpID. Класс ChangeEmployeeTransaction реализует это поведение, вызывая метод Execute, а затем посылая самому себе сообщение Change. Метод Change объявлен виртуальным и реализован в подклассах, как показано на рис. 27.15 и 27.16.

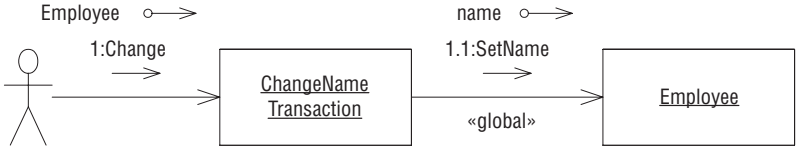


Рис. 27.15. *Динамическая модель операции ChangeNameTransaction*

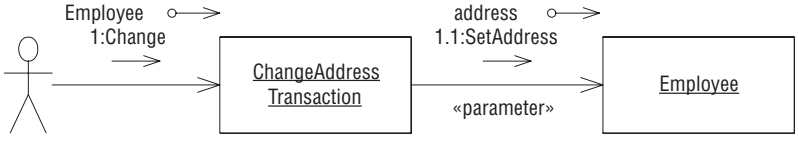


Рис. 27.16. *Динамическая модель операции ChangeAddressTransaction*

В листинге 27.13 приведен тест для класса ChangeNameTransaction. В нем для создания работника с почасовой оплатой по имени Bill используется операция AddHourlyEmployee. Затем создается объект ChangeNameTransaction и вызывается его метод Execute, который должен изменить имя работника с Bill на Bob. И наконец, этот объект Employee извлекается из базы данных PayrollDatabase и проверяется, что имя действительно было изменено.

Листинг 27.13. PayrollTest.TestChangeNameTransaction()

```
[Test]
public void TestChangeNameTransaction()
{
  int empId = 2;
  AddHourlyEmployee t =
    new AddHourlyEmployee(empId, "Bill", "Home", 15.25);
  t.Execute();
  ChangeNameTransaction cnt =
    new ChangeNameTransaction(empId, "Bob");
  cnt.Execute();
  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);
  Assert.AreEqual("Bob", e.Name);
}
```

В листинге 27.14 приведена реализация абстрактного базового класса ChangeEmployeeTransaction. Легко узнаваема структура паттерна Шаблон-

ный метод. Метод Execute() просто читает нужный экземпляр Employee из PayrollDatabase и если все прошло успешно, то вызывает абстрактный метод Change().

Листинг 27.14. ChangeEmployeeTransaction.cs

```
using System;

namespace Payroll
{
  public abstract class ChangeEmployeeTransaction : Transaction
  {
    private readonly int empId;

    public ChangeEmployeeTransaction(int empId)
    {
      this.empId = empId;
    }

    public void Execute()
    {
      Employee e = PayrollDatabase.GetEmployee(empId);
      if(e != null)
        Change(e);
      else
        throw new InvalidOperationException(
          "Работник не найден.");
    }

    protected abstract void Change(Employee e);
  }
}
```

В листинге 27.15 показана реализация класса ChangeNameTransaction. Тут мы видим вторую половину паттерна Шаблонный метод. Метод Change() изменяет имя в объекте Employee, переданном в качестве аргумента. Структура класса ChangeAddressTransaction очень похожа, его реализация оставлена вам в качестве упражнения.

Листинг 27.15. ChangeNameTransaction.cs

```
namespace Payroll
{
  public class ChangeNameTransaction :
    ChangeEmployeeTransaction
  {
    private readonly string newName;

    public ChangeNameTransaction(int id, string newName)
      : base(id)
    {
```

```
        this.newName = newName;
      }

      protected override void Change(Employee e)
      {
        e.Name = newName;
      }
    }
  }
```

Изменение тарификации. На рис. 27.17 изображена иерархия операции `ChangeClassificationTransaction`. Снова применен паттерн Шаблонный метод. Каждая операция такого вида должна создать новый объект `PaymentClassification` и передать его объекту `Employee`. Для этого объект посылает себе самому сообщение `GetClassification`. Это абстрактный метод, который реализован во всех подклассах `ChangeClassificationTransaction`, как показано на рис. 27.18–27.20.

В листинге 27.16 приведен тест для класса `ChangeHourlyTransaction`. В нем с помощью операции `AddCommissionedEmployee` создается работник с комиссионной оплатой, после чего создается объект `ChangeHourlyTransaction` и вызывается его метод `Execute()`. Затем из базы данных читается запись об измененном работнике и проверяется, что хранящийся в ней объект `PaymentClassification` действительно имеет тип `HourlyClassification`, причем в нем находится правильная почасовая ставка, а объект `PaymentSchedule` имеет тип `WeeklySchedule`.

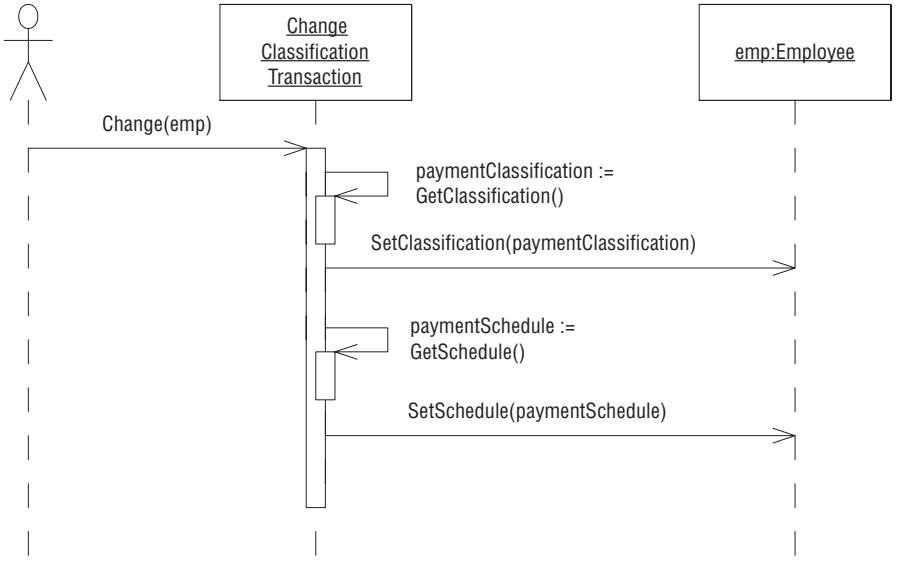


Рис. 27.17. *Динамическая модель операции ChangeClassificationTransaction*

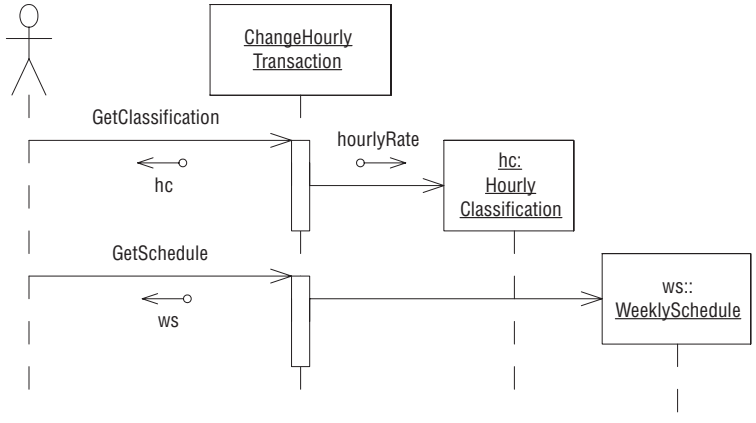


Рис. 27.18. *Динамическая модель операции ChangeHourlyTransaction*

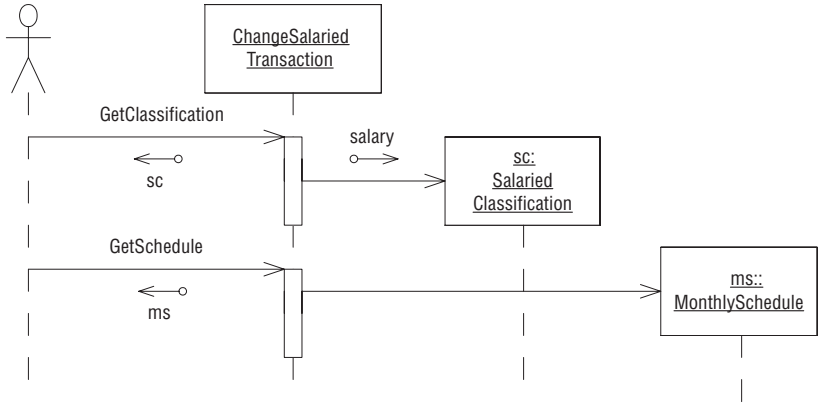


Рис. 27.19. *Динамическая модель операции ChangeSalariedTransaction*

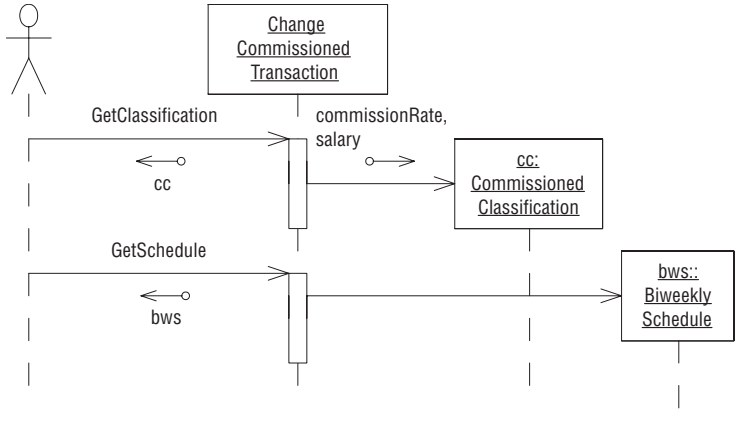


Рис. 27.20. *Динамическая модель операции ChangeCommissionedTransaction*

Листинг 27.16. PayrollTest.TestChangeHourlyTransaction()

```
[Test]
public void TestChangeHourlyTransaction()
{
  int empId = 3;
  AddCommissionedEmployee t =
    new AddCommissionedEmployee(
      empId, "Lance", "Home", 2500, 3.2);
  t.Execute();
  ChangeHourlyTransaction cht =
    new ChangeHourlyTransaction(empId, 27.52);
  cht.Execute();
  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);
  PaymentClassification pc = e.Classification;
  Assert.IsNotNull(pc);
  Assert.IsTrue(pc is HourlyClassification);
  HourlyClassification hc = pc as HourlyClassification;
  Assert.AreEqual(27.52, hc.HourlyRate, .001);
  PaymentSchedule ps = e.Schedule;
  Assert.IsTrue(ps is WeeklySchedule);
}
```

В листинге 27.17 приведена реализация абстрактного базового класса `ChangeClassificationTransaction`**. И на этот раз отчетливо просматривается паттерн Шаблонный метод. Метод** `Change()` **вызывает два абстрактных свойства,** `Classification` **и** `Schedule`**, и полученные от них значения записывает в объект** `Employee` **в качестве тарификации и графика выплат.**

Листинг 27.17. ChangeClassificationTransaction.cs

```
namespace Payroll
{
  public abstract class ChangeClassificationTransaction
    : ChangeEmployeeTransaction
  {
    public ChangeClassificationTransaction(int id)
      : base (id)
    {}

    protected override void Change(Employee e)
    {
      e.Classification = Classification;
      e.Schedule = Schedule;
    }

    protected abstract
      PaymentClassification Classification { get; }
    protected abstract PaymentSchedule Schedule { get; }
  }
}
```

Решение воспользоваться свойствами, а не методами `get` было принято по ходу написания кода. В очередной раз мы встречаемся с расхождением между диаграммой и кодом.

В листинге 27.18 приведена реализация класса `ChangeHourlyTransaction`. Он завершает паттерн Шаблонный метод, реализуя `get`-свойства `Classification` и `Schedule`, унаследованные от `ChangeClassificationTransaction`. В данном случае свойство `Classification` возвращает вновь созданный объект `HourlyClassification`, а свойство `Schedule` – вновь созданный объект `WeeklySchedule`.

Листинг 27.18. ChangeHourlyTransaction.cs

```
namespace Payroll
{
  public class ChangeHourlyTransaction
    : ChangeClassificationTransaction
  {
    private readonly double hourlyRate;

    public ChangeHourlyTransaction(int id, double hourlyRate)
      : base(id)
    {
      this.hourlyRate = hourlyRate;
    }

    protected override PaymentClassification Classification
    {
      get { return new HourlyClassification(hourlyRate); }
    }

    protected override PaymentSchedule Schedule
    {
      get { return new WeeklySchedule(); }
    }
  }
}
```

Как и раньше, реализацияя классов `ChangeSalariedTransaction` и `ChangeCommissionedTransaction` оставлена вам в качестве упражнения.

Аналогичный механизм используется для реализации класса `ChangeMethodTransaction`. Абстрактное свойство `Method` применяется для выбора объекта подходящего подкласса `PaymentMethod`, который затем передается объекту `Employee` (рис. 27.21–27.24).

Реализация этих классов не сулит никаких сюрпризов и оставлена вам в качестве упражнения.

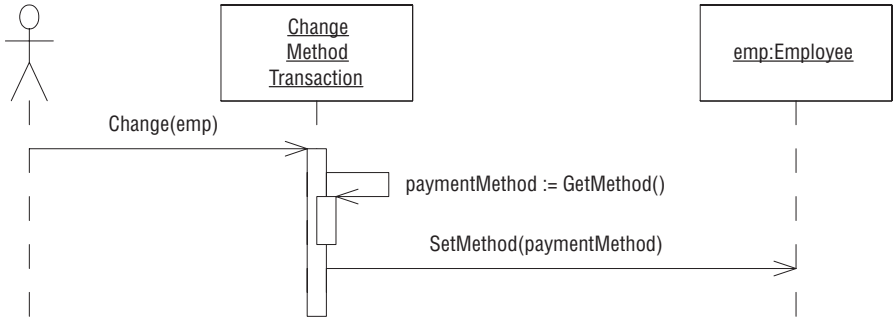


Рис. 27.21. *Динамическая модель операции ChangeMethodTransaction*

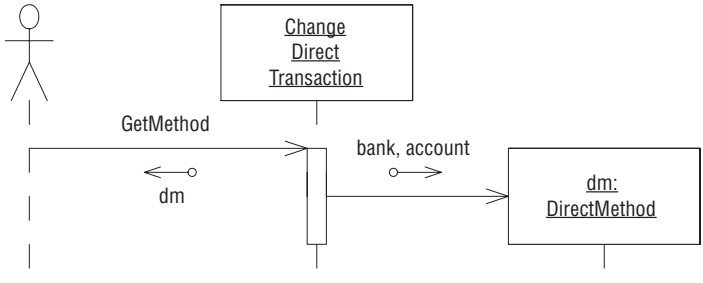


Рис. 27.22. *Динамическая модель операции ChangeDirectTransaction*

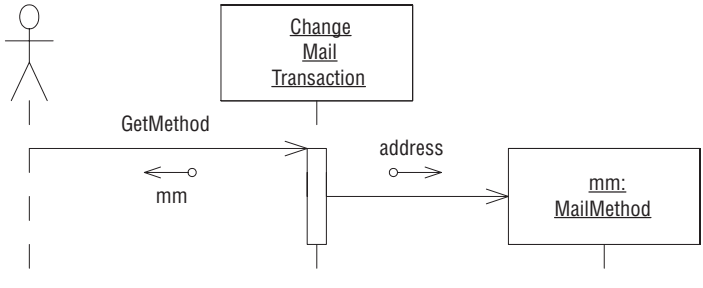


Рис. 27.23. *Динамическая модель операции ChangeMailTransaction*

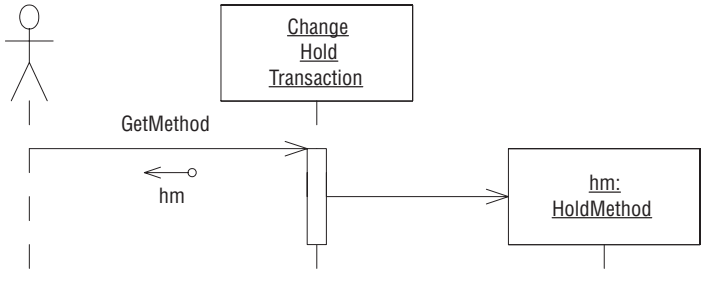


Рис. 27.24. *Динамическая модель операции ChangeHoldTransaction*

На рис. 27.25 показана реализация класса `ChangeAffiliationTransaction`. И здесь паттерн Шаблонный метод применяется для выбора объекта одного из подклассов `Affiliation`, который затем следует передать объекту `Employee` (рис. 27.26 и 27.27).

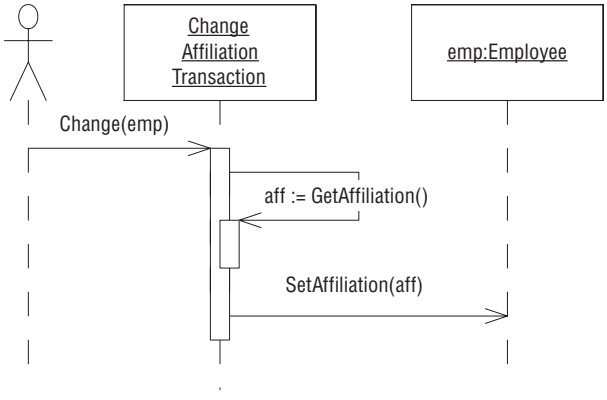


Рис. 27.25. *Динамическая модель операции ChangeAffiliationTransaction*

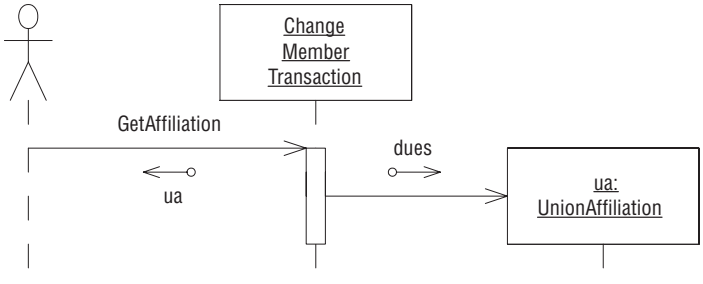


Рис. 27.26. *Динамическая модель операции ChangeMemberTransaction*

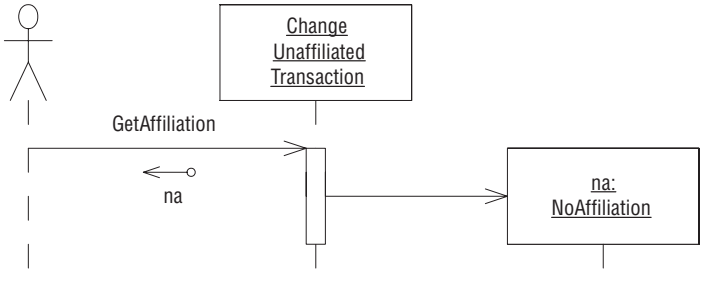


Рис. 27.27. *Динамическая модель операции ChangeUnaffiliatedTransaction*

Что я курил?

Приступив к реализации этого дизайна, я был немало удивлен. Взгляните внимательнее на динамические диаграммы для операций изменения членства в организациях. Улавливаете проблему?

Как обычно, я начал с написания теста для класса `ChangeMemberTransaction`. Он показан в листинге 27.19. Поначалу все стандартно. Создается работник Билл с почасовой оплатой, затем создается объект `ChangeMemberTransaction` и вызывается его метод `Execute()`, который делает Билла членом профсоюза. Потом проверяется, что с Биллом связан объект `UnionAffiliation` и в этом объекте хранится правильная ставка взносов.

Листинг 27.19. PayrollTest.ChangeUnionMember()

```
[Test]
public void ChangeUnionMember()
{
  int empId = 8;
  AddHourlyEmployee t =
    new AddHourlyEmployee(empId, "Билл", "Домашний", 15.25);
  t.Execute();
  int memberId = 7743;
  ChangeMemberTransaction cmt =
    new ChangeMemberTransaction(empId, memberId, 99.42);
  cmt.Execute();
  Employee e = PayrollDatabase.GetEmployee(empId);
  Assert.IsNotNull(e);
  Affiliation affiliation = e.Affiliation;
  Assert.IsNotNull(affiliation);
  Assert.IsTrue(affiliation is UnionAffiliation);
  UnionAffiliation uf = affiliation as UnionAffiliation;
  Assert.AreEqual(99.42, uf.Dues, .001);
  Employee member = PayrollDatabase.GetUnionMember(memberId);
  Assert.IsNotNull(member);
  Assert.AreEqual(e, member);
}
```

Сюрприз скрыт в последних строках теста. В них проверяется, действительно ли в базе `PayrollDatabase` сохранилась информация о том, что Билл – член профсоюза. Однако в существующих UML-диаграммах на это нет никаких указаний. Из диаграмм мы видим лишь, что с объектом `Employee` связывается объект подходящего подкласса `Affiliation`. Я не заметил этого упущения. А вы?

Я спокойно занимался кодированием операций в полном соответствии с диаграммами, как вдруг этот автономный тест не прошел. В чем ошибка, я понял сразу. А вот как ее исправить, было неясно. Как сделать, чтобы информация о членстве запоминалась операцией `ChangeMemberTransaction`, но стиралась операцией `ChangeUnaffiliatedTransaction`?

Ответ в том, чтобы добавить в класс ChangeAffiliationTransaction еще один абстрактный метод RecordMembership(Employee). В подклассе ChangeMemberTransaction он будет записывать memberId в объект Employee, а в подклассе ChangeUnaffiliatedTransaction – стирать информацию о членстве.

В листинге 27.20 показана окончательная реализация абстрактного базового класса ChangeAffiliationTransaction. И на этот раз очевидно использование паттерна Шаблонный метод.

Листинг 27.20. ChangeAffiliationTransaction.cs

```
namespace Payroll
{
  public abstract class ChangeAffiliationTransaction :
    ChangeEmployeeTransaction
  {
    public ChangeAffiliationTransaction(int empId)
      : base(empId)
    {}

    protected override void Change(Employee e)
    {
      RecordMembership(e);
      Affiliation affiliation = Affiliation;
      e.Affiliation = affiliation;
    }

    protected abstract Affiliation Affiliation { get; }
    protected abstract void RecordMembership(Employee e);
  }
}
```

В листинге 27.21 приведена реализация класса ChangeMemberTransaction. Ничего особо сложного или интересного в нем нет. А вот реализация ChangeUnaffiliatedTransaction в листинге 27.22 чуть более любопытна. Метод RecordMembership должен определить, является ли текущий работник членом профсоюза. Если да, то он получает memberId из объекта UnionAffiliation и стирает запись о членстве.

Листинг 27.21. ChangeMemberTransaction.cs

```
namespace Payroll
{
  public class ChangeMemberTransaction :
    ChangeAffiliationTransaction
  {
    private readonly int memberId;
    private readonly double dues;

    public ChangeMemberTransaction(
      int empId, int memberId, double dues)
      : base(empId)
```

```
    {
      this.memberId = memberId;
      this.dues = dues;
    }

    protected override Affiliation Affiliation
    {
      get { return new UnionAffiliation(memberId, dues); }
    }

    protected override void RecordMembership(Employee e)
    {
      PayrollDatabase.AddUnionMember(memberId, e);
    }
  }
}
```

Листинг 27.22. ChangeUnaffiliatedTransaction.cs

```
namespace Payroll
{
  public class ChangeUnaffiliatedTransaction
    : ChangeAffiliationTransaction
  {
    public ChangeUnaffiliatedTransaction(int empId)
      : base(empId)
    {}

    protected override Affiliation Affiliation
    {
      get { return new NoAffiliation(); }
    }

    protected override void RecordMembership(Employee e)
    {
      Affiliation affiliation = e.Affiliation;
      if(affiliation is UnionAffiliation)
      {
        UnionAffiliation unionAffiliation =
          affiliation as UnionAffiliation;
        int memberId = unionAffiliation.MemberId;
        PayrollDatabase.RemoveUnionMember(memberId);
      }
    }
  }
}
```

Не могу сказать, что очень доволен таким дизайном. Мне не нравится, что класс `ChangeUnaffiliatedTransaction` должен знать о существовании `UnionAffiliation`. Эту проблему можно было бы решить, включив в класс `Affiliation` абстрактные методы `RecordMembership` и `EraseMembership`. Но

тогда `UnionAffiliation` и `NoAffiliation` должны были бы знать о `PayrollDatabase`. А это меня тоже как-то не радует.[1]

Но вообще-то показанная реализация довольно проста и принцип OCP нарушается лишь чуть-чуть. Хорошо то, что очень немногие модули в системе знают о существовании класса `ChangeUnaffiliatedTransaction`, поэтому дополнительные зависимости в нем большого вреда не нанесут.

Начисление зарплаты

Вот и пришло время для операции, составляющей весь смысл приложения: начисления зарплаты части работников. На рис. 27.28 изображена статическая структура класса `PaydayTransaction`, а на рис. 27.29 и 27.30 – его динамическое поведение.

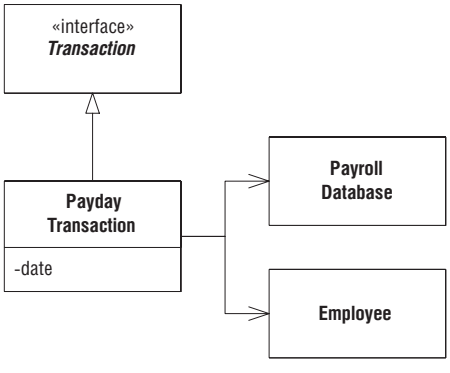


Рис. 27.28. *Статическая модель класса PaydayTransaction*

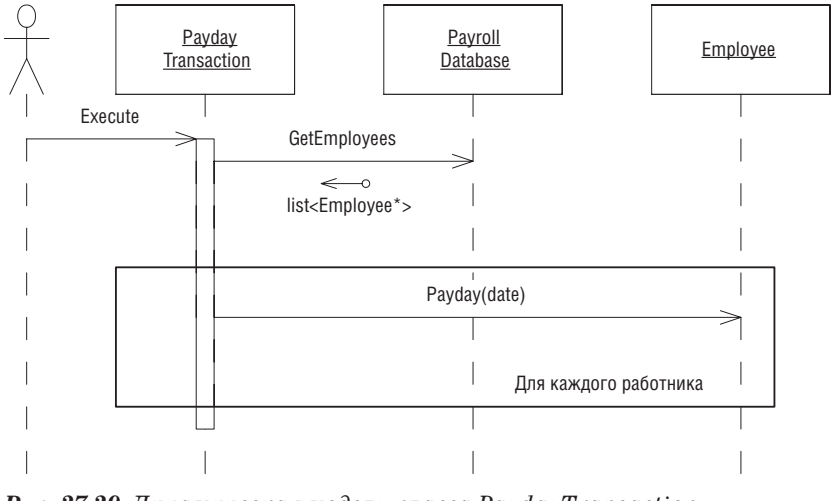


Рис. 27.29. *Динамическая модель класса PaydayTransaction*

[1] Можно было бы решить проблему за счет применения шаблона Посетитель, но это, пожалуй, станет чрезмерным усложнением.

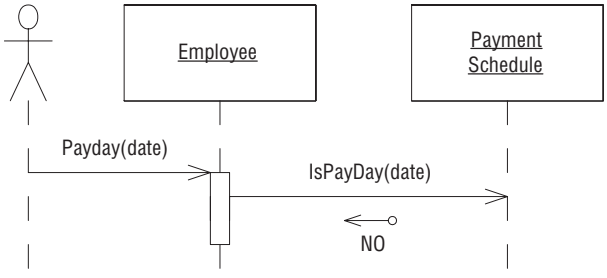


Рис. 27.30. *Динамическая модель сценария «Сегодня еще не пора начислять зарплату»*

В динамической модели явственно видно полиморфное поведение. Работа метода `CalculatePay` зависит от типа объекта `PaymentClassification`, хранящегося в объекте `Employee`. Алгоритм, применяемый для определения того, наступила ли дата выплаты, зависит от типа объекта `PaymentSchedule`. Наконец, метод выплаты начисленной суммы зависит от типа объекта `PaymentMethod`. Благодаря высокой степени абстракции все эти алгоритмы закрыты относительно добавления нового способа тарификации, графика выплат, определения принадлежности к внешним организациям или способа платежа.

В алгоритмах на рис. 27.31 и 27.32 вводится понятие *запоминания даты* (Post). После того как величина зарплаты рассчитана и передана объекту `Employee`, дата расчета запоминается, то есть обновляются записи, имеющие отношение к расчету. Следовательно, можно сказать, что метод `CalculatePay` рассчитывает зарплату от последней запомненной даты до указанной.

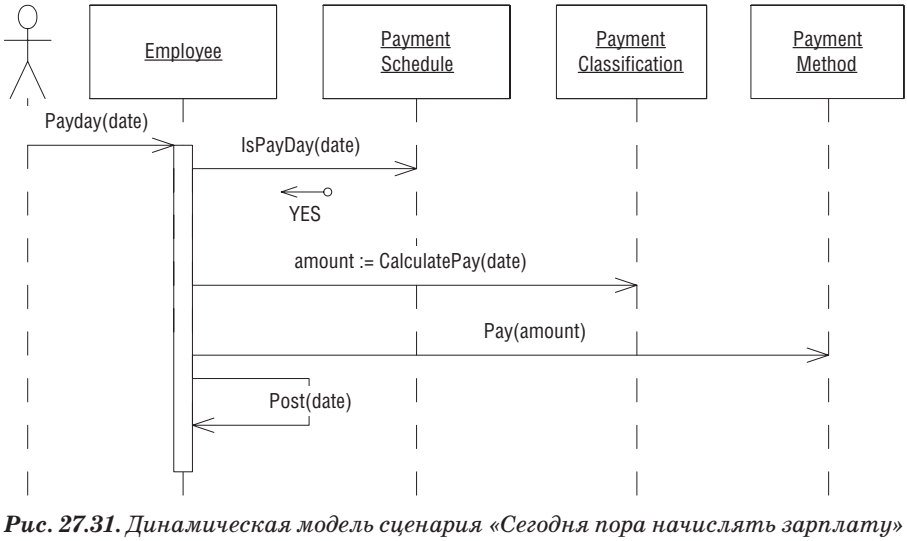


Рис. 27.31. *Динамическая модель сценария «Сегодня пора начислять зарплату»*

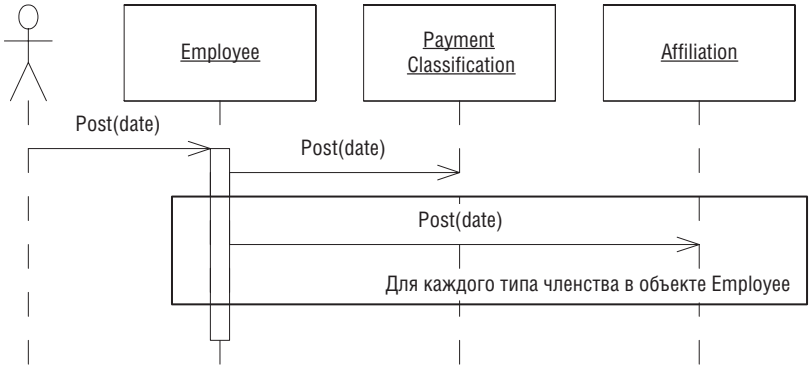


Рис. 27.32. *Динамическая модель сценария «Запоминание даты начисления зарплаты»*

Разработчики и бизнес-решения. Откуда взялась идея о запоминании даты? Совершенно точно она не упоминалась ни в пользовательских историях, ни в прецедентах. Получается, что я придумал ее для решения встретившейся проблемы. Меня смущало, что метод `Payday` может несколько раз вызываться с одной и той же датой или в одном и том же расчетном периоде, и я хотел предотвратить многократное начисление зарплаты одному работнику. Я сделал это по собственной инициативе, не советуясь с заказчиком. Мне просто показалось, что так будет правильно.

По сути дела, я принял бизнес-решение, считая, что разные прогоны программы начисления зарплаты должны давать разные результаты. Но следовало бы проконсультироваться с заказчиком или менеджером проекта, потому что у них на этот счет могут быть другие соображения.

Поговорив с заказчиком[1], я понял, что идея запоминания даты идет вразрез с его намерениями. Заказчик хочет, чтобы можно было запустить программу и посмотреть на сгенерированные ею чеки. Он хочет, чтобы при обнаружении ошибок можно было исправить входную информацию и запустить программу снова. Заказчик сказал, что карточки табельного учета или справки о продажах с датами вне текущего расчетного периода никогда не должны приниматься во внимание.



Поэтому всю схему с запоминанием даты расчета придется отбросить. Мне подумалось, что это хорошая идея, но заказчик решил иначе.

[1] Которым в данном случае я сам и являюсь.

Начисление зарплаты работникам с твердым окладом

Два теста в листинге 27.23 проверяют правильность начисления зарплаты работникам с твердым окладом. В первом случае проверяется, что зарплата действительно начисляется в последний день месяца, а во втором – что ни в какой день месяца, кроме последнего, она не начисляется.

Листинг 27.23. PayrollTest.PaySingleSalariedEmployee и прочие

```
[Test]
public void PaySingleSalariedEmployee()
{
  int empId = 1;
  AddSalariedEmployee t = new AddSalariedEmployee(
    empId, "Bob", "Home", 1000.00);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 30);
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  Paycheck pc = pt.GetPaycheck(empId);
  Assert.IsNotNull(pc);
  Assert.AreEqual(payDate, pc.PayDate);
  Assert.AreEqual(1000.00, pc.GrossPay, .001);
  Assert.AreEqual("Hold", pc.GetField("Disposition"));
  Assert.AreEqual(0.0, pc.Deductions, .001);
  Assert.AreEqual(1000.00, pc.NetPay, .001);
}

[Test]
public void PaySingleSalariedEmployeeOnWrongDate()
{
  int empId = 1;
  AddSalariedEmployee t = new AddSalariedEmployee(
    empId, "Bob", "Home", 1000.00);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 29);
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  Paycheck pc = pt.GetPaycheck(empId);
  Assert.IsNull(pc);
}
```

В листинге 27.24 показан метод `Execute()` из класса `PaydayTransaction`. Он перебирает все объекты `Employee` в базе данных и у каждого спрашивает, надо ли ему начислять зарплату в день, указанный в операции. Если да, то метод создает новый платежный чек для данного работника и просит объект `Employee` заполнить его поля.

Листинг 27.24. PaydayTransaction.Execute()

```
public void Execute()
{
  ArrayList empIds = PayrollDatabase.GetAllEmployeeIds();
```

```
    foreach(int empId in empIds)
    {
      Employee employee = PayrollDatabase.GetEmployee(empId);
      if (employee.IsPayDate(payDate)) {
        Paycheck pc = new Paycheck(payDate);
        paychecks[empId] = pc;
        employee.Payday(pc);
      }
    }
  }
```

В листинге 27.25 приведен файл `MonthlySchedule.cs`**. Обратите внимание, что метод** `IsPayDate` **возвращает** `true`**, только если дата, переданная в аргументе, является последним днем месяца.**

Листинг 27.25. MonthlySchedule.cs

```
using System;

namespace Payroll
{
  public class MonthlySchedule : PaymentSchedule
  {
    private bool IsLastDayOfMonth(DateTime date)
    {
      int m1 = date.Month;
      int m2 = date.AddDays(1).Month;
      return (m1 != m2);
    }

    public bool IsPayDate(DateTime payDate)
    {
      return IsLastDayOfMonth(payDate);
    }
  }
}
```

В листинге 27.26 приведена реализация метода `Employee.PayDay()`**. Это общий алгоритм расчета и доставки зарплаты любому работнику. Обратите внимание на повсеместное использование паттерна Стратегия. Все конкретные вычисления поручаются классам стратегий: выбора порядка оплаты и способа платежа, определения членства.**

Листинг 27.26. Employee.PayDay()

```
public void PayDay(Paycheck paycheck)
{
  double grossPay = classification.CalculatePay(paycheck);
  double deductions =
    affiliation.CalculateDeductions(paycheck);
  double netPay = grossPay - deductions;
  paycheck.GrossPay = grossPay;
  paycheck.Deductions = deductions;
```

```
    paycheck.NetPay = netPay;
    method.Pay(paycheck);
  }
```

Начисление зарплаты работникам с почасовой оплатой

Расчет зарплаты для работников с почасовой оплатой – хороший пример пошаговой природы разработки через тестирование. Я начал с тривиальных тестов и постепенно шел ко все более и более сложным. Сначала я покажу сами тесты, а потом получившийся на их основе код.

В листинге 27.27 рассмотрен простейший случай. Мы добавляем в базу данных работника с почасовой оплатой, а затем рассчитываем его зарплату. Поскольку никаких карточек табельного учета еще нет, мы ожидаем получить нуль. Вспомогательный метод `ValidateHourlyPaycheck` появился в ходе проведенного позднее рефакторинга. Поначалу его код был частью тестового метода. Этот тест проходил при условии, что метод `WeeklySchedule.IsPayDate()` возвращал `true`.

Листинг 27.27. PayrollTest.TestPaySingleHourlyEmployeeNoTimeCards()

```
[Test]
public void PayingSingleHourlyEmployeeNoTimeCards()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Билл", "Домашний", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 9);
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  ValidateHourlyPaycheck(pt, empId, payDate, 0.0);
}

private void ValidateHourlyPaycheck(PaydayTransaction pt,
  int empid, DateTime payDate, double pay)
{
  Paycheck pc = pt.GetPaycheck(empid);
  Assert.IsNotNull(pc);
  Assert.AreEqual(payDate, pc.PayDate);
  Assert.AreEqual(pay, pc.GrossPay, .001);
  Assert.AreEqual("Hold", pc.GetField("Disposition"));
  Assert.AreEqual(0.0, pc.Deductions, .001);
  Assert.AreEqual(pay, pc.NetPay, .001);
}
```

В листинге 27.28 приведены еще два теста. В первом проверяется, как начисляется зарплата после добавления одной карточки табельного учета, во втором – что для карточки с переработкой (более 8 часов) начисляются сверхурочные. Разумеется, я не писал эти тесты одновременно. Сначала написал первый, заставил его работать, а потом принялся за второй.

Листинг 27.28. PayrollTest.PaySingleHourlyEmployee...()

```
[Test]
public void PaySingleHourlyEmployeeOneTimeCard()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Билл", "Домашний", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 9); // пятница
  TimeCardTransaction tc =
    new TimeCardTransaction(payDate, 2.0, empId);
  tc.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  ValidateHourlyPaycheck(pt, empId, payDate, 30.5);
}

[Test]
public void PaySingleHourlyEmployeeOvertimeOneTimeCard()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Билл", "Домашний", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 9); // пятница
  TimeCardTransaction tc =
    new TimeCardTransaction(payDate, 9.0, empId);
  tc.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  ValidateHourlyPaycheck(pt, empId, payDate,
    (8 + 1.5)*15.25);
}
```

Мне удалось заставить первый тест работать, изменив метод `HourlyClassification.CalculatePay` так, чтобы он перебирал все карточки для данного работника, суммировал часы и умножал на почасовую ставку. Второй тест заработал, когда я изменил метод так, чтобы он отдельно учитывал урочные и сверхурочные часы.

Тест в листинге 27.29 проверяет, что мы начисляем зарплату почасовикам, если конструктору класса `PaydayTransaction` передана дата с любым днем, кроме пятницы.

Листинг 27.29. PayrollTest.PaySingleHourlyEmployeeOnWrongDate()

```
[Test]
public void PaySingleHourlyEmployeeOnWrongDate()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
```

```
    empId, "Билл", "Домашний", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 8); // четверг
  TimeCardTransaction tc =
    new TimeCardTransaction(payDate, 9.0, empId);
  tc.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  Paycheck pc = pt.GetPaycheck(empId);
  Assert.IsNull(pc);
}
```

В листинге 27.30 показан тест, проверяющий правильность начисления зарплаты работнику с несколькими карточками табельного учета.

Листинг 27.30. PayrollTest.PaySingleHourlyEmployeeTwoTimeCards()

```
[Test]
public void PaySingleHourlyEmployeeTwoTimeCards()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Билл", "Домашний", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 9); // пятница
  TimeCardTransaction tc =
    new TimeCardTransaction(payDate, 2.0, empId);
  tc.Execute();
  TimeCardTransaction tc2 =
    new TimeCardTransaction(payDate.AddDays(-1), 5.0,empId);
  tc2.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  ValidateHourlyPaycheck(pt, empId, payDate, 7*15.25);
}
```

Наконец, тест в листинге 27.31 проверяет, что зарплата начисляется только по карточкам с датами в расчетном периоде. Все остальные карточки игнорируются.

Листинг 27.31. PayrollTest.Test...WithTimeCardsSpanningTwoPayPeriods()

```
[Test]
public void
TestPaySingleHourlyEmployeeWithTimeCardsSpanningTwoPayPeriods()
{
  int empId = 2;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Билл", "Домашний", 15.25);
  t.Execute();
  DateTime payDate = new DateTime(2001, 11, 9); // пятница
  DateTime dateInPreviousPayPeriod =
```

```
    new DateTime(2001, 11, 2);
  TimeCardTransaction tc =
    new TimeCardTransaction(payDate, 2.0, empId);
  tc.Execute();
  TimeCardTransaction tc2 = new TimeCardTransaction(
    dateInPreviousPayPeriod, 5.0, empId);
  tc2.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  ValidateHourlyPaycheck(pt, empId, payDate, 2*15.25);
}
```

Чтобы все эти тесты завершались успешно, мы писали код по частям, выполняя по одному тесту за раз. Та структура, которую вы видите в последующих листингах, совершенствовалась от теста к тесту. В листинге 27.32 показаны фрагменты файла `HourlyClassification.cs`. Мы просто перебираем в цикле карточки табельного учета и для каждой проверяем, попадает ли она в расчетный период. Если да, то вычисляется вклад этой карточки в величину зарплаты.

Листинг 27.32. HourlyClassification.cs (фрагмент)

```
public double CalculatePay(Paycheck paycheck)
{
  double totalPay = 0.0;
  foreach(TimeCard timeCard in timeCards.Values)
  {
    if(IsInPayPeriod(timeCard, paycheck.PayDate))
      totalPay += CalculatePayForTimeCard(timeCard);
  }
  return totalPay;
}

private bool IsInPayPeriod(TimeCard card,
  DateTime payPeriod)
{
  DateTime payPeriodEndDate = payPeriod;
  DateTime payPeriodStartDate = payPeriod.AddDays(-5);
  return card.Date <= payPeriodEndDate &&
    card.Date >= payPeriodStartDate;
}

private double CalculatePayForTimeCard(TimeCard card)
{
  double overtimeHours = Math.Max(0.0, card.Hours - 8);
  double normalHours = card.Hours - overtimeHours;
  return hourlyRate * normalHours +
  hourlyRate * 1.5 * overtimeHours;
}
```

Из листинга 27.33 видно, что класс `WeeklySchedule` платит только по пятницам.

Листинг 27.33. WeeklySchedule.IsPayDate()

```
public bool IsPayDate(DateTime payDate)
{
  return payDate.DayOfWeek == DayOfWeek.Friday;
}
```

Расчет зарплаты для работников с комиссионной оплаты оставлен вам в качестве упражнения. Никаких серьезных сюпризов здесь не ожидается.

Расчетные периоды: проблема проектирования. Теперь пора заняться взносами и платой за услуги профсоюза. Я подумываю о тесте, который добавит работника с твердым окладом, сделает его членом профсоюза, а затем рассчитает ему зарплату и проверит, что из нее вычтена сумма взносов. Код показан в листинге 27.34.

Листинг 27.34. PayrollTest.SalariedUnionMemberDues()

```
[Test]
public void SalariedUnionMemberDues()
{
  int empId = 1;
  AddSalariedEmployee t = new AddSalariedEmployee(
    empId, "Билл", "Домашний", 1000.00);
  t.Execute();
  int memberId = 7734;
  ChangeMemberTransaction cmt =
  new ChangeMemberTransaction(empId, memberId, 9.42);
    cmt.Execute();
  DateTime payDate = new DateTime(2001, 11, 30);
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  Paycheck pc = pt.GetPaycheck(empId);
  Assert.IsNotNull(pc);
  Assert.AreEqual(payDate, pc.PayDate);
  Assert.AreEqual(1000.0, pc.GrossPay, .001);
  Assert.AreEqual("Hold", pc.GetField("Disposition"));
  Assert.AreEqual(???, pc.Deductions, .001);
  Assert.AreEqual(1000.0 - ???, pc.NetPay, .001);
}
```

Обратите внимание на символы **???** в последних двух строках. Что вместо них подставить? Пользовательские истории говорят, что членские взносы должны рассчитываться еженедельно, но работникам на окладе платят раз в месяц. Сколько недель в месяце? Следует ли просто умножить величину взноса на 4? Это не совсем точно. Надо спросить у заказчика, чего хочет он.[1]

[1] И снова Боб ведет беседу сам с собой. Зайдите на сайт google.com/groups и поищите «Schizophrenic Robert Martin».

Заказчик сообщает, что профсоюзные взносы начисляются каждую пятницу. Поэтому я должен подсчитать количество пятниц в расчетном периоде и умножить его на величину еженедельного взноса. В ноябре 2001 года, для которого написан тест, было пять пятниц, поэтому я соответственно модифицирую тест.

Для подсчета пятниц в расчетном периоде нужно знать его начальную и конечную даты. Я уже проделывал такие вычисления раньше в методе `IsInPayPeriod` в листинге 27.32. (А вы, наверное, написали нечто подобное в методе `CommissionedClassification`.) Этот метод вызывается из метода `CalculatePay` объекта `HourlyClassification`, чтобы учитывались только карточки табельного учета, попадающие в расчетный период. Теперь выясняется, что в классе `UnionAffiliation` он тоже нужен.

Стоп-стоп-стоп! А что этот метод вообще делает в классе `HourlyClassification`? Мы уже выяснили, что ассоциация между графиком выплат и порядком оплаты несущественна. Метод, имеющий дело с расчетным периодом, должен находиться в классе `PaymentSchedule`, а не `PaymentClassification`!

Интересно, что UML-диаграммы не помогли заметить эту проблему. Она стала очевидной, когда я начал обдумывать тесты для класса `UnionAffiliation`. Это еще один пример того, как важно поверять любой дизайн кодом. Диаграммы – вещь полезная, но полагаться только на них, не подтверждая свои наблюдения кодом, рискованно.

Ну и как же мы перенесем расчетный период из иерархии `PaymentSchedule` в иерархии `PaymentClassification` и `Affiliation`? Они ведь ничего не знают друг о друге. Но есть одна мысль. Можно было бы поместить даты расчетного периода в объект `Paycheck`. В данный момент в `Paycheck` хранится только дата конца периода. Почему бы не добавить туда и дату начала?

В листинге 27.35 показаны изменения в методе `PaydayTransaction.Execute()`. Обратите внимание, что при создании объекта `Paycheck` его конструктору передаются даты начала и конца расчетного периода. А вычисляет обе даты объект `PaymentSchedule`. Изменения в классе `Paycheck` очевидны.

Листинг 27.35. PaydayTransaction.Execute()

```
public void Execute()
{
  ArrayList empIds = PayrollDatabase.GetAllEmployeeIds();
  foreach(int empId in empIds)
  {
    Employee employee = PayrollDatabase.GetEmployee(empId);
    if (employee.IsPayDate(payDate))
    {
      DateTime startDate =
        employee.GetPayPeriodStartDate(payDate);
```

```
        Paycheck pc = new Paycheck(startDate, payDate);
        paychecks[empId] = pc;
        employee.Payday(pc);
      }
    }
  }
```

Два метода в классах `HourlyClassification` **и** `CommissionedClassification`**, которые определяли, попадает ли в расчетный период дата, указанная в объектах** `TimeCard` **и** `SalesReceipt`**, теперь объединены и перенесены в базовый класс** `PaymentClassification`**. См. листинг 27.36.**

Листинг 27.36. PaymentClassification.IsInPayPeriod(...)

```
public bool IsInPayPeriod(DateTime theDate, Paycheck paycheck)
{
  DateTime payPeriodEndDate = paycheck.PayPeriodEndDate;
  DateTime payPeriodStartDate = paycheck.PayPeriodStartDate;
  return (theDate >= payPeriodStartDate)
    && (theDate <= payPeriodEndDate);
}
```

Теперь мы готовы написать метод `UnionAffilliation.CalculateDeductions`**, в котором вычисляется сумма взносов (листинг 27.37). Из объекта** `paycheck` **извлекаются даты начала и конца периода и передаются вспомогательному методу, который подсчитывает число пятниц между ними. Затем полученное число умножается на недельную величину взносов, это и будет окончательный результат.**

Листинг 27.37. UnionAffiliation.CalculateDeductions(...)

```
public double CalculateDeductions(Paycheck paycheck)
{
  double totalDues = 0;
  int fridays = NumberOfFridaysInPayPeriod(
    paycheck.PayPeriodStartDate, paycheck.PayPeriodEndDate);
  totalDues = dues * fridays;
  return totalDues;
}

private int NumberOfFridaysInPayPeriod(
  DateTime payPeriodStart, DateTime payPeriodEnd)
{
  int fridays = 0;
  for (DateTime day = payPeriodStart;
    day <= payPeriodEnd; day.AddDays(1))
  {
    if (day.DayOfWeek == DayOfWeek.Friday)
      fridays++;
  }
  return fridays;
}
```

В последних двух тестах рассматривается плата за услуги профсоюза. Тест в листинге 27.38 проверяет, что плата за услуги вычитается правильно.

Листинг 27.38. PayrollTest.HourlyUnionMemberServiceCharge()

```
[Test]
public void HourlyUnionMemberServiceCharge()
{
  int empId = 1;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Билл", "Домашний", 15.24);
  t.Execute();
  int memberId = 7734;
  ChangeMemberTransaction cmt =
    new ChangeMemberTransaction(empId, memberId, 9.42);
  cmt.Execute();
  DateTime payDate = new DateTime(2001, 11, 9);
  ServiceChargeTransaction sct =
    new ServiceChargeTransaction(memberId, payDate, 19.42);
  sct.Execute();
  TimeCardTransaction tct =
    new TimeCardTransaction(payDate, 8.0, empId);
  tct.Execute();
  PaydayTransaction pt = new PaydayTransaction(payDate);
  pt.Execute();
  Paycheck pc = pt.GetPaycheck(empId);
  Assert.IsNotNull(pc);
  Assert.AreEqual(payDate, pc.PayPeriodEndDate);
  Assert.AreEqual(8*15.24, pc.GrossPay, .001);
  Assert.AreEqual("Hold", pc.GetField("Disposition"));
  Assert.AreEqual(9.42 + 19.42, pc.Deductions, .001);
  Assert.AreEqual((8*15.24)-(9.42 + 19.42),pc.NetPay, .001);
}
```

Второй тест, который выявил некую проблему, показан в листинге 27.39. В нем проверяется, что платежные требования с датой вне расчетного периода не учитываются.

Листинг 27.39. PayrollTest.ServiceChargesSpanningMultiplePayPeriods()

```
[Test]
public void ServiceChargesSpanningMultiplePayPeriods()
{
  int empId = 1;
  AddHourlyEmployee t = new AddHourlyEmployee(
    empId, "Билл", "Домашний", 15.24);
  t.Execute();
  int memberId = 7734;
  ChangeMemberTransaction cmt =
    new ChangeMemberTransaction(empId, memberId, 9.42);
  cmt.Execute();
```

```
    DateTime payDate = new DateTime(2001, 11, 9);
    DateTime earlyDate =
      new DateTime(2001, 11, 2); // предыдущая пятница
    DateTime lateDate =
      new DateTime(2001, 11, 16); // следующая пятница
    ServiceChargeTransaction sct =
      new ServiceChargeTransaction(memberId, payDate, 19.42);
    sct.Execute();
    ServiceChargeTransaction sctEarly =
      new ServiceChargeTransaction(memberId,earlyDate,100.00);
    sctEarly.Execute();
    ServiceChargeTransaction sctLate =
      new ServiceChargeTransaction(memberId,lateDate,200.00);
    sctLate.Execute();
    TimeCardTransaction tct =
      new TimeCardTransaction(payDate, 8.0, empId);
    tct.Execute();
    PaydayTransaction pt = new PaydayTransaction(payDate);
    pt.Execute();
    Paycheck pc = pt.GetPaycheck(empId);
    Assert.IsNotNull(pc);
    Assert.AreEqual(payDate, pc.PayPeriodEndDate);
    Assert.AreEqual(8*15.24, pc.GrossPay, .001);
    Assert.AreEqual("Hold", pc.GetField("Disposition"));
    Assert.AreEqual(9.42 + 19.42, pc.Deductions, .001);
    Assert.AreEqual((8*15.24) - (9.42 + 19.42),
      pc.NetPay, .001);
  }
```

Для реализации я изначально собирался вызывать `IsInPayPeriod` **из метода** `UnionAffiliation.CalculateDeductions`**. Но, увы, мы только что поместили** `IsInPayPeriod` **в класс** `PaymentClassification` **(см. листинг 27.36). Это было удобно до тех пор, пока данный метод был нужен лишь классам, производным от** `PaymentClassification`**. Но оказалось, что ими дело не исчерпывается. Поэтому я перенес этот метод в класс** `DateUtil`**. В конце концов, это всего лишь функция, определяющая, попадает ли одна дата в интервал между двумя другими (см. листинг 27.40).**

Листинг 27.40. DateUtil.cs

```
using System;

namespace Payroll
{
  public class DateUtil
  {
    public static bool IsInPayPeriod(
      DateTime theDate, DateTime startDate, DateTime endDate)
    {
      return (theDate >= startDate) && (theDate <= endDate);
    }
```

```
    }
  }
```

Теперь наконец мы можем закончить метод `UnionAffiliation.CalculateDeductions`**. Оставляю это вам в качестве упражнения.**

В листинге 27.41 приведена реализация класса `Employee`**.**

Листинг 27.41. Employee.cs

```
using System;

namespace Payroll
{
  public class Employee
  {
    private readonly int empid;
    private string name;
    private readonly string address;
    private PaymentClassification classification;
    private PaymentSchedule schedule;
    private PaymentMethod method;
    private Affiliation affiliation = new NoAffiliation();

    public Employee(int empid, string name, string address)
    {
      this.empid = empid;
      this.name = name;
      this.address = address;
    }

    public string Name
    {
      get { return name; }
      set { name = value; }
    }

    public string Address
    {
      get { return address; }
    }

    public PaymentClassification Classification
    {
      get { return classification; }
      set { classification = value; }
    }

    public PaymentSchedule Schedule
    {
      get { return schedule; }
      set { schedule = value; }
```

```
    }

    public PaymentMethod Method
    {
      get { return method; }
      set { method = value; }
    }

    public Affiliation Affiliation
    {
      get { return affiliation; }
      set { affiliation = value; }
    }

    public bool IsPayDate(DateTime date)
    {
      return schedule.IsPayDate(date);
    }

    public void Payday(Paycheck paycheck)
    {
      double grossPay = classification.CalculatePay(paycheck);
      double deductions =
      affiliation.CalculateDeductions(paycheck);
      double netPay = grossPay - deductions;
      paycheck.GrossPay = grossPay;
      paycheck.Deductions = deductions;
      paycheck.NetPay = netPay;
      method.Pay(paycheck);
    }
    public DateTime GetPayPeriodStartDate(DateTime date)
    {
      return schedule.GetPayPeriodStartDate(date);
    }
  }
}
```

Головная программа

Головную программу теперь можно записать в виде цикла, который читает входные записи из источника и выполняет их. На рис. 27.33 и 27.34 представлены статическая и динамическая диаграммы головной программы. Идея очень проста: объект `PayrollApplication` в цикле запрашивает входные записи, описывающие операции, у объекта `TransactionSource`, а затем передает полученные объекты `Transaction` методу `Execute`. Отметим, что эта картина отличается от диаграммы, изображенной на рис. 27.1, поскольку мы перешли к использованию более абстрактного механизма.

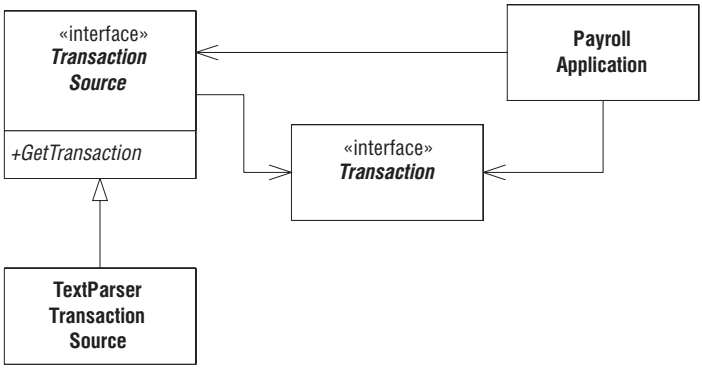


Рис. 27.33. *Статическая модель головной программы*

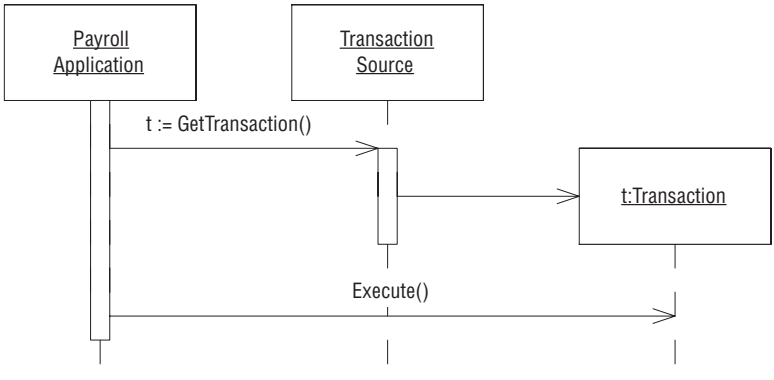


Рис. 27.34. *Динамическая модель головной программы*

`TransactionSource` – это интерфейс, который можно реализовать разными способами. На статической диаграмме показан производный от него класс `TextParserTransactionSource`, который читает данные из текстового потока, разбивая его на входные записи, как описано в прецедентах. Затем объект этого класса создает подходящие объекты `Transaction` и передает их объекту `PayrollApplication`.

Отделение интерфейса от реализации в классе `TransactionSource` позволяет менять источник записей об операциях. Например, можно было бы без труда соединить `PayrollApplication` с `GUITransactionSource` или `RemoteTransactionSource`.

База данных

Теперь, когда большая часть приложения проанализирована, спроектирована и реализована, можно вернуться к вопросу о базе данных. Ясно, что класс `PayrollDatabase` инкапсулирует идею постоянного хра-

нения. Объекты, находящиеся в PayrollDatabase, очевидно, не должны уничтожаться после завершения работы приложения. Как это реализовать? Есть несколько вариантов.

Можно реализовать PayrollDatabase, воспользовавшись какой-нибудь объектно-ориентированной системой управления базами данных (ООСУБД). Это дало бы возможность помещать объекты на постоянное хранение в базу. Нам как проектировщикам не пришлось бы прилагать особых усилий, потому что ООСУБД почти ничего не добавила бы к дизайну. Одно из серьезных достоинств ООСУБД заключается в том, что они мало влияют на объектную модель приложения. С точки зрения дизайна, базы данных как бы и не существует.[1]

Другой вариант – воспользоваться для хранения данных плоскими файлами. На этапе инициализации объект PayrollDatabase мог бы прочитать файл и создать необходимые объекты в памяти. А в конце работы он просто записал бы новую версию файла. Конечно, это решение не подойдет, если в компании сотни или тысячи работников или если необходим оперативный одновременный доступ к базе данных о работниках. Однако для небольшой компании его может оказаться достаточно, и уж точно оно годится как механизм тестирования всех остальных классов приложения без инвестирования в большую СУБД.

Третий вариант – подключить к объекту PayrollDatabase реляционную СУБД (РСУБД). Тогда реализация PayrollDatabase свелась бы к выполнению запросов к СУБД для временного создания необходимых объектов в памяти.

Важно, что любой из этих подходов будет работать. Мы спроектировали приложение так, что оно не знает и не интересуется механизмом реализации базы данных. С точки зрения приложения, база данных – это просто средство для управления постоянным хранением.

Обычно базы данных не следует рассматривать как решающий фактор при проектировании и реализации. Как мы только что показали, этот вопрос можно отложить до последнего момента и трактовать базу данных как деталь реализации.[2] При этом мы оставляем открытым ряд интересных возможностей для реализации постоянного хранения

---

[1] Это чересчур оптимистичный взгляд на вещи. В таком простом приложении, как расчет зарплаты, ООСУБД действительно оказала бы очень небольшое влияние на дизайн программы. По мере усложнения приложений влияние ООСУБД на их дизайн возрастает. И все же оно гораздо меньше, чем в случае РСУБД.

[2] Иногда природа базы данных является одним из требований к приложению. РСУБД предоставляют мощные средства для запросов и составления отчетов, которые могут быть включены в состав требований. Но даже если такое требование сформулировано явно, проектировщик все равно должен отделять дизайн приложения от дизайна базы данных. Дизайн приложения не должен зависеть от конкретной СУБД.

и создания механизмов тестирования остальных частей приложения. Мы также не связываем себя с конкретной технологией баз данных или продуктом. Мы свободны выбрать потребную базу данных, исходя из получившегося дизайна, а впоследствии заменить один продукт на другой.

Заключение

На 32 диаграммах в главах 26 и 27 документированы дизайн и реализация системы расчета заработной платы. По ходу дела мы активно пользовались абстракциями и полиморфизмом. В результате крупные фрагменты системы оказались закрытыми относительно изменения политики расчетов. Например, приложение можно доработать для поддержки работников с поквартальной выплатой, включающей премию. Это потребовало бы *добавлений* в дизайн, но уже существующий код изменился бы мало.

Во время процесса разработки мы редко задавались вопросом, что, собственно, делаем: анализируем, проектируем или реализуем, предпочитая уделять внимание ясности и управлению зависимостями. Мы старались отыскивать абстракции, лежащие в основе приложения. В результате получился добротный дизайн системы расчета зарплаты и набор основных классов, относящихся к данной предметной области.

Об этой главе

Представленные в этой главе диаграммы построены на базе диаграмм Буча в соответствующей главе издания этой книги 1995 года.[1] Эти диаграммы созданы в 1994 году. По мере их создания я писал реализующий код, дабы убедиться, что диаграммы осмысленны. Однако по своему объему он даже не приближается к тому, что представлен здесь. Поэтому те диаграммы не были так же тщательно поверены кодом и тестами. И это заметно.

Эта глава вошла в мою книгу 2002 года.[2] Код для нее я писал на C++ в том же порядке, что и здесь. Написанию промышленного кода всякий раз предшествовали тесты. Во многих случаях тесты создавались инкрементно и эволюционировали вместе с кодом. При написании кода я следовал диаграммам в той мере, в какой это имело смысл. Были случаи, когда это не получалось, и тогда я изменял дизайн.

Впервые это случилось, когда я решил избавиться от нескольких объектов `Affiliation` в объекте `Employee`. Во второй раз я обнаружил, что упустил из виду запись информации о членстве в профсоюзе в классе `ChangeMemberTransaction`.

---

1 [Martin1995]

2 [Martin2002]

Это нормально. Проектирование без обратной связи не может обходиться без ошибок. Именно обратная связь, обеспечиваемая тестами и написанным кодом, помогла эти ошибки найти.

Код в этой главе был переведен с C++ на C# моим соавтором Микой Мартином. Особое внимание уделялось соглашениям и стилю написания программ на C#, чтобы не получился код на несуществующем языке C#++. (Окончательную версию кода можно найти по адресу www.objectmentor.com/PPP/payroll.net.zip.) Диаграммы были оставлены без изменения, разве что отношения композиции были заменены ассоциациями.

Библиография

[Jacobson92] Ivar Jacobson «Object-Oriented Software Engineering: A Use Case Driven Approach», Addison-Wesley, 1992.

[Martin1995] «Designing Object-Oriented C++ Aplications Using the Booch Method», Prentice Hall, 1995.

[Martin2002] «Agile Software Development: Principles, Patterns, and Practices», Prentice Hall, 2002.

IV

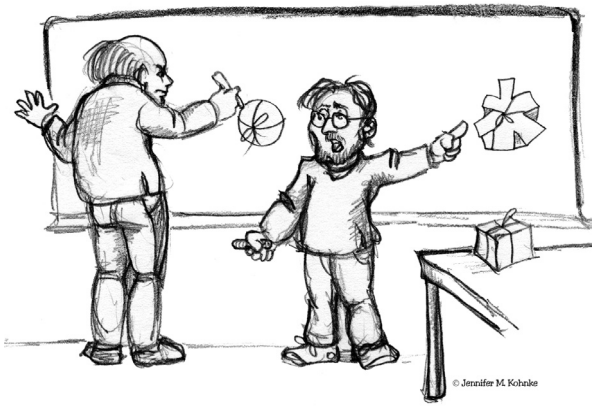
Пакетирование системы расчета заработной платы



В этом разделе мы рассмотрим принципы проектирования, касающиеся разбиения большой программной системы на отдельные пакеты. В главе 28 обсуждаются сами принципы. В главе 29 описывается паттерн, который поможет нам улучшить структуру пакетов. В главе 30 показано, как эти принципы и паттерн можно применить к системе расчета заработной платы.

28

Принципы проектирования пакетов и компонентов



Симпатичный пакет.

Энтони

По мере роста размера и сложности программных приложений возникает необходимость как-то организовывать их на более высоком уровне. Классы – это удобная единица для организации небольших приложений, но в качестве единственного организационного элемента масштабного приложения они слишком мелки. Нужно что-то более крупное, чем класс. Это что-то называется *пакетом*, или *компонентом*.

Пакеты и компоненты

В области программного обеспечения термин *пакет* семантически сильно перегружен. Нас здесь будет интересовать один конкретный вид пакетов, который часто называют *компонентом*. Компонент – это независимо развертываемая *двоичная* единица. В .NET компоненты часто называют *сборками*, а содержатся они обычно в DLL-файле.

Компоненты являются жизненно важными элементами крупных программных систем, поскольку позволяют раскладывать их на более мелкие двоичные комплектующие. Если зависимости между компонентами хорошо продуманы, то при исправлении ошибок и добавлении новых функций можно повторно развертывать только изменившиеся компоненты. Но важнее тот факт, что *дизайн крупных систем критически зависит от удачного дизайна компонентов*, поскольку это позволяет разбить коллектив разработчиков на отдельные команды, каждая из которых занимается только своими компонентами, а не системой в целом.

В UML пакеты можно использовать как контейнеры для групп классов. Такие пакеты могут соответствовать подсистемам, библиотекам или компонентам. Группировка классов в пакеты позволяет рассуждать о дизайне на более высоком уровне абстракции. Если пакеты являются компонентами, то их можно применять для управления разработкой и распространением системы. Наша цель в этой главе – научиться выделять по некоторому критерию группы классов, а затем оформлять эти группы в виде независимо развертываемых компонентов.

Но одни классы часто зависят от других, и зависимости нередко пересекают границы компонентов. Поэтому и между компонентами существуют отношения зависимости. Они выражают высокоуровневую организацию приложения и тоже нуждаются в управлении.

Таким образом, возникает длинный ряд вопросов.

1. Каковы принципы разбиения множества классов на компоненты?
2. Какими принципами проектирования следует руководствоваться, обдумывая отношения между компонентами?
3. Следует ли проектировать компоненты раньше классов (сверху вниз)? Или классы раньше компонентов (снизу вверх)?
4. Как физически оформляются компоненты? В языке C#? В среде разработки?
5. Пусть компоненты созданы; для какой цели мы будем их использовать?

В этой главе формулируется шесть принципов управления содержимым компонентов и зависимостями между ними. Первые три – принципы сцепленности пакетов – помогают при решении вопроса о группировке

классов в пакеты. Оставшиеся три касаются связанности пакетов и помогают решить, как пакеты должны быть взаимосвязаны. В последних двух принципах описываются также *метрики управления зависимостями*, с помощью которых разработчики могут оценивать и количественно характеризовать структуру зависимостей в данном дизайне.

Принципы сцепленности компонентов: детальность

Принципы сцепленности компонентов помогают разработчикам решить, как разбивать множество классов на компоненты. Эти принципы опираются на тот факт, что выявлены по крайней мере какие-то классы и взаимные зависимости между ними. Поэтому подход к разбиению – снизу вверх.

Принцип эквивалентности повторного использования и выпуска (Reuse/Release Equivalence Principle – REP)

> **Единица повторного использования равна единице выпуска.**

Чего вы ожидаете от автора библиотеки классов, которую планируете повторно использовать? Разумеется, хотелось бы иметь хорошую документацию, работающий код, четко специфицированные интерфейсы и т. д. Но это еще не все.

Во-первых, чтобы повторное использование написанного этим автором кода вообще имело смысл, вы хотите иметь гарантию, что автор будет поддерживать свое детище. Ведь если поддерживать код предстоит вам, то придется потратить на его изучение массу времени, а тогда уж лучше спроектировать компонент поменьше, зато под свои конкретные нужды.

Во-вторых, вы хотите, чтобы автор извещал вас обо всех планируемых изменениях в интерфейсе и функциональности кода. Но одного извещения недостаточно. Нужно еще, чтобы автор обеспечил возможность отказаться от использования новых версий. А то, не дай бог, он выпустит новую версию как раз тогда, когда у вас трещит по швам график, или внесет изменения, несовместимые с вашей системой.

Но если вы решите отказаться от новой версии, то автор должен гарантировать поддержку старой версии в течение какого-то периода времени. Может быть, всего три месяца, может быть, год – это вопрос обсуждаемый. Но автор не может просто порвать с вами, отказав во всякой поддержке. Если автор не соглашается поддерживать старые версии, то вы всерьез задумайтесь, стоит ли использовать такой код и целиком зависеть от капризов автора.

Этот вопрос в основном политический. Он касается организационных аспектов процесса поддержки, призванного защитить интересы людей, собирающихся использовать сторонний код. Но политические и организационные вопросы оказывают глубокое влияние на способ пакетирования ПО. Для того чтобы дать гарантии, в которых нуждаются пользователи, автор организует свою программу в виде набора повторно используемых компонентов и присваивает им номера выпуска.

Таким образом, принцип REP гласит, что единица повторного использования, компонент, не может быть меньше единицы выпуска. Все, что предназначено для повторного использования, должно управляться какой-то системой учета выпусков. Не может быть так, чтобы разработчик просто написал какой-то класс и объявил его повторно используемым. О возможности повторного использования можно говорить только тогда, когда внедрена некая система учета выпусков и обеспечены гарантии извещения, надежности и поддержки, в которых так нуждаются потенциальные пользователи.

Принцип REP дает первую подсказку, как разбить дизайн на компоненты. Раз в основе повторного использования должны лежать компоненты, значит, повторно используемые компоненты должны состоять из повторно используемых классов. Поэтому по крайней мере некоторые компоненты должны содержать повторно используемые наборы классов.

Тот факт, что разбиение программы определяется политическими факторами, поначалу выглядит настораживающим, но ведь ПО – не математически безупречная сущность, которую можно структурировать, руководствуясь исключительно математически строгими правилами. ПО – продукт деятельности человека, призванный служить человеческим нуждам. ПО создается людьми и для людей. И если мы собираемся его повторно использовать, то и разбивать на компоненты должны так, чтобы людям было удобно.

И какие из этого можно сделать выводы касательно внутренней структуры компонента? Рассматривать его состав следует с точки зрения потенциальных пользователей. Если компонент содержит ПО, которое допускает повторное использование, то в нем не должно быть частей, спроектированных без учета повторного использования. *Либо все классы, включенные в компонент, можно использовать повторно, либо ни один.*

Далее, сам факт повторной используемости не является единственным критерием; следует еще принимать во внимание, кто будет использовать компонент. Конечно, библиотеку контейнерных классов можно использовать повторно. Как и каркас для построения финансового ПО. Но вряд ли стоит делать их частями одного и того же компонента, потому что далеко не все заинтересованные в библиотеке контейнерных классов нуждаются в финансовом каркасе. Таким образом, желательно, чтобы все классы в компоненте были ориентированы на одну и ту же аудиторию.

Плохо если потенциальные пользователи обнаружат в компоненте как необходимые, так и совершенно бесполезные для них классы.

Принцип совместного повторного использования (Common Reuse Principle – CRP)

> **Все классы внутри компонента используются совместно. Если вы можете повторно использовать один класс, то можете использовать и все остальные.**

Этот принцип помогает решить, какие классы включать в компонент. CRP гласит, что классы, которые, скорее всего, будут использоваться совместно, принадлежат одному компоненту.

Классы редко используются изолированно. Обычно повторно используемые классы кооперируются с другими классами, являющимися частями одной и той же абстракции. CRP утверждает, что такие классы должны входить в один компонент. Естественно ожидать, что между классами в этом компоненте будет много взаимных зависимостей. Простой пример – класс контейнера и сопутствующие ему итераторы. Эти классы используются совместно, потому что они тесно связаны. И, значит, должны принадлежать одному компоненту.

Но CRP говорит не только о том, какие классы помещать в один компонент, а также о том, какие *не* помещать туда. Если один компонент пользуется другим, то между ними образуется зависимость. Может случиться так, что первому компоненту нужен всего один класс из второго. Но от этого зависимость отнюдь не становится слабее. Первый компонент все равно зависит от второго. При каждом новом выпуске второго компонента придется заново проверять и выпускать первый. И это справедливо даже в том случае, когда причиной нового выпуска используемого компонента является изменение в классе, который использующему компоненту не интересен.

Очень часто компоненты оформляются в виде DLL-библиотек. Если используемый компонент выпущен в виде DLL, то использующий его код зависит от этой DLL в целом. Любая модификация данной DLL, даже если речь идет об изменении класса, который никак не касается использующей программы, обязательно приводит к выпуску новой версии DLL. Новую DLL необходимо повторно развертывать и заново тестировать все использующие ее программы.

Поэтому если уж мой код зависит от некоторого компонента, то я хотел бы, чтобы он зависел от всех классов в этом компоненте. Иными словами, я хочу, чтобы классы, включенные в компонент, были неразделимы, чтобы не получилось так, что клиент зависит от одних и не зависит от других классов. Иначе мне придется заниматься повторным

развертыванием и проверкой чаще, чем необходимо, тратя на это несоразмерно много времени.

Итак, принцип CRP даже больше говорит о том, какие классы не следует включать в один компонент, чем о том, какие следует. Смысл CRP в том, что классы, не являющиеся тесно связанными, не должны находиться в одном компоненте.

Принцип общей закрытости (Common Closure Principle – CCP)

> **Все классы внутри компонента должны быть закрыты относительно изменений одного и того же вида. Изменение, затрагивающее компонент, должно затрагивать все классы в этом компоненте и только в нем.**

Это не что иное как принцип единственной обязанности (SRP) в применении к компонентам. SRP говорит, что у класса не должно быть более одной причины для изменения, а CCP – что то же самое справедливо и для компонента.

В большинстве приложений пригодность для сопровождения важнее повторной используемости. Если код приложения приходится изменять, то хотелось бы, чтобы все изменения были локализованы в одном компоненте, а не разбросаны по многим, поскольку это позволит ограничиться повторным развертыванием только одного изменившегося компонента. Другие компоненты, не зависящие от изменившегося, не нужно ни развертывать, ни заново проверять.

Принцип CCP рекомендует собирать в одном месте классы, которые могут изменяться по одним и тем же причинам. Если два класса настолько тесно связаны, физически или концептуально, что всегда изменяются вместе, то их естественно поместить в один компонент. Это уменьшит трудозатраты на повторный выпуск, проверку и распространение ПО.

Этот принцип неотъемлем от принципа открытости/закрытости (OCP). Ибо именно о «закрытости» в смысле OCP и идет речь в CCP. OCP гласит, что классы должны быть закрыты для модификации, но открыты для расширения. Но, как мы узнали, достичь стопроцентной закрытости невозможно. Закрытость должна быть стратегической целью. Мы проектируем систему так, чтобы она была закрыта относительно типичных изменений, с которыми нам приходилось ранее сталкиваться.

CCP развивает эту идею, рекомендуя включать классы, открытые для изменений одного и того же вида, в один компонент. Тогда в случае появления новых требований возрастают шансы на то, что удастся ограничиться изменением минимального числа компонентов.

Резюме

В прошлом наши представления о сцепленности были гораздо проще. Мы привыкли считать сцепленность свойством модуля, означающим, что он выполняет одну и только одну функцию. Однако три принципа сцепленности компонентов описывают гораздо более сложные отношения. Принимая решение о том, какие классы включить в компонент, мы должны помнить, что повторная используемость и удобство разработки – силы, направленные в противоположные стороны.

Уравновешивание этих сил с учетом потребностей приложения – нетривиальная задача. К тому же равновесие почти всегда динамическое, то есть разбиение, признанное оптимальным сегодня, в будущем году может оказаться непригодным. Поэтому состав компонента флуктуирует и эволюционирует со временем по мере того, как удобство разработки, являющееся одной из главных целей проекта, уступает место возможности повторного использования.

Принципы связанности компонентов: устойчивость

Следующие три принципа относятся к связям между компонентами. Здесь мы снова сталкиваемся с противоречием между удобством разработки и логическим дизайном. На архитектуру компонента оказывают влияние как технические и политические факторы, так и соображения изменчивости.

Принцип ацикличности зависимостей (Acyclic Dependencies Principle – ADP)

> **В графе зависимостей между компонентами не должно быть циклов.**



Бывало с вами так, что вы целый день упорно трудились, отладили какую-то часть программы, а придя на следующее утро, обнаружили, что она уже не работает? А почему не работает? Потому, что кто-то вчера задержался дольше, чем вы, и изменил нечто, от чего зависит ваша часть! Я называю это «синдромом следующего утра».

«Синдром следующего утра» характерен для ситуаций, когда несколько разработчиков модифицируют одни и те же ис-

ходные файлы. В относительно небольших проектах, где разработчиков мало, эта проблема не слишком серьезна. Но по мере того как размер проекта и, соответственно, команды разработчиков растет, каждое следующее утро может приносить все новые кошмары. Довольно часто бывает, что проходит неделя за неделей, а собрать стабильно работающую версию проекта никак не удается. Все время уходит на то, чтобы вносить в свой код бесконечные изменения, стараясь согласовать его с последними изменениями, внесенными кем-то другим.

За прошедшие десятилетия выработалось два решения этой проблемы: еженедельная сборка и принцип ADP. Оба пришли из индустрии телекоммуникаций.

Еженедельная сборка. Еженедельные сборки обычно применяются в проектах среднего размера. Работает это так: первые четыре дня недели все разработчики игнорируют друг друга. Каждый работает со своей копией кода, не думая об интеграции с коллегами. Но в пятницу все изменения собираются вместе и производится сборка системы. Достоинство такого подхода в том, что четыре дня из пяти разработчикам позволено жить в изолированных мирах. Ну а недостаток, конечно, в больших расходах на интеграцию, которые приходится нести каждую пятницу.

К сожалению, с ростом проекта закончить интеграцию в пятницу становится все сложнее. Приходится переносить ее на субботу. Нескольких таких суббот достаточно, чтобы разработчики пришли к выводу, что интеграцию следует начинать в четверг. И постепенно начало интеграции сползает к середине недели.

По мере того как все больше усилий тратится не на разработку, а на интеграцию, эффективность команды падает. И в конце концов ситуация ухудшается настолько, что сами разработчики или руководители проекта объявляют о переходе на сборку раз в две недели. На некоторое время проблему удается сгладить, но продолжительность интеграции с разрастанием проекта по-прежнему растет.

Кончается это кризисом. Чтобы обеспечивать приемлемую эффективность, цикл сборки приходится раз от разу увеличивать. Но увеличение промежутков между сборками несет в себе риск для проекта. Интегрировать и тестировать становится все труднее, и команда теряет преимущество оперативности.

Исключение циклических зависимостей. Решение проблемы заключается в том, чтобы выделить независимо выпускаемые компоненты. Единицей работы становится компонент, за который отвечает один разработчик или их группа. Когда компонент готов, он выпускается и передается в пользование другим разработчикам. Ему присваивают номер выпуска и помещают в каталог, доступный всем командам. Разработчики продолжают модифицировать его код, хранящийся в закрытых для других областях, а все остальные работают с выпущенной версией.

Когда выпускается новая версия компонента, прочие команды решают, стоит ли им сразу переходить на нее. Если они не захотят, то будут и дальше работать со старой версией. А когда решат, что готовы, перейдут на новую.

Таким образом, ни одна команда не зависит от милостей другой. Изменения, внесенные в один компонент, необязательно сразу же оказывают влияние на другие команды. Каждая команда сама решает когда переходить на новые версии используемых ею компонентов. И интеграция происходит постепенно. Не существует такого момента когда все разработчики должны собраться вместе и интегрировать все сделанное.

Этот очень простой и рациональный процесс широко распространен. Но, чтобы он работал, необходимо *управлять* структурными зависимостями компонентов. *Не должно быть циклов*. Если образуется циклическая зависимость, то синдрома следующего утра не избежать.

Рассмотрим диаграмму компонентов на рис. 28.1. Это довольно типичная структура компонентов, собранных в приложение. Назначение этого приложения сейчас неважно. А важна лишь структура зависимостей между компонентами. Обратите внимание, что она представляет собой *ориентированный граф*, или *орграф*. Компоненты являются его *вершинами*, а отношения зависимости – *ориентированными ребрами*.

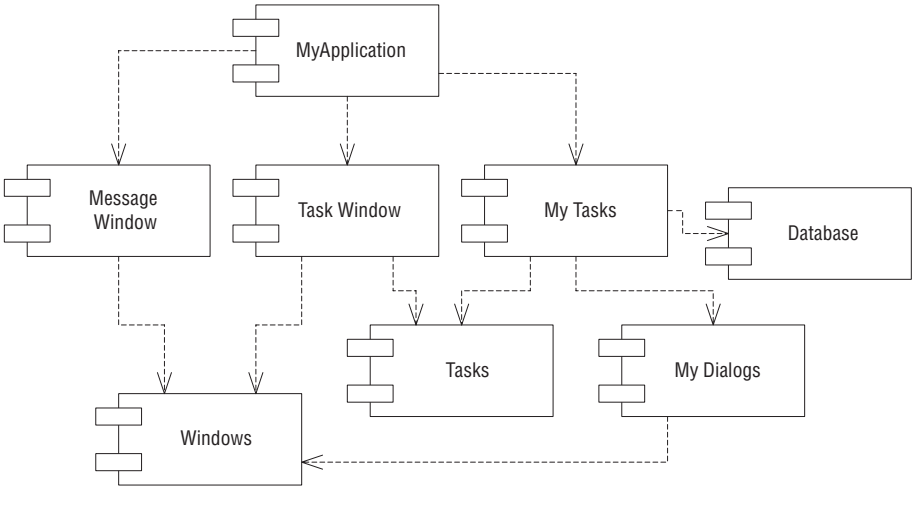


***Рис. 28.1.** Компоненты представлены ориентированным ациклическим графом*

Отметим еще одну особенность. С какого бы компонента ни начать, невозможно, следуя вдоль ребер, вернуться к тому же компоненту. В графе нет циклов. Это *ациклический ориентированный граф*.

Теперь посмотрим, что произойдет, когда команда, отвечающая за компонент `MyDialogs`, выпустит его новую версию. Легко понять, кто при

этом будет затронут; достаточно проследовать по стрелкам, обозначающим зависимости, в обратном направлении. В данном случае затронуты компоненты MyTasks и MyApplication. Разработчики, занятые этими компонентами, могут решить, стоит ли сразу интегрировать новую версию MyDialogs.

Выпуск MyDialogs не оказывает влияния на большинство других компонентов системы. Они просто ничего не знают о MyDialogs, им безразлично, что он изменился. И это хорошо, так как означает, что влияние выпуска новых версий MyDialogs сравнительно невелико.

Когда разработчики, работающие над компонентом MyDialogs, захотят протестировать свой компонент, им нужно будет лишь собрать его вместе с той версией компонента Windows, которую они в данный момент используют. Больше никакие компоненты не задействованы. И это хорошо, так как означает, что разработчикам MyDialogs для настройки тестовой среды придется сделать сравнительно немного, ибо количество принимаемых во внимание переменных невелико.

Когда наступает момент выпуска всей системы, это делается снизу вверх. Сначала компилируется, тестируется и выпускается компонент Windows, потом MessageWindow, MyDialogs, Task, TaskWindow и Database, MyTasks и наконец MyApplication. Процесс абсолютно прозрачен и удобен. Мы знаем, как собирать систему, потому что понимаем зависимости между ее частями.

Влияние циклов на граф зависимостей между компонентами. Предположим, что появилось новое требование, заставляющее изменить один из классов в компоненте MyDialogs, так что он начинает использовать некоторый класс в MyApplication. При этом создается цикл зависимостей, как показано на диаграмме на рис. 28.2.

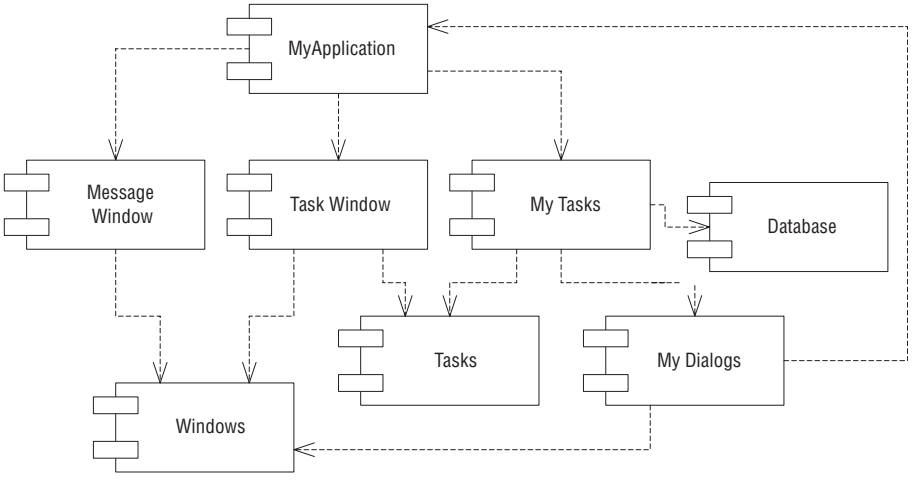


***Рис. 28.2.** Диаграмма компонентов, содержащая цикл*

Наличие этого цикла сразу же создает проблемы. Например, разработчики компонента MyTasks знают, что для выпуска необходимо обеспечить совместимость с Tasks, MyDialogs, Database и Windows. Но, коль скоро есть цикл, то следует поддержать совместимость также с MyApplication, TaskWindow и MessageWindow. Таким образом, MyTasks теперь зависит от *всех компонентов системы*. Из-за этого выпустить версию MyTasks становится очень трудно. Та же беда преследует компонент MyDialogs. Фактически наличие цикла заставляет выпускать компоненты MyApplication, MyTasks и MyDialogs только совместно. По существу, они превратились в один большой компонент. Все работающие над этими компонентами снова столкнутся с синдромом следующего утра. Они все время будут наступать друг другу на пятки, поскольку обязаны пользоваться одинаковыми версиями разрабатываемых ими компонентов.

Но неприятности на этом не заканчиваются. Посмотрим, что произойдет когда мы захотим протестировать компонент Mydialogs. Выясняется, что необходимо сослаться на все остальные компоненты системы, включая и Database. А это означает, что для тестирования одного лишь компонента MyDialogs придется произвести *полную сборку*. Это никуда не годится.

Вы никогда не задавались вопросом, почему для выполнения простого автономного теста одного из своих классов приходится ссылаться на такое большое количество чужих библиотек? Вероятно, потому, что в графе зависимостей есть циклы. Из-за таких циклов очень трудно изолировать модули. Автономное тестирование и выпуск версий становятся сложными и подверженными ошибкам задачами. Да и время компиляции растет в геометрической прогрессии с увеличением числа модулей. Более того, когда в графе зависимостей есть циклы, оказывается крайне сложно понять, в каком порядке собирать компоненты. Собственно, правильного порядка может не быть вообще, что ведет к весьма неприятным последствиям.

Разрыв цикла. Всегда есть возможность разорвать цикл зависимостей и вернуться к ациклическому орграфу. Основных способов два.

1. Применить принцип инверсии зависимости (DIP). В случае, изображенном на рис. 28.2, можно было бы ввести абстрактный базовый класс с интерфейсом, необходимым компоненту MyDialogs. Затем мы поместили бы этот класс в MyDialogs и унаследовали ему в MyApplication. Тем самым зависимость между MyDialogs и MyApplication инвертируется и цикл разрывается. См. рис. 28.3.

   Отметим еще раз, что в имя интерфейса включено имя клиента, а не сервера. Это подтверждает тезис о том, что интерфейсы принадлежат клиентам.

2. Создать новый компонент, от которого зависят MyDialogs и MyApplication. Перенести в этот компонент те классы, от которых зависят оба компонента. См. рис. 28.4.

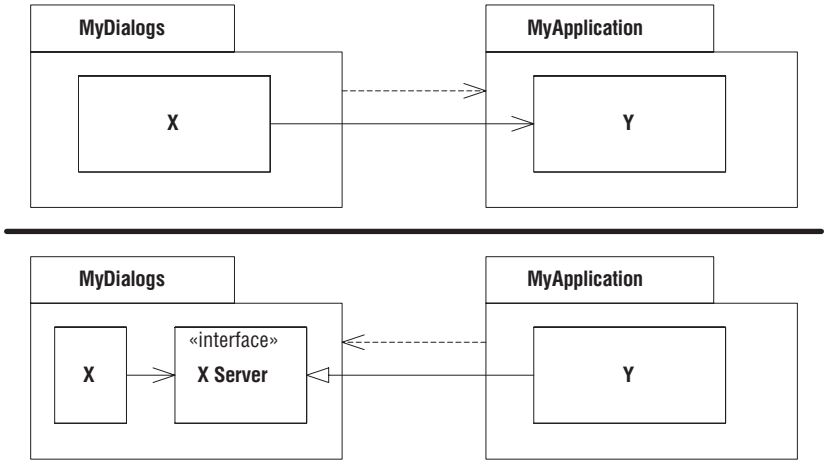


Рис. 28.3. *Разрыв цикла с помощью инверсии зависимости*

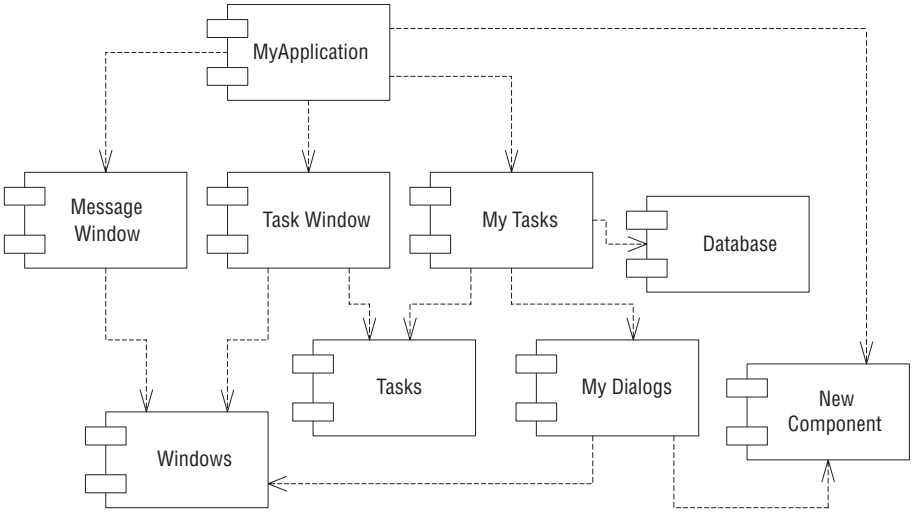


Рис. 28.4. *Разрыв цикла путем добавления нового компонента*

Второе решение предполагает, что структуру компонентов можно изменять при появлении новых требований. И действительно, с ростом приложения структура зависимостей между компонентами флуктуирует и развивается. Поэтому нужно постоянно следить за тем, чтобы в ней не появлялись циклы. Как только образуется цикл, его следует немедленно разорвать тем или иным способом. Иногда это приводит к созданию нового компонента и росту графа зависимостей.

Сверху вниз или снизу вверх? Рассмотренные выше вопросы неизбежно приводят к выводу о том, что структуру компонентов невозможно

спроектировать сверху вниз, не имея самого кода. Эта структура эволюционирует по мере роста и изменения системы.

Возможно, кому-то это покажется противоречащим здравому смыслу. Мы привыкли думать, что декомпозиция на такие крупные составные части, как компоненты, является также высокоуровневой *функциональной* декомпозицией. Глядя на структуру зависимостей между компонентами, мы невольно считаем, что сами компоненты как-то связаны с функциями системы. Но, хотя компоненты действительно предоставляют друг другу службы и функции, этим дело не ограничивается.

Структура зависимостей между компонентами – это карта *собираемости* системы. Именно поэтому ее нельзя полностью спроектировать в начале работы над проектом. И именно поэтому она не строго следует функциональной декомпозиции. По мере того как на ранних стадиях проектирования и реализации образуются все новые классы, растет потребность в управлении зависимостями, чтобы работу можно продолжать без синдрома следующего утра. К тому же мы хотим, чтобы изменения были как можно более локальными, поэтому начинаем обращать внимание на принципы SRP и CCP и размещать рядом классы, которые, скорее всего, будут изменяться вместе.

Приложение продолжает расти, и возникает интерес к созданию повторно используемых элементов. Теперь принцип CRP начинает диктовать состав компонентов. Наконец появляются циклы, мы применяем принцип ADP, и граф зависимостей между компонентами видоизменяется по причинам, относящимся скорее к структуре зависимостей, а не к функциям.

Попытавшись спроектировать структуру зависимостей между компонентами раньше, чем классы, мы, скорее всего, потерпели бы жестокую неудачу. На этой стадии практически ничего не известно об общей закрытости, о повторно используемых элементах, и почти наверняка мы создали бы компоненты, порождающие циклические зависимости. Поэтому структура зависимостей между компонентами должна расти и развиваться параллельно логическому проектированию системы.

Впрочем, на совершенствование структуры компонентов до такой степени когда возможна разработка с участием нескольких команд, требуется не так уж много времени. Как только это произойдет, команды могут сконцентрироваться на собственных компонентах. Взаимодействие между командами можно ограничить границами компонентов. В этом случае команды могут параллельно работать над одним проектом, практически не мешая друг другу.

Однако следует помнить, что структура компонентов продолжает изменяться все время, пока идет разработка. Это препятствует полной изоляции команд, работающих над разными компонентами. Они вынуждены согласовывать свои действия всякий раз, как компоненты перестают мирно уживаться друг с другом.

Принцип устойчивых зависимостей (Stable–Dependencies Principle – SDP)

> **Зависимости должны быть направлены в сторону устойчивости.**

Дизайн не может постоянно оставаться статичным. Если систему приходится сопровождать, то некоторая изменчивость неизбежна. Достигается это путем следования принципу CCP, который побуждает создавать компоненты, восприимчивые к изменениям определенного вида. Такие компоненты *изначально* спроектированы с учетом изменчивости; мы *ожидаем*, что они будут изменяться.

Компонент, который предполагается изменять, не должен зависеть от компонентов, изменение которых затруднено! Иначе изменчивый компонент тоже будет трудно модифицировать.

Но разработка ПО – дело своеобразное; бывает так, что вы разработали модуль, который изменять легко, а кто-то другой сделал его трудноизменяемым, просто «повесив» на него зависимость. Ни одна строка в исходном коде вашего модуля не изменилась, но внезапно он стал трудноизменяемым. Принцип SDP гласит, что от модулей, которые спроектированы так, чтобы облегчить внесение изменений, не должны зависеть модули, которые изменить труднее.

Устойчивость. Что понимается под устойчивостью? Поставьте монетку на ребро. В таком положении она устойчива? Нет, конечно. Однако если ее не трогать, то она может оставаться в этом положении очень долго. Поэтому устойчивость напрямую не связана с частотой изменений. Положение монеты не изменяется, но назвать его устойчивым язык не поворачивается.

В технике «устойчивость» определяется как «способность объекта противостоять усилиям, стремящимся вывести его из исходного состояния равновесия». Устойчивость связана с количеством работы, необходимой для изменения. Монетка неустойчива, потому что для ее опрокидывания нужно приложить совсем немного усилий. С другой стороны, стол очень устойчив – чтобы его перевернуть, придется здорово потрудиться.

Какое отношение все это имеет к программному обеспечению? Есть много факторов, затрудняющих изменение ПО: его размер, сложность, ясность и т. д. Но мы не будем на них останавливаться, а поговорим совсем о другом. Верный способ сделать программный компонент трудноизменяемым – создать много зависящих от него компонентов. Компонент, в сторону которого направлено много зависимостей, очень устойчив, потому что для увязки любых изменений в нем с зависимыми компонентами требуется много работы.

На рис. 28.5 показан устойчивый компонент X. От него зависят три компонента, поэтому есть три причины его не изменять. Мы говорим, что он *несет ответственность* за эти три компонента. С другой стороны, сам X ни от чего не зависит, поэтому для его изменения нет внешних стимулов. Такой компонент мы называем *независимым*.

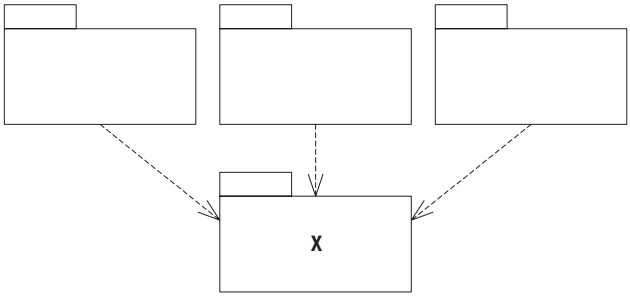


Рис. 28.5. *X – устойчивый компонент*

С другой стороны, на рис. 28.6 показан очень неустойчивый компонент Y, от которого ничто не зависит; мы называем его *неответственным*. Однако Y сам зависит от трех компонентов, поэтому стимулы к его изменению могут исходить из трех внешних источников. Мы говорим, что Y – зависимый компонент.

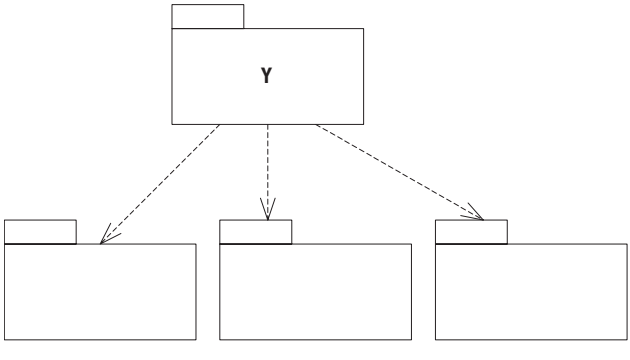


Рис. 28.6. *Y – неустойчивый компонент*

Метрики устойчивости. Как измеряется устойчивость компонента? Один из способов – подсчитать количество входящих и исходящих зависимостей. Вместе они позволяют вычислить *позиционную* устойчивость компонента:

- *Ca* (входящие связи): количество классов вне данного компонента, зависящих от классов внутри компонента.
- *Ce* (исходящие связи): количество классов внутри данного компонента, зависящих от классов вне компонента.

- I (неустойчивость): $I = \dfrac{Ce}{Ca + Ce}$

  Эта величина изменяется в диапазоне [0,1]. Если $I = 0$, то компонент максимально устойчив если $I = 1$ – максимально неустойчив.

Метрики *Ca* и *Ce* вычисляются путем подсчета количества *классов*, находящихся вне данного компонента и зависящих от классов внутри компонента. Рассмотрим пример на рис. 28.7:

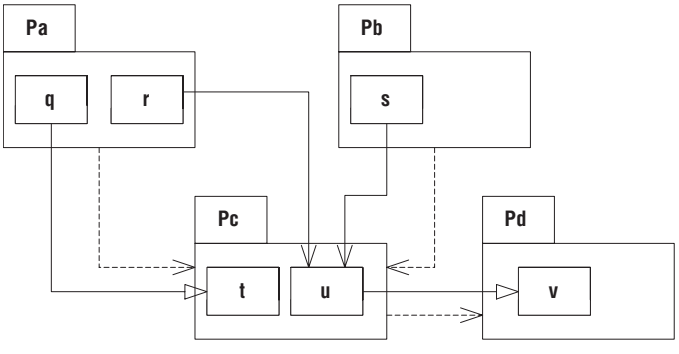


Рис. 28.7. *Вычисление Ca, Ce и I*

Пунктирные стрелки, соединяющие компоненты, обозначают зависимости. Отношения между классами внутри этих компонентов показывают, как эти зависимости реализованы. Существуют отношения наследования и ассоциации.

Пусть мы хотим вычислить устойчивость компонента `Pc`. Имеется три класса вне `Pc`, зависящих от классов внутри `Pc`. Следовательно, $Ca = 3$. Есть также один класс вне `Pc`, от которого зависят классы внутри `Pc`. Поэтому $Ce = 1$ и $I = 1/4$.

В языке C# такие зависимости обычно представляются предложениями `using`. Метрику I проще всего вычислить если исходный код организован так, что в каждом исходном файле находится по одному классу. Тогда в C# для вычисления I нужно подсчитать число предложений `using` и полных имен.

Если величина I равна 1, значит, не существует компонентов, зависящих от данного ($Ca = 0$), а сам этот компонент зависит от других ($Ce > 0$). Компонент настолько неустойчив, насколько это вообще возможно; он является *неответственным* и *зависимым*. Отсутствие зависящих от него компонентов означает, что у него нет причин *не* изменяться, и компоненты, от которых зависит он сам, могут создать массу причин для изменения.

С другой стороны, если I равно 0, то существуют компоненты, зависящие от данного ($Ca > 0$), но сам он ни от кого не зависит ($Ce = 0$). Компо-

нент *ответственный* и *независимый*. Такой компонент максимально устойчив. Зависящим от него трудно измениться, а зависимостей, которые заставили бы его самого измениться, не существует.

Согласно принципу SDP метрика *I* любого компонента должна быть больше метрики *I* компонентов, от которых он зависит. Иными словами, *I* должна уменьшаться в направлении зависимости.

Компоненты с различной устойчивостью. Если бы все компоненты системы были максимально устойчивы, то система оказалась бы неизменяемой. Такая ситуация нежелательна. Проектировать структуру компонентов следует так, чтобы одни компоненты были устойчивы, а другие – нет. На рис. 28.8 представлена идеальная конфигурация системы с тремя компонентами.

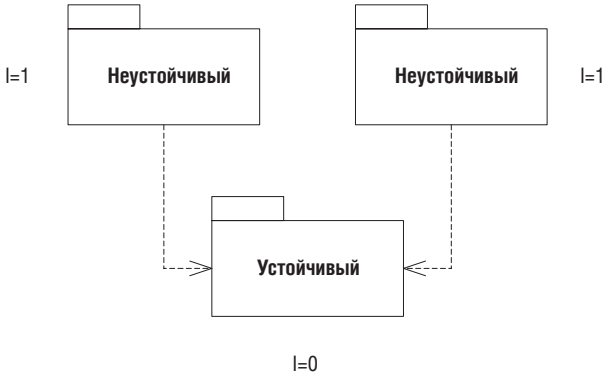


***Рис. 28.8.** Идеальная конфигурация компонентов*

Изменяемые компоненты находятся наверху и зависят от устойчивого компонента внизу. Размещать неустойчивые компоненты в верхней части диаграммы – полезное соглашение, поскольку в таком случае любая стрелка, направленная *вверх*, будет нарушать принцип SDP.

На рис. 28.9 показано, как можно нарушить принцип SDP. Компонент `Flexible` спроектирован с учетом простоты изменения. Мы хотим, чтобы он был неустойчивым, то есть метрика *I* была близка к 0. Однако человек, работающий над компонентом `Stable`, сделал его зависимым от `Flexible`. это нарушает SDP, поскольку величина *I* для `Stable` существенно меньше, чем для `Flexible`. А в результате изменить `Flexible` стало сложно, ибо это повлечет за собой пересмотр `Stable` и всех компонентов, которые от него зависят.

Чтобы исправить положение, необходимо как-то разорвать зависимость `Stable` от `Flexible`. Почему такая зависимость вообще существует? Допустим, что внутри `Flexible` есть класс `C`, которым пользуется некоторый класс `U` внутри `Stable`. См. рис. 28.10.

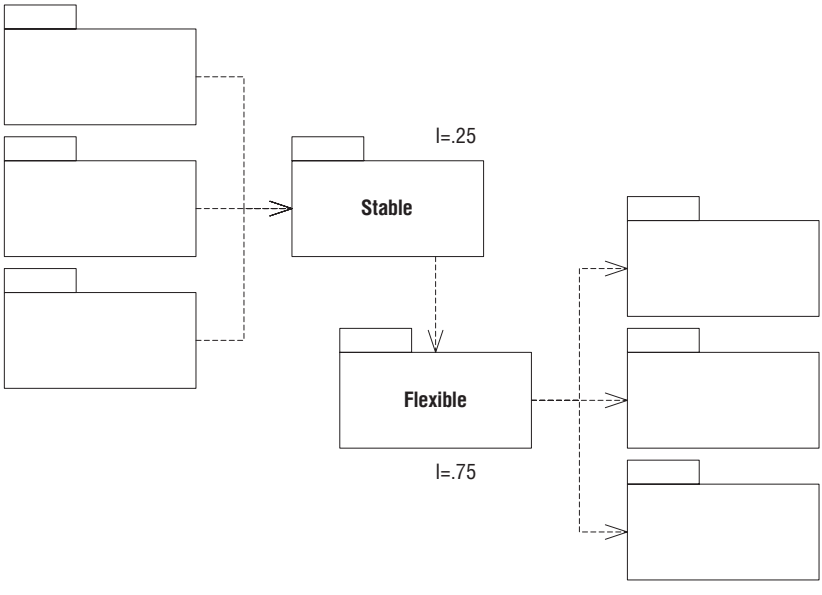


Рис. 28.9. *Нарушение принципа SDP*

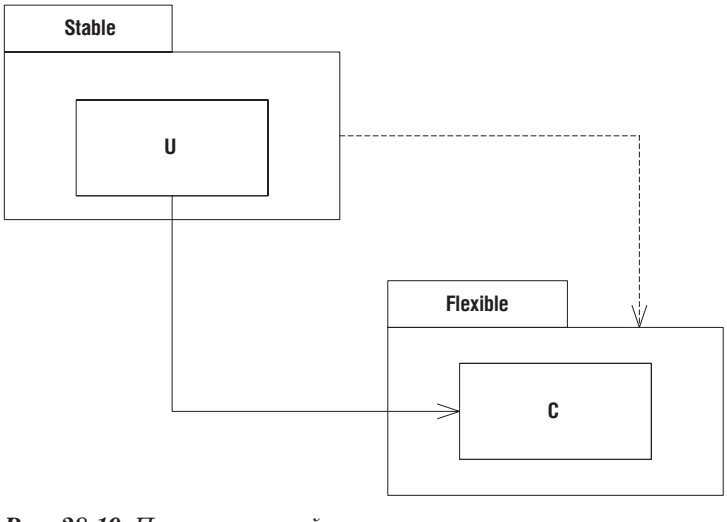


Рис. 28.10. *Причина плохой зависимости*

Чтобы разрешить проблему, воспользуемся принципом DIP. Создадим интерфейс `IU` и поместим его в компонент с именем `UInterface`. Позаботимся о том, чтобы в интерфейсе были объявлены все методы, необходимые `U`. Теперь унаследуем `C` от этого интерфейса (рис. 28.11). В результате зависимость `Stable` от `Flexible` разорвана и оба компонента зависят

от `UInterface`. Компонент `UInterface` очень устойчив ($I = 0$), а `Flexible` сохранил необходимую нам неустойчивость ($I = 1$). Все зависимости направлены в сторону *уменьшения I*.

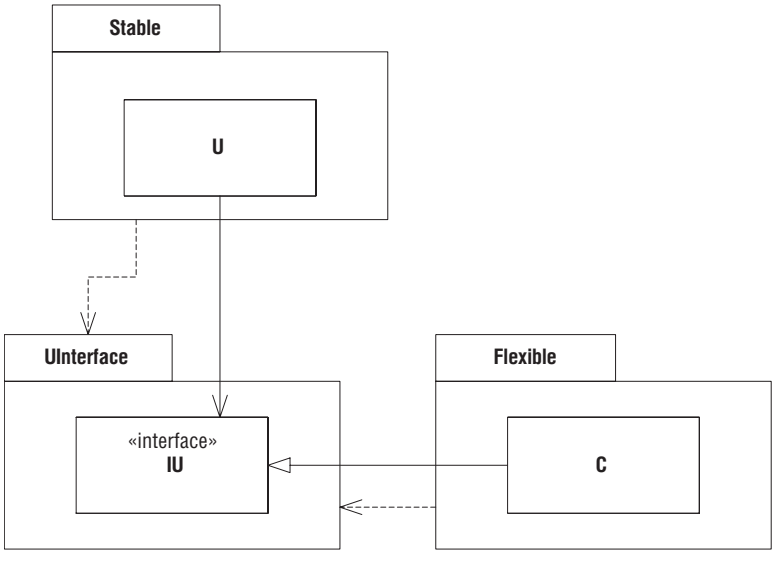


Рис. 28.11. *Исправление нарушения принципа SDP путем применения DIP*

Место дизайна верхнего уровня. Некоторые части системы изменяются не слишком часто. Они представляют архитектуру и проектные решения верхнего уровня. Мы не хотим, чтобы эти архитектурные решения были чрезмерно изменчивы. Поэтому части системы, инкапсулирующие высокоуровневый дизайн системы, следует помещать в устойчивые компоненты ($I = 0$). Неустойчивые компоненты ($I = 1$) должны содержать только те части, которые, скорее всего, будут изменяться.

Однако если высокоуровневый дизайн поместить в устойчивые компоненты, то его исходный код будет трудно изменять, что сделает дизайн негибким. Как добиться, чтобы максимально устойчивый компонент ($I = 0$) оставался достаточно гибким и мог выдерживать изменения? Ответ дает принцип OCP, гласящий, что возможно и даже желательно создавать классы, которые можно расширять без модификации. Какие классы удовлетворяют этому принципу? *Абстрактные*.

Принцип устойчивых абстракций (Stable–Abstractions Principle – SAP)

> **Компонент должен быть столь же абстрактным, сколь и устойчивым.**

Этот принцип устанавливает соотношение между устойчивостью и абстрактностью. Он говорит, что устойчивый компонент должен быть также и абстрактным, чтобы устойчивость не препятствовала его расширению. С другой стороны, он утверждает, что неустойчивый компонент должен быть конкретным, так как сама его неустойчивость позволяет легко изменять конкретный код.

Итак, если компонент предполагается сделать устойчивым, то он должен состоять из абстрактных классов, допускающих расширение. Расширяемые устойчивые компоненты оказываются гибкими и не налагают чрезмерных ограничений на дизайн.

В сочетании принципы SAP и SDP дают аналог принципа DIP для компонентов. Действительно, SDP говорит, что зависимости должны быть направлены в сторону устойчивости, а SAP – что из устойчивости вытекает абстрактность. Следовательно, зависимости направлены в сторону абстрактности.

Однако принцип DIP относится к классам. Класс не допускает никаких градаций: он либо абстрактный, либо нет. Принципы же SDP и SAP имеют дело с компонентами, которые могут быть отчасти абстрактными и отчасти устойчивыми.

Измерение абстрактности. Метрика A, являющаяся мерой абстрактности компонента, равна отношению количества абстрактных классов к общему количеству классов в компоненте. Пусть

Nc – количество классов в компоненте,

Na – количество абстрактных классов в нем же. Напомним, что абстрактным называется класс, содержащий хотя бы один абстрактный метод, так что создать его экземпляр невозможно.

Тогда абстрактностью A называется величина $A = \frac{Na}{Nc}$.

Метрика A изменяется от 0 до 1, причем 0 означает, что в компоненте вообще нет абстрактных классов, а 1 – что компонент содержит только абстрактные классы.

Главная последовательность. Теперь мы можем определить соотношение между устойчивостью (I) и абстрактностью (A). Нарисуем график, на котором по вертикальной оси отложим A, а по горизонтальной – I. Если представить на этом графике два «хороших» вида компонентов, то выяснится, что максимально устойчивые и абстрактные компоненты находятся в левом верхнем углу, в точке (0,1), а максимально неустойчивые и конкретные – в правом нижнем углу, в точке (1,0). См. рис. 28.12.

Но не все компоненты оказываются в этих точках. У разных компонентов степени абстрактности и устойчивости различны. Например, очень часто бывает, что один абстрактный класс является производным от другого, тоже абстрактного. Тогда подкласс, несмотря на свою абстрактность, имеет зависимость, и, следовательно, будучи максималь-

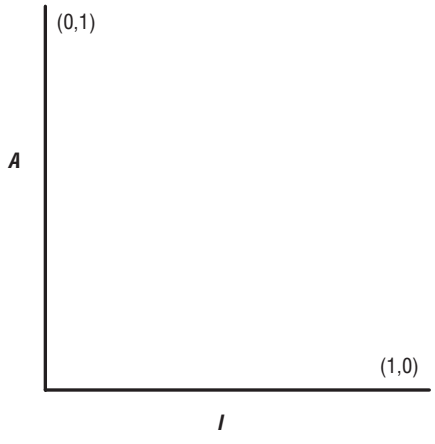


Рис. 28.12. *График зависимости A от I*

но абстрактным, он в то же время не является максимально устойчивым. Наличие зависимости уменьшает степень абстрактности.

Поскольку мы не можем гарантировать, что любой компонент находится либо в точке (0,1), либо в точке (1,0), то должны предположить наличие кривой на графике *A*/*I*, которая определяет разумные места расположения компонентов. Составить представление о том, как выглядит эта кривая, можно, найдя области, где *не* должно быть никаких компонентов, то есть зоны *исключения*. См. рис. 28.13.

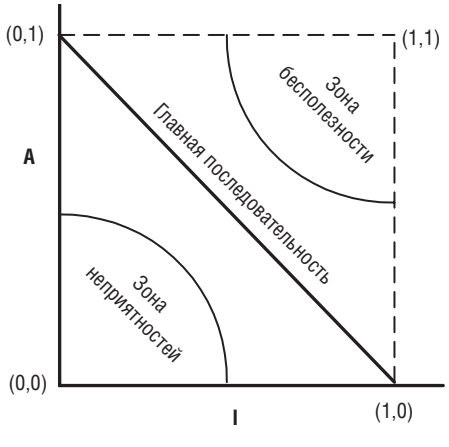


Рис. 28.13. *Зоны исключения*

Рассмотрим компонент в окрестности точки (0,0). Он в высокой степени устойчив и конкретен. Такой компонент нежелателен, потому что он слишком жесткий. Расширить его нельзя, так как он не абстракт-

ный. И изменить трудно в силу его устойчивости. Поэтому в обычной ситуации хорошо спроектированных компонентов вблизи точки (0,0) не будет. Окрестность этой точки является зоной исключения, или *зоной неприятностей*.

Следует отметить, что иногда компоненты все же попадают в зону неприятностей. Примером может служить компонент, представляющий схему базы данных. Схемы баз данных известны своей изменчивостью, очень конкретны, и от них зависят многие компоненты. Именно поэтому так трудно организовать интерфейс между объектно-ориентированными приложениями и базами данных, а изменение схемы приносит так много неприятностей.

Другой пример компонента в окрестности точки (0,0) – библиотека конкретных вспомогательных классов. Хотя для такого компонента метрика *I* равна 1, на деле он может оказаться неизменяемым. Возьмем, к примеру, компонент «строка». Хотя все классы в нем конкретны, изменению он не подлежит. Помещать такие компоненты в окрестность точки (0,0) безопасно, потому что вероятность их изменения мала. Можно даже ввести третью ось графика – изменчивость. В таком случае график на рис. 28.13 соответствует плоскости, для которой изменчивость равна 1.

Теперь рассмотрим компонент в окрестности точки (1,1). Это место нежелательно, потому что находящийся там компонент является максимально абстрактным и в то же время от него никто не зависит. Такие компоненты бесполезны, поэтому данная зона называется *зоной бесполезности*.

Представляется очевидным, что изменчивые компоненты должны быть максимально удалены от обеих зон исключения. Геометрическое место точек, максимально удаленных от обеих зон, – прямая, соединяющая точки (1,0) и (0,1). Она называется *главной последовательностью*.[1]

Компонент, находящийся на главной последовательности, не является ни «слишком абстрактным» (что помешало бы устойчивости), ни «слишком неустойчивым» (что помешало бы абстрактности). Он не является ни бесполезным, ни источником неприятностей. Он одновременно абстрактен, поэтому от него зависят другие компоненты, и конкретен, так что сам зависит от других компонентов.

Понятно, что лучше всего когда компонент находится в одной из конечных точек главной последовательности. Но мой опыт показывает, что такими идеальными характеристиками обладает меньше половины компонентов в проекте. Остальные компоненты можно считать хорошими, если они находятся близко к конечным точкам.

---

[1] Название *«главная последовательность»* отражает мой интерес к астрономии и диаграммам Герцшпрунга–Рассела.

Расстояние от главной последовательности. И теперь мы подходим к последней метрике. Если желательно, чтобы компонент находился на главной последовательности или близко к ней, то можно определить метрику, которая измеряет, насколько далеко компонент отстоит от идеала.

D (расстояние). $D = \frac{|A + I - 1|}{\sqrt{2}}$. Эта величина изменяется в диапазоне [0; ~0,707].

D' (нормированное расстояние). $D' = |A + I - 1|$. Эта метрика гораздо удобнее, чем D, так как изменяется в диапазоне [0,1]. При этом 0 означает, что компонент находится на главной последовательности, а 1 – что он максимально удален от нее.

С помощью этой метрики можно оценить общую сходимость дизайна к главной последовательности. Вычисляем D для каждого компонента. Компоненты, для которых D далеко от 0, следует пересмотреть на предмет изменения структуры. Подобный анализ действительно помогал мне строить компоненты, удобные для сопровождения и восприимчивые к изменениям.

Возможен также статистический анализ дизайна. Можно вычислить математическое ожидание и дисперсию метрик D для всех компонентов системы. Для хорошего дизайна обе величины будут близки к нулю. Дисперсию можно использовать для задания «предельных допусков», позволяющих выявить «исключительные» компоненты (рис. 28.14).

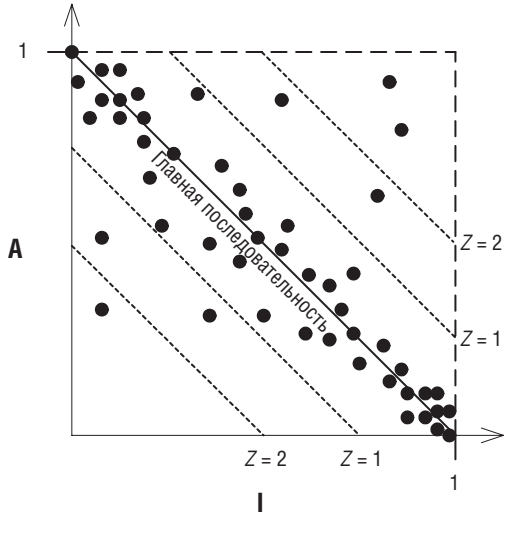


Рис. 28.14. *График рассеяния метрик компонента D*

На этом графике рассеяния (не основанном на реальных данных) мы видим, что большая часть компонентов концентрируется вдоль главной последовательности, но для некоторых стандартное отклонение от

среднего больше 1 (Z = 1). К таким отклонениям следует присмотреться повнимательнее. По-видимому, они либо очень абстрактны, но от них мало что зависит, либо очень конкретны, и при этом велико число зависящих от них компонентов.

Другой способ воспользоваться этими метриками состоит в том, чтобы изобразить изменение D' каждого компонента во времени. На рис. 28.15 показано, как может выглядеть такой график. Мы видим, что в последних нескольких выпусках в компонент `Payroll` вкрались какие-то странные зависимости. На графике показан контрольный порог $D' = 0{,}1$. Точка R2.1 вышла за допустимые пределы, поэтому имеет смысл разобраться, с чего вдруг этот компонент так удалился от главной последовательности.

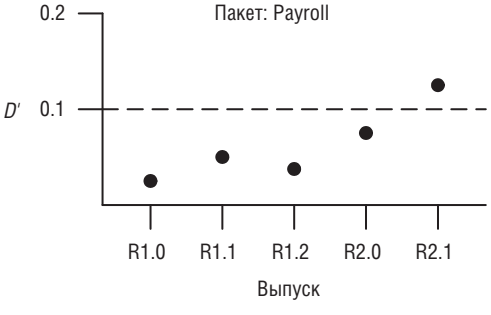


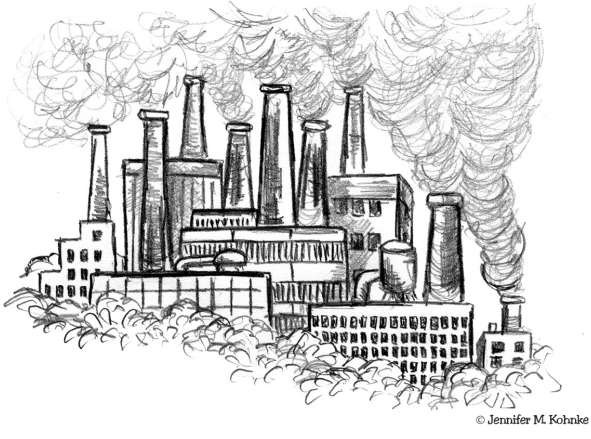
Рис. 28.15. *График зависимости метрики компонента D′ от времени*

Заключение

Описанные в этой главе *метрики управления зависимостями* позволяют численно оценить близость дизайна к тому соотношению зависимости и абстрактности, которое мне представляется «хорошим». Опыт свидетельствует, что некоторые зависимости хороши, тогда как другие плохи. Указанное соотношение отражает этот опыт. Но никакая метрика не абсолютна, она дает лишь представление об отклонении от произвольно выбранного стандарта. Вполне возможно, что стандарт, принятый в этой главе, подходит для одних приложений и не годится для других. Не исключено также, что для измерения качества дизайна можно подобрать гораздо лучшие метрики.

29

Фабрика



Кто строит фабрику – строит храм.

Кэлвин Кулидж (1872–1933)

Принцип инверсии зависимости (DIP) (глава 11) гласит, что следует предпочитать зависимости от абстрактных классов и избегать зависимостей от конкретных классов, особенно когда последние изменчивы. Поэтому следующий фрагмент нарушает принцип DIP:

```
Circle c = new Circle(origin, 1);
```

Здесь `Circle` – конкретный класс, значит, модули, которые создают экземпляры `Circle`, обязательно нарушают DIP. И вообще, любой код, в котором используется ключевое слово `new`, не согласуется с принципом DIP.

Довольно часто нарушение принципа DIP практически безвредно. Чем выше вероятность того, что конкретный класс будет изменяться, тем вероятнее, что зависимость от него приведет к неприятностям. Но если конкретный класс не склонен к изменениям, то ничего страшного в зависимости от него нет. Так, создание объектов типа `string` не вызывает у меня беспокойства. Зависимость от класса `string` вполне безопасна, потому что в обозримом будущем этот класс не изменится.

С другой стороны, во время активной разработки приложения многие конкретные классы часто изменяются, поэтому зависимость от них может стать источником проблем. Лучше зависеть от абстрактного интерфейса, тогда мы будем изолированы от большинства изменений.

Паттерн Фабрика (Factory) позволяет создавать конкретные объекты, не выходя за рамки зависимости от абстрактного интерфейса. Стало быть, он очень полезен на тех этапах разработки приложения когда конкретные классы еще очень изменчивы.

На рис. 29.1 показан сценарий, чреватый неприятностями. Имеется класс `SomeApp`, зависящий от интерфейса `Shape`. `SomeApp` обращается к экземплярам `Shape` исключительно через интерфейс `Shape` и не пользуется методами, специфичными для классов `Square` или `Circle`. К сожалению, `SomeApp` также создает экземпляры `Square` и `Circle` и, следовательно, зависит от этих конкретных классов.

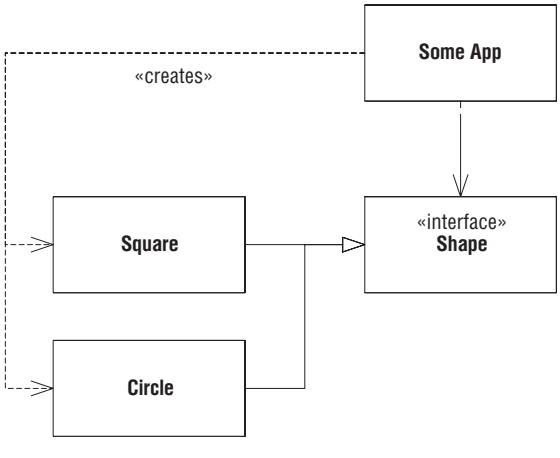


Рис. 29.1. *Приложение, нарушающее принцип DIP ради создания конкретных классов*

Ситуацию можно привести в норму, применив к `SomeApp` паттерн Фабрика, как показано на рис. 29.2. Здесь мы видим интерфейс `ShapeFactory`, в котором объявлены два метода: `MakeSquare` и `MakeCircle`. Метод `MakeSquare` возвращает экземпляр `Square`, а `MakeCircle` – экземпляр `Circle`. Однако в обоих случаях возвращаемое значение имеет тип `Shape`.

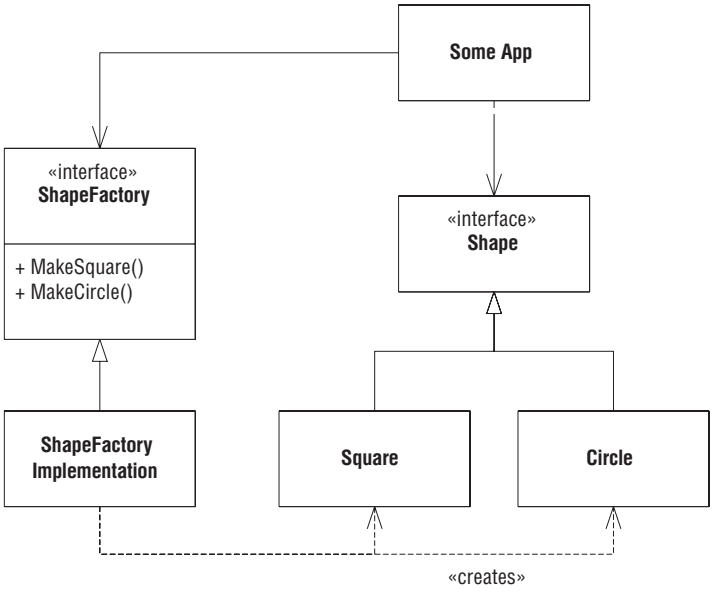


***Рис. 29.2.** Применение паттерна Фабрика к классу SomeApp*

В листинге 29.1 приведен код интерфейса `ShapeFactory`, а в листинге 29.2 – код его реализации `ShapeFactoryImplementation`.

Листинг 29.1. ShapeFactory.cs

```
public interface ShapeFactory
{
  Shape MakeCircle();
  Shape MakeSquare();
}
```

Листинг 29.2. ShapeFactoryImplementation.cs

```
public class ShapeFactoryImplementation : ShapeFactory
{
  public Shape MakeCircle()
  {
    return new Circle();
  }

  public Shape MakeSquare()
  {
    return new Square();
  }
}
```

Этот прием полностью решает проблему зависимости от конкретных классов. Код приложения больше не зависит ни от `Circle`, ни от `Square` и тем не менее может успешно создавать объекты обоих классов. Он ма-

нипулирует этими объектами через интерфейс Shape и никогда не вызывает методов, специфичных только для Square или Circle.

Проблема зависимости от конкретного класса перенесена в другое место. Где-то, конечно, должен быть создан экземпляр ShapeFactoryImplementation, но больше ни в одном месте объекты Square и Circle напрямую не создаются. Экземпляр ShapeFactoryImplementation, скорее всего, будет создан в головной программе или в специальном методе инициализации, который вызывается из Main.

Проблема зависимости

Внимательный читатель наверняка заметил, что в описанном выше варианте паттерна Фабрика имеется проблема. В интерфейсе ShapeFactory объявлены методы для всех классов, производных от Shape. Это приводит к *зависимости по именам*, усложняющей добавление новых подклассов Shape. При добавлении каждого нового подкласса нужно также добавить новый метод в интерфейс ShapeFactory. Обычно это означает, что придется заново откомпилировать и развернуть все клиенты ShapeFactory.[1]

Можно избавиться от этой проблемы, частично пожертвовав безопасностью типов. Вместо того чтобы вводить в ShapeFactory по одному методу для каждого подкласса Shape, можно оставить всего один метод, принимающий строку, как в листинге 29.3. При таком подходе в классе ShapeFactoryImplementation нужно будет использовать цепочку предложений if/else, в которых входной аргумент анализируется на предмет выбора подходящего подкласса Shape. Эта техника иллюстрируется в листингах 29.4 и 29.5.

Листинг 29.3. Фрагмент кода, в котором создается круг

```
[Test]
public void TestCreateCircle()
{
  Shape s = factory.Make("Circle");
  Assert.IsTrue(s is Circle);
}
```

Листинг 29.4. ShapeFactory.cs

```
public interface ShapeFactory
{
  Shape Make(string name);
}
```

[1] Мы уже говорили, что в языке C# это, строго говоря, необязательно. Можно обойтись без повторной компиляции и развертывания клиентов изменившегося интерфейса. Но это рискованное дело.

Листинг 29.5. ShapeFactoryImplementation.cs

```
public class ShapeFactoryImplementation : ShapeFactory
{
  public Shape Make(string name)
  {
    if(name.Equals("Circle"))
      return new Circle();
    else if(name.Equals("Square"))
      return new Square();
    else
      throw new Exception(
        "ShapeFactory не может создать: {0}", name);
  }
}
```

Можно возразить, что такой прием опасен, потому что в случае неправильного написания названия фигуры клиент получит ошибку на этапе выполнения, а не компиляции. Верно. Но если вы пишете автономные тесты и применяете методику разработки через тестирование, то такие ошибки времени выполнения будут обнаружены задолго до того, как породят проблемы.

Статическая и динамическая типизация

Только что рассмотренный компромисс между безопасностью типов и гибкостью – типичное проявление непрекращающихся споров по поводу подходов к разработке языков программирования. С одной стороны, есть статически типизированные языки, такие как C#, C++ и Java, где проверка типов производится на этапе компиляции и компилятор выдает ошибку если объявленные типы несовместимы. С другой стороны, есть динамически типизированные языки, такие как Python, Ruby, Groovy и Smalltalk, в которых типы проверяются во время выполнения. Компилятор не настаивает на совместимости типов, да и синтаксис языка не допускает такой проверки.

На примере паттерна Фабрика мы видели, что статическая типизация может приводить к такой конфигурации зависимостей, что единственно ради поддержания совместимости типов придется модифицировать исходные файлы. В рассмотренном случае мы были вынуждены изменять интерфейс `ShapeFactory` при каждом добавлении нового подкласса `Shape`. А в результате получаем повторную сборку и развертывание, которых можно было бы избежать. Мы решили проблему, ослабив требования к безопасности типов и переложив ответственность за обнаружение ошибок на автономные тесты; это позволило нам добавлять новые подклассы `Shape`, не изменяя `ShapeFactory`.

Сторонники статически типизированных языков говорят, что безопасность относительно типов на этапе компиляции стоит небольшого увеличения частоты редактирования исходного кода и количества циклов

сборки и развертывания. Противная сторона возражает, что автономные тесты обнаружат большую часть ошибок, которые находит компилятор статически типизированного языка, поэтому тащить за собой груз модификации кода, сборки и развертывания вовсе не обязательно.

Мне кажется интересным, что рост популярности динамически типизированных языков коррелирует с распространением методик разработки через тестирование (TDD). Быть может, программисты, воспринявшие идеи TDD, считают, что они изменяют соотношение между безопасностью и гибкостью. А может, эти программисты постепенно приходят к убеждению, что гибкость динамически типизированных языков перевешивает достоинства статической проверки типов.

Возможно, сейчас на пике популярности находятся статически типизированные языки. Если наблюдаемая тенденция продолжится, то не исключено, что следующий основной промышленный язык будет ближе к Smalltalk, чем к C++.

Взаимозаменяемые фабрики

Одно из крупных достоинств фабрик – возможность подменять одну реализацию фабрики другой. Это позволяет подставлять в приложение различные семейства объектов.

Представьте, например, приложение, которое должно адаптироваться к нескольким реализациям базы данных. Допустим, что можно либо работать с плоскими файлами, либо купить адаптер к СУБД Oracle. Чтобы изолировать приложение от реализации базы данных, можно воспользоваться паттерном Заместитель (Proxy).[1] А для создания экземпляров заместителей можно применить фабрики. Структура показана на рис. 29.3.

Обратите внимание на две реализации интерфейса `EmployeeFactory`. Одна создает объекты-заместители для работы с плоскими файлами, другая – для работы с Oracle. При этом приложение не знает о том, какая реализация используется.

Использование фабрик в тестовых фикстурах

При написании автономных тестов часто бывает необходимо протестировать поведение модуля изолированно от тех модулей, которыми он пользуется. Пусть, например, есть приложение `Payroll`, в котором используется база данных (рис. 29.4). Может возникнуть необходимость протестировать работу модуля `Payroll`, не связываясь ни с какой конкретной базой данных.

[1] Паттерн Заместитель мы будем изучать в главе 34. А пока достаточно знать, что речь идет о классе, который умеет читать конкретные объекты из конкретных баз данных.

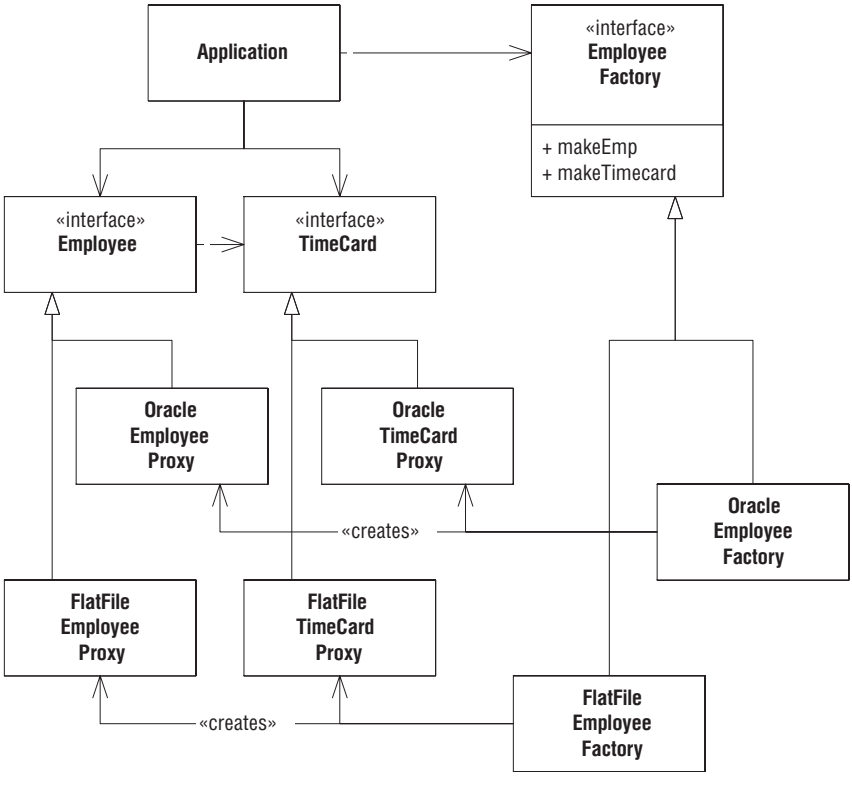


Рис. 29.3. *Взаимозаменяемые фабрики*



Рис. 29.4. *Приложение Payroll использует базу данных*

Этого можно добиться, создав абстрактный интерфейс для базы данных. Одной из его реализаций будет класс для работы с настоящей базой. А другой – тестовый код, написанный специально для имитации поведения базы данных и проверки того, что все обращения к ней правильны. Эта структура изображена на рис. 29.5. Модуль `PayrollTest` тестирует `PayrollModule`, обращаясь к нему, и одновременно реализует интерфейс `Database`, перехватывающий все обращения `Payroll` к базе. Это позволяет убедиться, что `Payroll` ведет себя надлежащим образом. Кроме того, `PayrollTest` может имитировать различные ошибки базы данных, которые иначе было бы трудно воспроизвести. Такой способ тестирования известен под названием Автошунтирование (Self-Shunt). Иногда его называют также *имитацией*, *mock-объектами* (Mocking) или *подменой* (Spoofing).

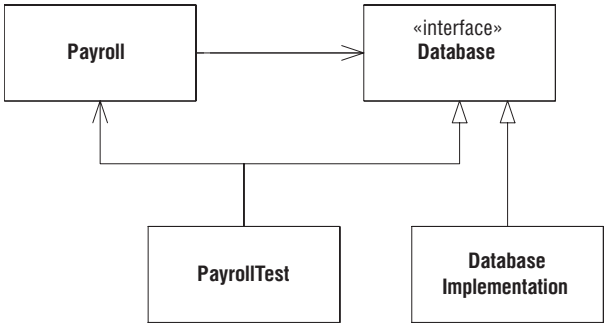


Рис. 29.5. *Автошунтирование базы данных в тесте PayrollTest*

Как объект `Payroll` получает экземпляр `PayrollTest`, используемый в качестве `Database`? Разумеется, `Payroll` не создает `PayrollTest` самостоятельно. Но ведь как-то получить ссылку на реализацию `Database` он должен.

Иногда ссылку на `Database` может передать объект `PayrollTest`. В других случаях `PayrollTest` должен установить глобальную переменную, ссылающуюся на `Database`. А бывает, что экземпляр `Database` обязан создать сам объект `Payroll`. Вот тогда мы можем воспользоваться фабрикой, которая будет создавать тестовую версию `Database`, и передадим экземпляр такой «подложной» фабрики объекту `Payroll`.

Одна из возможных структур изображена на рис. 29.6. Модуль `Payroll` получает фабрику из глобальной переменной – или статической переменной в глобальном классе – `GdatabaseFactory`. Модуль `PayrollTest` реализует интерфейс `DatabaseFactory` и записывает ссылку на себя в переменную `GdatabaseFactory`. Когда `Payroll` пользуется фабрикой для создания `Database`, модуль `PayrollTest` перехватывает обращение и возвращает ссылку на себя. Таким образом, `Payroll` думает, что создал `PayrollDatabase`, тогда как на самом деле все обращения к базе данных попадают к модулю `PayrollTest`.

Важность фабрик

Строгая интерпретация принципа DIP означала бы, что нужно создавать фабрику для любого изменчивого класса. При этом паттерн Фабрика выглядит очень соблазнительно. Поэтому разработчики могут поддаться искушению использовать фабрики по умолчанию. Но этой крайность, которую я не рекомендую.

Лично я поначалу не пользуюсь фабриками. А ввожу их в систему только при возникновении настоятельной необходимости. Например, когда приходится прибегать к паттерну Заместитель, возможно, имеет смысл завести и фабрику для создания сохраняемых объектов. Или если в процессе автономного тестирования возникает ситуация, когда

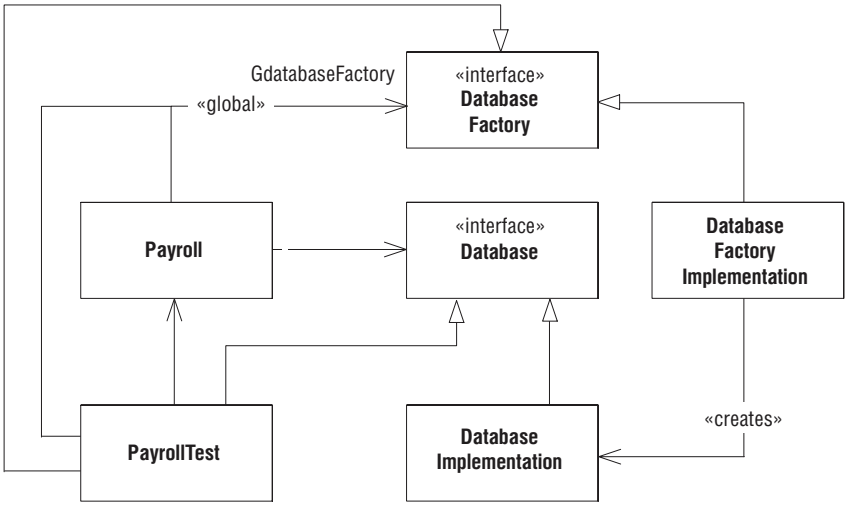


Рис. 29.6. *Подмена фабрики*

нужно подменить создателя объектов. Но пока полезность фабрики не стала очевидной, я стараюсь обходиться без нее.

Фабрики привносят сложность, которой часто можно избежать, особенно на ранних стадиях проектирования. Если использовать их к месту и не к месту, то развитие дизайна может сильно осложниться. Для создания всего одного содержательного класса придется вводить целых четыре сущности: два интерфейса, представляющих сам новый класс и его фабрику, и два конкретных класса, реализующих эти интерфейсы.

Заключение

Фабрика – это мощный инструмент. Она может оказаться ценным подспорьем, обеспечивающим согласование с принципом DIP, поскольку позволяет модулям стратегии верхнего уровня создавать экземпляры классов, не становясь зависимыми от конкретных реализаций этих классов. Кроме того, фабрики дают возможность подменять целые семейства реализаций групп классов. Но вместе с тем фабрики привносят сложность, без которой часто можно обойтись. Повсеместное их применение редко бывает оптимальным курсом.

Библиография

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

30

Система расчета заработной платы: анализ пакетов



Эвристическое правило: если какой-то предмет кажется вам искусным и изощренным, остерегитесь: возможно, это самообман.

Дональд Норман «Дизайн привычных вещей» (1990)

Мы много занимались анализом, проектированием и реализацией системы расчета зарплаты. Но и осталось сделать еще немало. Во-первых, сейчас над задачей работают всего два программиста – Боб и Мика. Принятая организация среды разработки этому вполне соответствует.

Все программные файлы находятся в одном каталоге. Структуры более высокого уровня не существует. Нет ни пакетов, ни подсистем, ни независимо выпускаемых компонентов. Но так мы далеко не продвинемся.

Мы должны предполагать, что с разрастанием проекта будет увеличиваться и количество занятых в нем людей. Чтобы нескольким разработчикам было удобно работать, мы должны разбить исходный код на компоненты – сборки, DLL-файлы, – которые можно было бы без труда снимать с контроля, изменять и тестировать.

В данный момент приложение насчитывает 4382 строки кода, организованные в 63 класса, которые находятся в 80 файлах. Цифры не поражают воображение, но создают некую организационную проблему. Как нам управлять этими исходными файлами и как разбить их на независимо развертываемые компоненты?

И еще – как распределить задачи реализации между программистами, чтобы они не мешали друг другу? Хотелось бы разбить множество классов на группы, которые отдельные разработчики или команды могли бы удобно извлекать из системы управления версиями и поддерживать.

Структура компонентов и обозначения

На рис. 30.1 изображена возможная структура компонентов в системе расчета зарплаты. Хороша она или плоха, мы обсудим позже. А пока займемся вопросом о том, как ее документировать и использовать.

По принятому соглашению диаграммы компонентов рисуются так, чтобы стрелки зависимостей были направлены вниз. Наверху находятся зависимые компоненты, внизу – те, от которых что-то зависит.

Мы видим, что система разбита на восемь компонентов. Компонент `PayrollApplication` содержит классы `PayrollApplication`, `TransactionSource` и `TextParserTransactionSource`. Компонент `Transactions` содержит всю иерархию классов `Transaction`. Состав остальных компонентов должен быть понятен из диаграммы.

Зависимости тоже ясны. Компонент `PayrollApplication` зависит от компонента `Transactions`, потому что класс `PayrollApplication` вызывает метод `Transaction.Execute`. Компонент `Transactions` зависит от `PayrollDatabase`, поскольку каждый из многочисленных подклассов `Transaction` напрямую общается с классом `PayrollDatabase`. Остальные зависимости рассматриваются аналогично.

По какому критерию мы объединили классы в компоненты? Да просто интуитивно определили, какие классы вроде бы должны быть вместе. Но в главе 28 мы выяснили, что такой подход не всегда уместен.

Посмотрите, что получится если мы изменим компонент `Classifications`. Нам придется заново компилировать и тестировать компонент `EmployeeDatabase`, и это оправданно. Однако далее возникает необходимость в повторной компиляции и тестировании компонента `Transac-`

tions. Ладно, класс ChangeClassificationTransaction и три его подкласса (см. рис. 27.13) *действительно* нужно откомпилировать и протестировать заново, но остальные-то зачем?

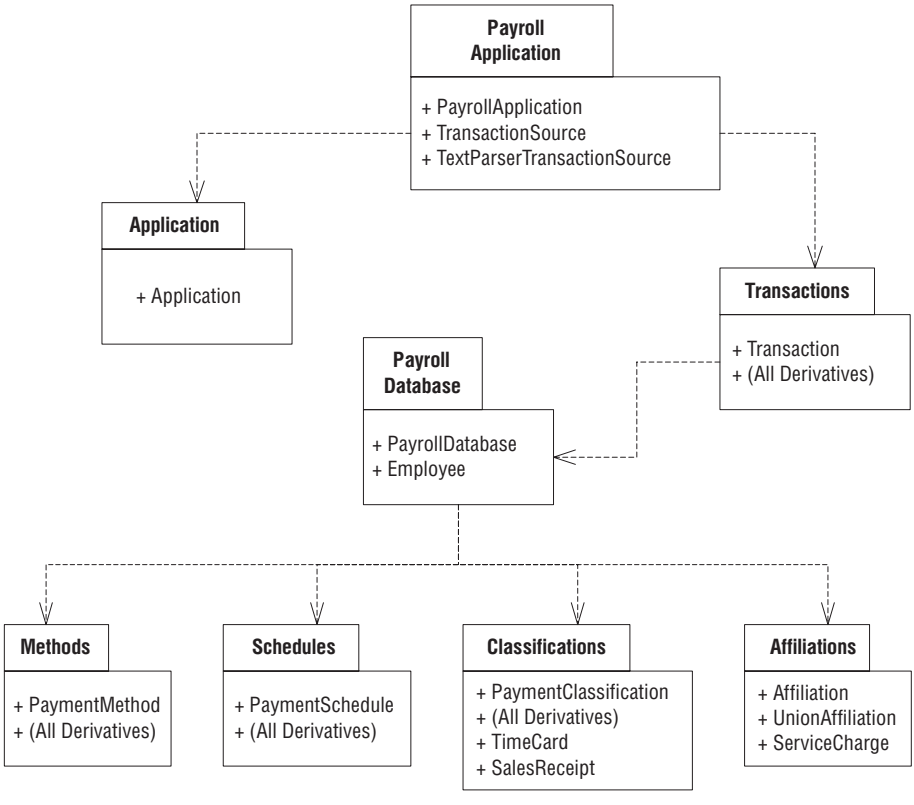


Рис. 30.1. *Возможная диаграмма компонентов в системе расчета зарплаты*

Технически классы остальных операций не нуждаются в компиляции и тестировании. Но они входят в компонент Transactions, и если мы собираемся выпустить новую его версию в связи с изменениями компонента Classifications, то было бы безответственно пренебречь этими действиями. Даже если не подвергать повторной компиляции и тестированию все операции, то сам пакет необходимо заново выпустить и развернуть, а тогда все его клиенты придется как минимум заново проверить и, возможно, тоже перекомпилировать.

Классы в компоненте Transactions закрыты относительно разных изменений. Каждый восприимчив к изменениям какого-то одного вида. ServiceChargeTransaction открыт относительно изменений в классе ServiceCharge, тогда как TimeCardTransaction – относительно изменений в классе TimeCard. Фактически из рис. 30.1 следует, что какая-то часть компонента Transactions зависит чуть ли не от всех остальных частей

системы. Следовательно, этот компонент понадобится часто перевыпускать. Всякий раз, как изменяется что-то ниже него, компонент Transactions придется заново проверить и выпустить.

Пакет PayrollApplication еще более чувствителен. На нем отражается любое изменение в любой части системы, поэтому частота перевыпуска будет очень высока. Возможно, вам кажется, что это неизбежно, – чем выше мы поднимаемся по иерархии зависимостей пакетов, тем чаще будут выпуски. Но, к счастью, это не так, а борьба с этим явлением – одна из основных целей объектно-ориентированного проектирования.

Применение принципа общей закрытости (CCP)

Рассмотрим рис. 30.2, на котором классы сгруппированы в соответствии с тем, от каких изменений они закрыты. Например, компонент PayrollApplication содержит классы PayrollApplication и TransactionSource. Оба они зависят от абстрактного класса Transaction, который находится в компоненте PayrollDomain. Отметим, что класс TextParserTransactionSource находится в другом компоненте, зависящем от абстрактного класса PayrollApplication. Тем самым мы создали нисходящую структуру, в которой детальные компоненты зависят от общих, а общие независимы. Это согласуется с принципом DIP.

Самый замечательный пример общности и независимости дает компонент PayrollDomain. В нем сосредоточена *самая суть* системы, и при этом он ни от чего не зависит! Рассмотрим этот компонент внимательнее. Он содержит классы Employee, PaymentClassification, PaymentMethod, PaymentSchedule, Affiliation и Transaction, то есть все основные абстракции модели. Как же он получился независимым? Да потому, что все эти классы абстрактны.

Теперь взгляните на компонент Classifications, который содержит все три класса, производных от PaymentClassification, а также класс ChangeClassificationTransaction с тремя его подклассами и классы TimeCard и SalesReceipt. Обратите внимание, что любое изменение, внесенное в эти девять классов, изолировано; за исключением TextParser, ни один компонент не затрагивается! Такая же изолированность наблюдается для компонентов Methods, Schedules и Affiliations. Впечатляет!

Отметим, что большая часть детального кода, который рано или поздно будет написан, находится в компонентах, для которых мало или вовсе нет зависимостей. Поскольку эти компоненты почти не имеют зависимостей, мы называем их *неответственными*. Код внутри таких компонентов обладает высочайшей гибкостью; его можно изменять, не затрагивая почти никаких частей системы. Также отметим, что большинство общих пакетов содержит очень мало кода. От них зависят почти все остальные, но сами они не зависят ни от чего. Поскольку от них зависят многие компоненты, мы называем их *ответственными*, а коль скоро они сами не зависят от других, то являются *независимы-*

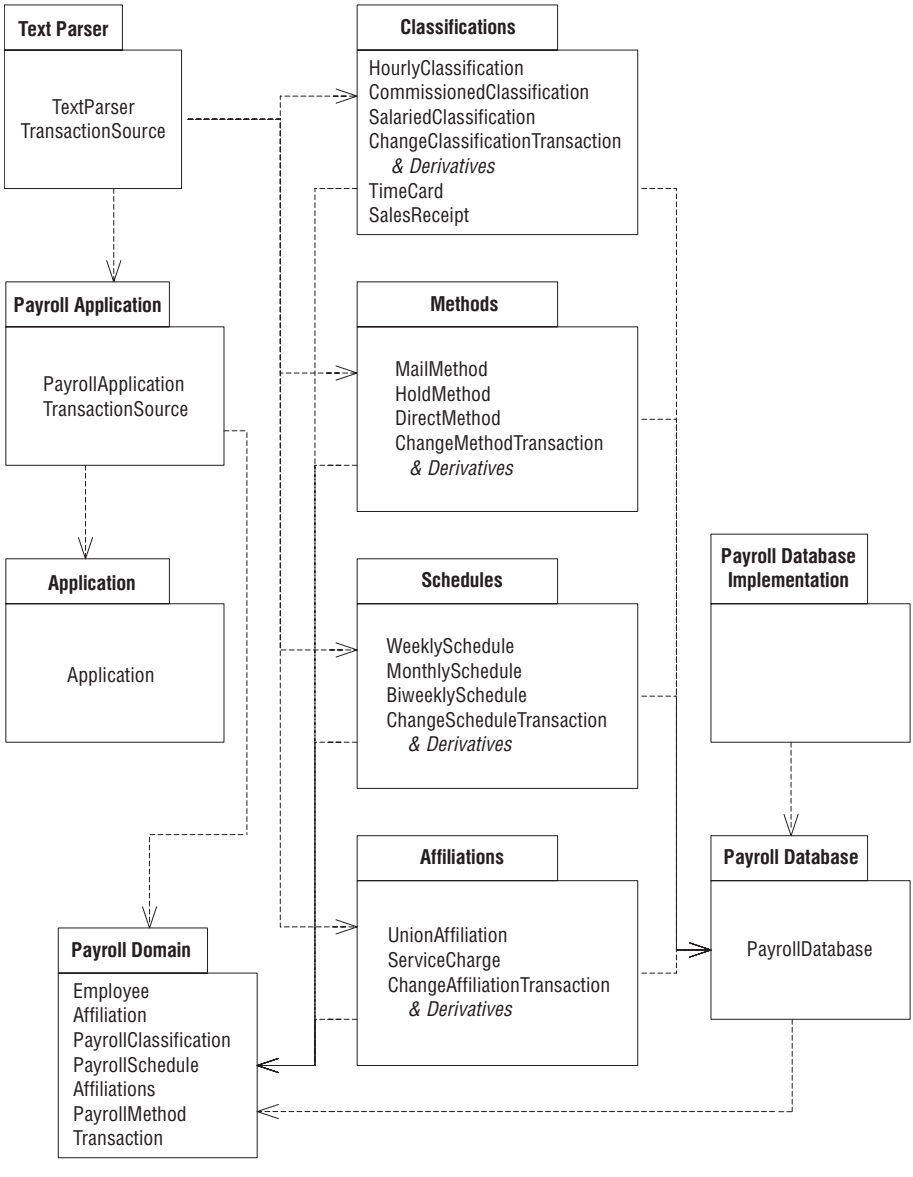


Рис. 30.2. *Иерархия закрытых компонентов системы расчета зарплаты*

ми. Таким образом, объем ответственного кода (то есть такого, изменение которого влияет на многие части системы) очень мал. К тому же этот небольшой объем кода еще и независим, то есть нет модулей, которые стимулировали бы его изменение. Такая нисходящая структура, в которой независимые и ответственные общие компоненты находятся

внизу, а неответственные и зависимые – вверху, является характерным признаком объектно-ориентированного дизайна.

Сравним рисунки 30.1 и 30.2. Обратите внимание, что детальные компоненты в нижней части рис. 30.1 независимы и в высокой степени ответственны. Такое место не для деталей! Детали должны зависеть от основных архитектурных решений, принятых в системе, а от них ничто не должно зависеть. Отметим также, что общие компоненты – те, что определяют архитектуру системы, – неответственны и очень зависимы. Получается, что компоненты, определяющие архитектуру, зависят от компонентов, содержащих детали реализации и, следовательно, ограничены ими. Это нарушение принципа SAP. Архитектура должна ограничивать детали, а не наоборот!

Применение принципа эквивалентности повторного использования и выпуска (REP)

Какие части системы расчета зарплаты можно использовать повторно? Другое подразделение компании, которое хотело бы воспользоваться нашей системой, но руководствуется иным набором политик, не смогло бы использовать компоненты `Classifications`, `Methods`, `Schedules` и `Affiliations`, но вполне способно взять на вооружение компоненты `PayrollDomain`, `PayrollApplication`, `Application`, `PayrollDatabase` и, возможно, `PDImplementation`. С другой стороны, отдел, имеющий потребность в программе для анализа текущей базы данных работников, мог бы воспользоваться компонентами `PayrollDomain`, `Classifications`, `Methods`, `Schedules`, `Affiliations`, `PayrollDatabase` и `PDImplementation`. В каждом случае минимальной единицей повторного использования является компонент.

Очень редко возникает необходимость повторно воспользоваться каким-то одним классом из компонента. Причина проста: классы внутри компонента должны быть сцепленными. Иначе говоря, они зависят друг от друга и разделить их достаточно трудно. Так, вряд ли имело бы смысл использовать класс `Employee` без класса `PaymentMethod`. На самом деле для выполнения этого приема нужно было бы модифицировать класс `Employee` так, чтобы он не содержал экземпляра `PaymentMethod`. Разумеется, мы не хотим поддерживать такой вид повторного использования, при котором были бы вынуждены изменять повторно используемые компоненты. Следовательно, единицей повторного использования остается компонент. Это дает нам еще один критерий сцепленности, который можно применять при объединении классов в компоненты: классы должны быть не только закрыты от общих изменений, но и допускать совместное повторное использование в соответствии с принципом REP.

Снова рассмотрим первоначальную диаграмму компонентов на рис. 30.1. Компоненты, которые мы захотели бы использовать повторно, например `Transactions` или `PayrollDatabase`, плохо приспособлены для этого, так

как несут с собой много лишнего багажа. Компонент PayrollApplication зависит от всего вообще. Если бы мы пожелали создать новую систему расчета зарплаты, в которой по-другому устроены графики выплат, способы платежа, членство во внешних организациях и тарификация, то не смогли использовать этот пакет целиком. Пришлось бы выдергивать отдельные классы из компонентов PayrollApplication, Transactions, Methods, Schedules, Classifications и Affiliations. Разобрав эти компоненты на части, мы свели бы на нет политику их выпуска. Мы уже не смогли бы сказать, что версия 3.2 компонента PayrollApplication допускает повторное использование.

Так как рис. 30.1 нарушает принцип CRP, то разработчик, согласившийся взять повторно используемые фрагменты из разных компонентов, не смог бы полагаться на нашу систему выпуска версий. Воспользовавшись классом PaymentMethod, он оказался бы зависимым от новой версии компонента Methods. Как правило, изменения будут касаться классов, которые ему не нужны, но все равно придется отслеживать новые версии и, возможно, заново компилировать и тестировать свой код.

Управлять всем этим будет настолько сложно, что разработчик, скорее всего, скопирует к себе повторно используемые компоненты и будет развивать их независимо от нас. Но это уже не будет повторным использованием. Две ветки кода разойдутся, потребуют независимой поддержки, то есть в конечном итоге затраты на сопровождение удвоятся.

Структура на рис. 30.2 от таких проблем избавлена. Показанные на нем компоненты легко использовать повторно. PayrollDomain тянет за собой не так много. Его можно использовать независимо от любых классов, производных от PaymentMethod, PaymentClassification, PaymentSchedule и т. д.

Проницательный читатель заметит, что диаграмма компонентов на рис. 30.2 не вполне согласуется с принципом CRP. Точнее, классы в компоненте PayrollDomain не образуют минимальную повторно используемую единицу. Класс Transaction необязательно использовать вместе с прочими классами, входящими в состав компонента. Можно разработать много приложений, которые обращаются к классу Employee и его полям, но не используют Transaction.

Это наводит на мысль переделать диаграмму компонентов, как показано на рис. 30.3. Теперь операции отделены от элементов, которыми манипулируют. Например, классы внутри компонента MethodTransactions манипулируют классами в компоненте Methods. Мы перенесли класс Transaction в новый компонент, который назвали TransactionApplication. Туда вошли также классы TransactionSource и TransactionApplication. Втроем они образуют повторно используемую единицу. Класс PayrollApplication стал теперь великим объединителем. Он содержит головную программу и производный от TransactionApplication класс, названный PayrollApplication, который связывает TextParserTransactionSource с TransactionApplication.

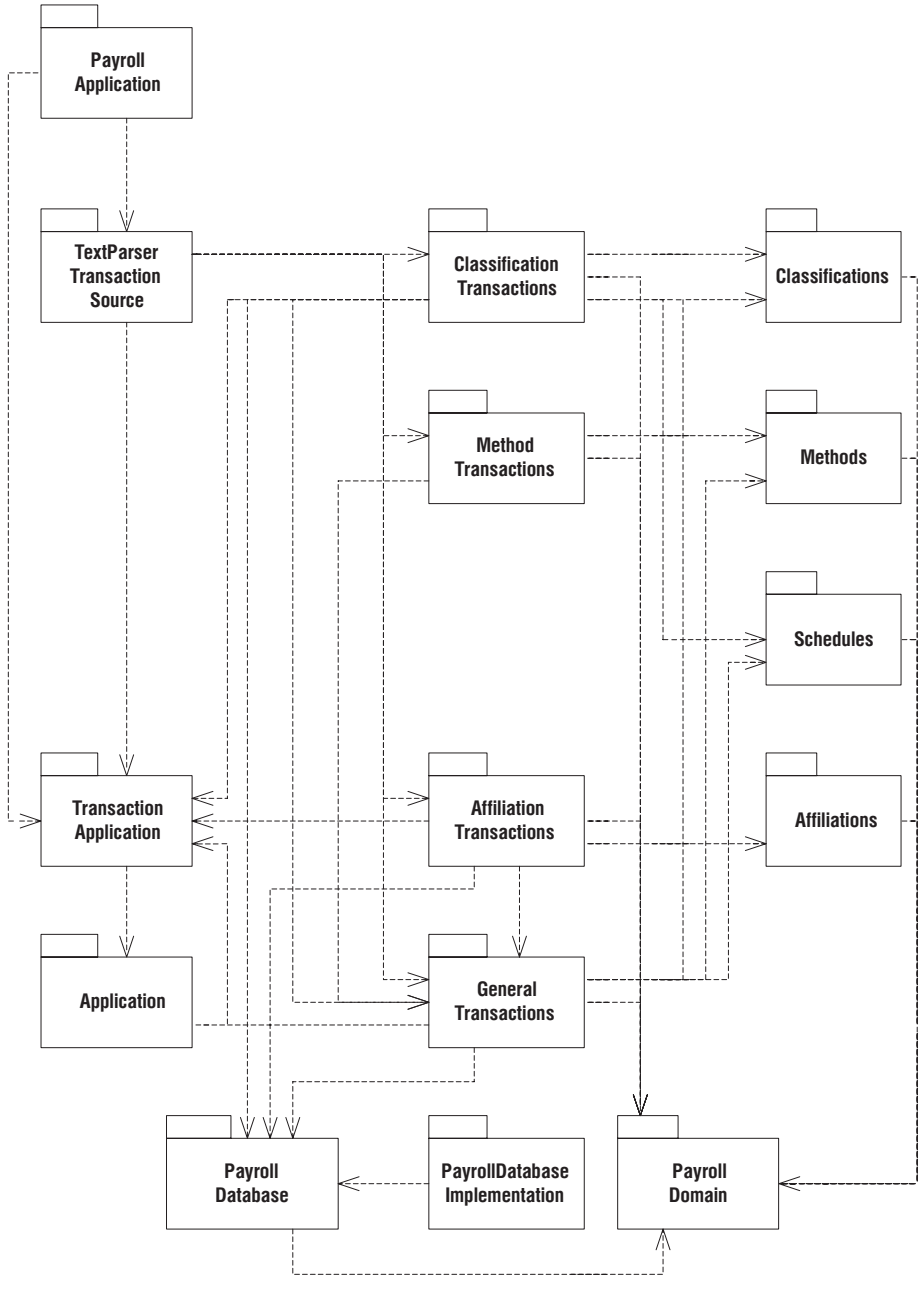


Рис. 30.3. *Исправленная диаграмма компонентов системы расчета зарплаты*

Эти манипуляции добавили в дизайн еще один уровень абстракции. Компонент TransactionApplication теперь можно повторно использовать в любом приложении, которое получает операции Transaction из источника TransactionSource и вызывает их метод Execute(). Компонент PayrollApplication более не является повторно используемым, потому что он очень сильно зависим. Однако его место занял более общий компонент TransactionApplication. Теперь мы можем использовать компонент PayrollDomain вообще без Transaction.

Конечно, это повышает степень повторной используемости и удобство сопровождения проекта, но ценой пяти дополнительных компонентов и усложнения архитектуры зависимостей. Ценность такого компромисса зависит от ожидаемых видов повторного использования и темпа эволюции приложения. Если приложение останется стабильным, а желающих повторно использовать его компоненты немного, то такое изменение, пожалуй, перебор. Но если жаждущих воспользоваться им будет много или приложение часто изменяется, то новая структура, наверное, окажется лучше старой. В общем, это субъективное решение, которое должно подтверждаться фактами, а не умозрительными рассуждениями. Лучше начать с простого и развивать структуру компонентов по мере необходимости. Когда придет нужда, структуру всегда можно будет усовершенствовать.

Связанность и инкапсуляция

Если связанность классов управляется в C# границами инкапсуляции, то связанностью компонентов можно управлять, объявляя помещенные в них классы открытыми или закрытыми. Если класс, находящийся в одном компоненте, может использоваться другим компонентом, то такой класс следует объявить открытым (public). Класс же, используемый только внутри самого компонента, объявляется внутренним (internal).

Скрывать некоторые классы внутри компонента полезно во избежание входящих связей. Classifications – это детальный компонент, содержащий реализацию нескольких политик тарификации. Чтобы оставить его на главной последовательности, мы должны ограничить входящие в него связи, поэтому мы скрываем классы, о которых другим пакетам знать ни к чему.

Хорошими примерами внутренних классов могут служить TimeCard и SalesReceipt. Это детали реализации механизмов расчета зарплаты работника. Мы хотели бы сохранить свободу изменения этих деталей, а потому должны позаботиться о том, чтобы ничто от них не зависело.

Взглянув на рис. 27.7–27.10 и на листинг 27.10, мы обнаружим, что классы TimeCardTransaction и SalesReceiptTransaction уже зависят от Time-

`Card` и `SalesReceipt`. Но эту проблему легко решить, как показано на рис. 30.4 и 30.5.

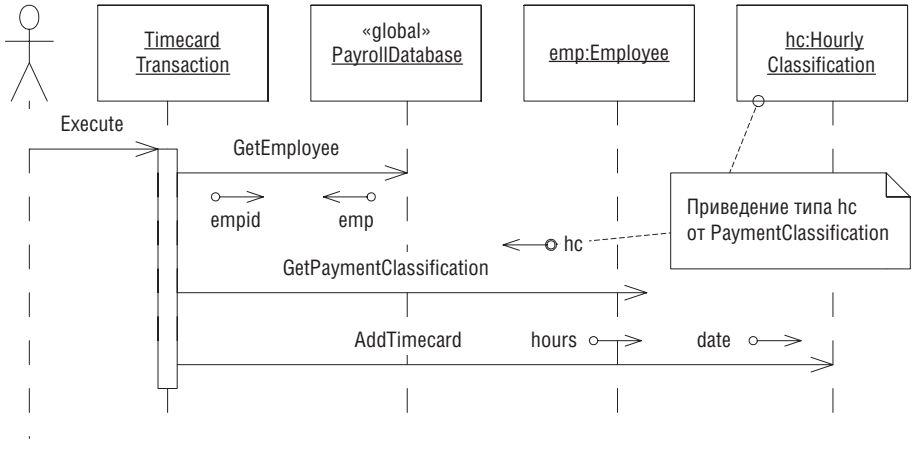


Рис. 30.4. *Пересмотр класса TimeCardTransaction с целью защиты приватности класса TimeCard*

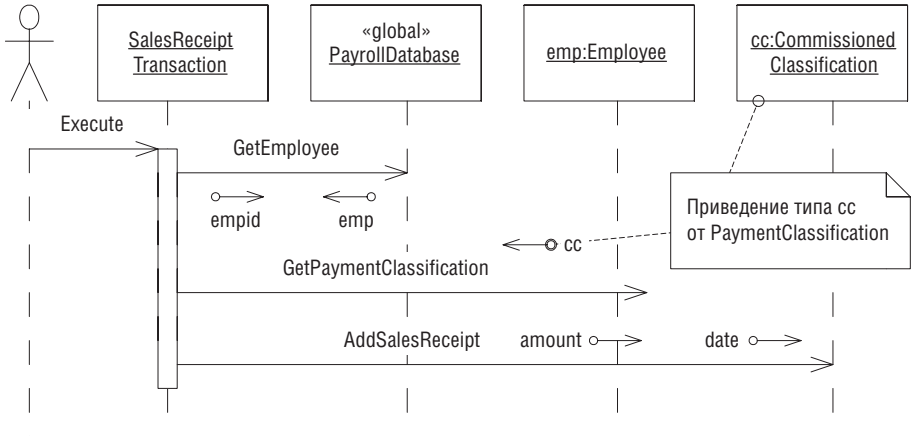


Рис. 30.5. *Пересмотр класса SalesReceiptTransaction с целью защиты приватности класса SalesReceipt*

Метрики

В главе 28 было показано, что сцепленность, связанность, устойчивость, общность и близость к главной последовательности можно количественно оценить с помощью нескольких простых метрик. Но зачем

нам это? Перефразируя Тома Демарко, можно сказать: «Нельзя управлять тем, что не контролируешь, и нельзя контролировать то, что не можешь измерить».[1] Работа программиста или менеджера будет эффективной, только если он способен контролировать разработку ПО на практике. А не умея измерять, мы ничего не сможем контролировать.

Применяя описанные ниже эвристики и вычислив некоторые фундаментальные метрики нашего объектно-ориентированного проекта, мы сможем скоррелировать их с измеренной производительностью ПО и участвующих в его разработке команд. Чем больше известно метрик, тем выше наша информированность и тем точнее контроль.

Рассматриваемые здесь метрики успешно применялись в ряде проектов с 1994 года. Есть несколько автоматических инструментов для их вычисления, но и вручную это проделать несложно. Кроме того, можно без труда написать простенький сценарий на языке оболочки, Python или Ruby, который просмотрит все исходные файлы и проведет подсчеты по метрикам.[2]

- H (*сцепленность по связям*) – это среднее число внутренних связей одного класса, входящего в компонент. Пусть R – число связей класса, не выходящих за пределы компонента (то есть соединяющих его с классами в том же компоненте). Пусть N – общее число классов в компоненте. Тогда величина

  $$H = \frac{R+1}{N}$$

  характеризует связь пакета со всеми своими классами. Дополнительная единица нужна для того, чтобы эта величина не обращалась в 0 при $N = 1$.
- C_a (*входящая связность*) вычисляется, как количество классов из других компонентов, которые зависят от классов в данном компоненте. Эти зависимости представляют собой такие отношения между классами, как наследование и ассоциация.
- C_e (*исходящая связность*) вычисляется, как количество классов из других компонентов, от которых зависят классы в данном компоненте. Как и в предыдущем случае, речь идет об отношениях между классами.
- A (*абстрактность*, или *общность*) вычисляется, как отношение числа абстрактных классов или интерфейсов в компоненте к общему

---

[1] [DeMarco82], стр. 3

[2] Один такой сценарий `depend.sh` на языке оболочки можно скачать из раздела бесплатных продуктов на сайте *www.objectmentor.com*.

числу классов или интерфейсов в нем же.[1] Эта метрика изменяется в диапазоне от 0 до 1.

A = Число абстрактных классов / Общее число классов

- *I* (*неустойчивость*) вычисляется, как отношение входящей связности к суммарной связности. Эта метрика также изменяется в диапазоне от 0 до 1.

$$I = \frac{C_e}{C_e + C_a}$$

- *D* (расстояние от главной последовательности) = $|(A + I - 1) / \sqrt{2}|$. Главная последовательность – это прямая, описываемая уравнением $A + I = 1$. Эта величина характеризует расстояние компонента от главной последовательности. Она изменяется в диапазоне от ~0,7 до 0; чем ближе к 0, тем лучше.[2]

$$D = \frac{|A + I - 1|}{\sqrt{2}}$$

- *D′* (нормированное расстояние от главной последовательности) – метрика *D*, приведенная к диапазону [0,1]. Она немного удобнее для вычисления и интерпретации. Значение 0 соответствует компонентам, лежащим на главной последовательности, значение 1 – компонентам, удаленным от нее на максимальное расстояние.

$$D' = |A + I - 1|$$

Применение метрик к задаче о расчете зарплаты

В табл. 30.1 показано, как распределены по компонентам классы из системы расчета зарплаты. А на рис. 30.6 изображена диаграмма компонентов со всеми метриками. Наконец, в табл. 30.2 приведены все метрики, вычисленные для каждого компонента.

---

[1] Может показаться, что лучше вычислять *A*, как отношение числа абстрактных методов к общему числу методов в пакете. Однако я обнаружил, что при таком способе подсчета метрика абстрактности значительно ослабляется. Достаточно всего одного абстрактного метода, чтобы класс считался абстрактным, и важность так определенной абстрактности не умаляется тем фактом, что в этом классе может быть еще несколько десятков конкретных методов, особенно если следовать принципу DIP.

[2] Все пакеты располагаются внутри единичного квадрата в осях A и I, поскольку ни A, ни I не могут быть больше 1. Главная последовательность соединяет точки (0,1) и (1,0) и, стало быть, делит квадрат пополам по диагонали. Наиболее удалены от главной последовательности углы квадрата с координатами (0,0) и (1,1).
Расстояние, на которое они удалены, составляет $\frac{\sqrt{2}}{2} = 0{,}70710678\ldots$

Таблица 30.1. Распределение классов по компонентам

Компонент	Классы в компоненте		
Affiliations	ServiceCharge	UnionAffiliation	
AffiliationTransactions	ChangeAffilliationTransaction	ChangeUnaffiliatedTransaction	ChangeMemberTransaction
	ServiceChargeTransaction		
Application	Application		
Classifications	CommissionedClassification	HourlyClassification	SalariedClassification
	SalesReceipt	Timecard	
ClassificationTransaction	ChangeClassificationTransaction	ChangeCommissionedTransaction	ChangeHourlyTransaction
	ChangeSalariedTransaction	SalesReceiptTransaction	TimecardTransaction
GeneralTransactions	AddCommissionedEmployee	AddEmployeeTransaction	AddHourlyEmployee
	AddSalariedEmployee	ChangeAddressTransaction	ChangeEmployeeTransaction
	ChangeNameTransaction	DeleteEmployeeTransaction	PaydayTransaction
Methods	DirectMethod	HoldMethod	MailMethod
MethodTransactions	ChangeDirectTransaction	ChangeHoldTransaction	ChangeMailTransaction
	ChangeMethodTransaction		
PayrollApplication	PayrollApplication		
PayrollDatabase	PayrollDatabase		
PayrollDatabaseImplementation	PayrollDatabaseImplementation		
PayrollDomain	Affiliation	Employee	PaymentClassification
	PaymentMethod	PaymentSchedule	
Schedules	BiweeklySchedule	MonthlySchedule	WeeklySchedule
TextParserTransactionSource	TextParserTransactionSource		
TransactionApplication	TransactionApplication	Transaction	TransactionSource

Таблица 30.2. Метрики для каждого компонента

Имя компонента	*N*	*A*	*Ca*	*Ce*	*R*	*H*	*I*	*A*	*D*	*D′*
Affiliations	2	0	2	1	1	1	0,33	0	0,47	0,67
AffilliationTransactions	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
PayrollDatabaseImpl...	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

Каждая зависимость на рис. 30.6 помечена двумя числами. Число, расположенное ближе к компоненту, из которого исходит стрелка, равно количеству классов в этом компоненте, зависящих от компонента на противоположном конце стрелки. Число на противоположном конце зависимости равно количеству классов в этом компоненте, от которых зависит компонент, на который указывает стрелка.

Для каждого компонента на рис. 30.6 указаны относящиеся к нему метрики. Многие из них радуют. Так, компоненты `PayrollApplication`, `PayrollDomain` и `PayrollDatabase` обладают высокой сцепленностью по связям и одновременно лежат на главной последовательности или близко к ней. Однако для компонентов `Classifications`, `Methods` и `Schedules` и сцепленность мала, и расстояние от главной последовательности почти максимально!

Отсюда мы делаем вывод, что разбиение классов на компоненты неудачно. Если мы не отыщем способа улучшить эти характеристики, то проект будет чувствителен к изменениям, а это потребует выпускать и тестировать компоненты чаще, чем необходимо. В нашем случае имеются компоненты с низкой абстрактностью, например `ClassificationTransaction`, которые сильно зависят от других классов с низкой абстрактностью, например `Classifications`. Классы с низкой абстрактностью содержат большую часть детального кода, поэтому, скорее всего, будут часто изменяться, что повлечет за собой перевыпуск зависящих от них ком-

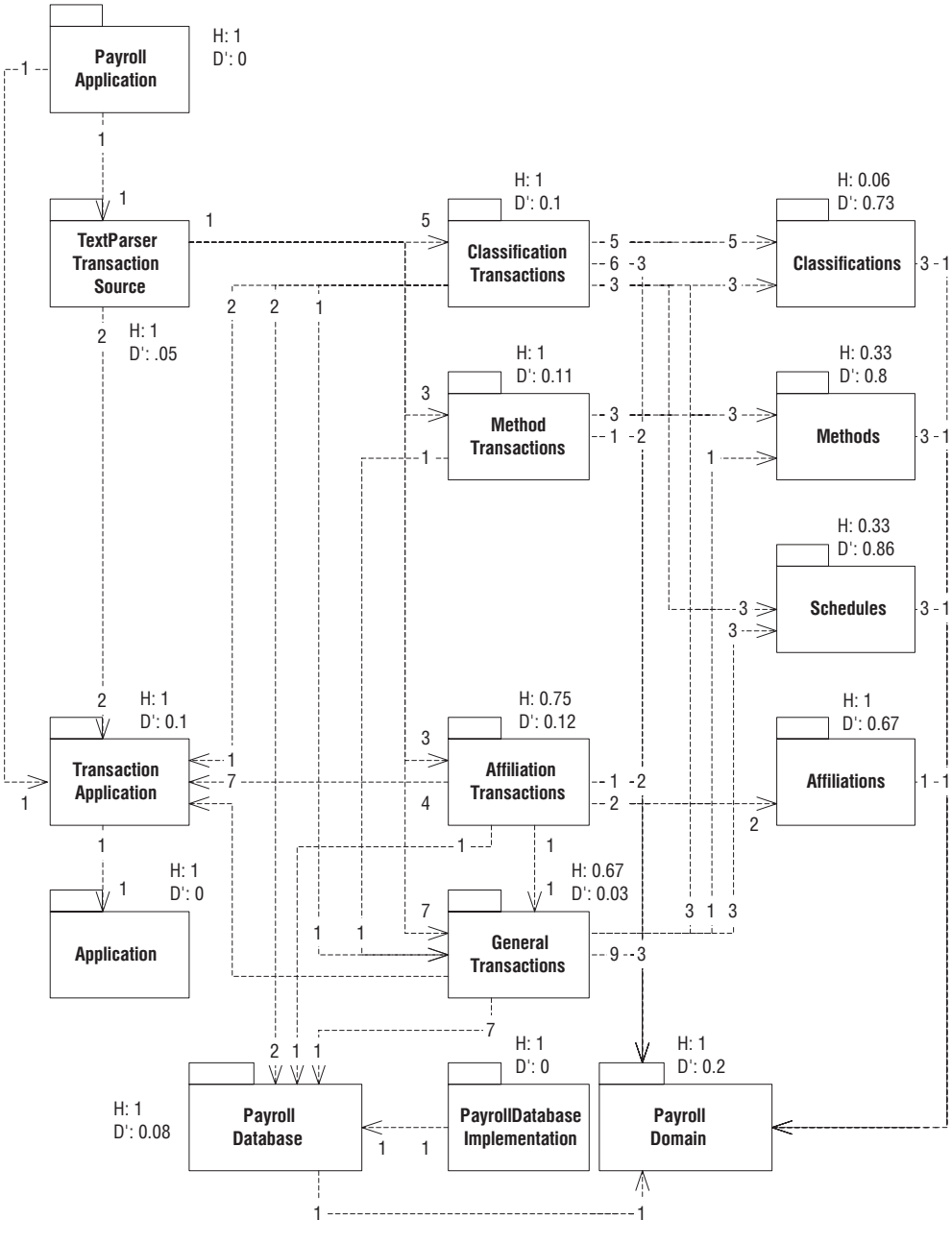


Рис. 30.6. *Диаграмма компонентов с метриками*

понентов. Например, частота выпуска для компонента ClassificationTransaction будет очень высока, поскольку это придется делать после изменений как в нем самом, так и в компоненте Classifications. А мы всеми средствами должны ограничивать чувствительность проекта к изменениям.

Конечно, если разработчиков всего двое или трое, то они смогут удержать весь проект в голове и стремление расположить все компоненты как можно ближе к главной последовательности в этом случае не слишком актуально. Но чем разработчиков больше, чем труднее удерживать процесс под контролем. К тому же получение этих метрик обходится гораздо дешевле, чем даже единственный цикл тестирования и выпуска.[1] Решайте сами, оправданы ли затраты на вычисление метрик в краткосрочной перспективе.

Фабрики объектов

Из компонентов Classifications и ClassificationTransaction исходит так много зависимостей, потому что содержащиеся в них классы приходится инстанцировать. Например, класс TextParserTransactionSource должен уметь создавать объекты типа AddHourlyEmployeeTransaction, поэтому существует входящая связь от пакета TextParserTransactionSource к пакету ClassificationTransactions. Точно так же, класс ChangeHourlyTransaction должен уметь создавать объекты HourlyClassification, поэтому существует входящая связь от ClassificationTransaction к Classifications.

Почти все остальные обращения к объектам внутри этих компонентов производятся через абстрактный интерфейс. Если бы не необходимость создавать каждый конкретный объект, то связей, входящих в эти компоненты, не было бы. Например, если бы классу TextParserTransactionSource не нужно было создавать различные операции, то он не зависел бы от четырех пакетов, содержащих реализации этих операций.

Эту проблему позволяет в значительной степени сгладить паттерн Фабрика. Каждый компонент предоставляет фабрику, которая отвечает за создание объектов всех находящихся внутри него открытых классов.

Фабрика объектов для компонента TransactionImplementation.
На рис. 30.7 показано, как построить фабрику объектов для компонента TransactionImplementation. Компонент TransactionFactory содержит абстрактный базовый класс, в котором объявлены абстрактные методы, играющие роль конструкторов для конкретных операций. А компонент TransactionImplementation содержит конкретный подкласс TransactionFactory и пользуется конкретными классами операций для создания их экземпляров.

---

[1] На ручной сбор статистики и вычисление метрик для системы расчета зарплаты у меня ушло примерно два часа. Если бы я воспользовался коммерческими инструментами, то результат был бы получен практически мгновенно.

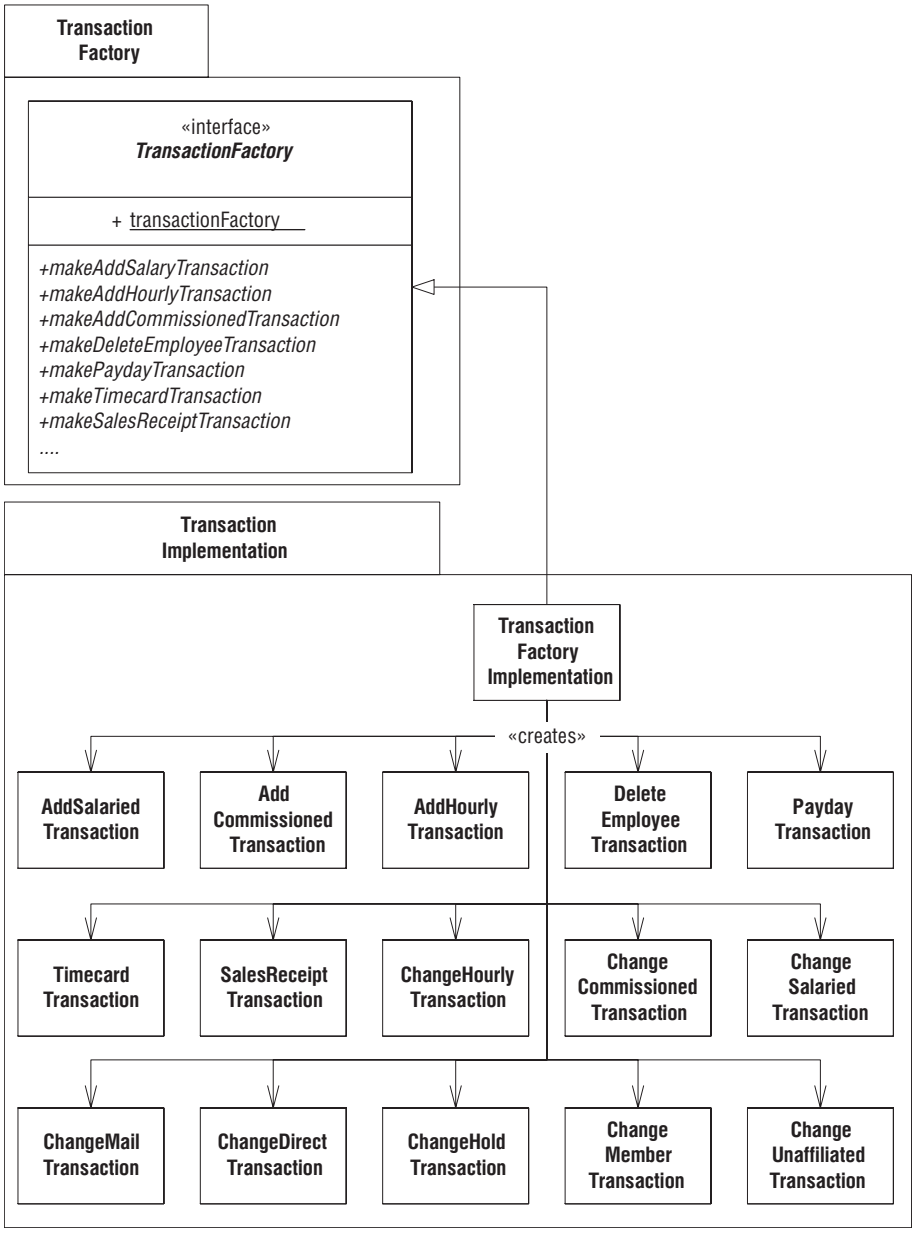


Рис. 30.7. *Фабрика объектов для операций*

В классе `TransactionFactory` имеется статический член, возвращающий ссылку на `TransactionFactory`. Этот член должен быть инициализирован в головной программе так, чтобы он указывал на экземпляр конкретного класса `TransactionFactoryImplementation`.[1]

Инициализация фабрик. Если одни фабрики собираются создавать объекты с помощью других фабрик, то статические члены абстрактных фабрик объектов следует инициализировать так, чтобы они ссылались на подходящую конкретную фабрику. Это необходимо сделать еще до первого обращения к фабрике. Обычно удобнее всего делать это в головной программе, а значит, головная программа будет зависеть от всех без исключения фабрик и от всех *конкретных* пакетов. Поэтому в каждый пакет будет входить хотя бы одна связь, исходящая из головной программы. Из-за этого все конкретные пакеты лежат чуть вне главной последовательности, но тут уж ничего не поделаешь.[2] Отсюда следует, что головную программу придется выпускать после каждого изменения любого конкретного компонента. Впрочем, это все равно пришлось бы делать после каждого изменения, потому что головную программу необходимо тестировать несмотря ни на что. На рис. 30.8 и 30.9 показаны статическая и динамическая структуры головной программы с точки зрения фабрик объектов.

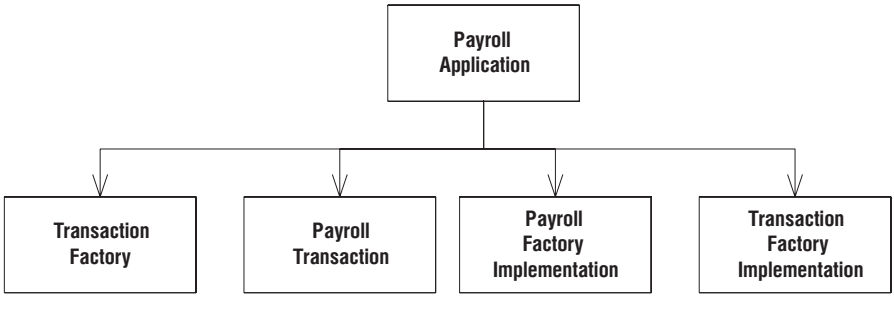


***Рис. 30.8.** Статическая структура головной программы и фабрик объектов*

Еще раз о границах сцепленности

В самом начале на рис. 30.1 мы разделили компоненты `Classifications`, `Methods`, `Schedules` и `Affiliations`. Тогда такое разбиение казалось разумным. Ведь могут же другие пользователи захотеть повторно использовать наши классы графиков и пренебречь классами для описания членства во внешних организациях. Это разбиение сохранилось и после того как операции были перенесены в отдельные компоненты, что привело

[1] Этот и следующий абзацы явно остались от предыдущего издания книги, ориентированного на язык C++. В C# для инициализации статических переменных-членов применяются другие механизмы. – *Прим. перев.*

[2] На практике я обычно игнорирую связи, исходящие из головной программы.

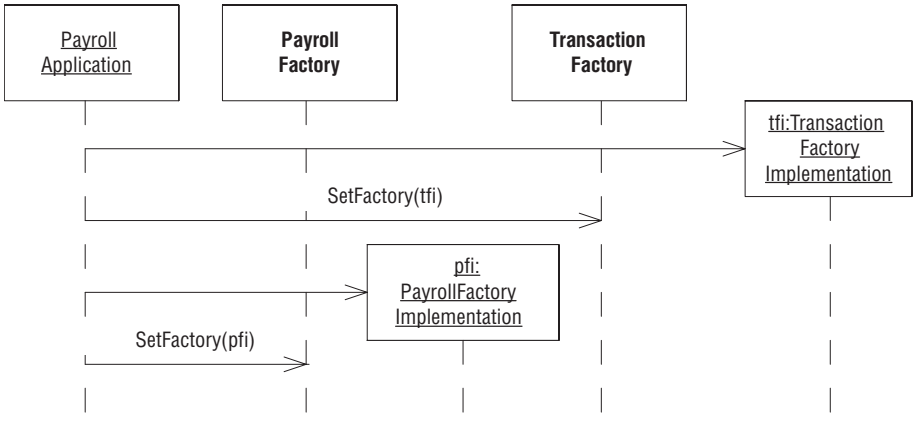


Рис. 30.9. *Динамическая структура головной программы и фабрик объектов*

к образованию двойной иерархии. Пожалуй, это уже чересчур. Диаграмма на рис. 30.6 получилась очень запутанной.

Запутанная диаграмма пакетов усложняет управление выпусками, если оно производится вручную. Хотя диаграммы компонентов неплохо работали бы совместно с автоматизированными средствами планирования проекта, большинство из нас не может позволить себе такой роскоши. Поэтому будем стараться, чтобы диаграммы компонентов были просты настолько, насколько это практически возможно.

На мой взгляд, выделение операций важнее функционального разбиения. Поэтому мы объединим все операции в один компонент `TransactionImplementation`. Кроме того, объединим компоненты `Classifications`, `Schedules`, `Methods` и `Affiliations` в один пакет `PayrollImplementation`.

Окончательная структура пакетов

В табл. 30.3 показано окончательное распределение классов по компонентам, а в табл. 30.4 приведена сводка метрик. На рис. 30.10 изображена окончательная структура компонентов, в которой для приближения конкретных компонентов к главной последовательности применяются фабрики объектов.

Метрики на этой диаграмме радуют глаз. Сцепленность по связям всюду очень высока отчасти благодаря связям фабрик с создаваемыми ими объектами, и заметных отклонений от главной последовательности не наблюдается. Таким образом, структура связей между компонентами способствует спокойной работе над проектом. Наши абстрактные компонен-

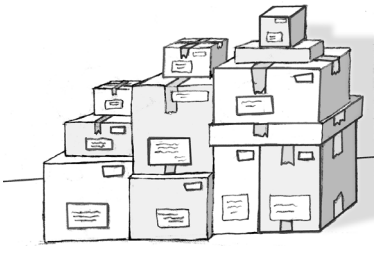


Таблица 30.3. Окончательное распределение классов по компонентам

Компоненты	Классы в компонентах		
AbstractTransactions	AddEmployeeTransaction	ChangeAffiliationTransaction	ChangeEmployeeTransaction
	ChangeClassificationTransaction	ChangeMethodTransaction	
Application	Application		
PayrollApplication	PayrollApplication		
PayrollDatabase	PayrollDatabase		
PayrollDatabaseImplementation	PayrollDatabaseImplementation		
PayrollDomain	Affiliation	Employee	PaymentClassification
	PaymentMethod	PaymentSchedule	
PayrollFactory	PayrollFactory		
PayrollImplementation	BiweeklySchedule	CommissionedClassification	DirectMethod
	HoldMethod	HourlyClassification	MailMethod
	MonthlySchedule	PayrollFactoryImplementation	SalariedClassification
	SalesReceipt	ServiceCharge	Timecard
	UnionAffiliation	WeeklySchedule	
TextParserTransactionSource	TextParserTransactionSource		
TransactionApplication	Transaction	TransactionApplication	TransactionSource
TransactionFactory	TransactionFactory		
TransactionImplementation	AddCommissionedEmployee	AddHourlyEmployee	AddSalariedEmployee
	ChangeAddressTransaction	ChangeCommissionedTransaction	ChangeDirectTransaction
	ChangeHoldTransaction	ChangeHourlyTransaction	ChangeMailTransaction
	ChangeMemberTransaction	ChangeNameTransaction	ChangeSalariedTransaction
	ChangeUnaffiliatedTransaction	DeleteEmployee	PaydayTransaction
	SalesReceiptTransaction	ServiceChargeTransaction	TimecardTransaction
	TransactionFactoryImplementation		

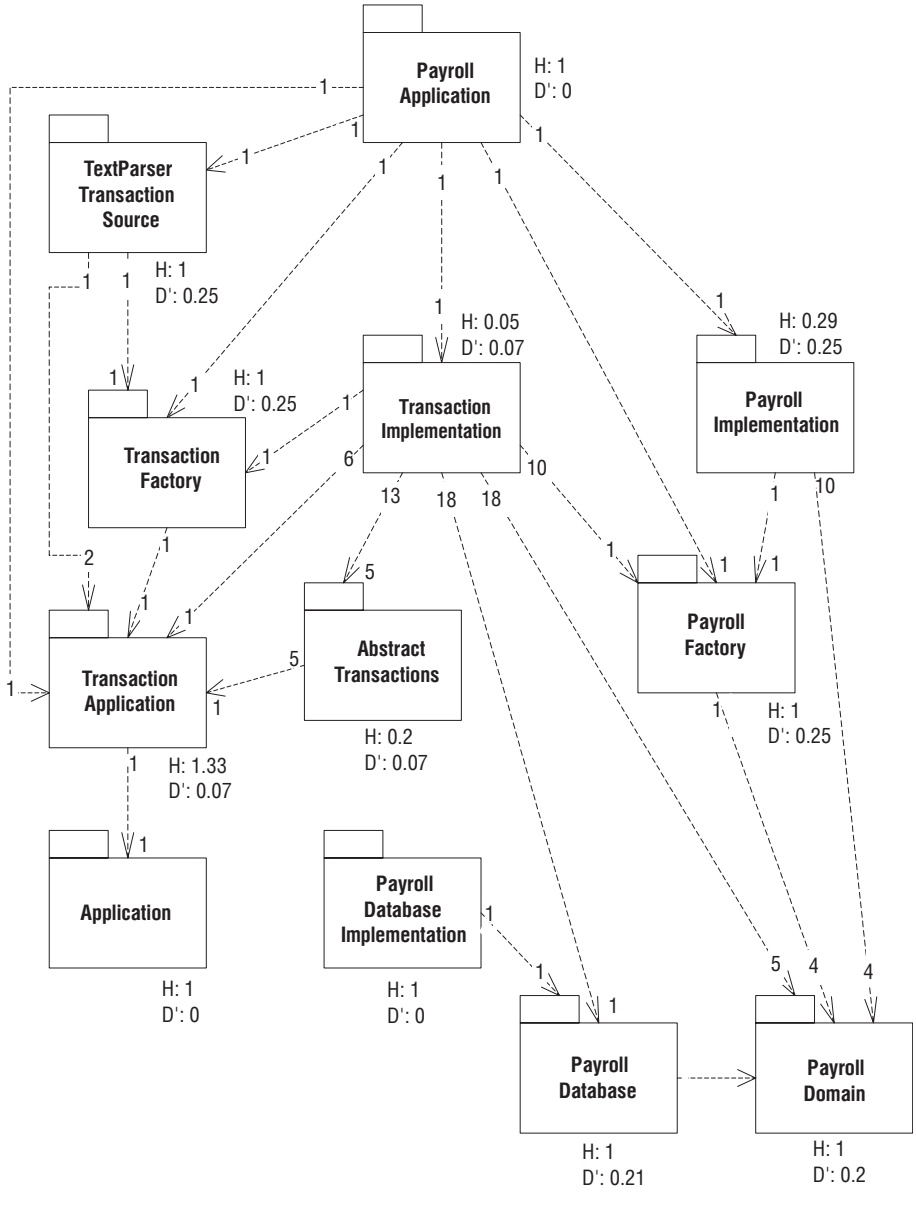


Рис. 30.10. *Окончательная структура компонентов системы расчета зарплаты*

ты закрыты, допускают повторное использование, от них зависят многие другие, но сами они не зависят почти ни от чего. Конкретные компоненты разделены в интересах удобства повторного использования, сильно зависят от абстрактных компонентов, но от них самих зависимостей мало.

Таблица 30.4. Метрики

Имя компонента	*N*	*A*	*Ca*	*Ce*	*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
PayrollDatabase Implementation	1	0	0	1	0	1	1	0	0	0
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	0,25
TransactionImplementation	19	0	1	14	0	0,05	0,93	0	0,05	0,07

Заключение

Потребность в управлении структурой компонентов появляется, когда размеры программы и коллектива разработчиков растут. Даже небольшие команды должны расчленять исходный код, чтобы отдельные программисты не мешали друг другу. А в крупных проектах неорганизованная совокупность исходных файлов может превратиться в сплошную массу, разобраться в которой невозможно. Описанные в этой главе принципы и метрики не раз помогали мне и многим другим командам успешно управлять структурой зависимостей.

Библиография

[Booch94] Grady Booch «Object-Oriented Analysis and Design with Applications», 2d ed., Addison-Wesley, 1994.

[DeMarco82] Tom DeMarco «Controlling Software Projects», Yourdon Press, 1982.

31

Компоновщик



Монтаж – это эвфемизм лжи. Это вносит путаницу. Это бесчестно, и это не журналистика.

Фред Френдли (1984)

Компоновщик (Composite) – очень простой паттерн, имеющий широкое применение. Его принципиальная структура показана на рис. 31.1. Здесь мы видим иерархию классов геометрических фигур. У базового класса `Shape` есть два подкласса: `Circle` и `Square`. Третьим подклассом является компоновщик. В классе `CompositeShape` хранится список объектов типа `Shape`. Метод `Draw()` в этом классе последовательно вызывает метод `Draw()` каждого объекта в списке.

Таким образом, экземпляр `CompositeShape` выглядит для системы как один объект `Shape`. Его можно передать любому методу, принимающему

Shape, и он будет вести себя, как Shape. Однако в действительности это заместитель[1] группы объектов Shape. В листингах 31.1 и 31.2 приведена одна из возможных реализаций класса CompositeShape.

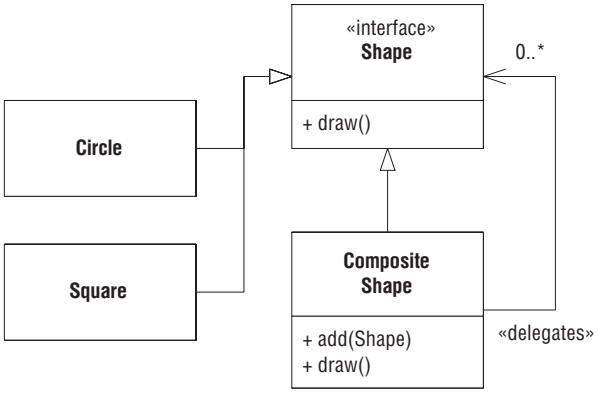


Рис. 31.1. *Паттерн Компоновщик*

Листинг 31.1. Shape.cs

```
public interface Shape
{
  void Draw();
}
```

Листинг 31.2. CompositeShape.cs

```
using System.Collections;

public class CompositeShape : Shape
{
  private ArrayList itsShapes = new ArrayList();

  public void Add(Shape s)
  {
    itsShapes.Add(s);
  }

  public void Draw()
  {
    foreach (Shape shape in itsShapes)
    shape.Draw();
  }
}
```

[1] Обратите внимание на сходство структуры этого паттерна и паттерна Заместитель.

Составные команды

Вспомним обсуждение объектов Sensor и Command в главе 21. На рис. 21.3 показан класс Sensor, использующий класс Command. Обнаружив событие, объект-датчик Sensor вызывал метод Do() ассоциированного с ним объекта Command.

Но в тот раз я не упомянул, что часто Sensor должен выполнять несколько команд. Например, дойдя до определенного места на тракте подачи, лист бумаги закрывает оптический датчик. В этот момент датчик останавливает один двигатель, запускает другой и включает определенную муфту.

Впервые столкнувшись с такой ситуацией, мы решили, что с каждым объектом Sensor надо ассоциировать список объектов Command (рис. 31.2). Но скоро поняли, что при необходимости выполнить несколько команд датчик обращается со всеми объектами Command одинаково. Он просто обходит список и для каждого объекта вызывает метод Do(). Идеальные предпосылки для применения паттерна Компоновщик.

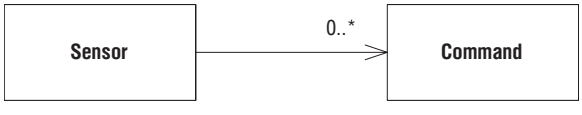


Рис. 31.2. *Объект Sensor, с которым ассоциировано несколько объектов Command*

Поэтому мы оставили класс Sensor в покое и ввели класс CompositeCommand, показанный на рис. 31.3. И значит, нам не пришлось изменять ни Sensor, ни Command. Мы сумели ассоциировать с объектом Sensor несколько объектов Command вместо одного, не изменив ни тот ни другой класс. Это полностью согласуется с принципом OCP.

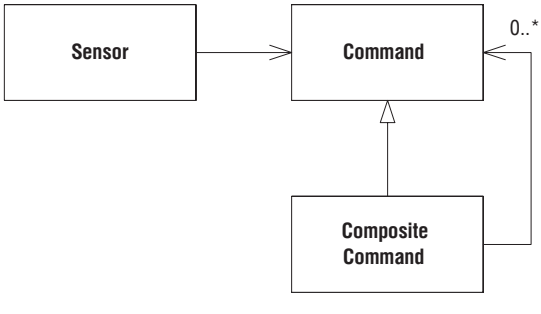


Рис. 31.3. *Класс CompositeCommand*

Кратный или не кратный

Это подводит нас к любопытному вопросу. Мы заставили `Sensor` вести себя так, будто с ним ассоциировано много команд, не изменив самого класса `Sensor`. При проектировании программ такая ситуация наверняка не является экстраординарной. Наверное, в некоторых случаях можно использовать Компоновщик вместо списка или массива объектов.

Иными словами, между классами `Sensor` и `Command` существует ассоциация один-к-одному. У нас появилось мимолетное искушение изменить ее тип на один-ко-многим. Но потом мы нашли способ имитировать поведение 1:*N*, не изменяя тип ассоциации. Связь 1:1 гораздо проще для понимания, кодирования и сопровождения, поэтому такое проектное решение, безусловно, правильно. А сколько ассоциаций 1:*N* можно заменить на 1:1 в вашем текущем проекте, если воспользоваться Компоновщиком?

Разумеется, Компоновщик позволяет преобразовать не все связи 1:*N* в 1:1, а только такие, где все объекты списка обрабатываются в точности одинаково. Например, если имеется список работников, в котором нужно найти тех, кому надо сегодня рассчитать зарплату, то вряд ли стоит применять паттерн Компоновщик, так как не все работники обрабатываются одинаково.

Заключение

Довольно много связей типа один-ко-многим имеет смысл упростить, воспользовавшись паттерном Компоновщик. Достоинства весьма существенны. Вместо того чтобы дублировать код обхода списка в каждом клиенте, его можно реализовать один раз в классе-компоновщике.

32

Наблюдатель: превращение в паттерн



Свою профессию я предпочитаю называть
«интерактивный наблюдатель современных
антропологических типов» – в этом есть изюминка.
К тому же «филер» – такое противное слово.

Анонимный автор

У этой главы особая цель. Да, я описываю паттерн Наблюдатель (Observer)[1], но не это главное. А главная задача – продемонстрировать, как в процессе эволюции дизайна и кода можно прийти к использованию паттерна.

В предыдущих главах мы много раз пользовались паттернами. Часто мы представляли решение задачи, ничего не говоря о том, как приш-

[1] [GOF95], p. 293

ли к применению паттерна. У вас могло сложиться впечатление, будто паттерны просто вставляются в дизайн и код в готовом виде. Но не это я рекомендую. Я предпочитаю развивать код, над которым работаю, в направлении паттерна. Возможно, паттерн проявится, а возможно, и нет. Все зависит от того, как разрешаются стоящие передо мной проблемы. Не так уж редко бывает, что я начинаю кодировать, имея в виду определенный паттерн, а в конце получается нечто совсем иное.

В этой главе мы поставим перед собой простую задачу и покажем, как в процессе ее решения эволюционировали дизайн и код. И плодом эволюции стал паттерн Наблюдатель. На каждом этапе я буду объяснять, какое проблемы собираюсь разрешить, а потом показывать шаги, приведшие к решению. И вот так, полагаясь на удачу, мы и доберемся до Наблюдателя.

Цифровые часы

Пусть уже имеется объект, представляющий часы. Каждую миллисекунду он получает от операционной системы прерывания, называемые тиками, и на их основе выводит текущее время. Объект знает, как преобразовывать миллисекунды в секунды, секунды в минуты, минуты в часы, часы в дни и т. д. Он знает, сколько дней в месяце и сколько месяцев в году. Ему известно о високосных годах. Короче говоря, о времени он знает все. См. рис. 32.1.

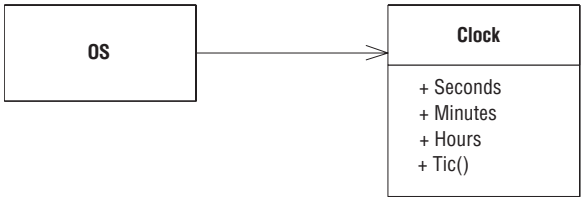


***Рис. 32.1.** Объект Clock*

Мы хотим написать приложение для отображения цифровых часов, которые будут находиться на рабочем столе и показывать время. Как это проще всего сделать? Можно написать что-то в таком роде:

```
public void DisplayTime()
{
  while (true)
  {
      int sec = clock.Seconds;
      int min = clock.Minutes;
      int hour = clock.Hours;
      ShowTime(hour, min, sec);
  }
}
```

Но это далеко не оптимальное решение. Такая программа потребляет все процессорные ресурсы, отображая время в бесконечном цикле. Большая часть итераций цикла бессмысленна, потому что время не изменяется. Быть может, в цифровых наручных или настенных часах это и приемлемо, потому что там экономить время процессора ни к чему. Но иметь такого «пожирателя» ресурсов на рабочем столе мы не желаем.

Похоже, путь от получения прерываний таймера до отображения времени на экране будет нелегким. Каким бы механизмом воспользоваться? Но первым делом я должен задаться другим вопросом: как проверить, делает ли выбранный механизм то, что мне требуется?

Стоящая передо мной задача заключается в том, как передать данные от `Clock` в `DigitalClock`. Предположим пока, что оба объекта уже существуют. Проблема в том, как их связать между собой. А протестировать эту связь я смогу, просто убедившись, что `DigitalClock` получает именно те данные, что отправил `Clock`.

Можно, например, создать два интерфейса: один будет представлять `Clock`, другой — `DigitalClock`. Потом я напишу специальные тестовые объекты-имитации, которые реализуют эти интерфейсы и проверят, что связь между ними работает, как и задумано. См. рис. 32.2.

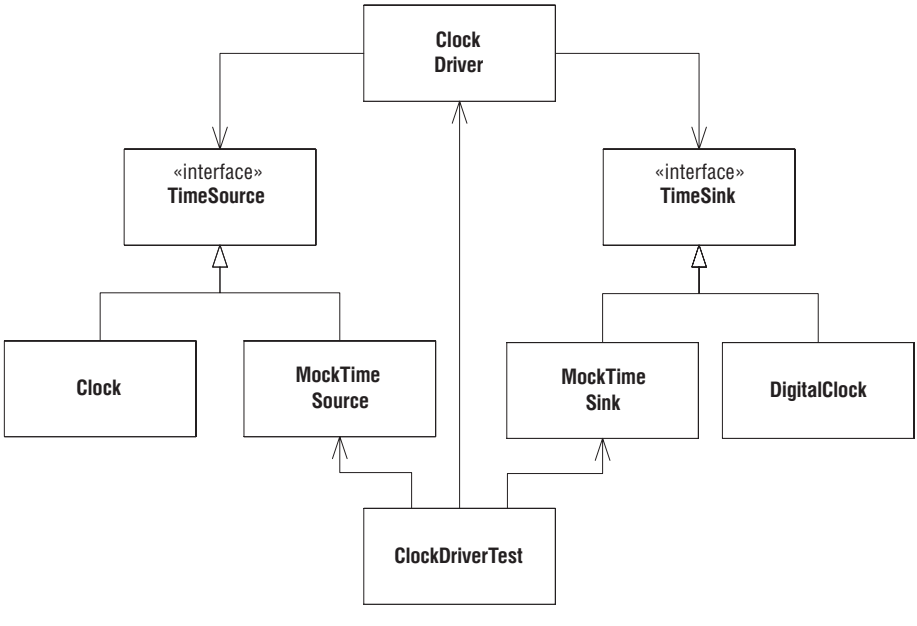


Рис. 32.2. *Тестирование DigitalClock*

Объект `ClockDriverTest` соединяет `ClockDriver` с двумя объектами-имитациями через интерфейсы `TimeSource` и `TimeSink` и затем опрашивает оба объекта, дабы убедиться, что `ClockDriver` правильно передает время от

источника к приемнику. Если необходимо, ClockDriverTest также позаботится о том, чтобы не страдала эффективность.

Интересно отметить следующее: интерфейсы появились в дизайне только потому, что мы задумались о том, как провести тестирование. Чтобы протестировать модуль, его необходимо изолировать от других модулей в системе, что мы и проделали, изолировав ClockDriver от Clock и DigitalClock. Начиная с разработки тестов, мы уменьшаем связанность проектируемой системы.

Но как все-таки работает ClockDriver? Понятно, что для обеспечения эффективности ClockDriver должен обнаруживать, когда изменилось время в объекте TimeSource. Тогда и только тогда он будет передавать данные о времени в объект TimeSink. Но как ClockDriver узнает, что время изменилось? Он мог бы постоянно опрашивать TimeSource, но тогда мы вернемся к исходной проблеме «пожирания» времени.

Проще всего если объект Clock будет извещать ClockDriver о том, что время изменилось. Мы могли бы передать в Clock ссылку на ClockDriver через интерфейс TimeSource; тогда при каждом изменении времени Clock сможет обновить ClockDriver. А ClockDriver, в свою очередь, установит время в ClockSink (рис. 32.3).

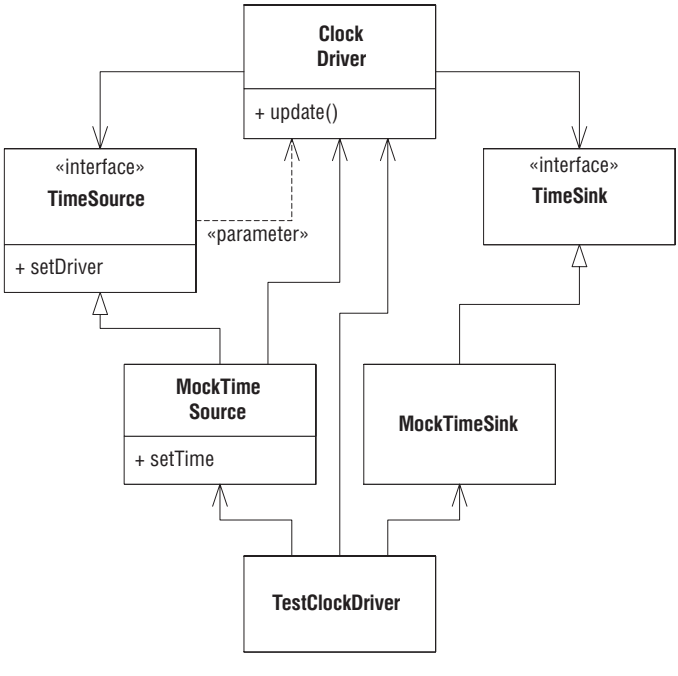


***Рис. 32.3.** TimeSource обновляет ClockDriver*

Обратите внимание на зависимость TimeSource от ClockDriver. Она существует, потому что аргумент метода SetDriver имеет тип ClockDriver. Не

могу сказать, что мне это нравится, так как объекты TimeSource должны будут использовать ClockDriver в любом случае. Но повременю решать эту проблему, пока не получу нечто работающее.

В листинге 32.1 приведен тест для ClockDriver. Мы создаем объект ClockDriver, привязываем к нему MockTimeSource и MockTimeSink, затем задаем время в источнике и ожидаем, что оно магическим образом попадет в приемник. Прочий код показан в листингах 32.2–32.6.

Листинг 32.1. ClockDriverTest.cs

```
using NUnit.Framework;

[TestFixture]
public class ClockDriverTest
{
  [Test]
  public void TestTimeChange()
  {
    MockTimeSource source = new MockTimeSource();
    MockTimeSink sink = new MockTimeSink();
    ClockDriver driver = new ClockDriver(source,sink);
    source.SetTime(3,4,5);
    Assert.AreEqual(3, sink.GetHours());
    Assert.AreEqual(4, sink.GetMinutes());
    Assert.AreEqual(5, sink.GetSeconds());

    source.SetTime(7,8,9);
    Assert.AreEqual(7, sink.GetHours());
    Assert.AreEqual(8, sink.GetMinutes());
    Assert.AreEqual(9, sink.GetSeconds());
  }
}
```

Листинг 32.2. ClockDriverTest.cs

```
public interface TimeSource
{
  void SetDriver(ClockDriver driver);
}
```

Листинг 32.3. TimeSink.cs

```
public interface TimeSink
{
  void SetTime(int hours, int minutes, int seconds);
}
```

Листинг 32.4. ClockDriver.cs

```
public class ClockDriver
{
  private readonly TimeSink sink;
```

```
  public ClockDriver(TimeSource source, TimeSink sink)
  {
    source.SetDriver(this);
    this.sink = sink;
  }

  public void Update(int hours, int minutes, int seconds)
  {
    sink.SetTime(hours, minutes, seconds);
  }
}
```

Листинг 32.5. MockTimeSource.cs

```
public class MockTimeSource : 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 32.6. MockTimeSink.cs

```
public class MockTimeSink : TimeSink
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;

  public int GetHours()
  {
    return itsHours;
  }

  public int GetMinutes()
  {
    return itsMinutes;
  }

  public int GetSeconds()
  {
    return itsSeconds;
  }

  public void SetTime(int hours, int minutes, int seconds)
  {
```

```
      itsHours = hours;
      itsMinutes = minutes;
      itsSeconds = seconds;
    }
  }
```

Ну вот, заработало, теперь можно подумать о том, как почистить код. Мне не по душе зависимость, идущая от `TimeSource` к `ClockDriver`. Я бы хотел, чтобы интерфейсом `TimeSource` мог пользоваться кто угодно, а не только объекты `ClockDriver`. Это можно поправить, создав интерфейс, который `TimeSource` сможет использовать, а `ClockDriver` – реализовать (рис. 32.4). Назовем этот интерфейс `ClockObserver`. См. листинги 32.7–32.10. Полужирным шрифтом выделен измененный код.

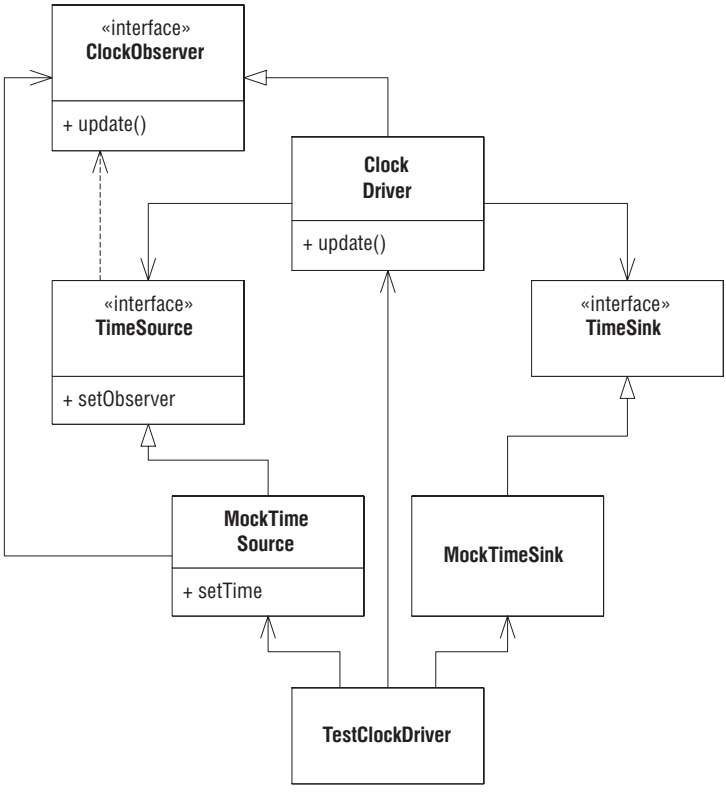


Рис. 32.4. *Разрыв зависимости TimeSource от ClockDriver*

Листинг 32.7. ClockObserver.cs

```
public interface ClockObserver
{
  void Update(int hours, int minutes, int secs);
}
```

Листинг 32.8. ClockDriver.cs

```
public class ClockDriver : ClockObserver
{
  private readonly TimeSink sink;

  public ClockDriver(TimeSource source, TimeSink sink)
  {
    source.SetObserver(this);
    this.sink = sink;
  }

  public void Update(int hours, int minutes, int seconds)
  {
    sink.SetTime(hours, minutes, seconds);
  }
}
```

Листинг 32.9. TimeSource.cs

```
public interface TimeSource
{
  void SetObserver(ClockObserver observer);
}
```

Листинг 32.10. MockTimeSource.cs

```
public class MockTimeSource : 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;
  }
}
```

Вот так-то лучше. Теперь воспользоваться `TimeSource` может любой объект, который реализует интерфейс `ClockObserver` и вызывает метод `SetObserver`, передавая себя в качестве аргумента.

Я хотел бы сделать так, чтобы одновременно получать время могли несколько объектов `TimeSink`. Один будет отображать цифровые часы, другой – служить источником времени для механизма напоминания, третий – запускать ночью резервное копирование. Короче говоря, один объект `TimeSource` должен обслуживать несколько объектов `TimeSink`.

Как это сделать? Сейчас при создании `ClockDriver` я задаю один `TimeSource` и один `TimeSink`. А как задать несколько экземпляров `TimeSink`?

Можно было бы изменить конструктор ClockDriver так, чтобы он принимал только TimeSource, и добавить метод AddTimeSink, который позволит подключать к источнику произвольное число объектов TimeSink.

Но мне не нравится в этом решении то, что теперь у меня целых два уровня косвенности. Сначала я информирую объект TimeSource о том, кто является наблюдателем ClockObserver, вызывая метод SetObserver. Затем я сообщаю ClockDriver, какие объекты TimeSink будут приемниками. Так ли уж необходима подобная двойная косвенность?

Посмотрев еще раз на типы ClockObserver и TimeSink, я замечаю, что в обоих есть метод SetTime. Получается, что TimeSink мог бы сам реализовать интерфейс ClockObserver. Если так и сделать, то моя тестовая программа могла бы создать MockTimeSink и передать его методу SetObserver объекта TimeSource. Тогда я вообще избавился бы от ClockDriver и TimeSink! В листинге 32.11 показаны изменения в коде ClockDriverTest.

Листинг 32.11. ClockDriverTest.cs

```
using NUnit.Framework;

[TestFixture]
public class ClockDirverTest
{
  [Test]
  public void TestTimeChange()
  {
    MockTimeSource source = new MockTimeSource();
    MockTimeSink sink = new MockTimeSink();
    source.SetObserver(sink);

    source.SetTime(3,4,5);
    Assert.AreEqual(3, sink.GetHours());
    Assert.AreEqual(4, sink.GetMinutes());
    Assert.AreEqual(5, sink.GetSeconds());

    source.SetTime(7,8,9);
    Assert.AreEqual(7, sink.GetHours());
    Assert.AreEqual(8, sink.GetMinutes());
    Assert.AreEqual(9, sink.GetSeconds());
  }
}
```

Это означает, что MockTimeSink теперь должен реализовать интерфейс ClockObserver, а не TimeSink. См. листинг 32.12. Все замечательно работает. С чего мы вдруг поначалу решили, что нам нужен класс ClockDriver? На рис. 32.5 изображена UML-диаграмма. Очевидно, она стала гораздо проще.

Листинг 32.12. MockTimeSink.cs

```
public class MockTimeSink : ClockObserver
{
```

```
    private int itsHours;
    private int itsMinutes;
    private int itsSeconds;

    public int GetHours()
    {
      return itsHours;
    }

    public int GetMinutes()
    {
      return itsMinutes;
    }

    public int GetSeconds()
    {
      return itsSeconds;
    }

    public void Update(int hours, int minutes, int secs)
    {
      itsHours = hours;
      itsMinutes = minutes;
      itsSeconds = secs;
    }
  }
```

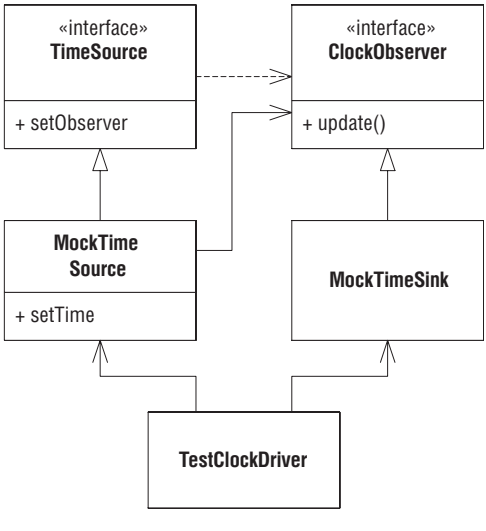


Рис. 32.5. *Исключение ClockDriver и TimeSink*

Ну вот, теперь мы можем поддержать несколько объектов `TimeSink`, переименовав метод `SetObserver` в `RegisterObserver` и помещая все зарегистрированные экземпляры `ClockObserver` в список, должным образом

обновляемый. Для этого потребуется внести еще одно изменение в тестовую программу (см. листинг 32.13). Заодно я немного переработал тестовую программу – она стала меньше и понятнее.

Листинг 32.13. ClockDriverTest.cs

```
using NUnit.Framework;

[TestFixture]
public class ClockDriverTest
{
  private MockTimeSource source;
  private MockTimeSink sink;

  [SetUp]
  public void SetUp()
  {
    source = new MockTimeSource();
    sink = new MockTimeSink();
    source.RegisterObserver(sink);
  }

  private void AssertSinkEquals(
    MockTimeSink sink, int hours, int mins, int secs)
  {
    Assert.AreEqual(hours, sink.GetHours());
    Assert.AreEqual(mins, sink.GetMinutes());
    Assert.AreEqual(secs, sink.GetSeconds());
  }

   [Test]
  public void TestTimeChange()
  {
    source.SetTime(3,4,5);
    AssertSinkEquals(sink, 3,4,5);

    source.SetTime(7,8,9);
    AssertSinkEquals(sink, 7,8,9);
  }

   [Test]
  public void TestMultipleSinks()
  {
    MockTimeSink sink2 = new MockTimeSink();
    source.RegisterObserver(sink2);
    source.SetTime(12,13,14);
    AssertSinkEquals(sink, 12,13,14);
    AssertSinkEquals(sink2, 12,13,14);
  }
}
```

Чтобы все это заработало, потребовалась довольно простая модификация. Мы изменили `MockTimeSource` так, чтобы все зарегистрированные наблюдатели хранились в списке `ArrayList`. При каждой смене времени мы обходим этот список и вызываем метод `Update` для каждого зарегистрированного объекта `ClockObserver`. Изменения показаны в листингах 32.14 и 32.15, а соответствующая UML-диаграмма – на рис. 32.6.

Листинг 32.14. TimeSource.cs

```
public interface TimeSource
{
  void RegisterObserver(ClockObserver observer);
}
```

Листинг 32.15. MockTimeSource.cs

```
using System.Collections;

public class MockTimeSource : TimeSource
{
  private ArrayList itsObservers = new ArrayList();

  public void SetTime(int hours, int mins, int secs)
  {
    foreach(ClockObserver observer in itsObservers)
      observer.Update(hours, mins, secs);
  }

  public void RegisterObserver(ClockObserver observer)
  {
    itsObservers.Add(observer);
  }
}
```

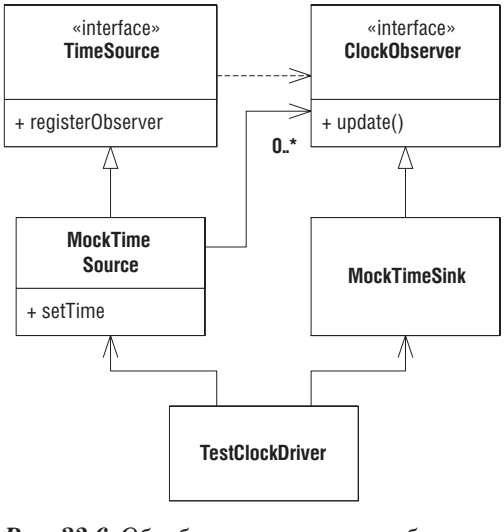


***Рис. 32.6.** Обработка нескольких объектов TimeSink*

Неплохо, но мне не нравится тот факт, что MockTimeSource должен заниматься регистрацией и обновлением списка. Отсюда следует, что класс Clock и любой другой класс, производный от TimeSource, должны будут дублировать код регистрации и обновления. Не думаю, что эта обязанность подходит классу Clock. Да и идея дублирования кода не приводит меня в восторг. Я хотел бы перенести все это в TimeSource. Конечно, это означает, что TimeSource придется преобразовать из интерфейса в класс. При этом в MockTimeSource почти ничего не останется. Требуемые изменения показаны в листингах 32.16 и 32.17, а также на рис. 32.7.

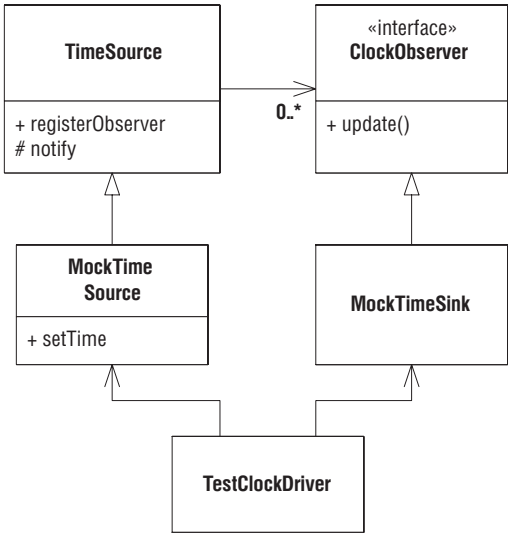


Рис. 32.7. *Перенос регистрации и обновления в класс TimeSource*

Листинг 32.16. TimeSource.cs

```
using System.Collections;

public abstract class TimeSource
{
  private ArrayList itsObservers = new ArrayList();

  protected void Notify(int hours, int mins, int secs)
  {
    foreach(ClockObserver observer in itsObservers)
      observer.Update(hours, mins, secs);
  }

  public void RegisterObserver(ClockObserver observer)
  {
    itsObservers.Add(observer);
  }
}
```

Листинг 32.17. MockTimeSource.cs

```
public class MockTimeSource : TimeSource
{
  public void SetTime(int hours, int mins, int secs)
  {
    Notify(hours, mins, secs);
  }
}
```

Очень хорошо. Теперь любой источник времени можно произвести от `TimeSource`. Чтобы известить наблюдателей, нужно лишь вызвать метод `Notify`. Но кое-что меня все-таки не устраивает. Класс `MockTimeSource` наследует непосредственно `TimeSource`. Это означает, что `Clock` также должен быть производным от `TimeSource`. Но с какой стати `Clock` должен зависеть от регистрации и обновления? Классу `Clock` необходимо знать только о времени и ни о чем больше. Его зависимость от `TimeSource` выглядит ненужной и нежелательной.

Я знаю, как решил бы эту проблему на C++. Создал бы класс `ObservableClock`, наследующий одновременно `TimeSource` и `Clock`, и переопределил бы функции `Tic` и `SetTime` в `ObservableClock` так, чтобы они вызывали соответственно функции `Tic` и `SetTime` класса `Clock`, а затем — метод `Notify` класса `TimeSource`. См. рис. 32.8 и листинг 32.18.

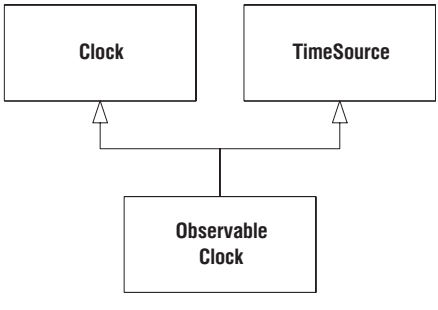


***Рис. 32.8.** Применение множественного наследования в C++ для разделения Clock и TimeSource*

Листинг 32.18. 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, в C# так не получится, потому что в этом языке множественное наследование не поддерживается. Поэтому придется либо оставить все, как есть, либо воспользоваться трюком с делегированием, который показан в листингах 32.19–32.21 и на рис. 32.9.

Обратите внимание, что класс `MockTimeSource` реализует интерфейс `TimeSource` и хранит ссылку на экземпляр `TimeSourceImplementation`. Еще заметьте, что все обращения к методу `RegisterObserver` объекта `MockTimeSource` делегируются объекту `TimeSourceImplementation`. А метод `MockTimeSource.SetTime` вызывает метод `Notify` объекта `TimeSourceImplementation`.

Листинг 32.19. TimeSource.cs

```
public interface TimeSource
{
  void RegisterObserver(ClockObserver observer);
}
```

Листинг 32.20. TimeSourceImplementation.cs

```
using System.Collections;

public class TimeSourceImplementation : TimeSource
{
  private ArrayList itsObservers = new ArrayList();

  public void Notify(int hours, int mins, int secs)
  {
    foreach(ClockObserver observer in itsObservers)
      observer.Update(hours, mins, secs);
  }

  public void RegisterObserver(ClockObserver observer)
  {
    itsObservers.Add(observer);
  }
}
```

Листинг 32.21. MockTimeSource.cs

```
public class MockTimeSource : TimeSource
{
  TimeSourceImplementation timeSourceImpl =
    new TimeSourceImplementation();

  public void SetTime(int hours, int mins, int secs)
  {
```

```
      timeSourceImpl.Notify(hours, mins, secs);
    }

    public void RegisterObserver(ClockObserver observer)
    {
      timeSourceImpl.RegisterObserver(observer);
    }
  }
```

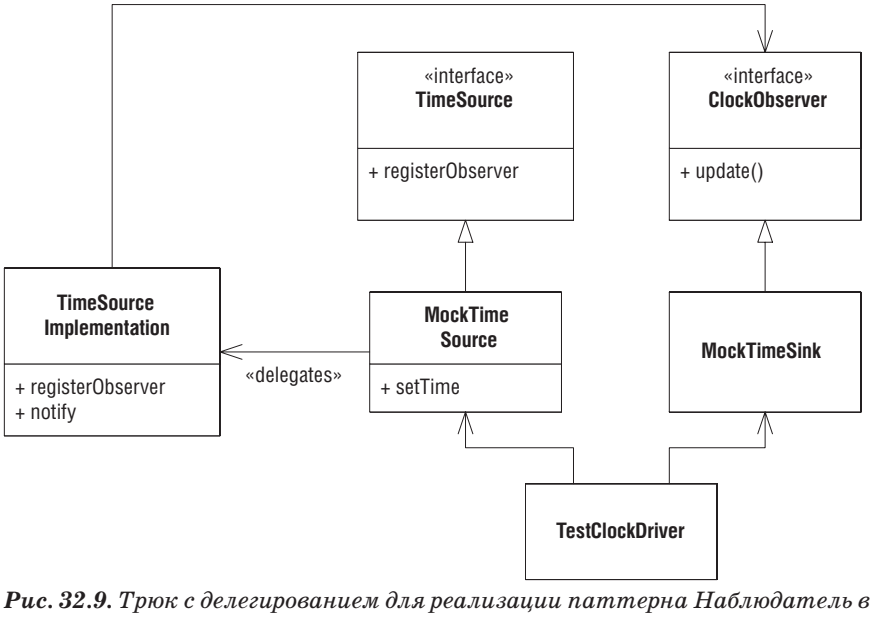


Рис. 32.9. *Трюк с делегированием для реализации паттерна Наблюдатель в C#*

Это не очень красивый подход, но у него есть то преимущество, что `MockTimeSource` не наследует никакому классу. Это означает, что если бы нам нужно было создать класс `ObservableClock`, то мы могли бы унаследовать `Clock`, реализовать интерфейс `TimeSource` и делегировать работу классу `TimeSourceImplementation` (см. рис. 32.10). Тем самым проблема зависимости `Clock` от регистрации и обновления решена, но ценой дополнительных затрат.

А теперь давайте вернемся к той ситуации, которая изображена на рис. 32.7, – до того, как мы отвлеклись. Нам просто придется смириться с тем фактом, что `Clock` должен зависеть от регистрации и обновления.

`TimeSource` – неудачное имя, не отражающее назначения класса. Когда у нас еще был `ClockDriver`, оно не вызывало нареканий. Но с тех пор все изменилось. Необходимо подобрать другое имя, отражающее функцию регистрации и обновления. В паттерне Наблюдатель такой класс называется `Subject`. Поскольку в нашем случае речь идет о времени, то можно было бы назвать его `TimeSubject`, но это имя интуитивно не понятно.

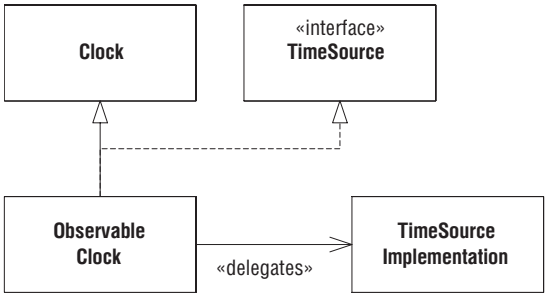


***Рис. 32.10.** Трюк с делегированием при реализации класса ObservableClock*

Можно было бы воспользоваться именем `Observable`, но лично мне оно тоже ничего не говорит. `TimeObservable`? Не подойдет.

Быть может, проблема возникла из-за того, что наблюдатель реализован согласно «модели проталкивания»?[1] Если перейти на «модель вытягивания», то класс можно было бы сделать общим. Тогда мы переименовали бы `TimeSource` в `Subject`, и всякий, кто знаком с паттерном Наблюдатель, понял бы, о чем речь.

Это неплохой вариант. Вместо того чтобы передавать время в методах `Notify` и `Update`, мы можем заставить `TimeSink` запрашивать время у `MockTimeSource`. Поскольку мы не хотим, чтобы `MockTimeSink` знал о `MockTimeSource`, то введем интерфейс, которым `MockTimeSink` сможет воспользоваться для получения времени. Классы `MockTimeSource` и `Clock` будут реализовывать этот интерфейс, который мы назовем `TimeSource`. Окончательный код и UML-диаграмма показаны на рис. 32.11 и в листингах 32.22–32.27.

Листинг 32.22. ObserverTest.cs

```
using NUnit.Framework;

[TestFixture]
public class ObserverTest
{
  private MockTimeSource source;
  private MockTimeSink sink;

  [SetUp]
  public void SetUp()
  {
    source = new MockTimeSource();
```

[1] В модели с проталкиванием наблюдатель «проталкивает» полученные от субъекта данные в виде аргументов методов `Notify` и `Update`. В модели с вытягиванием наблюдатель ничего не передает в методах `Notify` и `Update`, считая, что наблюдающий объект обратится к наблюдаемому для получения обновлений. См. [GOF95].

```
    sink = new MockTimeSink();
    source.RegisterObserver(sink);
  }

  private void AssertSinkEquals(
    MockTimeSink sink, int hours, int mins, int secs)
  {
    Assert.AreEqual(hours, sink.GetHours());
    Assert.AreEqual(mins, sink.GetMinutes());
    Assert.AreEqual(secs, sink.GetSeconds());
  }

  [Test]
  public void TestTimeChange()
  {
    source.SetTime(3,4,5);
    AssertSinkEquals(sink, 3,4,5);
    source.SetTime(7,8,9);
    AssertSinkEquals(sink, 7,8,9);
  }

  [Test]
  public void TestMultipleSinks()
  {
    MockTimeSink sink2 = new MockTimeSink();
    source.RegisterObserver(sink2);
    source.SetTime(12,13,14);
    AssertSinkEquals(sink, 12,13,14);
    AssertSinkEquals(sink2, 12,13,14);
  }
}
```

Листинг 32.23. Observer.cs

```
public interface Observer
{
  void Update();
}
```

Листинг 32.24. Subject.cs

```
using System.Collections;

public class Subject
{
  private ArrayList itsObservers = new ArrayList();

  public void NotifyObservers()
  {
    foreach(Observer observer in itsObservers)
      observer.Update();
  }
```

```
  public void RegisterObserver(Observer observer)
  {
    itsObservers.Add(observer);
  }
}
```

Листинг 32.25. TimeSource.cs

```
public interface TimeSource
{
  int GetHours();
  int GetMinutes();
  int GetSeconds();
}
```

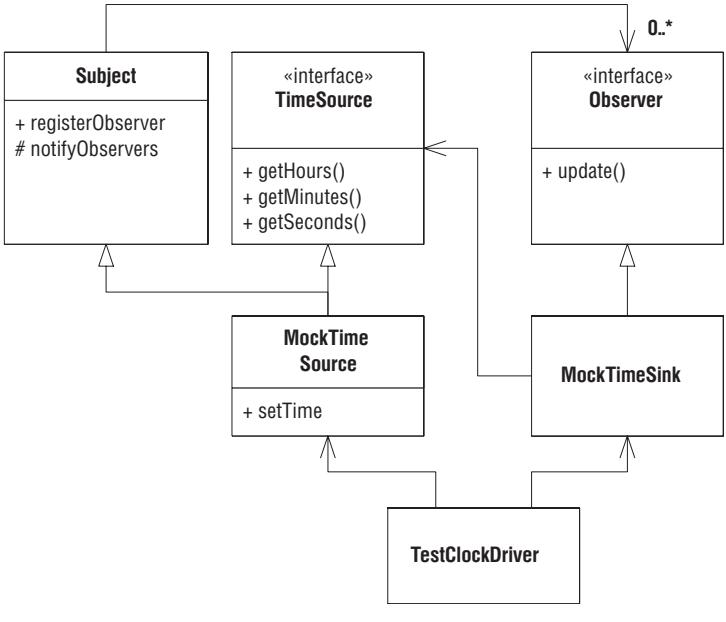


Рис. 32.11. *Окончательная версия паттерна Наблюдатель в применении к MockTimeSource и MockTimeSink*

Листинг 32.26. MockTimeSource.cs

```
public class MockTimeSource : Subject, TimeSource
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;

  public void SetTime(int hours, int mins, int secs)
  {
    itsHours = hours;
    itsMinutes = mins;
```

```
    itsSeconds = secs;
    NotifyObservers();
  }

  public int GetHours()
  {
    return itsHours;
  }

  public int GetMinutes()
  {
    return itsMinutes;
  }

  public int GetSeconds()
  {
    return itsSeconds;
  }
}
```

Листинг 32.27. MockTimeSink.cs

```
public class MockTimeSink : Observer
{
  private int itsHours;
  private int itsMinutes;
  private int itsSeconds;
  private TimeSource itsSource;

  public MockTimeSink(TimeSource source)
  {
    itsSource = source;
  }

  public int GetHours()
  {
    return itsHours;
  }

  public int GetMinutes()
  {
    return itsMinutes;
  }

  public int GetSeconds()
  {
    return itsSeconds;
  }

  public void Update()
  {
    itsHours = itsSource.GetHours();
```

```
      itsMinutes = itsSource.GetMinutes();
      itsSeconds = itsSource.GetSeconds();
    }
  }
```

Паттерн Наблюдатель

Отлично, после того как мы проработали пример и добрались в процессе эволюции кода до паттерна Наблюдатель, будет интересно узнать, что же он собой представляет. Каноническая форма этого паттерна показана на рис. 32.12. В нашем примере за объектом `Clock` наблюдал `DigitalClock`, который зарегистрировал себя с помощью интерфейса `Subject`, реализованного в классе `Clock`. При каждом изменении времени `Clock` вызывает метод `Notify` интерфейса `Subject`. Метод `Notify` вызывает метод `Update` каждого зарегистрированного объекта `Observer`. Следовательно, `DigitalClock` будет получать сообщения `Update` при каждой смене времени, после чего запросит текущее время у объекта `Clock` и отобразит его.

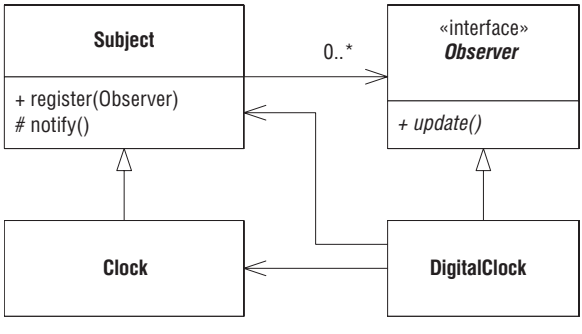


Рис. 32.12. *Канонический Наблюдатель с вытягиванием*

Наблюдатель – один из тех паттернов, которым вы будете находить применение повсюду, однажды усвоив его механику. Косвенность – великая вещь. Вы можете регистрировать наблюдателей для объектов, а не писать их так, чтобы они явно вызывали вас. Но хотя косвенность – полезный способ управления зависимостями, ее легко довести до абсурда. Злоупотребление паттерном Наблюдатель может сделать систему трудной для понимания и трассировки.

Модели

У паттерна Наблюдатель есть две основные модели. На рис. 32.12 показана *модель вытягивания*. Она получила такое название потому, что объект `DigitalClock` должен запрашивать (вытягивать) информацию о времени у объекта `Clock`, получив сообщение `Update`.

Достоинство модели вытягивания в простоте ее реализации и в том, что классы `Subject` и `Observer` могут быть стандартными повторно исполь-

зуемыми элементами и находиться в библиотеке. Но представьте себе, что вы наблюдаете за записью о работнике с тысячью полей и получили сообщение `Update`. Какое из тысячи полей изменилось?

Когда мы вызываем метод `Update` объекта `ClockObserver`, ответ на этот вопрос очевиден. `ClockObserver` должен запросить у `Clock` время и отобразить его. Но при обращении к методу `Update` объекта `EmployeeObserver` все уже не так просто. Мы не знаем, что произошло. И что делать, тоже не знаем. Может, изменилась фамилия работника, а может, зарплата. Может, у него сменился начальник. Или номер счета в банке. В общем, нужна помощь.

Такую помощь может оказать модель проталкивания для паттерна Наблюдатель. Ее структура показана на рис. 32.13. Отметим, что методы `Notify` и `Update` теперь принимают аргумент. Это подсказка, которую объект `Employee` передает объекту `SalaryObserver`. Она сообщает `SalaryObserver`, что именно изменилось в объекте `Employee`.

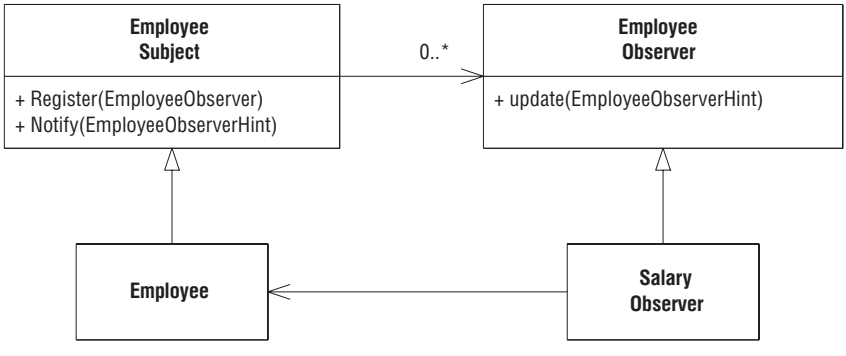


Рис. 32.13. *Модель проталкивания для паттерна Наблюдатель*

Тип `EmployeeObserverHint` аргумента методов `Notify` и `Update` может быть перечислением, строкой или более сложной структурой данных, содержащей старое и новое значения некоторого поля. Но в любом случае это значение передается наблюдателю.

Выбор той или иной модели зависит от сложности наблюдаемого объекта. Если наблюдаемый объект сложен и наблюдателю нужна подсказка, то больше подходит модель проталкивания. Если же объект прост, то достаточно и модели вытягивания.

Связь с принципами ООП

Паттерн Наблюдатель воплощает в себе принцип открытости/закрытости (OCP) во всей его полноте. Сама идея паттерна состоит в том, чтобы дать возможность добавлять новые наблюдающие объекты, не изменяя наблюдаемый объект. Поэтому наблюдаемый объект остается закрытым.

Из рис. 32.12 должно быть очевидно, что вместо `Subject` можно подставить `Clock`, а вместо `Observer` — `DigitalClock`. Значит, применим принцип подстановки Лисков (LSP).

`Observer` — абстрактный класс, от которого зависит конкретный класс `DigitialClock`, а также конкретные методы `Subject`. Следовательно, налицо принцип инверсии зависимости (DIP). Может показаться, что, раз в классе `Subject` нет абстрактных методов, значит, зависимость между `Clock` и `Subject` нарушает принцип DIP. Однако класс `Subject` никогда не должен инстанцироваться. Он имеет смысл только тогда, когда ему наследуют. Поэтому логически класс `Subject` абстрактный, пусть даже в нем нет ни одного абстрактного метода. Можно сделать его абстрактным принудительно, снабдив чисто виртуальным деструктором (в C++) или сделав все конструкторы защищенными.

На рис. 32.11 прослеживается также принцип разделения интерфейсов (ISP) . Классы `Subject` и `TimeSource` разделяют клиенты `MockTimeSource`, предоставляя специализированные интерфейсы для каждой категории клиентов.

Заключение

Итак, мы закончили. Начав с некоторой задачи проектирования и пройдя длинный путь эволюции, мы довольно близко подобрались к паттерну Наблюдатель. Вы можете упрекнуть меня в хитрости: ведь я с самого начала знал, что конечной точкой будет Наблюдатель, и в любом случае пришел бы к нему. Не стану этого отрицать. Но важно не это.

Если вы знакомы с паттернами проектирования, то мысль о нужном паттерне, скорее всего, сразу придет вам в голову, как только вы увидите задачу. Вопрос, следовательно, в том, следует ли реализовывать паттерн сразу или лучше подойти к нему мелкими шажками. В этой главе вы увидели, как выглядит второй вариант. Вместо того чтобы сразу заключить, что паттерн Наблюдатель подходит для этой задачи лучше всего, я медленно видоизменял код в этом направлении.

В любой промежуточной точке я мог бы счесть, что задача уже решена, и остановиться. Или обнаружить, что лучше выбрать другой курс, и сменить направление.

Некоторые диаграммы в этой главе нарисованы для вашего удобства. Я полагал, что следить за моей мыслью проще, когда рассматриваемые решения иллюстрируются на диаграммах. Если бы я не ставил себе целью подробно объяснять, что делаю, то опустил бы эти диаграммы. Однако *кое-какие* диаграммы я нарисовал для *себя*. Несколько раз возникали моменты, когда мне было просто необходимо взглянуть на уже созданную структуру и понять, куда двигаться дальше.

Если бы я не писал книгу, то набросал бы диаграммы на клочке бумаги или на доске. Тратить время на открытие какой-нибудь программы

рисования я не стал бы. Не могу вообразить ситуацию, когда самый совершенный инструмент оказался бы быстрее рисования на салфетке.

Воспользовавшись диаграммами, чтобы понять, как писать код дальше, я бы их выбросил. Ведь все те диаграммы, что я рисовал для себя, описывали промежуточные шаги.

Есть ли смысл сохранять диаграммы такого уровня детализации? Если вы пытаетесь проиллюстрировать ход своих мыслей, как я в этой книге, то они, конечно, полезны. Но обычно мы не документируем пути, по которым шли на протяжении нескольких часов кодирования. Как правило, это временные диаграммы, которые лучше выкинуть. На *таком* уровне детализации код вполне может документировать сам себя. На более высоких уровнях это верно уже не всегда.

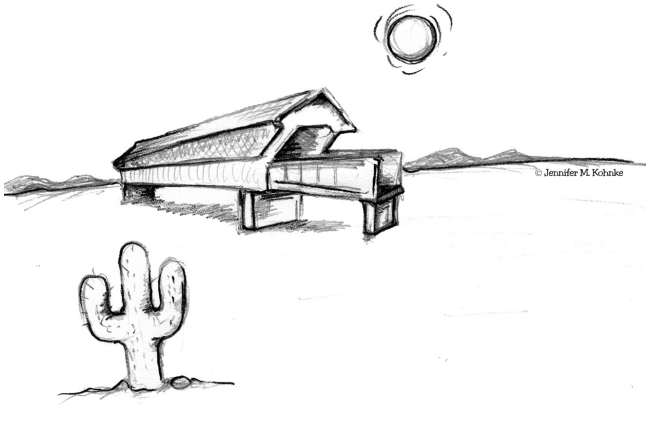
Библиография

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

[PLOPD3] Robert C. Martin, Dirk Riehle, Frank Buschmann, eds. «Pattern Languages of Program Design 3», Addison-Wesley, 1998.

33

Абстрактный сервер, адаптер и мост



Политики везде одинаковы: они обещают построить мост там, где и реки-то нет.

Никита Хрущев

В середине 1990-х годов я принимал активное участие в дискуссии, состоявшейся на конференции comp.object. Мы яростно спорили о различных стратегиях анализа и проектирования. В какой-то момент мы решили, что оценить позиции друг друга будет проще всего на конкретном примере. Поэтому мы выбрали очень простую задачу и продемонстрировали на ней свои любимые подходы.

Задача была элементарной – спроектировать программное обеспечение, управляющее работой простейшей настольной лампы. У лампы есть выключатель и источник света. Выключатель можно опросить, включен он или выключен, а источнику света приказать зажечься или погаснуть. Симпатичная, простенькая задачка.

Неистовые споры не утихали много месяцев. Кто-то ратовал за простой подход, когда есть всего два объекта: выключатель и источник света. Кто-то полагал, что должен быть объект «лампа», содержащий выключатель и источник света. Были люди, считавшие, что следует ввести в рассмотрение объект «электричество». А один человек даже предложил объект «электрический шнур».

Несмотря на абсурдность большинства аргументов, проектную модель исследовать интересно. Взгляните на рис. 33.1. Нет сомнений, что такой дизайн сможет работать. Объект `Switch` может опрашивать состояние реального выключателя и в зависимости от результата посылать сообщение `turnOn` или `turnOff` объекту `Light`.



Рис. 33.1. *Простая настольная лампа*

Что нам не нравится в этом дизайне? В нем нарушено два принципа проектирования: принцип инверсии зависимости (DIP) и принцип открытости/закрытости (OCP). Нарушение DIP бросается в глаза: зависимость, направленная от `Switch` к `Light`, – это зависимость от конкретного класса. Принцип DIP рекомендует отдавать предпочтение зависимости от абстрактных классов. Нарушение OCP не так очевидно, но более существенно. Причина нашего недовольства этим дизайном в том, что он заставляет тянуть за собой `Light` всюду, где нам понадобится `Switch`. Класс `Switch` невозможно расширить так, чтобы он управлял еще какими-то объектами, кроме `Light`.

Абстрактный сервер

Возможно, вам подумалось, что подкласс `Switch`, который будет управлять чем-то, помимо источников света, можно создать, как показано на рис. 33.2. Но это не решает проблему, потому что класс `FanSwitch` попрежнему наследует зависимость от `Light`. Всюду, где применяется `FanSwitch`, придется тащить и `Light`. Так что данное отношение наследования тоже нарушает принцип DIP.

Чтобы все же справиться с проблемой, мы воспользуемся одним из самых простых паттернов проектирования – Абстрактным сервером (Abstract Server) (см. рис. 33.3). Создав интерфейс между `Switch` и `Light`, мы позволяем классу `Switch` управлять всем, что реализует этот интерфейс. Тем самым оба принципа, DIP и OCP, оказываются удовлетворенными.

Кстати, интересное попутное замечание: обратите внимание, что интерфейс назван по имени клиента. Он называется `Switchable`, а не `Light`. Мы уже затрагивали эту тему раньше, но не грех и повторить. Интерфейсы принадлежат клиентам, а не производным классам. Логическая

связь между клиентом и интерфейсом сильнее, чем между интерфейсом и производными от него классами. Эта логическая связь настолько сильна, что не имеет смысла развертывать `Switch` без `Switchable`, тогда как развертывать `Switchable` без `Light` вполне допустимо. Сила логических связей конфликтует с силой связей физических. Наследование – гораздо более сильная физическая связь, чем ассоциация.

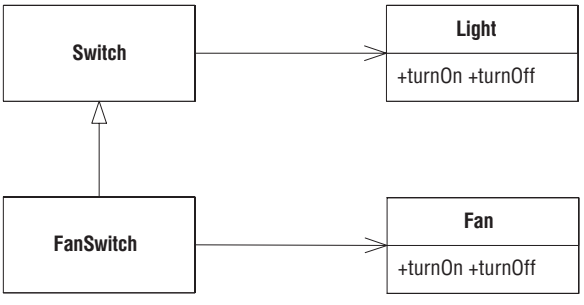


Рис. 33.2. *Плохой способ расширения класса Switch*

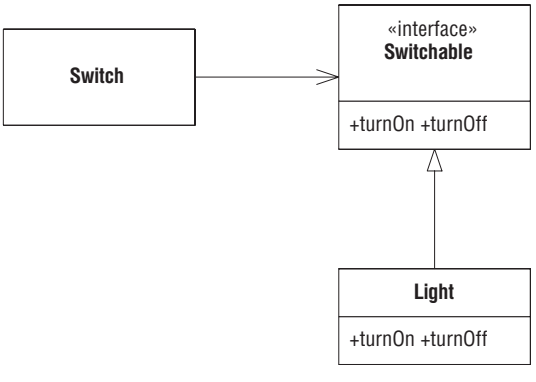


Рис. 33.3. *Решение задачи о лампе с помощью Абстрактного сервера*

В начале 1990-х годов мы думали, что миром правят физические связи. В авторитетных книгах рекомендовалось помещать иерархии наследования в один и тот же физический пакет. И это казалось разумным, потому что наследование – сильная физическая связь. Но за последние десять лет мы поняли, что физическая сила наследования обманчива и что иерархии наследования обычно не должны находиться в одном пакете. В один пакет нужно помещать клиенты вместе с теми интерфейсами, которыми они управляют.

Нестыковка между силой логических и физических связей – это артефакт статически типизированных языков, к которым относится и C#. В динамически типизированных языках – Smalltalk, Python или Ruby – этой нестыковки нет, потому что для достижения полиморфного поведения они не пользуются наследованием.

Адаптер

У дизайна на рис. 33.3 также имеется проблема – потенциальное нарушение принципа единственной обязанности (SRP). Мы связали вместе две сущности, `Light` и `Switchable`, которые могут изменяться в силу разных причин. Что если мы не сможем добавить отношение наследования в `Light`? Что если мы купили класс `Light` у сторонней фирмы без исходного кода? Что если `Switch` должен будет управлять классом, не являющимся производным от `Switchable`? На помощь приходит паттерн Адаптер (Adapter).[1]

На рис. 33.4 показано, как можно решить задачу с помощью паттерна Адаптер. Адаптер наследует интерфейсу `Switchable` и делегирует работу классу `Light`. В этом решении нет изъянов. Теперь `Switch` готов управлять любым объектом, который можно выключать и выключать. Нужно лишь создать подходящий адаптер. На самом деле у объекта может даже не быть методов `turnOn` и `turnOff`, которые объявлены в `Switchable`. Адаптер можно *адаптировать* к интерфейсу объекта.

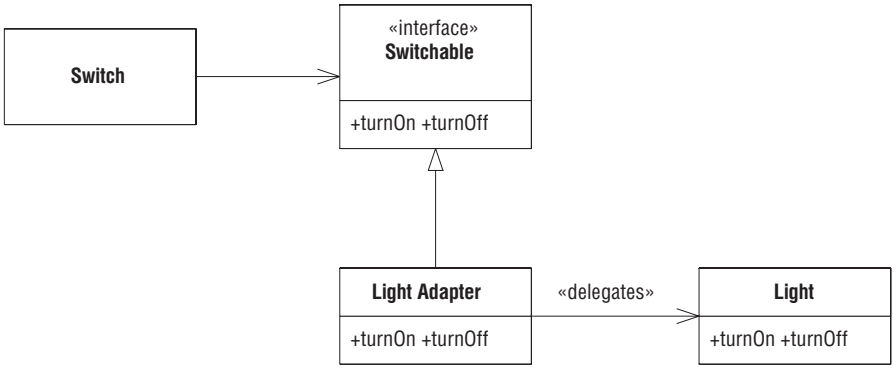


Рис. 33.4. *Решение задачи о лампе с помощью Адаптера*

Адаптеры обходятся не слишком дешево. Нужно написать новый класс, создать экземпляр адаптера и связать его с адаптируемым объектом. Затем при каждом обращении к адаптеру вы будете расплачиваться временем и памятью, необходимыми для реализации делегирования. Поэтому не стоит использовать адаптеры повсеместно. В большинстве случаев решение на основе Абстрактного сервера вполне приемлемо. Более того, даже самое первое из приведенных решений этой задачи (рис. 33.1) годится, если только вы заранее не *уверены*, что будут и другие объекты, которыми предстоит управлять с помощью `Switch`.

[1] Мы уже встречались с Адаптером выше, на рис. 10.2 и 10.3.

Адаптер класса

Класс `LightAdapter`, изображенный на рис. 33.4, принято называть *адаптером объекта*. Другой подход, называемый *адаптером класса*, представлен на рис. 33.5. В такой форме объект адаптера наследует интерфейсу `Switchable` и классу `Light`. Данный вариант немного эффективнее объектной формы и чуть проще в применении, но это достигается ценой использования сильной связи, какую представляет собой наследование.

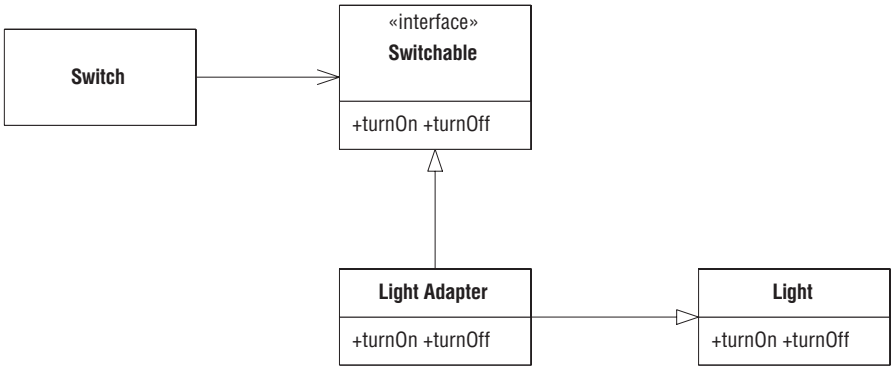


Рис. 33.5. *Решение задачи о лампе с помощью Адаптера*

Задача о модеме, адаптеры и принцип LSP

Рассмотрим ситуацию, изображенную на рис. 33.6. Имеется несколько модемных клиентов, и все они используют интерфейс `Modem`. Этот интерфейс реализован классами `HayesModem`, `USRoboticsModem` и `ErniesModem`. Довольно типичная ситуация. Она хорошо согласуется с принципами OCP, LSP и DIP. Появление новых модемов никак не отражается на клиентах. Допустим, что такое положение вещей сохранялось несколько лет. Написаны сотни клиентов, вполне довольных интерфейсом `Modem`.

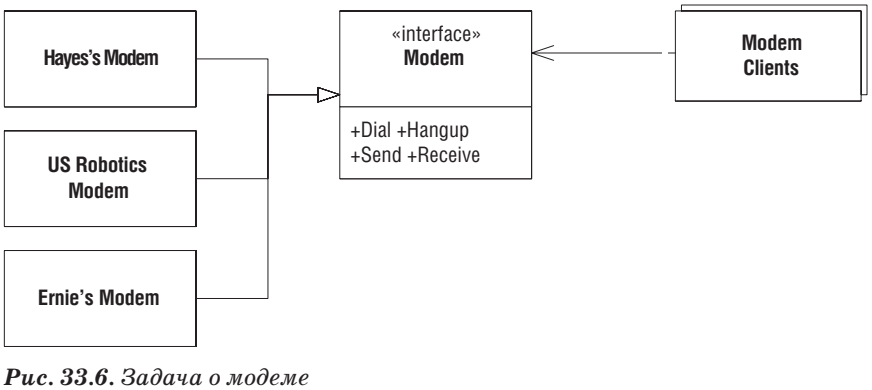


Рис. 33.6. *Задача о модеме*

А теперь предположим, что заказчик выставил новое требование. Некоторые модемы, называемые выделенными[1], не поддерживают набор номера, а устанавливаются на обоих концах выделенного соединения. Несколько новых приложений уже используют такие модемы и никаких номеров не набирают. Будем называть их `DedUsers`. Однако заказчик хочет, чтобы все существующие модемные клиенты получили возможность работать с выделенными модемами. И говорит, что не желает модифицировать сотни уже написанных приложений. Им нужно просто сказать, чтобы они набирали фиктивные телефонные номера.

Если бы у нас был выбор, то мы предпочли бы изменить дизайн системы, как показано на рис. 33.7. Мы воспользовались бы принципом ISP, чтобы разнести функции набора номера и связи по двум интерфейсам. Старые модемы реализовывали бы оба интерфейса, и модемные клиенты эти два интерфейса использовали бы. Клиенты `DedUsers` пользовались бы только интерфейсом `Modem`, а класс `DedicatedModem` только его и реализовывал бы. К сожалению, при этом пришлось бы вносить изменения во все модемные клиенты, а это как раз то, что заказчик категорически запретил.

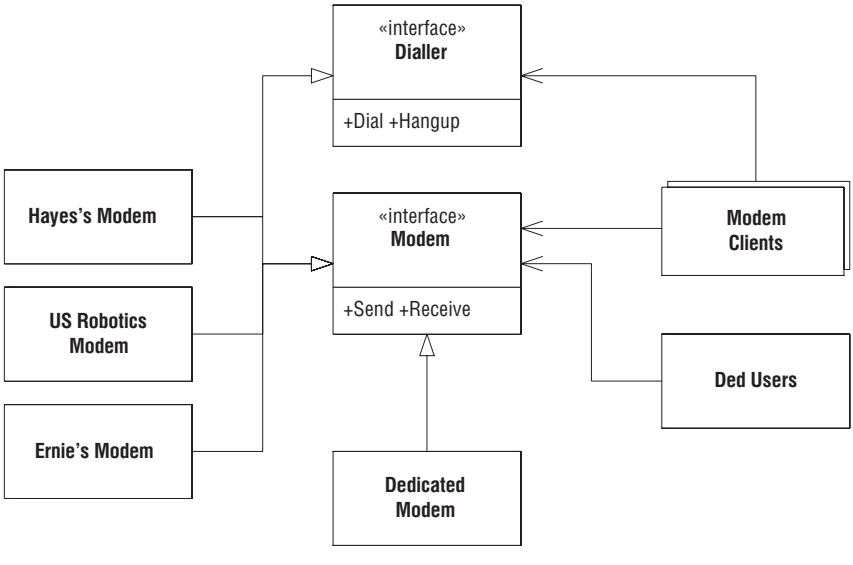


Рис. 33.7. *Идеальное решение задачи о модеме*

[1] Когда-то все модемы были выделенными; лишь в близкие к нам геологические эпохи модемы получили возможность набирать номер. А в раннем юрском периоде вы арендовали у телефонной компании модем размером с хлебницу и соединяли его с другим модемом выделенной линией, тоже арендуемой у телефонной компании. (Хорошо жили телефонные компании в юрском периоде.) А если возникало желание набирать номера, то надо было арендовать еще одно устройство примерно такого же размера, которое называлось автоматическим номеронабирателем.

И что нам делать? Разделить интерфейсы мы не можем, и тем не менее нужно как-то ухитриться сделать так, чтобы все модемные клиенты могли использовать класс DedicatedModem. Одно из решений – унаследовать DedicatedModem от Modem и реализовать методы Dial и Hangup так, чтобы они ничего не делали:

```
class DedicatedModem : Modem
{
  public virtual void Dial(char phoneNumber[10]) {}
  public virtual void Hangup() {}
  public virtual void Send(char c)
  {...}
  public virtual char Receive()
  {...}
}
```

Но вырожденные методы – признак нарушения принципа LSP. Пользователи базового класса могут ожидать, что Dial и Hangup существенно изменяют состояние модема. Вырожденные реализации в классе DedicatedModem не оправдывают этих ожиданий.

Допустим, что модемные клиенты были написаны в предположении, что модем находится в состоянии ожидания, пока не вызван метод Dial, и возвращаются в это состояние после вызова Hangup. Иными словами, они не ожидают поступления символов от модема, который еще не набрал номер. Класс DedicatedModem нарушает это предположение. Он начинает возвращать символы еще до того, как был вызван метод Dial, и продолжает это делать после вызова Hangup. Таким образом, DedicatedModem может привести к сбоям в работе некоторых клиентов.

Можно считать это ошибкой самих модемных клиентов. Они написаны плохо, раз происходит сбой в случае неожиданных входных данных. Готов с этим согласиться. Но нелегко будет убедить ребят, сопровождающих клиенты, внести изменения в их программы только потому, что мы добавляем новый режим. Это не только нарушает принцип OCP, но вообще не лезет ни в какие ворота. К тому же заказчик явно запретил изменять существующие клиенты.

Клудж. Мы можем имитировать состояние соединения в методах Dial и Hangup класса DedicatedModem. Можно отказаться возвращать символы если метод Dial еще не вызывался или если уже был вызван метод Hangup. При реализации этой идеи все модемные клиенты будут счастливы и изменять их не потребуется. *Нужно лишь убедить объекты* DedUsers *вызывать методы* Dial *и* Hangup (рис. 33.8).

Легко представить, как будут недовольны программисты, работающие над DedUsers. Они-то пользуются исключительно классом DedicatedModem. *И с какой стати им вызывать* Dial *и* Hangup*?* Однако они еще не написали свою программу, так что склонить их к нашему решению будет проще.

Запутанная сеть зависимостей. Много месяцев спустя, когда уже написаны сотни клиентов `DedUsers`, заказчик приходит с новым требованием. Похоже, что за все эти годы наши программы не сталкивались с набором международных номеров. Поэтому никто не обращал внимания на объявление `char[10]` в `dial`. Но теперь заказчик возжелал набирать номера произвольной длины. Ему требуется звонить в другие страны, по кредитной карточке, с идентификацией по ПИН-коду и т. д.

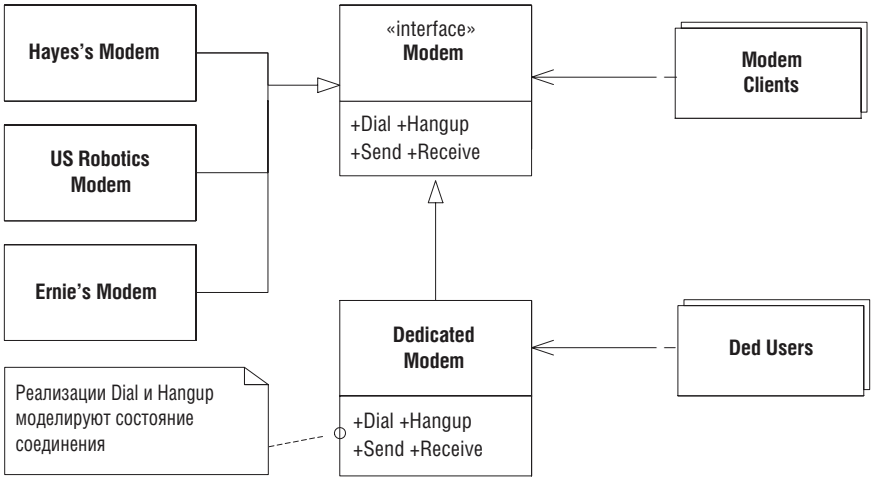


***Рис. 33.8.** Клудж – имитация состояния соединения в классе DedicatedModem*

Очевидно, что все модемные клиенты придется изменить. Они были написаны в предположении, что в номере телефона не больше десяти знаков. Заказчик дает добро на это изменение, поскольку выбора все равно нет, и орды программистов принимаются за дело. Столь же очевидно, что классы в иерархии `Modem` необходимо изменить с учетом новой длины номера телефона. Наша небольшая команда сумеет с этим справиться. *К сожалению, мы теперь вынуждены пойти к авторам* `DedUsers` *и поставить их в известность о том, что они снова должны изменить свой код!* Представьте себе их радость. Они не вызывали `Dial`, потому что это не требовалось. Они стали вызывать `Dial`, потому что мы так сказали. А теперь им предстоит тяжелая работа по сопровождению, потому что когда-то они сделали то, что мы велели.

В такую запутанную сеть зависимостей часто попадают многие проекты. Клудж в одной части системы создает опасную цепочку зависимостей, которая в конечном итоге вызывает проблемы в, казалось бы, совершенно не связанных местах.

Адаптер спешит на помощь. Этого фиаско можно было бы избежать, воспользовавшись для решения исходной проблемы паттерном Адаптер, как показано на рис. 33.9. В данном случае `DedicatedModem` не наследует `Modem`. Модемные клиенты используют `DedicatedModem` косвенно,

через объект DedicatedModemAdapter. Этот адаптер реализует методы Dial и Hangup так, что они имитируют состояние соединения. Вызовы же методов Send и Receieve адаптер делегирует объекту DedicatedModem.

Обратите внимание, что при таком подходе исчезли все трудности, с которыми мы сталкивались ранее. Модемные клиенты видят то поведение соединения, которого ожидают, а объектам DedUsers не приходится возиться с Dial и Hangup. Изменение требований к длине номера телефона не затрагивает DedUsers. Следовательно, вставив адаптер, мы устранили нарушения принципов LSP и OCP.

Впрочем, клудж никуда не делся. Адаптер все-таки имитирует состояние соединения. Если вы скажете, что это некрасиво, то я с вами, конечно, соглашусь. Однако же отметим, что все зависимости направлены *от* адаптера. Клудж изолирован, упрятан внутри адаптера, о котором почти никто не знает. Зависеть от этого адаптера будет разве что реализация какой-нибудь фабрики.[1]

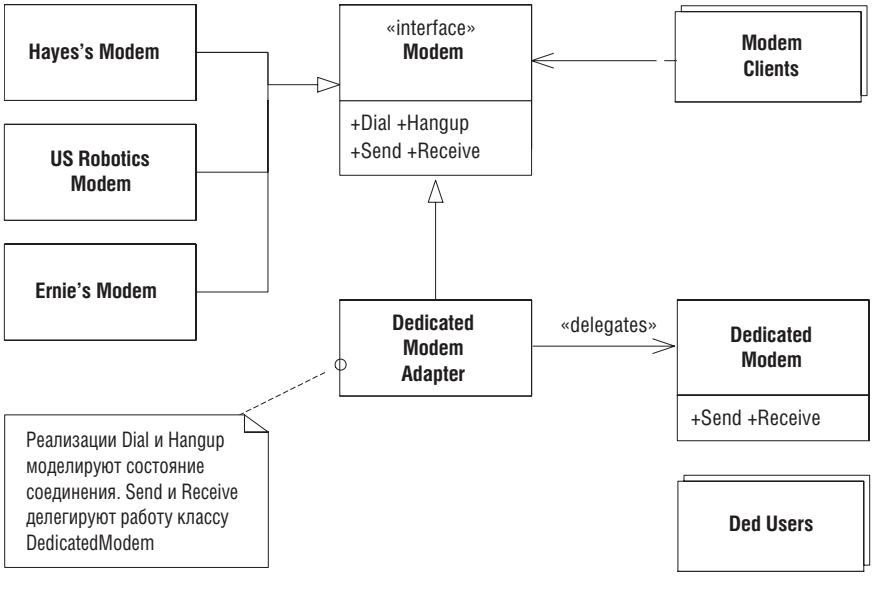


Рис. 33.9. *Решение задачи о модеме с помощью Адаптера*

Мост

На эту задачу можно взглянуть и по-другому. Необходимость поддержать выделенные модемы добавила новую степень свободы в иерархию типа Modem. Изначально тип Modem задумывался как простой интерфейс к различным аппаратным устройствам. Потому-то HayesModem, USRModem

[1] См. главу 29.

и ErniesModem и были подклассами базового класса Modem. Но, как оказалось, есть и другой способ организовать иерархию типа Modem. Можно ввести производные от Modem классы DialModem и DedicatedModem.

Объединить эти две независимых иерархии можно так, как показано на рис. 33.10. Каждый листовый узел дерева наделяет соответствующее устройство поведением модема с набором номера или выделенного. Объект DedicatedHayesModem управляет модемом Hayes в выделенном контексте.

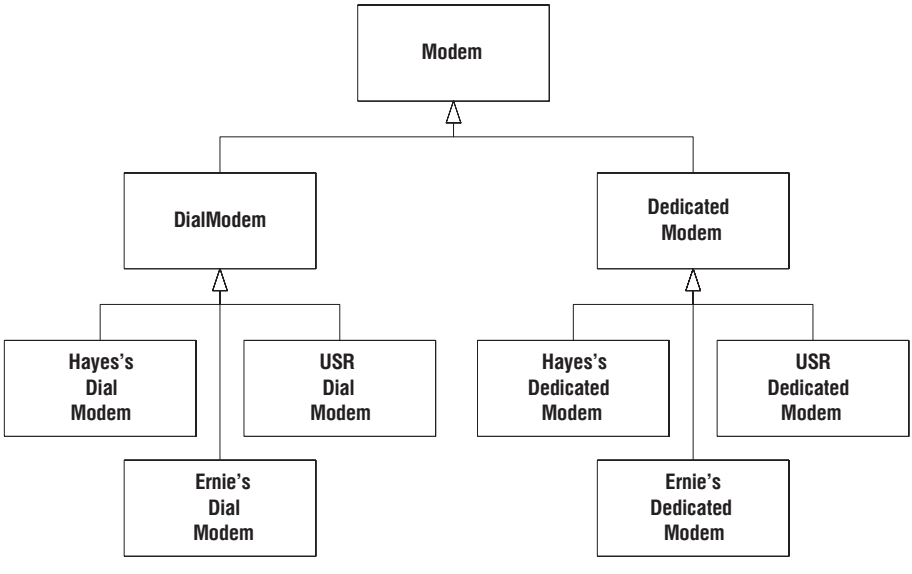


Рис. 33.10. *Решение задачи о модеме путем объединения иерархий типов*

Эта структура не идеальна. Всякий раз при добавлении нового оборудования мы должны создавать *два* новых класса: один для выделенного случая, второй – для набора номера. А если появится еще один вид соединения, то придется вводить *три* новых класса. Если изменения в том или ином направлении будут происходить довольно часто, то у нас появится куча производных классов.

Эту проблему позволяет разрешить паттерн Мост (Bridge). Он часто оказывается полезен в случаях когда степеней свободы больше одной. Вместо того чтобы объединять иерархии, мы можем разделить их и связать мостом.

Структура паттерна показана на рис. 33.11. Иерархия типа Modem разбита на две части: одна соответствует способу соединения, другая – оборудованию.

Пользователи обычных модемов продолжают работать с интерфейсом Modem. Класс ModemConnectionController реализует интерфейс Modem. А под-

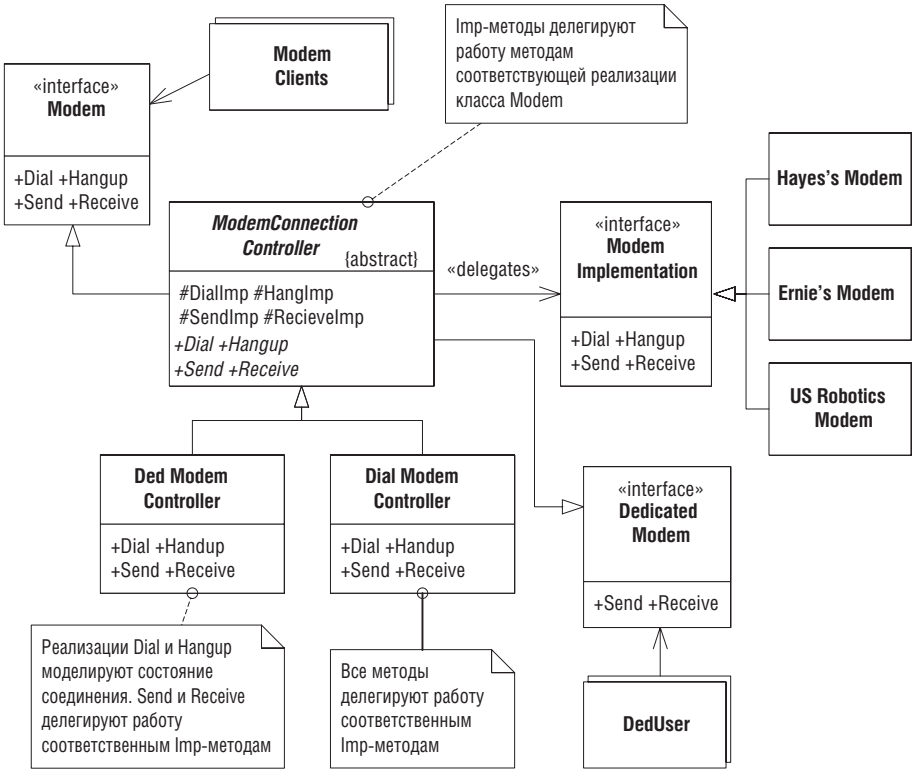


Рис. 33.11. *Решение задачи о модеме с помощью паттерна Мост*

классы `ModemConnectionController` отвечают за механизм соединения. `DialModemController` просто переадресует вызовы методов `Dial` и `Hangup` методам `DialImp` и `HangImp` из базового класса `ModemConnectionController`. А те, в свою очередь, делегируют работу классу `ModemImplementation`, который обращается к конкретному аппаратному устройству. Класс `DedModemController` реализует методы `Dial` и `Hangup`, имитируя состояние соединения. Вызовы `Send` и `Receive` переадресуются на `SendImp` и `ReceiveImp`, которые делегируют их иерархии `ModemImplementation`, как и выше.

Отметим, что четыре метода с суффиксом `Imp` в базовом классе `ModemConnectionController` – защищенные, они предназначены для вызова из подклассов `ModemConnectionController`, и только.

Эта структура сложна, но интересна. Ее можно создать, не затрагивая клиентов класса `Modem`, и при этом полностью отделить политику установления соединения от аппаратной реализации. Каждый класс, производный от `ModemConnectionController`, представляет одну политику соединения. Для ее реализации используются методы `SendImp`, `ReceiveImp`, `DialImp` и `HangImp`. Новые `Imp`-методы можно создавать, не мешая суще-

ствующим пользователям. Для добавления новых интерфейсов в классы управления соединением можно воспользоваться принципом ISP. Тем самым создается путь миграции для модемных клиентов, по которому они могут (необязательно быстро) перейти к API более высокого уровня, чем `Dial` и `Hangup`.

Заключение

Соблазнительно заявить следующее: вся проблема возникла из-за того, что исходный проект системы работы с модемами был никуда не годен. Проектировщики должны были понимать, что соединение и связь – разные концепции. Если бы они посвятили чуть больше времени анализу, то нашли и исправили бы свою ошибку. Ох, как хочется списать все трудности на недостаточно тщательный анализ.

Вздор! Нет такого понятия, как *достаточный* анализ. Сколько бы времени вы ни потратили на поиск идеальной структуры программы, все равно рано или поздно заказчик потребует внести изменение, которое в эту структуру не вписывается.

Избежать этого невозможно. Идеальных структур не бывает. Можно лишь попытаться найти баланс между затратами и результатами. По мере изменения требований к системе любая структура будет модифицироваться. Вся штука в том, как управлять этими изменениями, сохраняя по возможности простоту и гибкость системы.

Паттерн Адаптер предлагает простое и прямое решение. Все зависимости направлены в нужную сторону, а реализация совсем несложна. Решение на основе паттерна Мост чуть сложнее. Я бы не рекомендовал идти этим путем, если нет явных свидетельств в пользу необходимости полностью разделить политики связи и установления соединения, причем ожидается появление новых политик соединения.

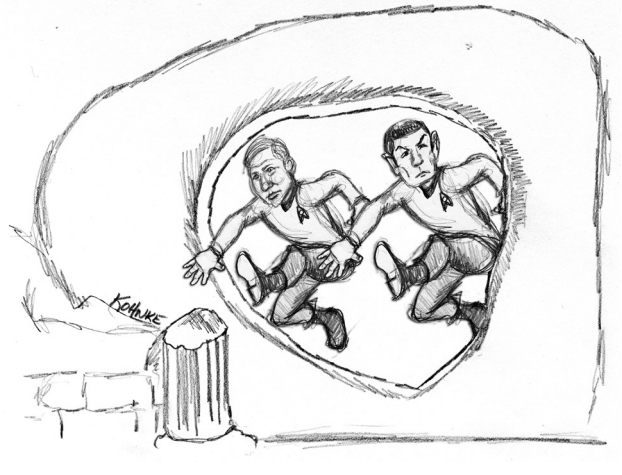
Урок здесь, как всегда, заключается в том, что у каждого паттерна своя цена. И выбирать нужно те паттерны, которые лучше всего соответствуют решаемой задаче.

Библиография

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

34

Заместитель и Шлюз: управление сторонними API



Мадам, я пытаюсь сконструировать мнемоническую схему с помощью каменного ножа и медвежьей шкуры.

Спок

В программных системах есть много барьеров. Перемещая данные из программы в базу данных, мы преодолеваем барьер базы данных. Посылая сообщения от одного компьютера другому – сетевой барьер.

Преодоление таких барьеров может оказаться непростым делом. Если проявить беспечность, то программа будет в основном сражаться с барьерами, а не решать задачу. Паттерн Заместитель (Proxy) помогает преодолевать барьеры, не забывая о цели, для которой пишется программа.

Заместитель

Допустим, что мы программируем электронный магазин на некотором сайте. В такой системе должны быть объекты, представляющие покупателя, заказ (корзину) и товары, включенные в заказ. На рис. 34.1 изображена возможная структура – упрощенная, но достаточная для наших целей.

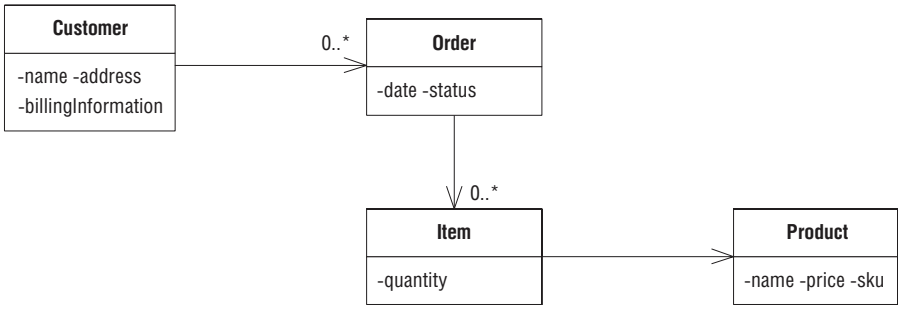


***Рис. 34.1.** Объектная модель простого электронного магазина*

Обдумывая задачу включения новой позиции в заказ, мы могли бы написать примерно такой код, как в листинге 34.1. Метод `AddItem` класса `Order` создает новый объект `Item`, содержащий ссылку на `Product` и количество, а потом добавляет его во внутренний список `ArrayList` объектов `Item`.

Листинг 34.1. Добавление нового товара

```
public class Order
{
  private ArrayList items = new ArrayList();

  public void AddItem(Product p, int qty)
  {
    Item item = new Item(p, qty);
    items.Add(item);
  }
}
```

А теперь допустим, что эти объекты представляют данные, хранящиеся в реляционной базе. На рис. 34.2 показаны соответствующие таблицы вместе с ключами. Чтобы найти все заказы данного покупателя, мы ищем в таблице `Order` записи, в которых поле `cusid` содержит идентификатор покупателя. Для нахождения всех позиций в данном заказе мы производим в таблице `Item` поиск по значению поля `orderId`. Чтобы узнать, какие товары соответствуют позициям заказа, мы используем поле `sku` (Stock Keeping Unit – артикул товара).

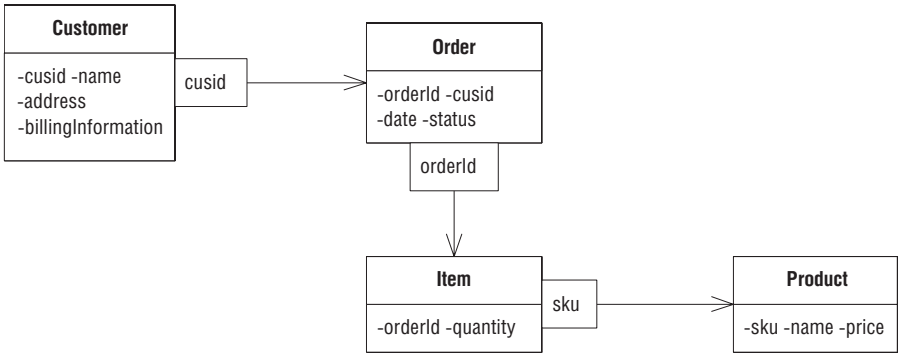


Рис. 34.2. *Реляционная модель электронного магазина*

Чтобы добавить строку, описывающую позицию заказа, нужно написать код, показанный в листинге 34.2. Здесь мы пользуемся библиотекой ADO.NET для прямого манипулирования данными в реляционной базе.

Листинг 34.2. Добавление позиции заказа в реляционную базу

```
public class AddItemTransaction : Transaction
{
  public void AddItem(int orderId, string sku, int qty)
  {
    string sql = "insert into items values(" +
      orderId + "," + sku + "," + qty + ")";
    SqlCommand command = new SqlCommand(sql, connection);
    command.ExecuteNonQuery();
  }
}
```

Показанные выше фрагменты очень различны, но решают одну и ту же логическую задачу – связывают позицию с заказом. В первом само существование базы данных игнорируется, во втором только оно и имеет значение.

Очевидно, что программа электронного магазина имеет дело с заказами, позициями и товарами. Но, увы, код в листинге 34.2 уводит нас в область SQL-команд, соединений с базой данных и составления запросов. Это существенное нарушение принципов SRP и, возможно, CCP. В листинге 34.2 смешаны две системы понятий с разными причинами изменения. Понятия позиций и заказов переплетены с понятиями реляционных схем и языка SQL. Если по какой-то причине произойдет изменение в одном аспекте, то придется изменять и другой. Код в листинге 34.2 нарушает также принцип DIP, поскольку стратегия программы зависит от деталей механизма хранения.

Эти болезни излечивает паттерн Заместитель. Чтобы его исследовать, подготовим тестовую программу, которая демонстрирует, как должны

себя вести операции создания заказа и вычисления итоговой стоимости. Содержательная часть программы приведена в листинге 34.3.

Простой код, при котором этот тест проходит, показан в листингах 34.4–34.6. В нем мы пользуемся объектной моделью, изображенной на рис. 34.1. Существование базы данных нигде не предполагается.

Листинг 34.3. Тестовая программа, которая создает заказ и проверяет правильность вычисления его стоимости

```
[Test]
public void TestOrderPrice()
{
  Order o = new Order("Bob");
  Product toothpaste = new Product("Toothpaste", 129);
  o.AddItem(toothpaste, 1);
  Assert.AreEqual(129, o.Total);
  Product mouthwash = new Product("Mouthwash", 342);
  o.AddItem(mouthwash, 2);
  Assert.AreEqual(813, o.Total);
}
```

Листинг 34.4. Order.cs

```
public class Order
{
  private ArrayList items = new ArrayList();

  public Order(string cusid)
  {}

  public void AddItem(Product p, int qty)
  {
    Item item = new Item(p,qty);
    items.Add(item);
  }

  public int Total
  {
    get
    {
      int total = 0;
      foreach(Item item in items)
      {
        Product p = item.Product;
        int qty = item.Quantity;
        total += p.Price * qty;
      }
      return total;
    }
  }
}
```

Листинг 34.5. Product.cs

```
public class Product
{
  private int price;

  public Product(string name, int price)
  {
    this.price = price;
  }

  public int Price
  {
    get { return price; }
  }
}
```

Листинг 34.6. Item.cs

```
public class Item
{
  private Product product;
  private int quantity;

  public Item(Product p, int qty)
  {
    product = p;
    quantity = qty;
  }

  public Product Product
  {
    get { return product; }
  }

  public int Quantity
  {
    get { return quantity; }
  }
}
```

На рис. 34.3 и 34.4 иллюстрируется работа паттерна Заместитель. Каждый подлежащий замещению объект разбивается на три части. Первая – интерфейс, в котором объявлены все методы, которые собирается вызывать клиент. Второй – реализация этих методов без каких-либо знаний о базе данных. Третий – собственно заместитель, который знает все о базе данных.

Рассмотрим класс `Product`. Мы заместили его, подменив интерфейсом. В интерфейсе объявлены все те методы, что имеются в классе `Product`. Класс `ProductImplementation` реализует этот интерфейс почти так же, как

и раньше. Класс `ProductDBProxy` реализует все методы `Product` таким образом, что они выбирают данные о товаре из базы, создают экземпляр `ProductImplementation` и делегируют ему дальнейшую работу.

Цепочка событий воспроизведена на диаграмме последовательности на рис. 34.4. Клиент посылает сообщение `Price` объекту, который ему представляется как `Product`, но на самом деле является `ProductDBProxy`. Объект `ProductDBProxy` извлекает `ProductImplementation` из базы данных и вызывает его свойство `Price`.

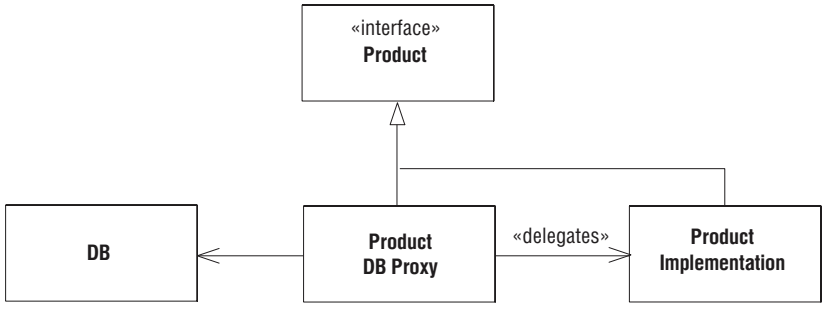


Рис. 34.3. *Статическая модель паттерна Заместитель*

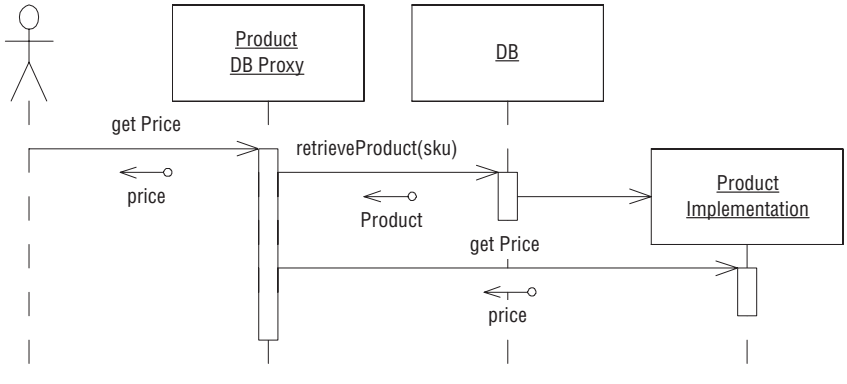


Рис. 34.4. *Динамическая модель паттерна Заместитель*

Ни клиент, ни объект `ProductImplementation` не знают, что происходит. База данных включена в приложение таким образом, что ни одна сторона о ней не подозревает. В этом и состоит прелесть паттерна Заместитель. Теоретически его можно поместить между любыми двумя взаимодействующими объектами, так что ни один не будет знать о его присутствии. Поэтому он пригоден для преодоления таких барьеров, как база данных или сеть, прозрачно для всех участников.

Но на практике применение заместителей нетривиально. Чтобы получить представление о возникающих проблемах, попробуем добавить этот паттерн в наш простой электронный магазин.

Реализация Заместителя

Проще всего написать заместитель для класса `Product`. Будем считать, что таблица товаров – это простой словарь. В каком-то одном месте в него будут загружены все товары. Никаких других манипуляций с этой таблицей не предвидится, поэтому заместитель будет сравнительно простым.

Для начала нам потребуется служебный класс для работы с базой данных, который будет отвечать за сохранение и выборку данных о товарах. Заместитель будет им пользоваться для операций с базой данных. В листинге 34.7 показан такой тестовый класс. А вместе с кодом в листингах 34.8 и 34.9 этот тест успешно проходит.

Листинг 34.7. DbTest.cs

```
[TestFixture]
public class DBTest
{
  [SetUp]
  public void SetUp()
  {
    DB.Init();
  }

  [TearDown]
  public void TearDown()
  {
    DB.Close();
  }

  [Test]
  public void StoreProduct()
  {
    ProductData storedProduct = new ProductData();
    storedProduct.name = "MyProduct";
    storedProduct.price = 1234;
    storedProduct.sku = "999";
    DB.Store(storedProduct);
    ProductData retrievedProduct =
      DB.GetProductData("999");
    DB.DeleteProductData("999");
    Assert.AreEqual(storedProduct, retrievedProduct);
  }
}
```

Листинг 34.8. ProductData.cs

```
public class ProductData
{
  private string name;
  private int price;
  private string sku;

  public ProductData(string name,
    int price, string sku)
  {
    this.name = name;
    this.price = price;
    this.sku = sku;
  }

  public ProductData() {}

  public override bool Equals(object o)
  {
    ProductData pd = (ProductData)o;
    return name.Equals(pd.name) &&
      sku.Equals(pd.sku) &&
      price==pd.price;
  }

  public override int GetHashCode()
  {
    return name.GetHashCode() ^
      sku.GetHashCode() ^
      price.GetHashCode();
  }
}
```

Листинг 34.9

```
public class Db
{
  private static SqlConnection connection;

  public static void Init()
  {
    string connectionString =
      "Initial Catalog=QuickyMart;" +
      "Data Source=marvin;" +
      "user id=sa;password=abc;";
    connection = new SqlConnection(connectionString);
    connection.Open();
  }

  public static void Store(ProductData pd)
  {
```

```
  SqlCommand command = BuildInsertionCommand(pd);
  command.ExecuteNonQuery();
}

private static SqlCommand
  BuildInsertionCommand(ProductData pd)
{
  string sql =
    "INSERT INTO Products VALUES (@sku, @name, @price)";
  SqlCommand command = new SqlCommand(sql, connection);
  command.Parameters.Add("@sku", pd.sku);
  command.Parameters.Add(«@name», pd.name);
  command.Parameters.Add(«@price», pd.price);
  return command;
}

public static ProductData GetProductData(string sku)
{
  SqlCommand command = BuildProductQueryCommand(sku);
  IDataReader reader = ExecuteQueryStatement(command);
  ProductData pd = ExtractProductDataFromReader(reader);
  reader.Close();
  return pd;
}

private static
  SqlCommand BuildProductQueryCommand(string sku)
{
  string sql = "SELECT * FROM Products WHERE sku = @sku";
  SqlCommand command = new SqlCommand(sql, connection);
  command.Parameters.Add("@sku", sku);
  return command;
}

private static ProductData
  ExtractProductDataFromReader(IDataReader reader)
{
  ProductData pd = new ProductData();
  pd.Sku = reader[«sku»].ToString();
  pd.Name = reader[«name»].ToString();
  pd.Price = Convert.ToInt32(reader[«price»]);
  return pd;
}

public static void DeleteProductData(string sku)
{
  BuildProductDeleteStatement(sku).ExecuteNonQuery();
}

private static SqlCommand
  BuildProductDeleteStatement(string sku)
```

```
    {
      string sql = «DELETE from Products WHERE sku = @sku»;
      SqlCommand command = new SqlCommand(sql, connection);
      command.Parameters.Add("@sku", sku);
      return command;
    }

    private static IDataReader
      ExecuteQueryStatement(SqlCommand command)
    {
      IDataReader reader = command.ExecuteReader();
      reader.Read();
      return reader;
    }

    public static void Close()
    {
      connection.Close();
    }
  }
```

Следующий шаг реализации заместителя – написать тест, показывающий, как он работает. В этом тесте мы добавим товар в базу данных, создадим объект `ProductProxy`, в котором поле `sku` равно артикулу сохраненного продукта, и попытаемся воспользоваться свойствами-акцессорами из класса `Product` для доступа к данным через заместителя. См. листинг 34.10.

Листинг 34.10. ProxyTest.cs

```
[TestFixture]
public class ProxyTest
{
  [SetUp]
  public void SetUp()
  {
    Db.Init();
    ProductData pd = new ProductData();
    pd.sku = "ProxyTest1";
    pd.name = "ProxyTestName1";
    pd.price = 456;
    Db.Store(pd);
  }

  [TearDown]
  public void TearDown()
  {
    Db.DeleteProductData("ProxyTest1");
    Db.Close();
  }
```

```
  [Test]
  public void ProductProxy()
  {
    Product p = new ProductProxy("ProxyTest1");
    Assert.AreEqual(456, p.Price);
    Assert.AreEqual("ProxyTestName1", p.Name);
    Assert.AreEqual("ProxyTest1", p.Sku);
  }
}
```

Чтобы все это заработало, необходимо отделить интерфейс класса `Product` **от его реализации. Поэтому я сделал** `Product` **интерфейсом, а для его реализации создал класс** `ProductImpl` **(листинги 34.11 и 34.12). Это вынудило меня изменить класс** `TestShoppingCart` **(не показан) так, чтобы вместо** `Product` **использовался** `ProductImpl`**.**

Листинг 34.11. Product.cs

```
public interface Product
{
  int Price {get;}
  string Name {get;}
  string Sku {get;}
}
```

Листинг 34.12. ProductImpl.cs

```
public class ProductImpl : Product
{
  private int price;
  private string name;
  private string sku;

  public ProductImpl(string sku, string name, int price)
  {
    this.price = price;
    this.name = name;
    this.sku = sku;
  }

  public int Price
  {
    get { return price; }
  }

  public string Name
  {
    get { return name; }
  }

  public string Sku
  {
```

```
    get { return sku; }
  }
}
```

Листинг 34.13

```
public class ProductProxy : Product
{
  private string sku;

  public ProductProxy(string sku)
  {
    this.sku = sku;
  }

  public int Price
  {
    get
    {
      ProductData pd = Db.GetProductData(sku);
      return pd.price;
    }
  }

  public string Name
  {
    get
    {
      ProductData pd = Db.GetProductData(sku);
      return pd.name;
    }
  }

  public string Sku
  {
    get { return sku; }
  }
}
```

Реализация заместителя тривиальна. На самом деле он не вполне соответствует канонической форме этого паттерна, показанной на рис. 34.3 и 34.4. Для меня, честно говоря, это стало сюрпризом. Я собирался реализовать паттерн Заместитель. Но когда это намерение материализовалось, оказалось, что канонический паттерн не имеет смысла.

Согласно канону объект `ProductProxy` должен был бы в каждом методе создавать объект `ProductImpl` и затем делегировать работу соответствующему методу или свойству этого объекта, как в примере ниже:

```
public int Price
{
  get
  {
```

```
      ProductData pd = Db.GetProductData(sku);
      ProductImpl p =
        new ProductImpl(pd.Name, pd.Sku, pd.Price);
      return pd.Price;
    }
  }
```

Но создание `ProductImpl` – пустая трата времени программиста и ресурсов компьютера. Объект `ProductProxy` уже имеет данные, которые вернули бы акцессоры `ProductImpl`. Поэтому ни создавать `ProductImpl`, ни делегировать ему мы не станем. Это еще один пример того, как реальный код может уклоняться от ожидаемых паттернов и моделей.

Отметим, что свойство `Sku` класса `ProductProxy` в листинге 34.13 – еще один шаг в том же направлении. Мы даже не даем себе труда обратиться к базе за получением значения `sku`. А зачем? Оно и так уже есть.

Возможно, у вас сложилось впечатление, что реализация `ProductProxy` крайне неэффективна. Мы обращаемся к базе из каждого акцессора. Не лучше ли было бы кэшировать объект `ProductData` в памяти?

Это изменение тривиально, но единственный побудительный мотив для него – наши опасения. На данном этапе разработки еще нет никаких свидетельств в пользу того, что программа окажется недостаточно производительной. Кроме того, мы знаем, что СУБД и сама что-то кэширует. Поэтому не вполне ясно, что даст поддержка собственного кэша. Лучше подождать, пока не появятся признаки замедления, и не искать бед на свою голову.

Следующий наш шаг – создать заместителя для класса `Order`. Каждый экземпляр `Order` может содержать несколько экземпляров `Item`. В реляционной схеме (рис. 34.2) это отношение отражено в таблице `Item`. Каждая строка этой таблицы содержит ключ соответствующей ей записи в таблице `Order`. Однако в объектной модели то же отношение реализовано списком `ArrayList` в классе `Order` (см. листинг 34.4). Каким-то образом заместитель должен преобразовать одну форму в другую.

Начнем с написания теста, который заместитель должен пройти. В нем мы добавляем в базу данных несколько фиктивных товаров, получаем для них заместителей и передаем их методу `AddItem` объекта `OrderProxy`. В самом конце тест запрашивает у `OrderProxy` полную стоимость заказа (листинг 34.14). Смысл теста в том, чтобы показать, что `OrderProxy` ведет себя в точности, как `Order`, хотя данные получает из базы, а не из объектов в памяти.

Листинг 34.14. ProxyTest.cs

```
[Test]
public void OrderProxyTotal()
{
  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);
    Assert.AreEqual(956, order.Total);
  }
```

Чтобы этот тест успешно прошел, нам необходимо реализовать несколько новых классов и методов. Сначала займемся методом `NewOrder` в классе `Db`. Похоже, он должен возвращать экземпляр некоего класса `OrderData`. Это полный аналог `ProductData`: простая структура данных, представляющая одну строку в таблице `Order`. Код метода приведен в листинге 34.15.

Листинг 34.15. OrderData.cs

```
public class OrderData
{
  public string customerId;
  public int orderId;

  public OrderData() {}

  public OrderData(int orderId, string customerId)
  {
    this.orderId = orderId;
    this.customerId = customerId;
  }
}
```

Пусть вас не смущает, что данные-члены открыты. Это не настоящий объект, а простой контейнер данных. Никакого интересного поведения, которое надо было бы инкапсулировать, в нем нет. Делать переменные экземпляра закрытыми, а потом писать для них свойства-акцессоры было бы пустой тратой времени. Можно было бы написать `struct` вместо `class`, но я хочу передавать `OrderData` по ссылке, а не по значению.

Теперь нужно написать метод `NewOrder` класса `Db`. Заметьте, что, вызывая его в листинге 34.14, мы передаем идентификатор покупателя-владельца заказа, но не указываем `orderId`. В каждой записи таблицы `Order` должен быть ключ, хранящийся в поле `orderId`. Более того, в каждой записи таблицы `Item` тоже имеется поле `orderId`, устанавливающее связь этой записи с записью в таблице `Order`. Ясно, что ключ `orderId` должен быть уникален. А как он создается? Взгляните на листинг 34.16.

Листинг 34.16. DbTest.cs

```
[Test]
public void OrderKeyGeneration()
{
```

```
    OrderData o1 = Db.NewOrder("Bob");
    OrderData o2 = Db.NewOrder("Bill");
    int firstOrderId = o1.orderId;
    int secondOrderId = o2.orderId;
    Assert.AreEqual(firstOrderId + 1, secondOrderId);
  }
```

Как видно из этого теста, мы ожидаем, что `orderId` каким-то образом автоматически будет увеличиваться на единицу при каждом создании нового заказа. Этого легко добиться, поручив `SqlServer` самостоятельно генерировать следующий `orderId`; чтобы получить это значение, нужно обратиться к функции базы данных `scope_identity()`. См. листинг 34.17.

Листинг 34.17

```
public static OrderData NewOrder(string customerId)
{
  string sql = "INSERT INTO Orders(cusId) VALUES(@cusId); " +
    "SELECT scope_identity()";
  SqlCommand command = new SqlCommand(sql, connection);
  command.Parameters.Add("@cusId", customerId);
  int newOrderId = Convert.ToInt32(command.ExecuteScalar());
  return new OrderData(newOrderId, customerId);
}
```

Вот теперь мы можем приступить к классу `OrderProxy`. Как и в случае `Product`, необходимо разделить `Order` на интерфейс и его реализацию. Следовательно, `Order` станет интерфейсом, а `OrderImpl` будет его реализовывать. См. листинги 34.18 и 34.19.

Листинг 34.18. Order.cs

```
public interface Order
{
  string CustomerId { get; }
  void AddItem(Product p, int quantity);
  int Total { get; }
}
```

Листинг 34.19. OrderImpl.cs

```
public class OrderImpl : Order
{
  private ArrayList items = new ArrayList();
  private string customerId;

  public OrderImp(string cusid)
  {
    customerId = cusid;
  }

  public string CustomerId
  {
```

```
    get { return customerId; }
  }

  public void AddItem(Product p, int qty)
  {
    Item item = new Item(p, qty);
    items.Add(item);
  }

  public int Total
  {
    get
    {
      int total = 0;
      foreach(Item item in items)
      {
        Product p = item.Product;
        int qty = item.Quantity;
        total += p.Price * qty;
      }
      return total;
    }
  }
}
```

Как реализовать в заместителе метод `AddItem`? Понятно, что заместитель не может делегировать работу методу `OrderImpl.AddItem`! Вместо этого он должен сам вставить строку в таблицу `Item`. С другой стороны, я *хочу*, чтобы свойство `OrderProxy.Total` делегировало свои функции свойству `OrderImpl.Total`, потому что бизнес-правило – политика вычисления итогов – должно быть инкапсулировано в `OrderImpl`. Сама идея создания заместителей заключается в том, чтобы отделить реализацию работы с базой данных от бизнес-правил.

Чтобы делегировать свойство `Total`, заместитель должен будет полностью построить объект `Order` со всеми содержащимися в нем позициями `Item`. Поэтому в свойстве `OrderProxy.Total` мы будем читать все позиции из базы данных и вызывать метод `AddItem` первоначально пустого объекта `OrderImpl` для каждой прочитанной позиции, а затем обратимся к свойству `Total` объекта `OrderImpl`. Таким образом, реализация `OrderProxy` выглядит примерно так, как показано в листинге 34.20.

В этом коде предполагается наличие класса `ItemData` и нескольких методов в классе `Db` для манипулирования таблицей `Item`. Они приведены в листингах 34.21–34.23.

Листинг 34.20

```
public class OrderProxy : Order
{
  private int orderId;
```

```
  public OrderProxy(int orderId)
  {
    this.orderId = orderId;
  }

  public int Total
  {
    get
    {
      OrderImp imp = new OrderImp(CustomerId);
      ItemData[] itemDataArray = Db.GetItemsForOrder(orderId);
      foreach(ItemData item in itemDataArray)
        imp.AddItem(new ProductProxy(item.sku), item.qty);
      return imp.Total;
    }
  }

  public string CustomerId
  {
    get
    {
      OrderData od = Db.GetOrderData(orderId);
      return od.customerId;
    }
  }

  public void AddItem(Product p, int quantity)
  {
    ItemData id =
      new ItemData(orderId, quantity, p.Sku);
    Db.Store(id);
  }

  public int OrderId
  {
    get { return orderId; }
  }
}
```

Листинг 34.21. ItemData.cs

```
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 override bool Equals(Object o)
  {
    if(o is ItemData)
    {
      ItemData id = o as ItemData;
      return orderId == id.orderId &&
        qty == id.qty &&
        sku.Equals(id.sku);
    }
    return false;
  }
}
```

Листинг 34.22

```
[Test]
public void StoreItem()
{
  ItemData storedItem = new ItemData(1, 3, "sku");
  Db.Store(storedItem);
  ItemData[] retrievedItems = Db.GetItemsForOrder(1);
  Assert.AreEqual(1, retrievedItems.Length);
  Assert.AreEqual(storedItem, retrievedItems[0]);
}

[Test]
public void NoItems()
{
  ItemData[] id = Db.GetItemsForOrder(42);
  Assert.AreEqual(0, id.Length);
}
```

Листинг 34.23

```
public static void Store(ItemData id)
{
  SqlCommand command = BuildItemInsersionStatement(id);
  command.ExecuteNonQuery();
}

private static SqlCommand
  BuildItemInsersionStatement(ItemData id)
{
  string sql = "INSERT INTO Items(orderId,quantity,sku) " +
    "VALUES (@orderID, @quantity, @sku)";
  SqlCommand command = new SqlCommand(sql, connection);
  command.Parameters.Add("@orderId", id.orderId);
  command.Parameters.Add("@quantity", id.qty);
  command.Parameters.Add("@sku", id.sku);
```

```
  return command;
}

public static ItemData[] GetItemsForOrder(int orderId)
{
  SqlCommand command =
    BuildItemsForOrderQueryStatement(orderId);
  IDataReader reader = command.ExecuteReader();
  ItemData[] id = ExtractItemDataFromResultSet(reader);
  reader.Close();
  return id;
}

private static SqlCommand
  BuildItemsForOrderQueryStatement(int orderId)
{
  string sql = «SELECT * FROM Items « +
    «WHERE orderid = @orderId»;
  SqlCommand command = new SqlCommand(sql, connection);
  command.Parameters.Add(«@orderId», orderId);
  return command;
}

private static ItemData[]
  ExtractItemDataFromResultSet(IDataReader reader)
{
  ArrayList items = new ArrayList();
  while (reader.Read())
  {
    int orderId = Convert.ToInt32(reader[«orderId»]);
    int quantity = Convert.ToInt32(reader[“quantity”]);
    string sku = reader[«sku»].ToString();
    ItemData id = new ItemData(orderId, quantity, sku);
    items.Add(id);
  }
  return (ItemData[]) items.ToArray(typeof (ItemData));
}

public static OrderData GetOrderData(int orderId)
{
  string sql = «SELECT cusid FROM orders « +
    «WHERE orderid = @orderId»;
  SqlCommand command = new SqlCommand(sql, connection);
  command.Parameters.Add(«@orderId», orderId);
  IDataReader reader = command.ExecuteReader();
  OrderData od = null;
  if (reader.Read())
    od = new OrderData(orderId, reader[«cusid»].ToString());
  reader.Close();
  return od;
}
```

```
public static void Clear()
{
  ExecuteSql(«DELETE FROM Items»);
  ExecuteSql(«DELETE FROM Orders»);
  ExecuteSql(«DELETE FROM Products»);
}

private static void ExecuteSql(string sql)
{
  SqlCommand command = new SqlCommand(sql, connection);
  command.ExecuteNonQuery();
}
```

Резюме

Этот пример должен был развеять всяческие вредные иллюзии относительно элегантности и простоты работы с заместителями. Использование заместителей – занятие нетривиальное. Простая модель делегирования, описанная в каноническом паттерне, на практике редко оказывается такой чистой. Иногда мы опускаем делегирование для тривиальных акцессоров. В случае методов, имеющих дело с отношениями 1:*N*, мы вынуждены *откладывать* делегирование и переносить его в другие методы, как показывает пример метода `AddItem` , делегирование которого было перенесено в `Total`. Наконец, перед нами маячит призрак кэширования.

В этом примере мы не стали заниматься кэшированием. На прогон всех тестов уходит меньше секунды, поэтому забивать себе голову вопросами производительности нет смысла. Но в реальном приложении проблема производительности и интеллектуального кэширования, скорее всего, встанет. Я не рекомендую без размышлений браться за реализацию какой-либо стратегии кэширования просто из опасения, что иначе программа будет работать медленно. Более того, мой опыт показывает, что преждевременное добавление кэширования – отличный способ *снизить* производительность. Так что если вы боитесь, что производительность может оказаться неадекватной, советую провести эксперименты, которые *доказали* бы, что проблема действительно имеет место. Вот тогда и *только тогда* нужно начинать думать о том, как ускорить работу программы.

Хотя работа с заместителями достаточно трудоемка, у них есть одно несомненное достоинство: *разделение обязанностей*. В нашем случае бизнес-правила и база данных полностью разделены. Класс `OrderImpl` никак не зависит от базы данных. Изменить схему базы или СУБД можно было бы, не затрагивая `Order`, `OrderImpl` и все остальные классы, относящиеся к предметной области.

Если отделение бизнес-правил от реализации базы данных критически важно, то паттерн Заместитель вполне подойдет. Вообще, его мож-

но применять для отделения бизнес-правил от *любых* аспектов реализации. Например, чтобы отделить их от таких технологий, как COM, CORBA, EJB и т. д. Это позволяет держать бизнес-правила (ценный актив организации) подальше от механизмов реализации, которые сейчас весьма популярны.

Базы данных, ПО промежуточного уровня и прочие сторонние интерфейсы

Сторонние API – это суровая реальность в жизни программистов. Мы покупаем СУБД, серверы промежуточного уровня, библиотеки классов, библиотеки для организации многопоточной работы и т. д. На первых порах мы обращаемся к этим API напрямую из своих программ (рис. 34.5).

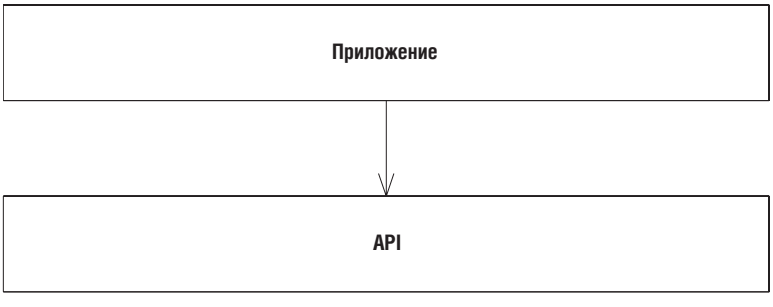


Рис. 34.5. *Первоначальная связь между приложением и сторонним API*

Но со временем мы замечаем, что код приложения все больше замусоривается такими обращениями. Например, в приложении, работающем с базой данных, код загромождается строками SQL-запросов, внутри которых скрыты бизнес-правила.

Настоящая проблема возникает при изменении стороннего API. В случае баз данных проблемой становится и изменение схемы. По мере выпуска новых версий API или схемы приходится перерабатывать все более объемные части приложения.

В конце концов разработчики приходят к выводу о том, что надо как-то оградить себя от этих изменений. И изобретают слой, отделяющий прикладные бизнес-правила от стороннего API (рис. 34.6). В этот слой они переносят весь код, в котором используется сторонний API и все концепции, относящиеся к этому API, а не к бизнес-правилам.

Иногда подобные слои, например ADO.NET, можно купить. Они отделяют код приложения от СУБД. Разумеется, они сами по себе являются сторонними API, поэтому может возникнуть необходимость экранироваться и от них.

Отметим, что между `Приложением` и `API` существует транзитивная зависимость. Иногда даже такой непрямой зависимости достаточно, чтобы

создать проблемы. Например, ADO.NET не изолирует приложение от деталей схемы.

Чтобы добиться лучшей изоляции, необходимо инвертировать зависимость между приложением и изолирующим слоем (см. рис. 34.7). Это экранирует приложение от знания, прямого или опосредованного, о самом существовании стороннего API. В случае базы данных приложение может ничего не знать о схеме. А в случае ПО промежуточного уровня – о типах данных, используемых соответствующим процессором.

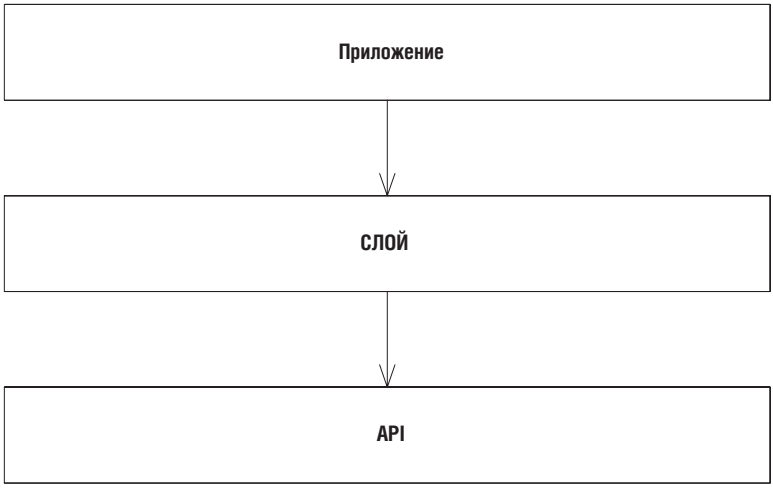


Рис. 34.6. *Включение изолирующего слоя*

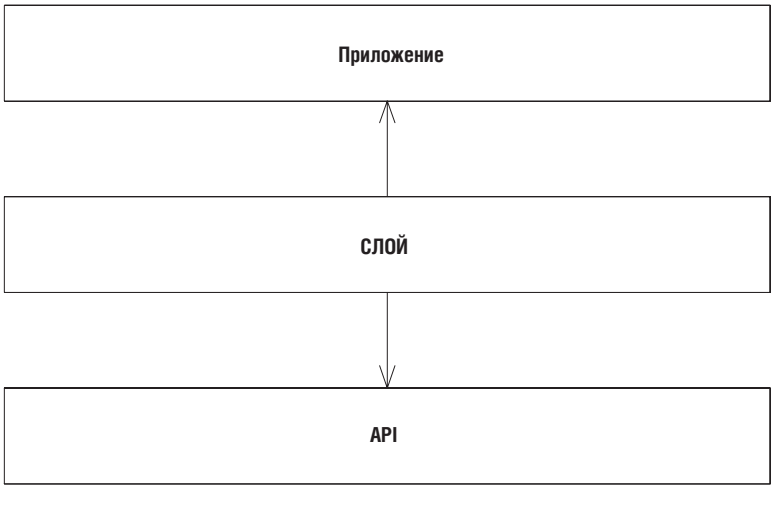


Рис. 34.7. *Инвертирование зависимости между приложением и слоем*

Именно такую организацию зависимостей и обеспечивает паттерн Заместитель. Приложение вообще не зависит от заместителей. Наоборот, все заместители зависят от приложения и API. Тем самым все знания о связях между приложением и API сосредоточены в заместителях.

Такое концентрированное знание означает, что заместители превращаются в ночной кошмар. Стоит API измениться, как тут же надо модифицировать заместители. Управляться с ними становится очень трудно.

Но знать, где сосредоточены кошмары, уже хорошо. Не будь заместителей, кошмары распространились бы по всему коду приложения.

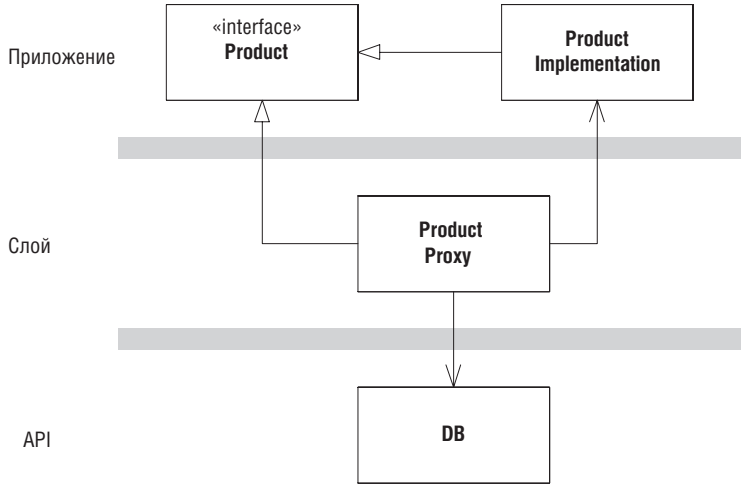


Рис. 34.8. *Как Заместитель инвертирует зависимость между приложением и слоем*

В большинстве приложений заместители не нужны. Это довольно тяжеловесный подход. Если я вижу решение, где используются заместители, то обычно советую избавиться от них и взять что-нибудь попроще. Но иногда жесткое разделение приложения и API, обеспечиваемое заместителями, действительно полезно. Такое встречается почти всегда в очень больших системах, где схема и/или API очень часто изменяются, а также в системах, способных работать поверх различных СУБД или ПО промежуточного уровня.

Шлюз к табличным данным

Паттерн Заместитель трудно использовать, и для большинства приложений он избыточен. Я не стал бы прибегать к нему, не будучи уверен в том, что необходимо безусловное отделение бизнес-правил от схемы базы данных. Обычно такое абсолютное разделение, какое дает Заместитель, не так уж нужно и с наличием небольшой связанности между

бизнес-правилами и схемой можно смириться. Паттерн Шлюз к табличным данным (Table Data Gateway – TDG), как правило, позволяет достичь приемлемого разделения без затрат, характерных для Заместителя. Этот паттерн также известен под названием «Объект доступа к данным» (Data Access Object – DAO). В нем для каждого объекта, который мы хотим хранить в базе данных, применяется специализированный Фасад (рис. 34.9).

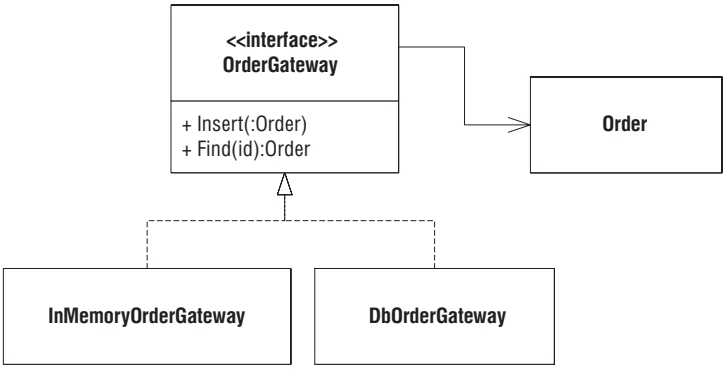


Рис. 34.9. *Паттерн Шлюз к табличным данным*

`OrderGateway` (листинг 34.24) – это интерфейс, которым приложение пользуется для доступа к слою, реализующему сохранение объектов `Order`. В нем объявлены методы `Insert` для добавления новых объектов и `Find` для выборки существующих.

Класс `DbOrderGateway` (листинг 34.25) реализует `OrderGateway` и перемещает экземпляры `Order` между объектной моделью и реляционной базой данных. Он устанавливает соединение с сервером `SqlServer` и работает с той же схемой, которую мы применяли при рассмотрении паттерна Заместитель.[1]

Листинг 34.24. OrderGateway.cs

```
public interface OrderGateway
{
```

[1] Должен сказать, что терпеть не могу системы доступа к базам данных на большинстве основных современных платформ. Сама мысль о составлении и выполнении SQL-запросов неудачна и неуместна. Позор, что нам приходится писать программы для генерации запросов на языке SQL, который изначально был задуман для того, чтобы на нем писали люди, а в результате анализируется и интерпретируется СУБД. Можно (и должно) придумать более прямой механизм. Многие команды разработчиков применяют каркасы, обеспечивающие сохранение объектов, например NHibernate, чтобы скрыть сакральные манипуляции с SQL, и это хорошо. Однако эти каркасы лишь скрывают то, что должно быть вообще исключено из употребления.

```
  void Insert(Order order);
  Order Find(int id);
}
```

Листинг 34.25. OrderGateway.cs

```
public class DbOrderGateway : OrderGateway
{
  private readonly ProductGateway productGateway;
  private readonly SqlConnection connection;

  public DbOrderGateway(SqlConnection connection,
    ProductGateway productGateway)
  {
    this.connection = connection;
    this.productGateway = productGateway;
  }

  public void Insert(Order order)
  {
    string sql = "insert into Orders (cusId) values (@cusId)" +
      "; select scope_identity()";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@cusId", order.CustomerId);
    int id = Convert.ToInt32(command.ExecuteScalar());
    order.Id = id;
    InsertItems(order);
  }

  public Order Find(int id)
  {
    string sql = "select * from Orders where orderId = @id";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@id", id);
    IDataReader reader = command.ExecuteReader();
    Order order = null;
    if(reader.Read())
    {
      string customerId = reader["cusId"].ToString();
      order = new Order(customerId);
      order.Id = id;
    }
    reader.Close();
    if(order != null)
      LoadItems(order);
    return order;
  }

  private void LoadItems(Order order)
  {
    string sql =
      "select * from Items where orderId = @orderId";
```

```
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@orderId", order.Id);
    IDataReader reader = command.ExecuteReader();
    while(reader.Read())
    {
      string sku = reader["sku"].ToString();
      int quantity = Convert.ToInt32(reader["quantity"]);
      Product product = productGateway.Find(sku);
      order.AddItem(product, quantity);
    }
  }

  private void InsertItems(Order order)
  {
    string sql = «insert into Items (orderId, quantity, sku)» +
      «values (@orderId, @quantity, @sku)»;
    foreach(Item item in order.Items)
    {
      SqlCommand command = new SqlCommand(sql, connection);
      command.Parameters.Add(«@orderId», order.Id);
      command.Parameters.Add("@quantity", item.Quantity);
      command.Parameters.Add("@sku", item.Product.Sku);
      command.ExecuteNonQuery();
    }
  }
}
```

Еще одна реализация `OrderGateway` **— класс** `InMemoryOrderGateway` **(листинг 34.26), который сохраняет и ищет объекты** `Order` **точно так же, как** `DbOrderGateway`**, но делает это в памяти, используя в качестве контейнера объект** `Hashtable`**. Сохранение данных в памяти на первый взгляд представляется глупостью, потому что все данные пропадут после завершения работы программы. Но, как мы вскоре увидим, такой подход очень полезен при тестировании.**

Листинг 34.26. InMemoryOrderGateway.cs

```
public class InMemoryOrderGateway : OrderGateway
{
  private static int nextId = 1;
  private Hashtable orders = new Hashtable();

  public void Insert(Order order)
  {
    orders[nextId++] = order;
  }

  public Order Find(int id)
  {
    return orders[id] as Order;
  }
}
```

Имеется также интерфейс `ProductGateway` **(листинг 34.27) вкупе с его реализациями для хранения в базе данных (листинг 34.28) и в памяти (листинг 34.29). Можно было бы ввести еще интерфейс** `ItemGateway` **для доступа к таблице** `Item`**, но это необязательно. Приложению не нужны позиции** `Items` **вне контекста заказа** `Order`**, поэтому класс** `DbOrderGateway` **обслуживает обе таблицы,** `Orders` **и** `Items`**.**

Листинг 34.27. ProductGateway.cs

```
public interface ProductGateway
{
  void Insert(Product product);
  Product Find(string sku);
}
```

Листинг 34.28. DbProductGateway.cs

```
public class DbProductGateway : ProductGateway
{
  private readonly SqlConnection connection;

  public DbProductGateway(SqlConnection connection)
  {
    this.connection = connection;
  }

  public void Insert(Product product)
  {
    string sql = "insert into Products (sku, name, price)" +
      " values (@sku, @name, @price)";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@sku", product.Sku);
    command.Parameters.Add("@name", product.Name);
    command.Parameters.Add("@price", product.Price);
    command.ExecuteNonQuery();
  }

  public Product Find(string sku)
  {
    string sql = "select * from Products where sku = @sku";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@sku", sku);
    IDataReader reader = command.ExecuteReader();
    Product product = null;
    if(reader.Read())
    {
      string name = reader[«name»].ToString();
      int price = Convert.ToInt32(reader[«price»]);
      product = new Product(name, sku, price);
    }
    reader.Close();
    return product;
```

```
  }
}
```

Листинг 34.29. InMemoryProductGateway.cs

```
public class InMemoryProductGateway : ProductGateway
{
  private Hashtable products = new Hashtable();

  public void Insert(Product product)
  {
    products[product.Sku] = product;
  }

  public Product Find(string sku)
  {
    return products[sku] as Product;
  }
}
```

Классы `Product` (листинг 34.30), `Order` (листинг 34.31) и `Item` (листинг 34.32) – простые объекты передачи данных (Data Transfer Objects – DTO), согласованные с исходной моделью.

Листинг 34.30. Product.cs

```
public class Product
{
  private readonly string name;
  private readonly string sku;
  private int price;

  public Product(string name, string sku, int price)
  {
    this.name = name;
    this.sku = sku;
    this.price = price;
  }

  public int Price
  {
    get { return price; }
  }

  public string Name
  {
    get { return name; }
  }

  public string Sku
  {
    get { return sku; }
  }
}
```

Листинг 34.31. Order.cs

```
public class Order
{
  private readonly string cusid;
  private ArrayList items = new ArrayList();
  private int id;

  public Order(string cusid)
  {
    this.cusid = cusid;
  }

  public string CustomerId
  {
    get { return cusid; }
  }

  public int Id
  {
    get { return id; }
    set { id = value; }
  }

  public int ItemCount
  {
    get { return items.Count; }
  }

  public int QuantityOf(Product product)
  {
    foreach(Item item in items)
    {
      if(item.Product.Sku.Equals(product.Sku))
        return item.Quantity;
    }
    return 0;
  }

  public void AddItem(Product p, int qty)
  {
    Item item = new Item(p,qty);
    items.Add(item);
  }

  public ArrayList Items
  {
    get { return items; }
  }

  public int Total
  {
```

```
      get
      {
        int total = 0;
        foreach(Item item in items)
        {
          Product p = item.Product;
          int qty = item.Quantity;
          total += p.Price * qty;
        }
        return total;
      }
    }
  }
```

Листинг 34.32. Item.cs

```
public class Item
{
  private Product product;
  private int quantity;

  public Item(Product p, int qty)
  {
    product = p;
    quantity = qty;
  }

  public Product Product
  {
    get { return product; }
  }

  public int Quantity
  {
    get { return quantity; }
  }
}
```

Тестирование и шлюз к табличным данным в памяти

Всякий, кто хоть раз на практике применял разработку через тестирование, согласится, что тесты имеют тенденцию быстро накапливаться. Не успеешь оглянуться, а их уже сотни. Время, необходимое для выполнения всех тестов, с каждым днем увеличивается. Многие тесты работают со слоем сохранения, поэтому если всякий раз обращаться к реальной базе данных, то за время выполнения всего комплекта тестов вполне можно было бы попить чайку. Сотни обращений к базе данных занимают много времени. Вот тут-то и оказывается полезен класс `InMemoryOrderGateway`. Поскольку все данные хранятся в памяти, то затраты на доступ к внешнему хранилищу равны нулю.

Использование объектов InMemoryGateway во время тестирования не только экономит массу времени, но и позволяет забыть о конфигурировании базы данных, что упрощает тестовый код. К тому же в конце выполнения теста не нужно приводить базу данных в исходное состояние, достаточно передать ее сборщику мусора.

Классы InMemoryGateway полезны и для приемочного тестирования. С их помощью можно выполнить все приложение, не имея настоящей базы данных. Я этим не раз пользовался. Вы увидите, что класс InMemoryOrderGateway совсем не велик и чрезвычайно прост.

Разумеется, часть автономных и приемочных тестов должна работать со шлюзами к настоящим базам данных. Убедиться в том, что система правильно работает с настоящей базой, *необходимо*. Но большую часть тестов все же можно выполнить со шлюзами к данным в памяти.

Учитывая все достоинства шлюзов к данным в памяти, имеет смысл применять их как можно шире. Лично я когда использую паттерн Шлюз к табличным данным, сначала пишу реализацию InMemoryGateway, а классы DbGateway откладываю на потом. Значительную часть приложения можно написать, ограничиваясь одними лишь классами InMemoryGateway. Код приложения не знает о том, что работает не с реальной базой данных. Поэтому неважно, какую СУБД вы в конечном итоге выберете и как будет выглядеть схема. На самом деле реализация классов DbGateway может быть одним из последних этапов разработки программы.

Тестирование шлюзов к базе данных

В листингах 34.34 и 34.35 показаны автономные тесты для классов DBProductGateway и DBOrderGateway. Их структура интересна, потому что они наследуют общему абстрактному базовому классу AbstractDBGatewayTest.

Отметим, что конструктору DbOrderGateway передается объект ProductGateway. Кроме того, обратите внимание, что в тестах используется класс InMemoryProductGateway, а не DbProductGateway. Несмотря на эту уловку, код прекрасно работает, и это избавляет от необходимости обращаться к реальной базе при выполнении тестов.

Листинг 34.33. AbstractDbGatewayTest.cs

```
public class AbstractDbGatewayTest
{
  protected SqlConnection connection;
  protected DbProductGateway gateway;
  protected IDataReader reader;

  protected void ExecuteSql(string sql)
  {
    SqlCommand command =
      new SqlCommand(sql, connection);
```

```
    command.ExecuteNonQuery();
    }

    protected void OpenConnection()
    {
      string connectionString =
        "Initial Catalog=QuickyMart;" +
        "Data Source=marvin;" +
        "user id=sa;password=abc;";
      connection = new SqlConnection(connectionString);
      this.connection.Open();
    }

    protected void Close()
    {
      if(reader != null)
        reader.Close();
      if(connection != null)
        connection.Close();
    }
  }
```

Листинг 34.34. DbProductGatewayTest.cs

```
  [TestFixture]
  public class DbProductGatewayTest : AbstractDbGatewayTest
  {
    private DbProductGateway gateway;

    [SetUp]
    public void SetUp()
    {
      OpenConnection();
      gateway = new DbProductGateway(connection);
      ExecuteSql("delete from Products");
    }

    [TearDown]
    public void TearDown()
    {
      Close();
    }

    [Test]
    public void Insert()
    {
      Product product = new Product("Peanut Butter", "pb", 3);
      gateway.Insert(product);
      SqlCommand command =
        new SqlCommand("select * from Products", connection);
      reader = command.ExecuteReader();
      Assert.IsTrue(reader.Read());
```

```
    Assert.AreEqual("pb", reader["sku"]);
    Assert.AreEqual("Peanut Butter", reader["name"]);
    Assert.AreEqual(3, reader["price"]);
    Assert.IsFalse(reader.Read());
  }

  [Test]
  public void Find()
  {
    Product pb = new Product("Peanut Butter", "pb", 3);
    Product jam = new Product("Strawberry Jam", "jam", 2);
    gateway.Insert(pb);
    gateway.Insert(jam);
    Assert.IsNull(gateway.Find("bad sku"));
    Product foundPb = gateway.Find(pb.Sku);
    CheckThatProductsMatch(pb, foundPb);
    Product foundJam = gateway.Find(jam.Sku);
    CheckThatProductsMatch(jam, foundJam);
  }

  private static void CheckThatProductsMatch(Product pb,
    Product pb2)
  {
    Assert.AreEqual(pb.Name, pb2.Name);
    Assert.AreEqual(pb.Sku, pb2.Sku);
    Assert.AreEqual(pb.Price, pb2.Price);
  }
}
```

Листинг 34.35. DbOrderGatewayTest.cs

```
[TestFixture]
public class DbOrderGatewayTest : AbstractDbGatewayTest
{
  private DbOrderGateway gateway;
  private Product pizza;
  private Product beer;

  [SetUp]
  public void SetUp()
  {
    OpenConnection();
    pizza = new Product("Pizza", "pizza", 15);
    beer = new Product("Beer", "beer", 2);
    ProductGateway productGateway =
      new InMemoryProductGateway();
    productGateway.Insert(pizza);
    productGateway.Insert(beer);
    gateway = new DbOrderGateway(connection, productGateway);
    ExecuteSql("delete from Orders");
    ExecuteSql("delete from Items");
  }
```

```
  [TearDown]
  public void TearDown()
  {
    Close();
  }

  [Test]
  public void Find()
  {
    string sql = "insert into Orders (cusId) " +
      "values ('Snoopy'); select scope_identity()";
    SqlCommand command = new SqlCommand(sql, connection);
    int orderId = Convert.ToInt32(command.ExecuteScalar());
    ExecuteSql(String.Format("insert into Items (orderId, " +
      "quantity, sku) values ({0}, 1, 'pizza')", orderId));
    ExecuteSql(String.Format("insert into Items (orderId, " +
      "quantity, sku) values ({0}, 6, 'beer')", orderId));
    Order order = gateway.Find(orderId);
    Assert.AreEqual("Snoopy", order.CustomerId);
    Assert.AreEqual(2, order.ItemCount);
    Assert.AreEqual(1, order.QuantityOf(pizza));
    Assert.AreEqual(6, order.QuantityOf(beer));
  }

  [Test]
  public void Insert()
  {
    Order order = new Order("Snoopy");
    order.AddItem(pizza, 1);
    order.AddItem(beer, 6);
    gateway.Insert(order);
    Assert.IsTrue(order.Id != -1);
    Order foundOrder = gateway.Find(order.Id);
    Assert.AreEqual("Snoopy", foundOrder.CustomerId);
    Assert.AreEqual(2, foundOrder.ItemCount);
    Assert.AreEqual(1, foundOrder.QuantityOf(pizza));
    Assert.AreEqual(6, foundOrder.QuantityOf(beer));
  }
}
```

Другие паттерны работы с базами данных

При работе с базами данных можно применять еще четыре паттерна: Объект расширения (Extension Object), Посетитель (Visitor), Декоратор (Decorator) и Фасад (Facade).[1]

1. **Объект расширения.** Представьте себе объект, который знает, как записать в базу данных расширенный (то есть дополненный некото-

[1] Первые три паттерна обсуждаются в главе 35, а Фасад – в главе 23.

рыми полями) объект. Чтобы выполнить операцию записи, вы должны будете запросить объект расширения с ключом `Database`, привести его к типу `DatabaseWriterExtension` и вызвать метод `Write`:

```
Product p = /* метод, возвращающий Product */
ExtensionObject e = p.GetExtension("Database");
if (e != null)
{
  DatabaseWriterExtension dwe = (DatabaseWriterExtension) e;
  e.Write();
}
```

2. **Посетитель.** Представьте себе класс посетителя, который знает, как записать посещенный объект в базу данных. Чтобы записать объект в базу, вы должны будете создать посетитель подходящего типа и вызвать метод `Accept` записываемого объекта:

```
Product p = /* метод, возвращающий Product */
DatabaseWriterVisitor dwv = new DatabaseWritierVisitor();
p.Accept(dwv);
```

3. **Декоратор.** Есть два способа применить декоратор для работы с базами данных. Можно декорировать бизнес-объект, пополнив его методами чтения и записи, или же декорировать объект данных, уже умеющий читать и записывать себя, снабдив его бизнес-правилами. Последний подход довольно часто применяется при работы с объектно-ориентированными базами данных. Бизнес-правила хранятся вне схемы ООБД и добавляются с помощью декораторов.

4. **Фасад.** Это моя любимая отправная точка. Посудите сами, паттерн Шлюз к табличным данным является частным случаем Фасада. Его недостаток в том, что он не полностью отделяет объекты бизнес-правил от базы данных. Его структура изображена на рис. 34.10. Класс `DatabaseFacade` предоставляет методы для чтения и записи всех

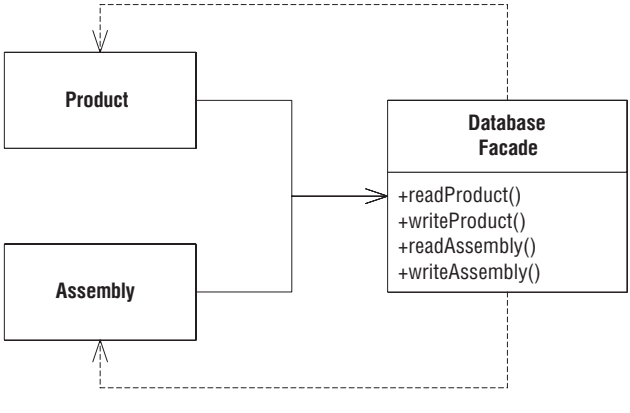


***Рис. 34.10.** Фасад Database*

необходимых объектов. Это связывает сами объекты с `DatabaseFacade` и наоборот. Объекты знают о фасаде, потому что именно они чаще всего и вызывают методы чтения и записи. Фасад знает об объектах, потому что должен обращаться к их свойствам-акцессорам из методов чтения и записи.

В больших приложениях такая связанность может стать помехой, но если приложение невелико или только начинает развиваться, то эта техника вполне эффективна. Если вы начали с фасада, а позднее решили перейти к другому паттерну, чтобы уменьшить связанность, то рефакторинг не вызовет особых затруднений.

Заключение

Очень заманчиво предвидеть необходимость в паттерне Заместитель задолго до того, как она реально возникла. Но эта идея очень редко бывает удачной. Я рекомендую начать со Шлюза к табличным данным или какой-то другой разновидности Фасада, а потом, если понадобится, прибегнуть к рефакторингу. Так вы сэкономите время и избежите неприятностей.

Библиография

[Fowler03] Martin Fowler «Patterns of Enterprise Application Architecture», Addison-Wesley, 2003.[1]

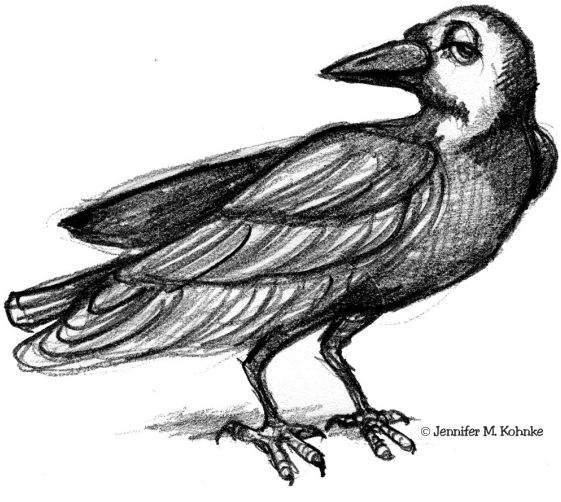
[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

[Martin97] Robert C. Martin «Design Patterns for Dealing with Dual Inheritance Hierarchies», *C++ Report*, April 1997.

[1] Мартин Фаулер «Архитектура корпоративных программных приложений». – Пер. с англ. – Вильямс, 2007; Мартин Фаулер и др. «Шаблоны корпоративных приложений». – Пер. с англ. – Вильямс, 2010.

35

Посетитель



«Это – гость, – пробормотал я, –
там, у входа моего, гость, – и больше ничего!»

Эдгар По «Ворон»

Вам необходимо добавить новый метод в иерархию классов, но сделать это трудно или нанесет ущерб дизайну. Проблема довольно типичная. Пусть, например, имеется иерархия классов `Modem`. В базовом классе есть методы, общие для всех модемов. Производные классы соответствуют модемам различных производителей и типов. Предположим, что требуется добавить в иерархию новый метод `configureForUnix`, который будет конфигурировать модем для работы в операционной системе UNIX.

В каждом подклассе он будет делать что-то свое, потому что у каждого модема собственные представления о задании конфигурации и о работе в UNIX.

К сожалению, добавление метода `ConfigureForUnix` порождает крайне неприятные вопросы. Как быть с Windows, OSX или Linux? Следует ли добавлять по одному методу для каждой операционной системы? Ясно, что это некрасиво. Эдак мы никогда не закроем интерфейс `Modem`. Стоит выйти новой ОС, как придется вновь модифицировать этот интерфейс и заново развертывать все программы, работающие с модемами.

Семейство паттернов Посетитель (Visitor) позволяет добавлять в существующую иерархию новые методы, не изменяя саму иерархию. В этой семейство входят следующие паттерны:[1]

- Посетитель
- Ациклический посетитель (Acyclic Visitor)
- Декоратор (Decorator)
- Объект расширения (Extension Object)

Посетитель

Рассмотрим иерархию классов `Modem`, показанную на рис. 35.1. Интерфейс `Modem` содержит общие методы, которые могут реализовать все модемы. На рисунке представлены также три производных от него класса: для модемов производства компаний Hayes и Zoom, а также разработанного нашим собственным инженером Эрни. Как сконфигурировать эти модемы для работы в UNIX, не вводя метод `ConfigureForUnix` в интерфейс `Modem`? Можно воспользоваться техникой *двойной диспетчеризации*, лежащей в основе паттерна Посетитель.

На рис. 35.2 изображена структура паттерна Посетитель, а в листингах 35.1–35.5 приведен соответствующий код на C#. В листинге 35.6 показан тестовый код, который проверяет работу Посетителя и демонстрирует другим программистам, как его следует использовать.

Обратите внимание, что в иерархии посетителей есть методы для каждого подкласса посещаемой иерархии. Тут мы наблюдаем поворот на 90 градусов: от производных классов к методам.

Из тестового кода видно, что для того, чтобы сконфигурировать модем для UNIX, программист должен создать экземпляр класса `UnixModemConfigurator` и передать его методу `Accept` класса `Modem`. Объект подкласса `Modem` затем вызовет метод `Visit(this)` класса `ModemVisitor`, базового для `UnixModemConfigurator`. Если это был подкласс `Hayes`, то `Visit(this)` в действительности является обращением к методу `public void Visit(Hayes)`

[1] [GOF95]. О паттернах Ациклический посетитель и Объект расширения см. [PLOPD3].

класса `UnixModemConfigurator`, который сконфигурирует модем Hayes для работы в UNIX.

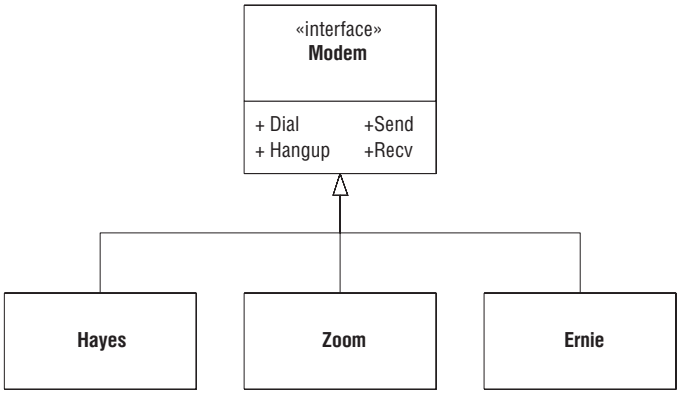


Рис. 35.1. *Иерархия классов Modem*

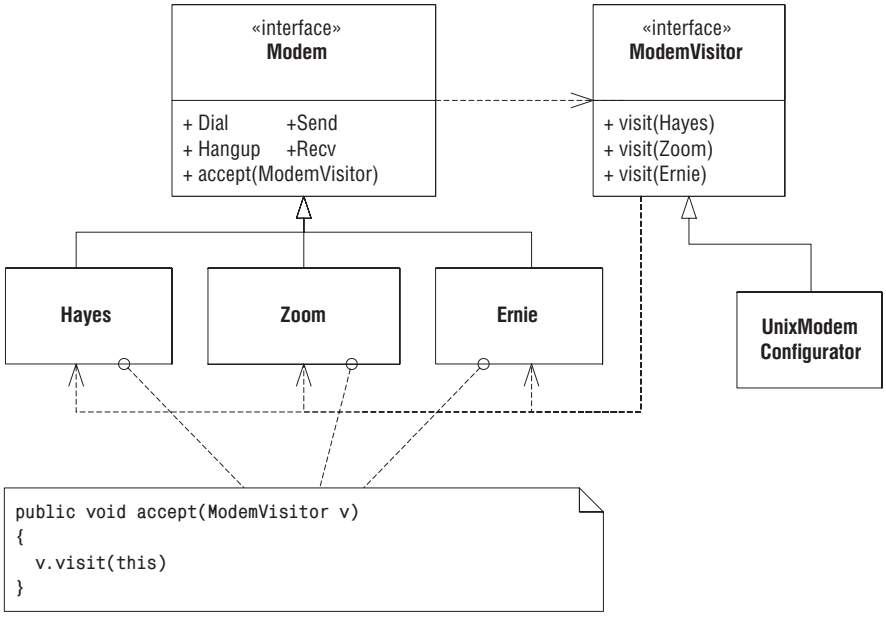


Рис. 35.2. *Паттерн Посетитель*

Листинг 35.1. Modem.cs

```
public interface Modem
{
  void Dial(string pno);
  void Hangup();
```

```
  void Send(char c);
  char Recv();
  void Accept(ModemVisitor v);
}
```

Листинг 35.2. HayesModem.cs

```
public class HayesModem : Modem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v) {v.Visit(this);}
  public string configurationString = null;
}
```

Листинг 35.3. ZoomModem.cs

```
public class ZoomModem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v) {v.Visit(this);}
  public int configurationValue = 0;
}
```

Листинг 35.4. ErnieModem.cs

```
public class ErnieModem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v) {v.Visit(this);}
  public string internalPattern = null;
}
```

Листинг 35.5. UnixModemConfigurator.cs

```
public class UnixModemConfigurator : ModemVisitor
{
  public void Visit(HayesModem m)
  {
    m.configurationString = “&s1=4&D=3”;
  }

  public void Visit(ZoomModem m)
  {
    m.configurationValue = 42;
  }
```

```
  public void Visit(ErnieModem m)
  {
    m.internalPattern = "C is too slow";
  }
}
```

Листинг 35.6. ModemVisitorTest.cs

```
[TestFixture]
public class ModemVisitorTest
{
  private UnixModemConfigurator v;
  private HayesModem h;
  private ZoomModem z;
  private ErnieModem e;

  [SetUp]
  public void SetUp()
  {
    v = new UnixModemConfigurator();
    h = new HayesModem();
    z = new ZoomModem();
    e = new ErnieModem();
  }

  [Test]
  public void HayesForUnix()
  {
    h.Accept(v);
    Assert.AreEqual("&s1=4&D=3", h.configurationString);
  }

  [Test]
  public void ZoomForUnix()
  {
    z.Accept(v);
    Assert.AreEqual(42, z.configurationValue);
  }

  [Test]
  public void ErnieForUnix()
  {
    e.Accept(v);
    Assert.AreEqual("C is too slow", e.internalPattern);
  }
}
```

Имея такую структуру, мы можем добавлять методы конфигурирования для новых операционных систем, просто порождая новые подклассы `ModemVisitor` безо всякого изменения иерархии `Modem`. Таким образом, паттерн Посетитель заменяет добавление новых методов в иерархию `Modem` введением новых подклассов `ModemVisitor`.

Двойная диспетчеризация сводится к двум полиморфным операциям. Первая – метод `Accept`, он по-разному ведет себя в зависимости от объекта, для которого вызван. Вторая – метод `Visit`, вызываемый из того или иного варианта `Accept`, – выбирает конкретный вариант исполняемой функции.

Две диспетчеризации в паттерне Посетитель образуют матрицу функций. В нашем примере строками матрицы являются различные типы модемов, а столбцами – различные типы операционных систем. В каждой клетке матрицы находится функция, описывающая способ инициализации одного модема для одной операционной системы.

Паттерн работает быстро. Для выполнения требуется всего два полиморфных вызова вне зависимости от ширины или глубины посещаемой иерархии.

Ациклический посетитель

Отметим, что базовый класс посещаемой иерархии (`Modem`) зависит от базового класса иерархии посетителей (`ModemVisitor`). Отметим также, что в базовом классе иерархии посетителей имеется по одному методу для каждого производного класса в посещаемой иерархии. Этот цикл зависимостей связывает все посещаемые подклассы – все типы модемов – воедино, затрудняя инкрементную компиляцию посетителей или добавление новых подклассов в посещаемую иерархию.

Паттерн Посетитель хорошо работает в случаях когда в посещаемую иерархию новые производные классы добавляются не слишком часто. Если не ожидается никаких модемов, кроме `Hayes`, `Zoom` и `Ernie`, или хотя бы новые подклассы `Modem` добавляются эпизодически, то Посетитель подойдет.

С другой стороны, если посещаемая иерархия изменяется очень часто и то и дело создаются новые подклассы, то базовый класс посетителей (например, `ModemVisitor`) придется постоянно модифицировать и перекомпилировать вместе со всеми производными от него.

Решить эти проблемы поможет паттерн Ациклический посетитель[1] (рис. 35.3). В этом варианте цикл зависимостей разрывается за счет того, что базовый класс `Visitor` (в нашем случае `ModemVisitor`) делается *вырожденным*, то есть не содержащим ни одного метода. Это означает, что он не зависит от подклассов, добавляемых в посещаемую иерархию.

Классы, производные от `Visitor`, наследуют также интерфейсам посетителя. Существует по одному такому интерфейсу для каждого подкласса в посещаемой иерархии. Тем самым мы совершаем поворот на 180 градусов от производных классов к интерфейсам. Методы `Accept` в классах посещаемой иерархии приводят базовый класс к типу соответствую-

[1] [PLOPD3], стр. 93

щего интерфейса посетителя. Если приведение прошло успешно, то вызывается соответствующий метод visit. Код показан в листингах 35.7–35.16.

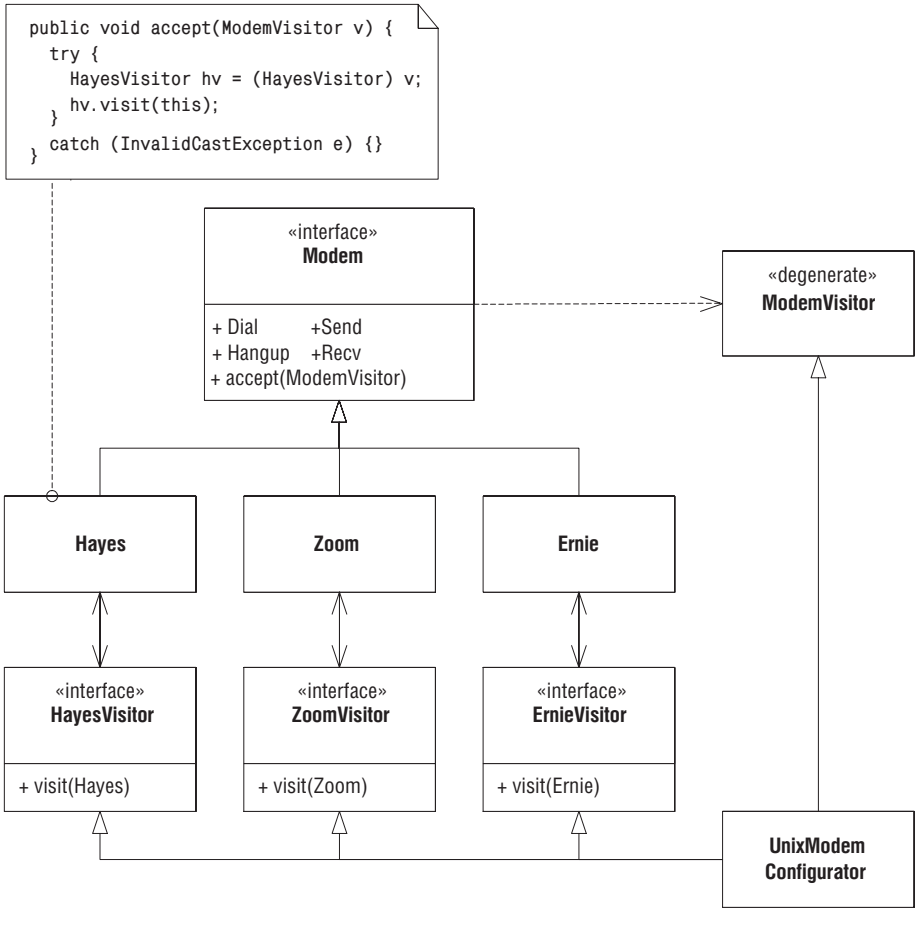


***Рис. 35.3.** Ациклический посетитель*

Листинг 35.7. Modem.cs

```
public interface Modem
{
  void Dial(string pno);
  void Hangup();
  void Send(char c);
  char Recv();
  void Accept(ModemVisitor v);
}
```

Листинг 35.8. ModemVisitor.cs

```
public interface ModemVisitor
{
}
```

Листинг 35.9. ErnieModemVisitor.cs

```
public interface ErnieModemVisitor : ModemVisitor
{
  void Visit(ErnieModem m);
}
```

Листинг 35.10. HayesModemVisitor.cs

```
public interface HayesModemVisitor : ModemVisitor
{
  void Visit(HayesModem m);
}
```

Листинг 35.11. ZoomModemVisitor.cs

```
public interface ZoomModemVisitor : ModemVisitor
{
  void Visit(ZoomModem m);
}
```

Листинг 35.12. ErnieModem.cs

```
public class ErnieModem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v)
  {
    if(v is ErnieModemVisitor)
      (v as ErnieModemVisitor).Visit(this);
  }

  public string internalPattern = null;
}
```

Листинг 35.13. HayesModem.cs

```
public class HayesModem : Modem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v)
  {
    if(v is HayesModemVisitor)
      (v as HayesModemVisitor).Visit(this);
  }
```

```
  public string configurationString = null;
}
```

Листинг 35.14. HayesModem.cs

```
public class ZoomModem
{
  public void Dial(string pno){}
  public void Hangup(){}
  public void Send(char c){}
  public char Recv() {return (char)0;}
  public void Accept(ModemVisitor v)
  {
    if(v is ZoomModemVisitor)
      (v as ZoomModemVisitor).Visit(this);
  }

  public int configurationValue = 0;
}
```

Листинг 35.15. UnixModemConfigurator.cs

```
public class UnixModemConfigurator
  : HayesModemVisitor, ZoomModemVisitor, ErnieModemVisitor
{
  public void Visit(HayesModem m)
  {
    m.configurationString = "&s1=4&D=3";
  }

  public void Visit(ZoomModem m)
  {
    m.configurationValue = 42;
  }

  public void Visit(ErnieModem m)
  {
    m.internalPattern = "C is too slow";
  }
}
```

Листинг 35.16. ModemVisitorTest.cs

```
[TestFixture]
public class ModemVisitorTest
{
  private UnixModemConfigurator v;
  private HayesModem h;
  private ZoomModem z;
  private ErnieModem e;

  [SetUp]
  public void SetUp()
```

```
    {
      v = new UnixModemConfigurator();
      h = new HayesModem();
      z = new ZoomModem();
      e = new ErnieModem();
    }

    [Test]
    public void HayesForUnix()
    {
      h.Accept(v);
      Assert.AreEqual("&s1=4&D=3", h.configurationString);
    }

    [Test]
    public void ZoomForUnix()
    {
      z.Accept(v);
      Assert.AreEqual(42, z.configurationValue);
    }

    [Test]
    public void ErnieForUnix()
    {
      e.Accept(v);
      Assert.AreEqual("C is too slow", e.internalPattern);
    }
  }
```

Итак, мы разорвали цикл зависимостей и упростили добавление новых посещаемых подклассов, решив одновременно вопрос с инкрементной компиляцией. К сожалению, решение оказалось гораздо сложнее. Хуже того, скорость приведения типов может зависеть от ширины и глубины посещаемой иерархии, а потому ее трудно предсказать.

В системах реального времени из-за длительного и непредсказуемого времени, требуемого для приведения типа, паттерн Ациклический посетитель может оказаться неприемлемым. В других системах причиной отказа от него может стать излишняя сложность. Однако если посещаемая иерархия изменчива, а время компиляции важно, то этот паттерн – все же неплохая альтернатива.

Выше я объяснял, что паттерн Посетитель порождает матрицу функций, в которой строки соответствуют посещаемому типу, а столбцы – выполняемой операции. В случае Ациклического посетителя эта матрица *разрежена*. Классы посетителей не обязаны реализовывать методы `Visit` для каждого класса в посещаемой иерархии. Например, если модемы Ernie нельзя сконфигурировать для UNIX, то класс `UnixModemConfigurator` не станет реализовывать интерфейс `ErnieVisitor`.

Применения паттерна Посетитель

Генерация отчетов. Паттерн Посетитель часто используется для обхода больших структур данных и генерации отчетов. Ценность его в этом случае заключается в том, что объекты, входящие в структуру данных, не обязаны содержать в себе код генерации отчета. Новый вид отчета можно реализовать путем добавления новых Посетителей, а не изменением кода в объектах структуры данных. Это означает, что отчеты можно поместить в отдельные компоненты и устанавливать только у тех заказчиков, которым они нужны.

Рассмотрим простую структуру данных, представляющую спецификацию изделия (рис. 35.4). На ее основе можно сгенерировать самые разные отчеты. Например, отчет о полной стоимости сборки или отчет обо всех входящих в сборку деталях.

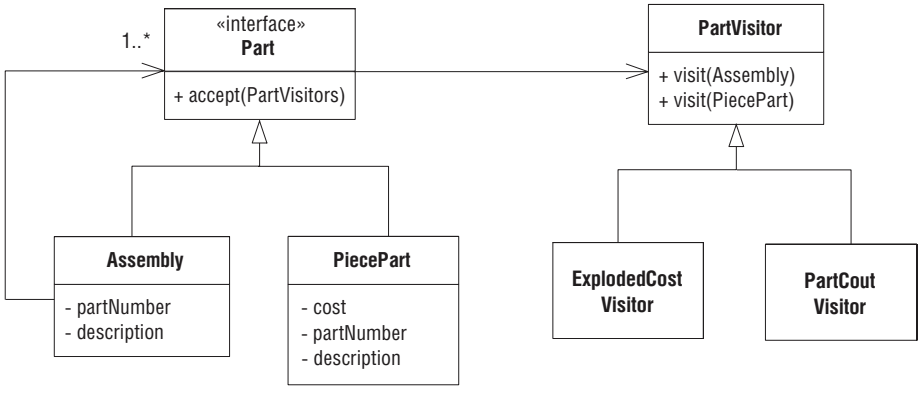


***Рис. 35.4.** Структура генератора отчетов для спецификации изделия*

Любой отчет можно было бы сгенерировать с помощью методов в классе `Part`. Например, в него можно добавить свойства `ExplodedCost` и `PieceCount` и реализовать их в каждом подклассе `Part` таким образом, чтобы получился нужный отчет. Увы, если пользователям понадобится новый отчет, то всю иерархию `Part` придется изменять.

Принцип единственной обязанности (SRP) гласит, что мы должны разделять код, который изменяется вследствие различных причин. Причиной изменения иерархии `Part` может служить добавление новых деталей. Но изменяться из-за того, что кому потребовались новые отчеты, она не должна. Следовательно, функциональность, связанную с отчетами, следует вынести из иерархии `Part`. Структура Посетителя, изображенная на рис. 35.4, показывает, как это сделать.

Каждый отчет реализуется в виде посетителя. Метод `Accept` в классе `Assembly` написан так, что вызывает метод посетителя `Visit`, а затем – метод `Accept` всех входящих в сборку экземпляров `Part`. Таким образом,

мы обходим все дерево и для каждого узла вызываем подходящий метод `Visit` генератора отчетов. По мере посещения узлов генератор собирает нужную ему информацию. Затем у него можно запросить интересующие данные и представить их пользователю.

Такая структура позволяет создавать неограниченное количество отчетов, не затрагивая саму иерархию деталей. Более того, каждый отчет можно откомпилировать и распространять независимо от других. Это удобно. В листингах 35.17–35.23 показано, как это решение выглядит на C#.

Листинг 35.17. Part.cs

```
public interface Part
{
  string PartNumber { get; }
  string Description { get; }
  void Accept(PartVisitor v);
}
```

Листинг 35.18. Assembly.cs

```
public class Assembly : Part
{
  private IList parts = new ArrayList();
  private string partNumber;
  private string description;

  public Assembly(string partNumber, string description)
  {
    this.partNumber = partNumber;
    this.description = description;
  }

  public void Accept(PartVisitor v)
  {
    v.Visit(this);
    foreach(Part part in Parts)
      part.Accept(v);
  }

  public void Add(Part part)
  {
    parts.Add(part);
  }

  public IList Parts
  {
    get { return parts; }
  }

  public string PartNumber
```

```
    {
      get { return partNumber; }
    }

    public string Description
    {
      get { return description; }
    }
  }
```

Листинг 35.19. PiecePart.cs

```
public class PiecePart : Part
{
  private string partNumber;
  private string description;
  private double cost;

  public PiecePart(string partNumber,
    string description,
    double cost)
  {
    this.partNumber = partNumber;
    this.description = description;
    this.cost = cost;
  }

  public void Accept(PartVisitor v)
  {
    v.Visit(this);
  }

  public string PartNumber
  {
    get { return partNumber; }
  }

  public string Description
  {
    get { return description; }
  }

  public double Cost
  {
    get { return cost; }
  }
}
```

Листинг 35.20. PartVisitor.cs

```
public interface PartVisitor
{
  void Visit(PiecePart pp);
```

```
  void Visit(Assembly a);
}
```

Листинг 35.21. ExplosiveCostExplorer.cs

```
public class ExplodedCostVisitor : PartVisitor
{
  private double cost = 0;

  public double Cost
  {
    get { return cost; }
  }

  public void Visit(PiecePart p)
  {
    cost += p.Cost;
  }

  public void Visit(Assembly a)
  {}
}
```

Листинг 35.22. PartCountVisitor.cs

```
public class PartCountVisitor : PartVisitor
{
  private int pieceCount = 0;
  private Hashtable pieceMap = new Hashtable();

  public void Visit(PiecePart p)
  {
    pieceCount++;
    string partNumber = p.PartNumber;
    int partNumberCount = 0;
    if (pieceMap.ContainsKey(partNumber))
      partNumberCount = (int)pieceMap[partNumber];
    partNumberCount++;
    pieceMap[partNumber] = partNumberCount;
  }

  public void Visit(Assembly a)
  {}

  public int PieceCount
  {
    get { return pieceCount; }
  }

  public int PartNumberCount
  {
    get { return pieceMap.Count; }
  }
```

```
  public int GetCountForPart(string partNumber)
  {
    int partNumberCount = 0;
    if (pieceMap.ContainsKey(partNumber))
      partNumberCount = (int)pieceMap[partNumber];
    return partNumberCount;
  }
}
```

Листинг 35.23. BOMReportTest.cs

```
[TestFixture]
public class BOMReportTest
{
  private PiecePart p1;
  private PiecePart p2;
  private Assembly a;

  [SetUp]
  public void SetUp()
  {
    p1 = new PiecePart("997624", "MyPart", 3.20);
    p2 = new PiecePart("7734", "Hell", 666);
    a = new Assembly("5879", "MyAssembly");
  }

  [Test]
  public void CreatePart()
  {
    Assert.AreEqual("997624", p1.PartNumber);
    Assert.AreEqual("MyPart", p1.Description);
    Assert.AreEqual(3.20, p1.Cost, .01);
  }

  [Test]
  public void CreateAssembly()
  {
    Assert.AreEqual("5879", a.PartNumber);
    Assert.AreEqual("MyAssembly", a.Description);
  }

  [Test]
  public void Assembly()
  {
    a.Add(p1);
    a.Add(p2);
    Assert.AreEqual(2, a.Parts.Count);
    PiecePart p = a.Parts[0] as PiecePart;
    Assert.AreEqual(p, p1);
    p = a.Parts[1] as PiecePart;
    Assert.AreEqual(p, p2);
  }
```

```
[Test]
public void AssemblyOfAssemblies()
{
  Assembly subAssembly = new Assembly("1324", "SubAssembly");
  subAssembly.Add(p1);
  a.Add(subAssembly);
  Assert.AreEqual(subAssembly, a.Parts[0]);
}

private class TestingVisitor : PartVisitor
{
  public IList visitedParts = new ArrayList();
  public void Visit(PiecePart p)
  {
    visitedParts.Add(p);
  }

  public void Visit(Assembly assy)
  {
    visitedParts.Add(assy);
  }
}

[Test]
public void VisitorCoverage()
{
  a.Add(p1);
  a.Add(p2);
  TestingVisitor visitor = new TestingVisitor();
  a.Accept(visitor);
  Assert.IsTrue(visitor.visitedParts.Contains(p1));
  Assert.IsTrue(visitor.visitedParts.Contains(p2));
  Assert.IsTrue(visitor.visitedParts.Contains(a));
}

private Assembly cellphone;
private 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);
}

[Test]
public void ExplodedCost()
{
  SetUpReportDatabase();
  ExplodedCostVisitor v = new ExplodedCostVisitor();
  cellphone.Accept(v);
  Assert.AreEqual(81.27, v.Cost, .001);
}

[Test]
public void PartCount()
{
  SetUpReportDatabase();
  PartCountVisitor v = new PartCountVisitor();
  cellphone.Accept(v);
  Assert.AreEqual(36, v.PieceCount);
  Assert.AreEqual(8, v.PartNumberCount);
  Assert.AreEqual(1, v.GetCountForPart(«DS-1428»), «DS-
    1428»);
  Assert.AreEqual(1, v.GetCountForPart(«SP-92»), «SP-92»);
  Assert.AreEqual(1, v.GetCountForPart(«MC-28»), «MC-28»);
  Assert.AreEqual(1, v.GetCountForPart(«CR-56»), «CR-56»);
  Assert.AreEqual(1, v.GetCountForPart(«RC-77»), «RC-77»);
  Assert.AreEqual(15, v.GetCountForPart(«CV-15»), «CV-15»);
```

```
        Assert.AreEqual(15, v.GetCountForPart(«CN-2»), «CN-2»);
        Assert.AreEqual(0, v.GetCountForPart(«Bob»), «Bob»);
      }
    }
```

Другие применения. Вообще говоря, паттерн Посетитель можно использовать в любом приложении, где имеется структура данных, которую можно интерпретировать разными способами. Компиляторы часто создают промежуточные структуры данных, представляющие синтаксически правильный исходный код. Затем на их основе генерируется двоичный код. Можно представить себе посетители, соответствующие различным процессорам и/или алгоритмам оптимизации. Можно также вообразить посетитель, который преобразует промежуточную структуру в список перекрестных ссылок или даже в UML-диаграмму.

Во многих приложениях применяются структуры конфигурационных данных. Можно представить себе, что различные подсистемы инициализируют себя на основе конфигурационных данных, обрабатывая их с помощью специализированных посетителей.

Для чего бы ни применялись посетители, неизменным остается одно: посещаемая структура данных не зависит от способа ее использования. Можно создавать новые посетители или изменять старые и устанавливать их у существующих пользователей без повторной компиляции и развертывания основных структур данных.

Декоратор

Паттерн Посетитель позволил нам добавлять в существующие иерархии новые методы, не изменяя самих иерархий. Другой способ решить ту же задачу дает паттерн Декоратор.

Снова рассмотрим иерархию `Modem` на рис. 35.1. Допустим, что имеется приложение, у которого много пользователей. Сидя за своим компьютером, пользователь может попросить систему позвонить на другой компьютер по модемной линии. Некоторые пользователи хотят слышать, как модем набирает номер, другие предпочитают, чтобы модем работал беззвучно.

Реализовать это можно, опрашивая пользовательские настройки во всех местах, где набирается номер. Если пользователь хочет слышать модем, то мы выставляем высокий уровень громкости, в противном случае отключаем динамик:

```
...
Modem m = user.Modem;
if (user.WantsLoudDial())
m.Volume = 11; // на единицу больше 10, так?
m.Dial(...);
...
```

Мысль о том, что этот фрагмент придется повторить сотни раз в коде приложения, вселяет ужас: видятся 80-часовые рабочие недели и чудовищные посиделки в обществе отладчика. Нет, этого нужно избежать любой ценой.

Другой вариант – установить флаг в самом объекте модема и заставить метод `Dial` проверять его и соответственно выставлять громкость:

```
...
public class HayesModem : Modem
{
  private bool wantsLoudDial = false;

  public void Dial(...)
  {
    if (wantsLoudDial)
    {
      Volume = 11;
    }
    ...
  }
  ...
}
```

Это лучше, но все равно код придется дублировать в каждом производном от `Modem` классе. Причем авторы этих классов должны не забыть продублировать этот код. Полагаться на память программиста – дело рискованное.

Можно было бы решить проблему, воспользовавшись паттерном Шаблонный метод.[1] Для этого нужно преобразовать интерфейс `Modem` в класс, создать в нем поле `wantsLoudDial` и проверять его в методе `Dial` перед вызовом метода `DialForReal`:

```
...
public abstract class Modem
{
  private bool wantsLoudDial = false;

  public void Dial(...)
  {
    if (wantsLoudDial)
    {
      Volume = 11;
    }
    DialForReal(...)
  }

  public abstract void DialForReal(...);
}
```

[1] См. главу 22.

Это еще лучше, но с какой стати класс `Modem` должен зависеть от капризов пользователя? Почему он должен знать о состоянии динамика? А если у пользователей появятся еще какие-то причуды, например желание выйти из системы перед тем, как повесить трубку?

И снова в игру вступает принцип общей закрытости (CCP). Мы хотим разделить явления, изменяющиеся в силу разных причин. Можно также вспомнить о принципе единственной обязанности (SRP), поскольку функция громкого набора не имеет ничего общего с внутренними функциями класса `Modem` и потому не должна быть его частью.

Паттерн Декоратор решает эту проблему путем создания совершенно нового класса: `LoudDialModem`. Он является производным от `Modem` и делегирует всю работу хранящемуся внутри него объекту `Modem`. Единственное отличие в том, что перед вызовом метода `Dial` он выставляет громкость звука. Соответствующая структура показана на рис. 35.5.

Теперь решение о том, включать ли динамик, принимается в одном месте. Там, где программа анализирует настройки, можно создать объект `LoudDialModem`, если запрашивается громкий набор, причем конструктору этого объекта передается экземпляр пользовательского модема. Метод `LoudDialModem` делегирует этому экземпляру все вызовы, так что пользователь не заметит никакой разницы. Однако метод `Dial` прежде, чем делегировать, установит громкость. После этого объект `LoudDialModem` может стать пользовательским модемом, и система об этом никогда не узнает. В листингах 35.24–35.27 приведен соответствующий код.

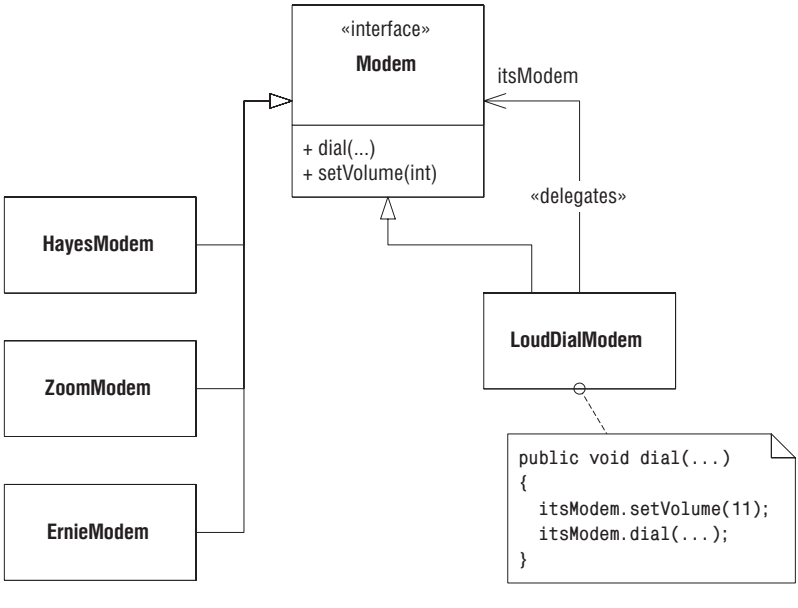


Рис. 35.5. *Паттерн Декоратор и класс LoudDialModem*

Листинг 35.24. Modem.cs

```
public interface Modem
{
  void Dial(string pno);
  int SpeakerVolume { get; set; }
  string PhoneNumber { get; }
}
```

Листинг 35.25. HayesModem.cs

```
public class HayesModem : Modem
{
  private string phoneNumber;
  private int speakerVolume;

  public void Dial(string pno)
  {
    phoneNumber = pno;
  }

  public int SpeakerVolume
  {
    get { return speakerVolume; }
    set { speakerVolume = value; }
  }

  public string PhoneNumber
  {
    get { return phoneNumber; }
  }
}
```

Листинг 35.26. LoudDialModem.cs

```
public class LoudDialModem : Modem
{
  private Modem itsModem;

  public LoudDialModem(Modem m)
  {
    itsModem = m;
  }

  public void Dial(string pno)
  {
    itsModem.SpeakerVolume = 10;
    itsModem.Dial(pno);
  }

  public int SpeakerVolume
  {
```

```
    get { return itsModem.SpeakerVolume; }
    set { itsModem.SpeakerVolume = value; }
  }

  public string PhoneNumber
  {
    get { return itsModem.PhoneNumber; }
  }
}
```

Листинг 35.27. ModemDecoratorTest.cs

```
[TestFixture]
public class ModemDecoratorTest
{
  [Test]
  public void CreateHayes()
  {
    Modem m = new HayesModem();
    Assert.AreEqual(null, m.PhoneNumber);
    m.Dial("5551212");
    Assert.AreEqual("5551212", m.PhoneNumber);
    Assert.AreEqual(0, m.SpeakerVolume);
    m.SpeakerVolume = 10;
    Assert.AreEqual(10, m.SpeakerVolume);
  }

  [Test]
  public void LoudDialModem()
  {
    Modem m = new HayesModem();
    Modem d = new LoudDialModem(m);
    Assert.AreEqual(null, d.PhoneNumber);
    Assert.AreEqual(0, d.SpeakerVolume);
    d.Dial("5551212");
    Assert.AreEqual("5551212", d.PhoneNumber);
    Assert.AreEqual(10, d.SpeakerVolume);
  }
}
```

Иногда для одной иерархии может быть два или даже больше декораторов. Например, можно декорировать иерархию `Modem` классом `LogoutExitModem`, который посылает строку `'exit'` перед вызовом метода `Hangup`. Этот второй декоратор должен будет продублировать весь код делегирования, который уже был написан в `LoudDialModem`. Устранить дублирование можно, создав новый класс `ModemDecorator`, который содержит код делегирования. Тогда реальные декораторы просто унаследуют классу `ModemDecorator` и переопределят только те методы, которые им нужны. Эта структура показана на рис. 35.6 и в листингах 35.28 и 35.29.

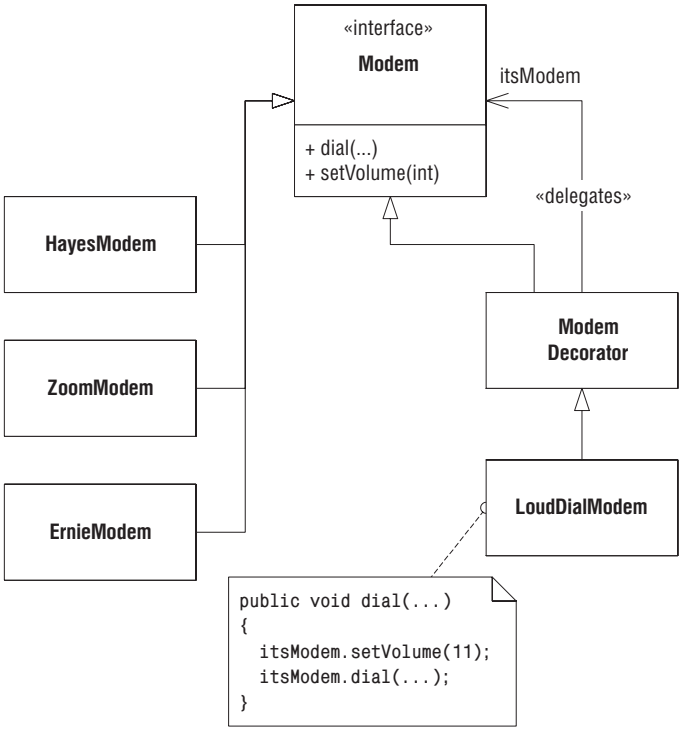


Рис. 35.6. *ModemDecorator*

Листинг 35.28. ModemDecorator.cs

```
public class ModemDecorator
{
  private Modem modem;

  public ModemDecorator(Modem m)
  {
    modem = m;
  }

  public virtual void Dial(string pno)
  {
    modem.Dial(pno);
  }

  public virtual int SpeakerVolume
  {
    get { return modem.SpeakerVolume; }
    set { modem.SpeakerVolume = value; }
  }
```

```
    public virtual string PhoneNumber
    {
      get { return modem.PhoneNumber; }
    }

    protected Modem Modem
    {
      get { return modem; }
    }
  }
```

Листинг 35.29. LoudDialModem.cs

```
public class LoudDialModem : ModemDecorator
{
  public LoudDialModem(Modem m) : base(m)
  {}

  public override void Dial(string pno)
  {
    Modem.SpeakerVolume = 10;
    Modem.Dial(pno);
  }
}
```

Объект расширения

Еще один способ добавить новую функциональность в имеющуюся иерархию, не изменяя ее, – воспользоваться паттерном Объект расширения. Он сложнее прочих, зато и гораздо мощнее и гибче. Каждый объект в иерархии хранит список специальных объектов расширения. Кроме того, каждый объект предоставляет метод, который позволяет запросить объект расширения по имени. Сам объект расширения содержит методы, манипулирующие исходным объектом в иерархии.

Снова рассмотрим задачу о спецификации изделия. Нам нужно, чтобы каждый объект в иерархии умел создавать свое XML-представление. Можно было бы включить в иерархию методы `toXML`, но это нарушит принцип ССР. Возможно, мы не хотим, чтобы части кода, относящиеся к спецификации и к XML-представлению, находились в одном классе. Можно было бы создавать XML с помощью паттерна Посетитель, но это не позволяет выделить код генерации XML для каждого типа объектов в спецификации. Ведь код генерации XML для всех классов в иерархии спецификации находился бы в одном и том же классе посетителя. А если хочется поместить код генерации XML одного вида объекта в отдельный класс?

Паттерн Объект расширения предлагает изящный способ добиться желаемого. Приведенный ниже код вводит в иерархию классов спецификации два вида объектов расширения: один преобразует объекты в фор-

мат XML, другой – в формат CSV (значения, разделенные запятыми). Для получения первого объекта мы вызываем метод `GetExtension("XML")`, второго – `GetExtension("CSV")`. Структура показана на рис. 35.7. Стереотип «marker» обозначает маркерный интерфейс, то есть интерфейс без методов.

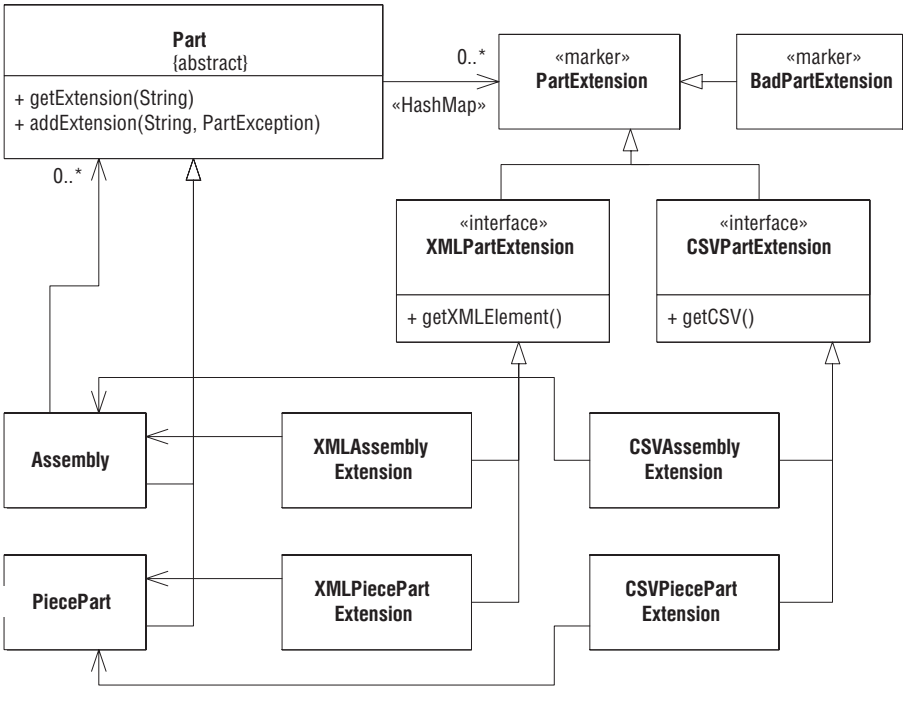


***Рис. 35.7.** Объект расширения*

Код приведен в листингах 35.30–35.41. Важно понимать, что я писал этот код не с нуля, а совершенствовал по мере разработки тестов. В листинге 35.30 показаны все тесты. Они были написаны именно в таком порядке. Каждый тест создавался до написания кода, с которым он успешно проходит. После того как тест был написан и завершался неудачно, я приступал к написанию самого кода. Код никогда не был сложнее, чем необходимо для успешного выполнения *существующих* тестов. Таким образом, код наращивался понемногу и на каждом шаге получалась работоспособная версия. Я знал, что собираюсь использовать паттерн Объект расширения, и руководствовался этим знанием при написании кода.

Листинг 35.30. BomXmlTest.CS

```
[TestFixture]
public class BomXmlTest
{
```

```
private PiecePart p1;
private PiecePart p2;
private Assembly a;

[SetUp]
public void SetUp()
{
  p1 = new PiecePart("997624", "MyPart", 3.20);
  p2 = new PiecePart("7734", "Hell", 666);
  a = new Assembly("5879", "MyAssembly");
}

[Test]
public void CreatePart()
{
  Assert.AreEqual("997624", p1.PartNumber);
  Assert.AreEqual("MyPart", p1.Description);
  Assert.AreEqual(3.20, p1.Cost, .01);
}

[Test]
public void CreateAssembly()
{
  Assert.AreEqual("5879", a.PartNumber);
  Assert.AreEqual("MyAssembly", a.Description);
}

[Test]
public void Assembly()
{
  a.Add(p1);
  a.Add(p2);
  Assert.AreEqual(2, a.Parts.Count);
  Assert.AreEqual(a.Parts[0], p1);
  Assert.AreEqual(a.Parts[1], p2);
}

[Test]
public void AssemblyOfAssemblies()
{
  Assembly subAssembly = new Assembly("1324", "SubAssembly");
  subAssembly.Add(p1);
  a.Add(subAssembly);
  Assert.AreEqual(subAssembly, a.Parts[0]);
}

private string ChildText(
  XmlElement element, string childName)
{
  return Child(element, childName).InnerText;
}
```

```
private XmlElement Child(XmlElement element, string childName)
{
  XmlNodeList children =
    element.GetElementsByTagName(childName);
  return children.Item(0) as XmlElement;
}

[Test]
public void PiecePart1XML()
{
  PartExtension e = p1.GetExtension("XML");
  XmlPartExtension xe = e as XmlPartExtension;
  XmlElement xml = xe.XmlElement;
  Assert.AreEqual(«PiecePart», xml.Name);
  Assert.AreEqual(«997624»,
  ChildText(xml, «PartNumber»));
  Assert.AreEqual(«MyPart»,
  ChildText(xml, «Description»));
  Assert.AreEqual(3.2,
  Double.Parse(ChildText(xml, «Cost»)), .01);
}

[Test]
public void PiecePart2XML()
{
  PartExtension e = p2.GetExtension(«XML»);
  XmlPartExtension xe = e as XmlPartExtension;
  XmlElement xml = xe.XmlElement;
  Assert.AreEqual(«PiecePart», xml.Name);
  Assert.AreEqual(«7734»,
  ChildText(xml, «PartNumber»));
  Assert.AreEqual(«Hell»,
  ChildText(xml, «Description»));
  Assert.AreEqual(666,
  Double.Parse(ChildText(xml, «Cost»)), .01);
}

[Test]
public void SimpleAssemblyXML()
{
  PartExtension e = a.GetExtension(«XML»);
  XmlPartExtension xe = e as XmlPartExtension;
  XmlElement xml = xe.XmlElement;
  Assert.AreEqual(«Assembly», xml.Name);
  Assert.AreEqual(«5879»,
  ChildText(xml, «PartNumber»));
  Assert.AreEqual(«MyAssembly»,
  ChildText(xml, «Description»));
  XmlElement parts = Child(xml, «Parts»);
  XmlNodeList partList = parts.ChildNodes;
  Assert.AreEqual(0, partList.Count);
}
```

```
[Test]
public void AssemblyWithPartsXML()
{
  a.Add(p1);
  a.Add(p2);
  PartExtension e = a.GetExtension("XML");
  XmlPartExtension xe = e as XmlPartExtension;
  XmlElement xml = xe.XmlElement;
  Assert.AreEqual(«Assembly», xml.Name);
  Assert.AreEqual(«5879»,
  ChildText(xml, «PartNumber»));
  Assert.AreEqual(«MyAssembly»,
  ChildText(xml, «Description»));
  XmlElement parts = Child(xml, «Parts»);
  XmlNodeList partList = parts.ChildNodes;
  Assert.AreEqual(2, partList.Count);
  XmlElement partElement =
    partList.Item(0) as XmlElement;
  Assert.AreEqual(«PiecePart», partElement.Name);
  Assert.AreEqual(«997624»,
  ChildText(partElement, «PartNumber»));
  partElement = partList.Item(1) as XmlElement;
  Assert.AreEqual(«PiecePart», partElement.Name);
  Assert.AreEqual(«7734»,
  ChildText(partElement, «PartNumber»));
}

[Test]
public void PiecePart1toCSV()
{
  PartExtension e = p1.GetExtension(«CSV»);
  CsvPartExtension ce = e as CsvPartExtension;
  String csv = ce.CsvText;
  Assert.AreEqual(«PiecePart,997624,MyPart,3.2», csv);
}

[Test]
public void PiecePart2toCSV()
{
  PartExtension e = p2.GetExtension(«CSV»);
  CsvPartExtension ce = e as CsvPartExtension;
  String csv = ce.CsvText;
  Assert.AreEqual(«PiecePart,7734,Hell,666», csv);
}

[Test]
public void SimpleAssemblyCSV()
{
  PartExtension e = a.GetExtension(«CSV»);
  CsvPartExtension ce = e as CsvPartExtension;
```

```
    String csv = ce.CsvText;
    Assert.AreEqual(«Assembly,5879,MyAssembly», csv);
  }

  [Test]
  public void AssemblyWithPartsCSV()
  {
    a.Add(p1);
    a.Add(p2);
    PartExtension e = a.GetExtension("CSV");
    CsvPartExtension ce = e as CsvPartExtension;
    String csv = ce.CsvText;
    Assert.AreEqual(«Assembly,5879,MyAssembly,» +
      «{PiecePart,997624,MyPart,3.2},» +
      «{PiecePart,7734,Hell,666}»
      , csv);
  }

  [Test]
  public void BadExtension()
  {
    PartExtension pe = p1.GetExtension(
      «ThisStringDoesn'tMatchAnyException»);
      Assert.IsTrue(pe is BadPartExtension);
  }
}
```

Листинг 35.31. Part.cs

```
public abstract class Part
{
  Hashtable extensions = new Hashtable();

  public abstract string PartNumber { get; }
  public abstract string Description { get; }
  public void AddExtension(string extensionType,
    PartExtension extension)
  {
    extensions[extensionType] = extension;
  }

  public PartExtension GetExtension(string extensionType)
  {
    PartExtension pe =
      extensions[extensionType] as PartExtension;
    if (pe == null)
      pe = new BadPartExtension();
    return pe;
  }
}
```

Листинг 35.32. PartExtension.cs

```
public interface PartExtension
{
}
```

Листинг 35.33. PiecePart.cs

```
public class PiecePart : Part
{
  private string partNumber;
  private string description;
  private double cost;

  public PiecePart(string partNumber,
    string description,
    double cost)
  {
    this.partNumber = partNumber;
    this.description = description;
    this.cost = cost;
    AddExtension("CSV", new CsvPiecePartExtension(this));
    AddExtension("XML", new XmlPiecePartExtension(this));
  }

  public override string PartNumber
  {
    get { return partNumber; }
  }

  public override string Description
  {
    get { return description; }
  }

  public double Cost
  {
    get { return cost; }
  }
}
```

Листинг 35.34. Assembly.cs

```
public class Assembly : Part
{
  private IList parts = new ArrayList();
  private string partNumber;
  private string description;

  public Assembly(string partNumber, string description)
  {
    this.partNumber = partNumber;
    this.description = description;
```

```
      AddExtension("CSV", new CsvAssemblyExtension(this));
      AddExtension("XML", new XmlAssemblyExtension(this));
    }

    public void Add(Part part)
    {
      parts.Add(part);
    }

    public IList Parts
    {
      get { return parts; }
    }

    public override string PartNumber
    {
      get { return partNumber; }
    }

    public override string Description
    {
      get { return description; }
    }
  }
```

Листинг 35.35. XmlPartExtension.cs

```
public abstract class XmlPartExtension : PartExtension
{
  private static XmlDocument document = new XmlDocument();

  public abstract XmlElement XmlElement { get; }

  protected XmlElement NewElement(string name)
  {
    return document.CreateElement(name);
  }

  protected XmlElement NewTextElement(
    string name, string text)
  {
    XmlElement element = document.CreateElement(name);
    XmlText xmlText = document.CreateTextNode(text);
    element.AppendChild(xmlText);
    return element;
  }
}
```

Листинг 35.36. XmlPiecePartExtension.cs

```
public class XmlPiecePartExtension : XmlPartExtension
{
  private PiecePart piecePart;
```

```
  public XmlPiecePartExtension(PiecePart part)
  {
    piecePart = part;
  }

  public override XmlElement XmlElement
  {
    get
    {
      XmlElement e = NewElement("PiecePart");
      e.AppendChild(NewTextElement(
        "PartNumber", piecePart.PartNumber));
      e.AppendChild(NewTextElement(
        "Description", piecePart.Description));
      e.AppendChild(NewTextElement(
        "Cost", piecePart.Cost.ToString()));
      return e;
    }
  }
}
```

Листинг 35.37. XmlAssemblyExtension.cs

```
public class XmlAssemblyExtension : XmlPartExtension
{
  private Assembly assembly;

  public XmlAssemblyExtension(Assembly assembly)
  {
    this.assembly = assembly;
  }

  public override XmlElement XmlElement
  {
    get
    {
      XmlElement e = NewElement("Assembly");
      e.AppendChild(NewTextElement(
        "PartNumber", assembly.PartNumber));
      e.AppendChild(NewTextElement(
        "Description", assembly.Description));

      XmlElement parts = NewElement("Parts");
      foreach(Part part in assembly.Parts)
      {
        XmlPartExtension xpe =
          part.GetExtension("XML")
          as XmlPartExtension;
        parts.AppendChild(xpe.XmlElement);
      }
      e.AppendChild(parts);
```

```
        return e;
      }
    }
  }
```

Листинг 35.38. CsvPartExtension.cs

```
public interface CsvPartExtension : PartExtension
{
  string CsvText { get; }
}
```

Листинг 35.39. CsvPiecePartExtension.cs

```
public class CsvPiecePartExtension : CsvPartExtension
{
  private PiecePart piecePart;

  public CsvPiecePartExtension(PiecePart part)
  {
    piecePart = part;
  }

  public string CsvText
  {
    get
    {
      StringBuilder b =
        new StringBuilder("PiecePart,");
      b.Append(piecePart.PartNumber);
      b.Append(",");
      b.Append(piecePart.Description);
      b.Append(",");
      b.Append(piecePart.Cost);
      return b.ToString();
    }
  }
}
```

Листинг 35.40. CsvAssemblyExtension.cs

```
public class CsvAssemblyExtension : CsvPartExtension
{
  private Assembly assembly;

  public CsvAssemblyExtension(Assembly assy)
  {
    assembly = assy;
  }

  public string CsvText
  {
    get
```

```
    {
      StringBuilder b =
        new StringBuilder("Assembly,");
      b.Append(assembly.PartNumber);
      b.Append(",");
      b.Append(assembly.Description);

      foreach(Part part in assembly.Parts)
      {
        CsvPartExtension cpe =
          part.GetExtension("CSV")
          as CsvPartExtension;
        b.Append(«,{«);
        b.Append(cpe.CsvText);
        b.Append(«}»);
      }
      return b.ToString();
    }
  }
}
```

Листинг 35.41. BadPartExtension.cs

```
public class BadPartExtension : PartExtension
{
}
```

Обратите внимание, что объекты расширения попадают в каждый объект спецификации через его конструктор. Это означает, что объекты спецификации в какой-то степени все-таки зависят от классов, относящихся к генерации XML- и CSV-представлений. Но если даже такую слабенькую зависимость необходимо разорвать, то можно создать Фабрику[1], которая будет порождать объекты спецификации и загружать в них расширения.

Тот факт, что объекты расширения можно загружать в основной объект, значительно повышает гибкость решения. В зависимости от состояния системы те или иные объекты расширения можно вставлять или удалять. Впрочем, такая гибкость может завести слишком далеко. Как правило, в большинстве случаев это не требуется. На самом деле первоначальная реализация метода `PiecePart.GetExtention(String extensionType)` выглядела примерно так:

```
public PartExtension GetExtension(String extensionType)
{
  if (extensionType.Equals("XML"))
    return new XmlPiecePartExtension(this);

  else if (extensionType.Equals("CSV"))
    return new XmlAssemblyExtension(this);
```

[1] См. главу 29.

```
    return new BadPartExtension();
  }
```

Меня это не особенно обрадовало, поскольку код практически идентичен тому, что включен в `Assembly.GetExtension`. Решение на основе хэш-таблицы `Hashtable` в классе `Part` позволяет избежать такого дублирования, и оно вообще проще. Любой, кто просмотрит его код, будет точно знать, как обращаться к объектам расширения.

Заключение

Семейство паттернов Посетитель предлагает несколько способов модифицировать поведение иерархии классов, не меняя сами эти классы. Тем самым они помогают не отступать от принципа открытости/закрытости. Кроме того, они дают механизмы разделения различных видов функциональности, позволяя не загромождать классы чрезмерно большим количеством разнородных функций. И следовательно, находятся в полном согласии с принципом общей закрытости. Разумеется, к структуре этого семейства применимы также принципы LSP и DIP.

Паттерны Посетитель очень заманчивы. Но главное – не увлекаться. Применяйте их когда это нужно, но не теряйте здоровый скептицизм. Зачастую задачу, решаемую с помощью Посетителя, можно выполнить и более простыми способами.[1]

Библиография

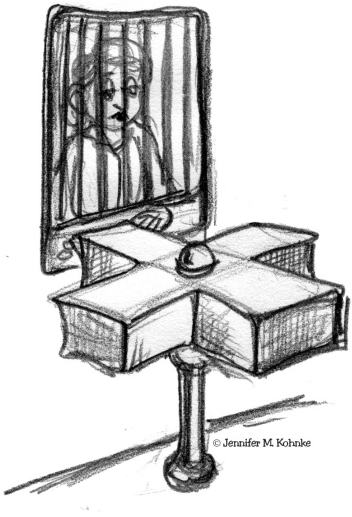
[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

[PLOPD3] Robert C. Martin, Dirk Riehle, and Frank Buschmann, eds. «Pattern Languages of Program Design 3», Addison-Wesley, 1998.

[1] Прочитав эту главу, можете вернуться к главе 9 и решить задачу об упорядочении геометрических фигур.

36

Состояние



То государство не имеет средств к сохранению,
которое не имеет средств к изменению.

Эдмунд Берк (1729 –1797)

Конечные автоматы (КА) – одна из наиболее полезных абстракций в арсенале программиста, которая находит чуть ли не универсальное применение. Это простой и элегантный способ изучения и определения поведения сложных систем, а также действенная стратегия реализации, которую легко понять и модифицировать. Я применял их для самых разных целей – от управления высокоуровневым ГИП до построения коммуникационных протоколов низкого уровня.

Нотацию и основные операции КА мы рассматривали в главе 15. А теперь обратимся к паттернам реализации этого механизма. Взгляните еще раз на схему работы турникета в метро, изображенную на рис. 36.1.

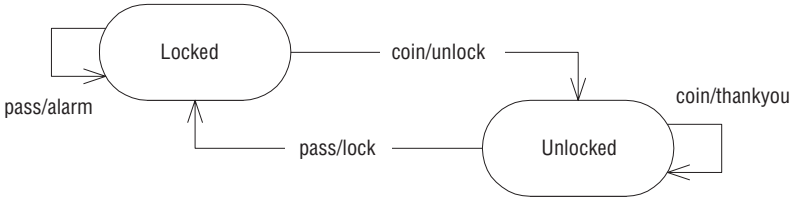


Рис. 36.1. *Конечный автомат турникета, включающий аномальные события*

Вложенные предложения switch/case

Существует несколько стратегий реализации КА. Первая – самая прямолинейная – использование вложенных предложений `switch/case`. Одна такая реализация показана в листинге 36.1.

Листинг 36.1. Turnstile.cs (реализация с использованием вложенных switch/case)

```
public enum State {LOCKED, UNLOCKED};
public enum Event {COIN, PASS};

public class Turnstile
{
  // Должно быть закрытым
  internal State state = State.LOCKED;
  private TurnstileController turnstileController;

  public Turnstile(TurnstileController action)
  {
    turnstileController = action;
  }

  public void HandleEvent(Event e)
  {
    switch (state)
    {
      case State.LOCKED:
        switch (e)
        {
          case Event.COIN:
            state = State.UNLOCKED;
            turnstileController.Unlock();
            break;
          case Event.PASS:
            turnstileController.Alarm();
```

```
            break;
          }
          break;
        case State.UNLOCKED:
          switch (e)
          {
            case Event.COIN:
              turnstileController.Thankyou();
              break;
            case Event.PASS:
              state = State.LOCKED;
              turnstileController.Lock();
              break;
          }
          break;
      }
    }
  }
```

Вложенное предложение `switch/case` разбивает код метода на четыре взаимно исключающих зоны, соответствующих переходам из одного состояния в другое. В каждой зоне производится необходимое изменение состояния, а затем вызывается ассоциированное с переходом действие. Так, в зоне «`Locked` и `Coin`» состояние изменяется на `Unlocked` и вызывается `Unlock`.

В этом фрагменте есть ряд интересных аспектов, никак не связанных с вложенным предложением `switch/case`. Чтобы осознать их, нужно посмотреть на автономный тест, с помощью которого я проверял этот код (см. листинги 36.2 и 36.3).

Листинг 36.2. TurnstileController.cs

```
public interface TurnstileController
{
  void Lock();
  void Unlock();
  void Thankyou();
  void Alarm();
}
```

Листинг 36.3. TurnstileTest.cs

```
[TestFixture]
public class TurnstileTest
{
  private Turnstile turnstile;
  private TurnstileControllerSpoof controllerSpoof;

  private class TurnstileControllerSpoof : TurnstileController
  {
    public bool lockCalled = false;
    public bool unlockCalled = false;
```

```
  public bool thankyouCalled = false;
  public bool alarmCalled = false;
  public void Lock(){lockCalled = true;}
  public void Unlock(){unlockCalled = true;}
  public void Thankyou(){thankyouCalled = true;}
  public void Alarm(){alarmCalled = true;}
}

[SetUp]
public void SetUp()
{
  controllerSpoof = new TurnstileControllerSpoof();
  turnstile = new Turnstile(controllerSpoof);
}

[Test]
public void InitialConditions()
{
  Assert.AreEqual(State.LOCKED, turnstile.state);
}

[Test]
public void CoinInLockedState()
{
  turnstile.state = State.LOCKED;
  turnstile.HandleEvent(Event.COIN);
  Assert.AreEqual(State.UNLOCKED, turnstile.state);
  Assert.IsTrue(controllerSpoof.unlockCalled);
}

[Test]
public void CoinInUnlockedState()
{
  turnstile.state = State.UNLOCKED;
  turnstile.HandleEvent(Event.COIN);
  Assert.AreEqual(State.UNLOCKED, turnstile.state);
  Assert.IsTrue(controllerSpoof.thankyouCalled);
}

[Test]
public void PassInLockedState()
{
  turnstile.state = State.LOCKED;
  turnstile.HandleEvent(Event.PASS);
  Assert.AreEqual(State.LOCKED, turnstile.state);
  Assert.IsTrue(controllerSpoof.alarmCalled);
}

[Test]
public void PassInUnlockedState()
{
```

```
        turnstile.state = State.UNLOCKED;
        turnstile.HandleEvent(Event.PASS);
        Assert.AreEqual(State.LOCKED, turnstile.state);
        Assert.IsTrue(controllerSpoof.lockCalled);
      }
    }
```

Поле State с внутренним доступом

Обратите внимание на функции `CoinInLockedState`, `CoinInUnlockedState`, `PassInLockedState` и `PassInUnlockedState`. Они независимо тестируют четыре перехода в КА, присваивая полю `state` объекта `Turnstile` то состояние, которое хотят проверять, а затем вызывая проверяемое событие. Поскольку тест обращается к полю `state`, оно не может быть закрытым. Поэтому я сделал его внутренним (модификатор доступа `internal`) и включил комментарий, говорящий о том, что, по идее, оно должно быть закрытым.

Догмат объектно-ориентированного программирования настаивает на том, чтобы все переменные экземпляров были закрытыми. Я же откровенно проигнорировал это правило и тем самым нарушил инкапсуляцию класса `Turnstile`.

Или не нарушил? Давайте уточним: я предпочел бы сделать поле `state` закрытым. Однако при этом я лишился бы возможности установить в тесте его значение. Можно было бы ввести свойство `CurrentState` с акцессорами `get` и `set` и модификатором доступа `internal`, но это как-то нелепо. Я же не собираюсь раскрывать поле `state` какому-нибудь классу, кроме `TestTurnstile`, так зачем тогда создавать свойство, предполагающее, что любой класс в пределах сборки может читать и изменять это поле?

Тестирование действий

Обратите внимание на интерфейс `TurnstileController` в листинге 36.2. Он создан специально для того, чтобы класс `TestTurnstile` мог гарантировать, что вызываются нужные методы класса `Turnstile` в должном порядке. Без этого интерфейса удостовериться в правильности работы конечного автомата было бы гораздо сложнее.

Это пример того, как тестирование оказывает влияние на дизайн. Если бы я просто писал конечный автомат, не задумываясь о том, как его тестировать, то вряд ли создал бы интерфейс `TurnstileController`. И это было бы печально. Интерфейс `TurnstileController` прекрасно отделяет логику КА от действий, который он должен выполнять. Тот же интерфейс вполне мог бы использовать другой КА с совершенно иной логикой.

Потребность в тесте, который мог бы автономно проверить все операции, заставила нас разделить код способом, о котором мы иначе даже не помыслили бы. Удобство тестирования – это стимул, благодаря которому дизайн становится менее связанным.

Достоинства и недостатки

Для простых конечных автоматов реализация с помощью вложенных предложений switch/case оказывается изящной и эффективной. Все состояния и события располагаются на одной-двух страницах кода. Однако если КА побольше, то ситуация меняется. Когда количество состояний и событий исчисляется десятками, программа разрастается и занимает несколько страниц. Не существует ориентиров, позволяющих понять, к какому месту автомата относится фрагмент кода, на который мы смотрим. Сопровождение длинных вложенных предложений switch/case затруднительно и чревато ошибками.

Еще один недостаток вложенных switch/case состоит в том, что не существует четкого разделения между логикой КА и кодом, реализующим действия. В листинге 36.1 такое разделение наглядно просматривается, потому что действия реализованы в классе, производном от Turnstile-Controller. Однако в большинстве реализаций КА на базе switch/case, с которыми мне приходилось сталкиваться, действия упрятаны в глубине предложений case. Собственно, и в листинге 36.1 такое возможно.

Таблицы переходов

Стандартный прием реализации КА заключается в том, чтобы создать таблицу с описанием переходов. Она интерпретируется ядром, обрабатывающим события. Ядро ищет в таблице переход, соответствующий событию, вызывает ассоциированное с ним действие и изменяет состояние. В листинге 36.4 приведен код создания таблицы переходов, а в листинге 36.5 – ядро системы. И то и другое – фрагменты полной реализации (листинг 36.6).

Листинг 36.4. Построение таблицы переходов состояний турникета

```
public Turnstile(TurnstileController controller)
{
  Action unlock = new Action(controller.Unlock);
  Action alarm = new Action(controller.Alarm);
  Action thankYou = new Action(controller.Thankyou);
  Action lockAction = new Action(controller.Lock);

  AddTransition(
    State.LOCKED, Event.COIN, State.UNLOCKED, unlock);
  AddTransition(
    State.LOCKED, Event.PASS, State.LOCKED, alarm);
  AddTransition(
    State.UNLOCKED, Event.COIN, State.UNLOCKED, thankYou);
  AddTransition(
    State.UNLOCKED, Event.PASS, State.LOCKED, lockAction);
}
```

Листинг 36.5. Ядро конечного автомата

```
public void HandleEvent(Event e)
{
  foreach(Transition transition in transitions)
  {
    if(state == transition.startState &&
      e == transition.trigger)
    {
      state = transition.endState;
      transition.action();
    }
  }
}
```

Интерпретация таблицы

В листинге 36.6 приведена полная реализация конечного автомата с помощью поиска в списке структур данных, описывающих переходы. Этот код полностью совместим с классами `TurnstileController` **(листинг 36.2) и** `TurnstileTest` **(листинг 36.3).**

Листинг 36.6. Полная реализация Turnstile.cs

```
public enum State {LOCKED, UNLOCKED};
public enum Event {COIN, PASS};

public class Turnstile
{
  // Должно быть закрытым
  internal State state = State.LOCKED;
  private IList transitions = new ArrayList();
  private delegate void Action();

  public Turnstile(TurnstileController controller)
  {
    Action unlock = new Action(controller.Unlock);
    Action alarm = new Action(controller.Alarm);
    Action thankYou = new Action(controller.Thankyou);
    Action lockAction = new Action(controller.Lock);

    AddTransition(
      State.LOCKED, Event.COIN, State.UNLOCKED, unlock);
    AddTransition(
      State.LOCKED, Event.PASS, State.LOCKED, alarm);
    AddTransition(
      State.UNLOCKED, Event.COIN, State.UNLOCKED, thankYou);
    AddTransition(
      State.UNLOCKED, Event.PASS, State.LOCKED, lockAction);
  }
```

```
    public void HandleEvent(Event e)
    {
      foreach(Transition transition in transitions)
      {
        if(state == transition.startState &&
          e == transition.trigger)
        {
          state = transition.endState;
          transition.action();
        }
      }
    }

    private void AddTransition(State start, Event e, State end,
      Action action)
    {
      transitions.Add(new Transition(start, e, end, action));
    }

    private class Transition
    {
      public State startState;
      public Event trigger;
      public State endState;
      public Action action;

      public Transition(State start, Event e, State end,Action a)
      {
        this.startState = start;
        this.trigger = e;
        this.endState = end;
        this.action = a;
      }
    }
  }
```

Достоинства и недостатки

Одно несомненное достоинство состоит в том, что код построения таблицы читается как каноническая таблица переходов состояний. Четыре строки `AddTransition` не вызывают никаких вопросов. Вся логика конечного автомата находится в одном месте и не замусорена реализацией действий.

Поддерживать такой КА гораздо проще, чем реализацию на основании вложенных `switch/case`. Если понадобится новое состояние, достаточно будет добавить еще одну строку `AddTransition` в конструктор класса `Turnstile`.

Еще одно достоинство заключается в том, что таблицу очень легко изменить во время выполнения. Следовательно, мы можем динамически

изменять логику конечного автомата. Я часто применял подобный прием для изменения КА «на лету».

Наконец, при таком подходе можно создавать несколько таблиц, каждая из которых реализует КА со своей логикой. Нужная таблица выбирается на этапе выполнения в зависимости от начальных условий.

Недостатком этого решения является в первую очередь низкая производительность. На поиск в таблице переходов уходит время. Если конечный автомат очень велик, то время поиска может оказаться заметным.

Паттерн Состояние

Еще одну технику реализации КА предлагает паттерн Состояние (State)[1]. Он сочетает эффективность вложенных `switch/case` с гибкостью таблиц переходов.

Структура решения представлена на рис. 36.2. В классе `Turnstile` имеются открытые методы для событий и защищенные методы для действий. Здесь же хранится ссылка на интерфейс `TurnstileState`. Два производных от `TurnstileState` класса представляют два состояния КА.

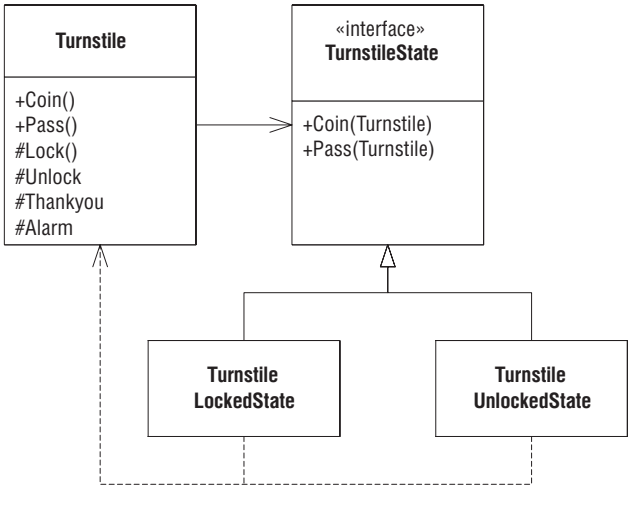


Рис. 36.2. *Паттерн Состояние для класса Turnstile*

При вызове любого из двух методов `Turnstile`, представляющих события, это событие передается экземпляру подкласса `TurnstileState`. Методы класса `TurnstileLockedState` реализуют действия в состоянии `Locked`, а методы класса `TurnstileUnlockedState` – в состоянии `Unlocked`. Чтобы изменить состояние КА, достаточно записать в объект `Turnstile` ссылку на нужный подкласс `TurnstileState`.

[1] [GOF95], стр. 305

В листинге 36.7 приведен интерфейс TurnstileState и два производных от него класса. Структура конечного автомата наглядно описывается четырьмя методами этих классов. Например, метод Coin класса LockedTurnstileState говорит, что состояние объекта Turnstile изменяется на Unlocked (Закрыт), после чего вызывается метод Unlock класса Turnstile.

Листинг 36.7. Turnstile.cs

```
public interface TurnstileState
{
  void Coin(Turnstile t);
  void Pass(Turnstile t);
}

internal class LockedTurnstileState : TurnstileState
{
  public void Coin(Turnstile t)
  {
    t.SetUnlocked();
    t.Unlock();
  }

  public void Pass(Turnstile t)
  {
    t.Alarm();
  }
}

internal class UnlockedTurnstileState : TurnstileState
{
  public void Coin(Turnstile t)
  {
    t.Thankyou();
  }

  public void Pass(Turnstile t)
  {
    t.SetLocked();
    t.Lock() ;
  }
}
```

Класс Turnstile показан в листинге 36.8. Обратите внимание на статические переменные, в которых хранятся ссылки на классы, производные от TurnstileState. В этих классах нет никаких переменных-членов, поэтому создавать больше чем по одному экземпляру каждого не имеет смысла. Хранение ссылок на экземпляры подклассов TurnstileState в переменных-членах позволяет не создавать новый экземпляр при каждом изменении состояния. А поскольку эти переменные статические, то нам не нужно создавать отдельные экземпляры подклассов в случае когда требуется более одного экземпляра Turnstile.

Листинг 36.8. Turnstile.cs

```
public class Turnstile
{
  internal static TurnstileState lockedState =
    new LockedTurnstileState();
  internal static TurnstileState unlockedState =
    new UnlockedTurnstileState();
  private TurnstileController turnstileController;
  internal TurnstileState state = unlockedState;

  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 bool IsLocked()
  {
    return state == lockedState;
  }

  public bool IsUnlocked()
  {
    return state == unlockedState;
  }

  internal void Thankyou()
  {
    turnstileController.Thankyou();
  }

  internal void Alarm()
```

```
    {
      turnstileController.Alarm();
    }

    internal void Lock()
    {
      turnstileController.Lock();
    }

    internal void Unlock()
    {
      turnstileController.Unlock();
    }
  }
```

Паттерны Состояние и Стратегия

Рисунок 36.2 очень сильно напоминает паттерн Стратегия.[1] В обоих случаях есть класс, представляющий контекст, а вся работа делегируется полиморфному базовому классу, у которого есть несколько производных. Различие (см. рис. 36.3) состоит в том, что в паттерне Состояние производный класс хранит обратную ссылку на контекстный. Основное назначение производных классов в том, чтобы выбрать метод контекстного класса и вызвать его через эту ссылку. В паттерне Стратегия такого намерения и, соответственно, ограничения нет. Производные классы не обязаны хранить ссылку на контекстный класс и вызывать методы последнего. Поэтому любое решение, удовлетворяющее паттерну Состояние, одновременно удовлетворяет и паттерну Стратегия, но обратное неверно.

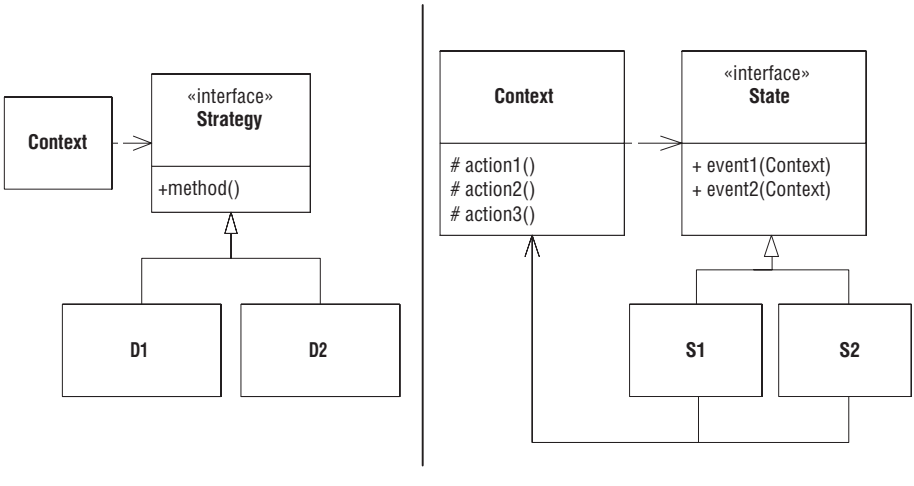


Рис. 36.3. *Состояние и Стратегия*

[1] См. главу 22.

Достоинства и недостатки

Паттерн Состояние обеспечивает строгое разделение действий и логики конечного автомата. Действия реализуются в классе `Context`, а логика распределена между классами, производными от `State`. Поэтому очень просто изменить одно, не затрагивая другого. Например, было бы несложно повторно использовать действия из класса `Context` в автомате с другой логикой переходов: достаточно заменить набор классов, производных от `State`. Наоборот, можно было бы создать подклассы `Context`, где некоторые действия переопределены, и это никак не повлияло бы на логику, реализованную в подклассах `State`.

Дополнительное достоинство этого подхода – высокая эффективность. Пожалуй, он не менее эффективен, чем вложенные предложения `switch/case`. Таким образом, мы имеем гибкость табличного решения в сочетании с эффективностью вложенных `switch/case`.

Недостатков два. Во-первых, программировать подклассы `State`, мягко говоря, утомительно; от конечного автомата с 20 состояниями можно просто с ума сойти. Во-вторых, налицо распределенность логики – нет такого единого места, где она была бы видна целиком. Поэтому код трудно сопровождать. Это сродни непрозрачности подхода на основе вложенных `switch/case`.

Компилятор конечных автоматов (SMC)

Трудоемкость написания производных от `State` классов и желание иметь одно место, в котором сосредоточена логика конечного автомата, побудили меня создать компилятор SMC, описанный в главе 15. Данные, подаваемые на вход компилятора, показаны в листинге 36.9. Синтаксис такой:

```
currentState
{
  event newState action
  ...
}
```

Четыре строки в начале листинга 36.9 задают имя конечного автомата, имя контекстного класса, начальное состояние и имя исключения, возбуждаемого в случае недопустимого события.

Листинг 36.9. 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 воспользоваться компилятором, вы должны написать класс, в котором объявлены методы действий. Имя этого класса задано в строке `Context`. Я назвал его `TurnstileActions`. См. листинг 36.10.

Листинг 36.10. TurnstileActions.cs

```
public abstract class TurnstileActions
{
  public virtual void Lock() {}
  public virtual void Unlock() {}
  public virtual void Thankyou() {}
  public virtual void Alarm() {}
}
```

Компилятор генерирует класс, производный от контекстного. Его имя задано в строке `FSMName`. Я назвал его `Turnstile`. Можно было бы реализовать методы действий прямо в классе `TurnstileActions`, но я предпочитаю написать другой класс, производный от сгенерированного, и реализовать действия уже в нем.

Листинг 36.11. TurnstileFSM.cs

```
public class TurnstileFSM : Turnstile
{
  private readonly TurnstileController controller;

  public TurnstileFSM(TurnstileController controller)
  {
    this.controller = controller;
  }

  public override void Lock()
  {
    controller.Lock();
  }

  public override void Unlock()
  {
    controller.Unlock();
  }

  public override void Thankyou()
  {
    controller.Thankyou();
```

```
    }

    public override void Alarm()
    {
      controller.Alarm();
    }
  }
```

Больше нам ничего писать не нужно, все остальное берет на себя SMC. Получившаяся структура изображена на рис. 36.4. Она называется трехуровневым КА.[1]

Три уровня обеспечивают максимальную гибкость при очень низких затратах. Мы можем породить много разных КА, просто создавая новые классы, производные от `TurnstileActions`. А если нужны другие действия, достаточно унаследовать класс `Turnstile` и переопределить его методы.

Отметим, что сгенерированный код полностью изолирован от кода, который пишете вы сами. Модифицировать сгенерированный код никогда не придется; на него даже смотреть необязательно. Он заслуживает не больше внимания, чем какой-нибудь двоичный код.

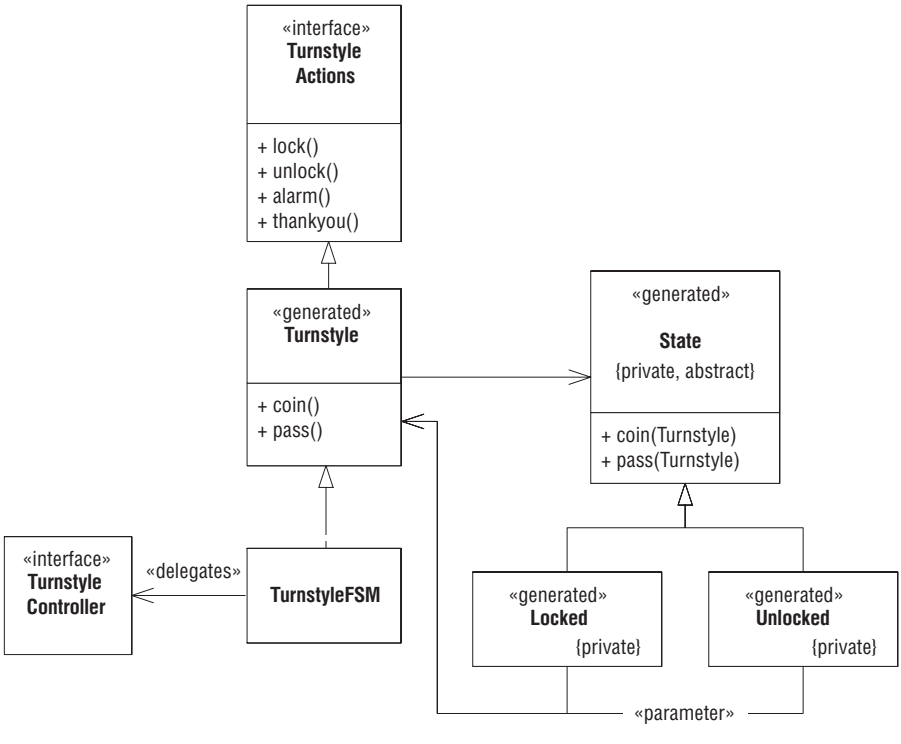


Рис. 36.4. *Трехуровневый КА*

[1] [PLOPD1], стр. 383

Файл Turnstile.cs, сгенерированный SMC, и другие вспомогательные файлы

В листингах 36.12–36.14 приведен полный код для задачи о турникете. Файл Turnstile.cs сгенерирован компилятором SMC. Кое-какие рудименты машинной генерации присутствуют, но в целом код не так уж плох.

Листинг 36.12. Turnstile.cs

```
//---------------------------------------------
//
// FSM: Turnstile
// Context: TurnstileActions
// Exception: FSMError
// Version:
// Generated: Monday 07/18/2005 at 20:57:53 CDT
//
//---------------------------------------------

//---------------------------------------------
//
// class Turnstile
// This is the Finite State Machine class
//
public class Turnstile : TurnstileActions
{
  private State itsState;
  private static string itsVersion = "";

  // instance variables for each state
  private Unlocked itsUnlockedState;
  private Locked itsLockedState;

  // constructor
  public Turnstile()
  {
    itsUnlockedState = new Unlocked();
    itsLockedState = new Locked();
    itsState = itsLockedState;
    // Entry functions for: Locked
  }

  // accessor functions

  public string GetVersion()
  {
    return itsVersion;
  }
  public string GetCurrentStateName()
  {
```

```
    return itsState.StateName();
  }
  public State GetCurrentState()
  {
    return itsState;
  }
  public State GetItsUnlockedState()
  {
    return itsUnlockedState;
  }
  public State GetItsLockedState()
  {
    return itsLockedState;
  }

  // Mutator functions

  public void SetState(State value)
  {
    itsState = value;
  }

  // event functions - forward to the current State

  public void Pass()
  {
    itsState.Pass(this);
  }
  public void Coin()
  {
    itsState.Coin(this);
  }
}

//-----------------------------------------
//
// public class State
// This is the base State class
//
public abstract class State
{
  public abstract string StateName();

  // default event functions

  public virtual void Pass(Turnstile name)
  {
    throw new FSMError( "Pass", name.GetCurrentState());
  }
  public virtual void Coin(Turnstile name)
```

```
    {
      throw new FSMError( "Coin", name.GetCurrentState());
    }
  }

  //-----------------------------------------
  //
  // class Unlocked
  // handles the Unlocked State and its events
  //
  public class Unlocked : State
  {
    public override string StateName()
      { return "Unlocked"; }
    //
    // responds to Coin event
    //
    public override void Coin(Turnstile name)
    {
      name.Thankyou();

      // change the state
      name.SetState(name.GetItsUnlockedState());
    }

    //
    // responds to Pass event
    //
    public override void Pass(Turnstile name)
    {
      name.Lock();

      // change the state
      name.SetState(name.GetItsLockedState());
    }
  }

  //-----------------------------------------
  //
  // class Locked
  // handles the Locked State and its events
  //
  public class Locked : State
  {
    public override string StateName()
      { return "Locked"; }

    //
    // responds to Coin event
    //
```

```
    public override void Coin(Turnstile name)
    {
      name.Unlock();

      // change the state
      name.SetState(name.GetItsUnlockedState());
    }

    //
    // responds to Pass event
    //
    public override void Pass(Turnstile name)
    {
      name.Alarm();

      // change the state
      name.SetState(name.GetItsLockedState());
    }
  }
```

`FSMError` — это класс исключения, которое SMC должен возбуждать в случае недопустимого события. Задача о турникете настолько проста, что недопустимых событий просто не может быть, так что этот класс бесполезен. Но в более сложных конечных автоматах бывают события, которые не должны возникать в определенных состояниях. Соответствующие переходы не упомянуты во входных данных для SMC. Если все же такое событие произойдет, то сгенерированный код возбудит исключение.

Листинг 36.13. FSMError.cs

```
public class FSMError : ApplicationException
{
  private static string message =
    "Undefined transition from state: {0} with event: {1}.";

  public FSMError(string theEvent, State state)
    : base(string.Format(message, state.StateName(), theEvent))
  {}
}
```

Код для тестирования сгенерированного SMC конечного автомата очень похож на все прочие тестовые программы в этой главе. Различия несущественны.

Листинг 36.14

```
[TestFixture]
public class SMCTurnstileTest
{
  private Turnstile turnstile;
```

```
private TurnstileControllerSpoof controllerSpoof;

private class TurnstileControllerSpoof : TurnstileController
{
  public bool lockCalled = false;
  public bool unlockCalled = false;
  public bool thankyouCalled = false;
  public bool alarmCalled = false;
  public void Lock(){lockCalled = true;}
  public void Unlock(){unlockCalled = true;}
  public void Thankyou(){thankyouCalled = true;}
  public void Alarm(){alarmCalled = true;}
  }

[SetUp]
public void SetUp()
{
  controllerSpoof = new TurnstileControllerSpoof();
  turnstile = new TurnstileFSM(controllerSpoof);
}

[Test]
public void InitialConditions()
{
  Assert.IsTrue(turnstile.GetCurrentState() is Locked);
}

[Test]
public void CoinInLockedState()
{
  turnstile.SetState(new Locked());
  turnstile.Coin();
  Assert.IsTrue(turnstile.GetCurrentState() is Unlocked);
  Assert.IsTrue(controllerSpoof.unlockCalled);
}

[Test]
public void CoinInUnlockedState()
{
  turnstile.SetState(new Unlocked());
  turnstile.Coin();
  Assert.IsTrue(turnstile.GetCurrentState() is Unlocked);
  Assert.IsTrue(controllerSpoof.thankyouCalled);
}

[Test]
public void PassInLockedState()
{
  turnstile.SetState(new Locked());
  turnstile.Pass();
```

```
    Assert.IsTrue(turnstile.GetCurrentState() is Locked);
    Assert.IsTrue(controllerSpoof.alarmCalled);
  }

  [Test]
  public void PassInUnlockedState()
  {
    turnstile.SetState(new Unlocked());
    turnstile.Pass();
    Assert.IsTrue(turnstile.GetCurrentState() is Locked);
    Assert.IsTrue(controllerSpoof.lockCalled);
  }
}
```

Класс `TurnstileController` ничем не отличается от одноименных классов, представленных в этой главе ранее. С его кодом можно ознакомиться в листинге 36.2.

Ниже приведена команда для запуска компилятора SMC. Обратите внимание, что SMC написан на языке Java. Но это не мешает ему генерировать код на C# (вдобавок к Java и C++):

```
java -classpath .\smc.jar smc.Smc -g smc.generator.csharp.SMCSharpGenerator
turnstileFSM.sm
```

Достоинства и недостатки

Очевидно, что нам удалось взять лучшее из различных подходов. Описание КА находится в одном месте, и сопровождать его очень удобно. Логика КА полностью отделена от реализации действий, что позволяет изменять одно, не затрагивая другого. Решение эффективно, элегантно и требует от нас минимального кодирования.

Недостаток – само применение SMC. Необходимо развернуть и освоить еще один инструмент. Впрочем, этот конкретный инструмент установить и использовать совсем просто, к тому же он бесплатен.

Виды приложений конечных автоматов

Я использую конечные автоматы и компилятор SMC в нескольких классах приложений.

Высокоуровневые политики приложения и ГИП

Одной из задач «графической революции», свершившейся в 1980-х годах, было создание человеко-машинных интерфейсов *без состояния*. В то время в программах применялся главным образом интерфейс на базе текстовых иерархических меню. Запутаться в структуре таких меню и перестать понимать, в каком *состоянии* находится экран, было очень просто. Графические интерфейсы пользователя (ГИП) сгладили эту проблему, доведя количество состояний окна программы до мини-

мума. При разработке современных ГИП много внимания уделяется тому, чтобы наиболее употребительные функции постоянно находились на экране, а также исключению скрытых состояний, сбивающих пользователя с толку.

По иронии судьбы программы, реализующие такие ГИП «без состояния», сами как раз изобилуют различными состояниями. Программа должна определять, какие пункты меню и кнопки сделать недоступными, какие подокна показать, какие вкладки активировать, куда поместить фокус ввода и т. д. Все это – решения о состоянии интерфейса.

Я уже давно понял, что управлять этими факторами, не имея какой-то единой структуры, – сплошной кошмар. Такую управляющую структуру проще всего выразить в виде конечного автомата. И с тех пор я почти все ГИП пишу с помощью КА, сгенерированных компилятором SMC или его предшественниками.

Рассмотрим конечный автомат в листинге 36.15. Он управляет графическим интерфейсом той части приложения, которая отвечает за вход в систему. Получив начальное событие, автомат выводит окно входа в систему. После того как пользователь нажмет клавишу Enter, автомат проверяет пароль. Если пароль правилен, то автомат переходит в состояние `loggedIn` и запускает пользовательский процесс (не показан). В противном случае выводится окно с сообщением об ошибке ввода пароля. Пользователь может повторить попытку, нажав кнопку OK, или отказаться от повторных попыток с помощью кнопки Отмена. Если три раза подряд введен неверный пароль (событие `thirdBadPassword`), то автомат блокирует экран, пока не будет введен пароль администратора.

Листинг 36.15. 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 высокоуровневая политика приложения представлена в виде конечного автомата. Она описана в одном месте, что удобно для сопровождения. Прочие части системы заметно упрощаются, потому что не перемешаны с кодом этой политики.

Очевидно, что такой подход применим не только к графическим интерфейсам. Я использовал его также при построении текстовых и машинно-машинных интерфейсов. Но поскольку ГИП обычно сложнее прочих интерфейсов, то и необходимость в таком решении более насущна.

Контроллеры взаимодействия с ГИП

Предположим, что вы хотите предоставить пользователю средства для рисования прямоугольников на экране. Допускаются следующие действия. Пользователь щелкает по иконке прямоугольника в палитре инструментов, помещает указатель мыши в то место на холсте, где должен находиться один из углов прямоугольника, нажимает кнопку мыши и перетаскивает указатель в направлении противоположного угла. В процессе перетаскивания на экране рисуется анимированный контур будущего прямоугольника. Пользователь изменяет его форму, продолжая перетаскивать указатель мыши с нажатой кнопкой. Когда прямоугольник примет желаемую форму, пользователь отпускает кнопку мыши. Программа прекращает анимацию и рисует окончательный прямоугольник.

Разумеется, пользователь в любой момент может отменить операцию, щелкнув по другому значку в палитре. Если указатель мыши в процессе перетаскивания выходит за пределы холста, то анимированный контур пропадает. Когда указатель возвращается в окно холста, контур снова появляется.

Наконец, закончив рисовать один прямоугольник, пользователь может нарисовать следующий, просто щелкнув мышью в окне холста и начав

перетаскивание. Снова щелкать по иконке прямоугольника в палитре необязательно.

Все вышеописанное – не что иное как конечный автомат. Диаграмма переходов его состояний изображена на рис. 36.5. Закрашенный кружок перед стрелкой обозначает начальное состояние КА.[1] Закрашенный кружок, обведенный окружностью, обозначает конечное состояние КА.

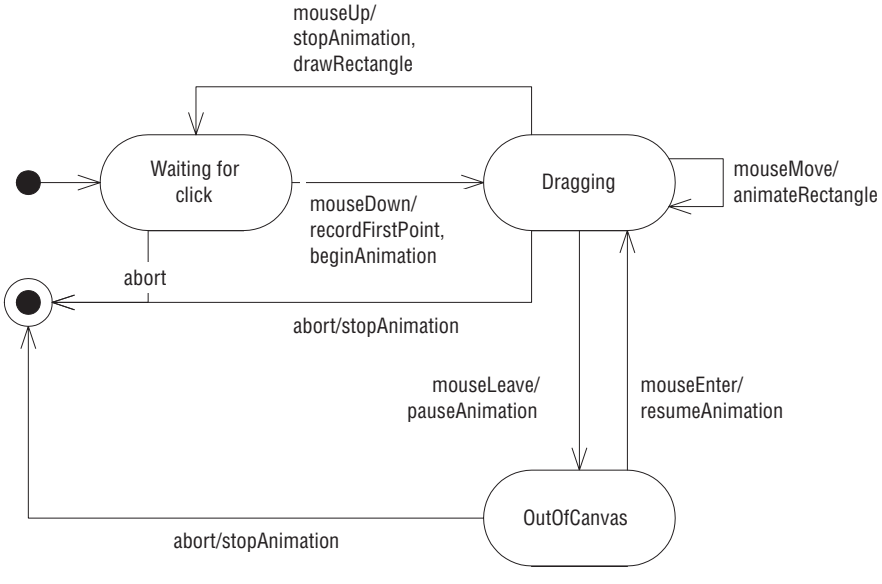


***Рис. 36.5.** Конечный автомат взаимодействия с прямоугольником*

Описание взаимодействий с ГИП – рай для конечных автоматов. Они управляются событиями, возникающими в результате действий пользователя. События вызывают изменения в состоянии взаимодействий.

Распределенная обработка

Распределенная обработка – еще одна ситуация, в которой состояние системы изменяется в зависимости от входных событий. Пусть, например, требуется передать большой блок информации из одного узла сети в другой. Предположим также, что время реакции сети – важный параметр, поэтому вы разбиваете блок на мелкие пакеты и передаете их группой.

На рис. 36.6 изображен конечный автомат, описывающий этот сценарий. Он начинает работу с запроса сеанса передачи, затем передает один

[1] См. раздел «Диаграммы состояний» в главе 13.

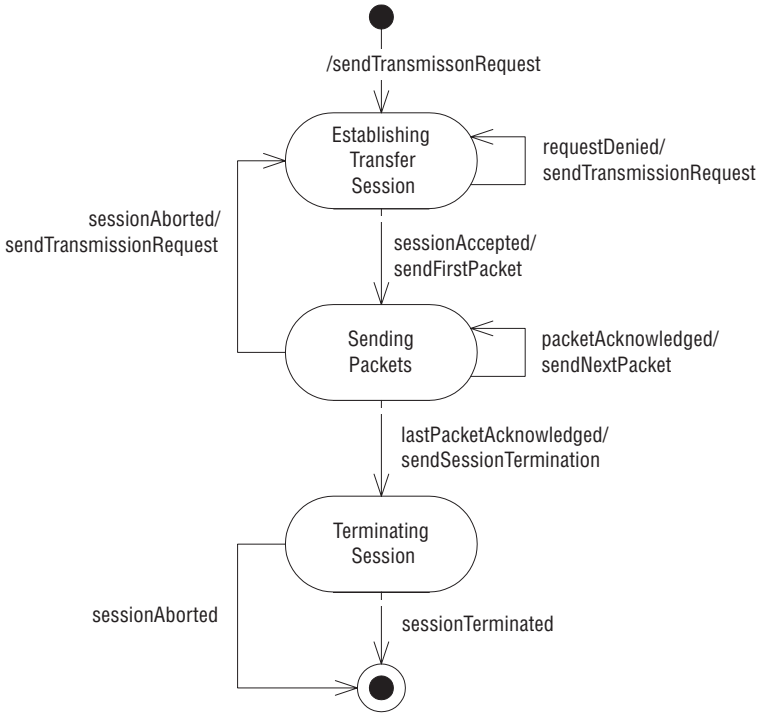


Рис. 36.6. *Отправка большого блока в виде группы мелких пакетов*

пакет за другим, каждый раз ожидая подтверждения, и заканчивает завершением сеанса.

Заключение

Конечные автоматы используются явно недостаточно. Во многих случаях они позволяют создать более понятный, простой, гибкий и аккуратный код. Значительным подспорьем в этом деле могут стать паттерн Состояние и простые инструменты генерации кода по таблице переходов состояний.

Библиография

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

[PLOPD1] James O. Coplien and Douglas C. Schmidt «Pattern Languages of Program Design», Addison-Wesley, 1995.

37

Система расчета заработной платы: база данных



Часто у экспертов данных больше, чем здравого смысла.

Колин Пауэлл

В предыдущих главах мы реализовали всю бизнес-логику системы расчета зарплаты. В состав реализации входил и класс `PayrollDatabase`, предназначенный для хранения необходимых данных в памяти. В тот момент его было вполне достаточно. Но ясно, что системе необходимо какое-то более долговременное хранилище данных. В этой главе объясняется, как обеспечить сохранение данных в реляционной базе.

Построение базы данных

Выбор технологии баз данных обычно диктуется скорее политическими, нежели техническими причинами. Компании-производители баз данных и платформ немало постарались, чтобы убедить рынок в критической важности такого выбора. Лояльность и преданность конкретному поставщику формируется по причинам, больше относящимся к человеческим отношениям, чем к технической целесообразности. Поэтому не следует придавать слишком большого значения нашему выбору Microsoft SQL Server в качестве СУБД для хранения данных приложения.

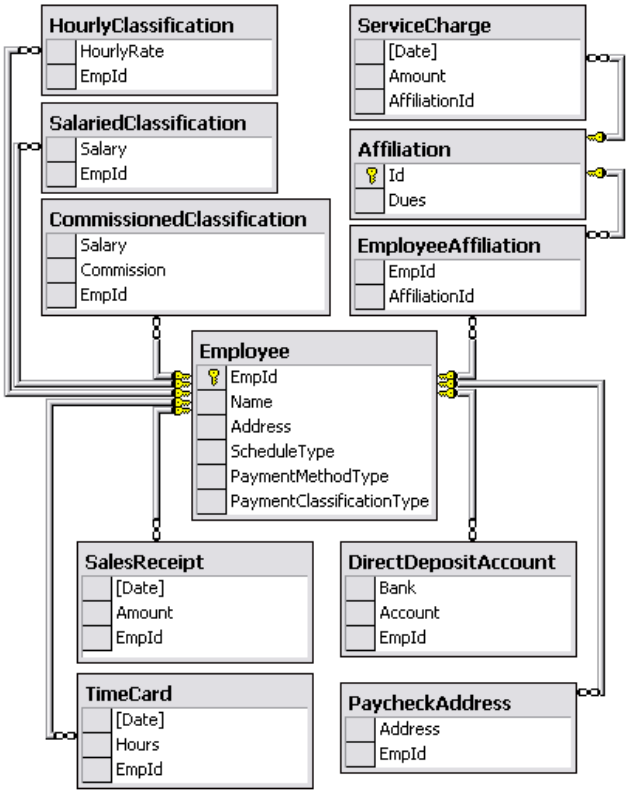


Рис. 37.1. *Схема базы данных для системы расчета зарплаты*

На рис. 37.1 показана схема базы данных, которой мы будем далее пользоваться. В центре находится таблица `Employee`. Здесь хранятся данные о работниках, а также строковые константы, определяющие график выплат (`ScheduleType`), метод платежа (`PaymentMethodType`) и тарификацию (`PaymentClassification`). Сами данные, относящиеся к тарификации,

хранятся в таблицах HourlyClassification, SalariedClassification и CommissionedClassification. Ссылка на соответствующую запись таблицы Employee находится в столбце EmpId. Для этого столбца определено ограничение, гарантирующее, что в таблице Employee существует запись с данным значением EmpId. В таблицах DirectDepositAccount и PaycheckAddress хранятся данные, относящиеся к соответствующему методу (перечисление на банковский счет или отправка чека почтой); на столбец EmpId наложено аналогичное ограничение. Таблицы SalesReceipt (Справки о продажах) и TimeCard (Карточки табельного учета) не нуждаются в пояснениях. В таблице Affiliation хранятся данные о членстве в профсоюзе, она связана с таблицей Employee через таблицу EmpoyeeAffiliation.

Изъян в дизайне программы

Возможно, вы еще не забыли, что в классе PayrollDatabase были только открытые статические методы. Как теперь выясняется, это решение не годится. Как приступить к работе с реальной базой данных, не испортив все тесты, в которых использовались статические методы? Мы не хотим замещать класс PayrollDatabase классом, обращающимся к реальной базе. Если так поступить, то все ранее написанные автономные тесты станут обращаться к реальной базе данных. Хорошо бы сделать PayrollDatabase интерфейсом, чтобы можно было легко подставлять альтернативные реализации. Одна реализация хранила бы данные в памяти, чтобы выполнение тестов занимало немного времени. А другая работала бы с данными в реальной базе.

Для этого нам придется прибегнуть к рефакторингу, выполняя после каждого шага автономные тесты, чтобы случайно не испортить код. Первым делом создадим экземпляр класса PayrollDatabase и сохраним его в статической переменной instance. Затем переименуем все статические методы в PayrollDatabase, добавив в имя слово Static. А тело каждого метода перенесем в новый не-статический метод с прежним именем (см. листинг 37.1).

Листинг 37.1. Пример рефакторинга

```
public class PayrollDatabase
{
  private static PayrollDatabase instance;

  public static void AddEmployee_Static(Employee employee)
  {
    instance.AddEmployee(employee);
  }

  public void AddEmployee(Employee employee)
  {
    employees[employee.EmpId] = employee;
  }
```

Теперь нужно найти все обращения к PayrollDatabase.AddEmployee_Static() и заменить их на PayrollDatabase.instance.AddEmployee(). Сделав это, мы сможем удалить статическую версию метода. То же самое следует проделать для каждого статического метода.

Таким образом, все обращения к базе данных производятся посредством переменной PayrollDatabase.instance. Мы хотели бы превратить PayrollDatabase в интерфейс. Поэтому нужно подыскать для этой переменной другого кладельца. Конечно, поместить ее следует в класс PayrollTest, потому что тогда ее можно будет использовать во всех тестах. Что касается приложения, то самым подходящим методом являются все классы, производные от Transaction. Экземпляр PayrollDatabase нужно будет передавать конструктору каждого такого класса и сохранять в переменной экземпляра. Но вместо того чтобы дублировать этот код, давайте просто поместим экземпляр PayrollDatabase в базовый класс Transaction. Правда, сейчас Transaction – интерфейс, поэтому нужно преобразовать его в абстрактный класс, как показано в листинге 37.2.

Листинг 37.2. Transaction.cs

```
public abstract class Transaction
{
  protected readonly PayrollDatabase database;

  public Transaction(PayrollDatabase database)
  {
    this.database = database;
  }

  public abstract void Execute();
}
```

Теперь никто не использует переменную PayrollDatabase.instance, так что ее можно удалить. Но прежде, чем преобразовать PayrollDatabase в интерфейс, нам понадобится новая реализация, которая наследует PayrollDatabase. Поскольку текущая реализация хранит все в памяти, назовем новый класс InMemoryPayrollDatabase (листинг 37.3) и будем использовать его там, где создавался экземпляр PayrollDatabase. Вот теперь PayrollDatabase можно превратить в интерфейс (листинг 37.4) и начать работу над реализацией доступа к настоящей базе данных.

Листинг 37.3. InMemoryPayrollDatabase.cs

```
public class InMemoryPayrollDatabase : PayrollDatabase
{
  private static Hashtable employees = new Hashtable();
  private static Hashtable unionMembers = new Hashtable();

  public void AddEmployee(Employee employee)
  {
```

```
      employees[employee.EmpId] = employee;
    }

    // и т. д...
  }
```

Листинг 37.4. PayrollDatabase.cs

```
public interface PayrollDatabase
{
  void AddEmployee(Employee employee);
  Employee GetEmployee(int id);
  void DeleteEmployee(int id);
  void AddUnionMember(int id, Employee e);
  Employee GetUnionMember(int id);
  void RemoveUnionMember(int memberId);
  ArrayList GetAllEmployeeIds();
}
```

Добавление работника

Переработав дизайн, мы можем создать класс `SqlPayrollDatabase`. Он реализует интерфейс `PayrollDatabase` таким образом, что данные сохраняются в базе данных SQL Server в соответствии со схемой на рис. 37.1. Одновременно с `SqlPayrollDatabase` мы создадим класс `SqlPayrollDatabaseTest` для автономного тестирования. В листинге 37.5 приведен первый тест.

Листинг 37.5. SqlPayrollDatabaseTest.cs

```
[TestFixture]
public class Blah
{
  private SqlPayrollDatabase database;

  [SetUp]
  public void SetUp()
  {
    database = new SqlPayrollDatabase();
  }

  [Test]
  public void AddEmployee()
  {
    Employee employee = new Employee(123,
      "George", "123 Baker St.");
    employee.Schedule = new MonthlySchedule();
    employee.Method =
      new DirectDepositMethod("Bank 1", "123890");
    employee.Classification =
      new SalariedClassification(1000.00);
```

```
    database.AddEmployee(123, employee);

    SqlConnection connection = new SqlConnection(
      "Initial Catalog=Payroll;Data Source=localhost;" +
      "user id=sa;password=abc");
    SqlCommand command = new SqlCommand(
      "select * from Employee", connection);
    SqlDataAdapter adapter = new SqlDataAdapter(command);
    DataSet dataset = new DataSet();
    adapter.Fill(dataset);

    DataTable table = dataset.Tables["table"];
    Assert.AreEqual(1, table.Rows.Count);
    DataRow row = table.Rows[0];
    Assert.AreEqual(123, row["EmpId"]);
    Assert.AreEqual("George", row["Name"]);
    Assert.AreEqual("123 Baker St.", row["Address"]);
  }
}
```

Здесь мы вызываем метод `AddEmployee()`, а затем обращаемся к базе данных и проверяем, что данные действительно сохранены. В листинге 37.6 показан прямолинейный код, с которым этот тест успешно проходит.

Листинг 37.6. SqlPayrollDatabase.cs

```
public class SqlPayrollDatabase : PayrollDatabase
{
  private readonly SqlConnection connection;

  public SqlPayrollDatabase()
  {
    connection = new SqlConnection(
      "Initial Catalog=Payroll;Data Source=localhost;" +
      "user id=sa;password=abc");
    connection.Open();
  }

  public void AddEmployee(Employee employee)
  {
    string sql = "insert into Employee values (" +
      "@EmpId, @Name, @Address, @ScheduleType, " +
      "@PaymentMethodType, @PaymentClassificationType)";
    SqlCommand command = new SqlCommand(sql, connection);

    command.Parameters.Add("@EmpId", employee.EmpId);
    command.Parameters.Add("@Name", employee.Name);
    command.Parameters.Add("@Address", employee.Address);
    command.Parameters.Add("@ScheduleType",
      employee.Schedule.GetType().ToString());
    command.Parameters.Add("@PaymentMethodType",
```

```
        employee.Method.GetType().ToString());
      command.Parameters.Add("@PaymentClassificationType",
        employee.Classification.GetType().ToString());
      command.ExecuteNonQuery();
    }
  }
```

Но проходит он только в первый раз, а во второй завершается неудачно. Мы получаем от SQL Server исключение с сообщением о попытке вставить дубликат ключа. Поэтому перед каждым выполнением теста таблицу `Employee` следует очищать. В листинге 37.7 показано, как добавить эту операцию в метод `SetUp`.

Листинг 37.7. SqlPayrollDatabaseTest.SetUp()

```
[SetUp]
public void SetUp()
{
  database = new SqlPayrollDatabase();
  SqlConnection connection = new SqlConnection(
    "Initial Catalog=Payroll;Data Source=localhost;" +
    "user id=sa;password=abc");connection.Open();
  SqlCommand command = new SqlCommand(
    "delete from Employee", connection);
  command.ExecuteNonQuery();
  connection.Close();
}
```

Так все работает, но код получился неряшливым. Мы создаем соединение в методе `SetUp`, а потом еще раз в тесте `AddEmployee`. Должно быть достаточно одного соединения, которое создается в `SetUp` и закрывается в `TearDown`. В листинге 37.8 показана исправленная версия.

Листинг 37.8. SqlPayrollDatabaseTest.cs

```
[TestFixture]
public class Blah
{
  private SqlPayrollDatabase database;
  private SqlConnection connection;

  [SetUp]
  public void SetUp()
  {
    database = new SqlPayrollDatabase();
    connection = new SqlConnection(
      "Initial Catalog=Payroll;Data Source=localhost;" +
      "user id=sa;password=abc");
    connection.Open();
    new SqlCommand("delete from Employee",
      this.connection).ExecuteNonQuery();
  }
```

```
  [TearDown]
  public void TearDown()
  {
    connection.Close();
  }

  [Test]
  public void AddEmployee()
  {
    Employee employee = new Employee(123,
      "George", "123 Baker St.");
    employee.Schedule = new MonthlySchedule();
    employee.Method =
      new DirectDepositMethod("Bank 1", "123890");
    employee.Classification =
      new SalariedClassification(1000.00);
    database.AddEmployee(employee);

  SqlCommand command = new SqlCommand(
      "select * from Employee", connection);
    SqlDataAdapter adapter = new SqlDataAdapter(command);
    DataSet dataset = new DataSet();
    adapter.Fill(dataset);
    DataTable table = dataset.Tables["table"];

    Assert.AreEqual(1, table.Rows.Count);
    DataRow row = table.Rows[0];
    Assert.AreEqual(123, row["EmpId"]);
    Assert.AreEqual("George", row["Name"]);
    Assert.AreEqual("123 Baker St.", row["Address"]);
  }
}
```

В листинге 37.6 видно, что в столбцы `ScheduleType`, `PaymentMethodType` **и** `PaymentClassificationType` **таблицы** `Employee` **записываются имена классов. Хотя это и работает, но имена чересчур длинные. Лучше было бы взять короткие ключевые слова. Начнем с графика выплат: в листинге 37.9 показано, как сохраняются данные в таблице** `MonthlySchedules`**. А в листинге 37.10 приведена часть класса** `SqlPayrollDatabase`**, отвечающая за успешное завершение этого теста.**

Листинг 37.9. SqlPayrollDatabaseTest.ScheduleGetsSaved()

```
[Test]
public void ScheduleGetsSaved()
{
  Employee employee = new Employee(123,
    "George", "123 Baker St.");
  employee.Schedule = new MonthlySchedule();
  employee.Method = new DirectDepositMethod();
  employee.Classification = new SalariedClassification(1000.00);
```

```
    database.AddEmployee(123, employee);

    SqlCommand command = new SqlCommand(
      "select * from Employee", connection);
    SqlDataAdapter adapter = new SqlDataAdapter(command);
    DataSet dataset = new DataSet();
    adapter.Fill(dataset);
    DataTable table = dataset.Tables["table"];

    Assert.AreEqual(1, table.Rows.Count);
    DataRow row = table.Rows[0];
    Assert.AreEqual("monthly", row["ScheduleType"]);
  }
```

Листинг 37.10. SqlPayrollDatabase.cs (фрагмент)

```
  public void AddEmployee(int id, Employee employee)
  {
    ...
    command.Parameters.Add("@ScheduleType",
      ScheduleCode(employee.Schedule));
    ...
  }

  private static string ScheduleCode(PaymentSchedule schedule)
  {
    if(schedule is MonthlySchedule)
      return "monthly";
    else
      return "unknown";
  }
```

Внимательный читатель наверняка обратил внимание на то, что в листинге 37.10 мы уже почти нарушили принцип OCP. Метод `ScheduleCode()` содержит предложение `if/else`, проверяющее, что график выплат `schedule` имеет тип `MonthlySchedule`. А вскоре мы добавим еще одну ветвь для сравнения с типом `WeeklySchedule` и другую — для `BiweeklySchedule`. И тогда каждый раз при добавлении нового вида графика платежа нам придется модифицировать цепочку `if/else`.

Возможная альтернатива — получать код графика из иерархии `PaymentSchedule`. Можно было бы добавить полиморфное свойство, например `string DatabaseCode`, которое возвращает нужное значение. Но тогда иерархия `PaymentSchedule` будет нарушать принцип SRP.

И это нарушение SRP отвратительно. Оно создает ненужную связь между базой данных и приложением и предлагает другим модулям эту связь всячески укрепить, пользуясь свойством `ScheduleCode`. С другой стороны, нарушение принципа OCP инкапсулировано в классе `SqlPayrollDatabase` и наружу вряд ли выйдет. Так что пока поживем с нарушением OCP.

При написании следующего теста у нас будет масса возможностей удалить дублирующийся код. В листинге 37.11 показан класс `SqlPayrollDatabaseTest` **после небольшого рефакторинга и добавления нового теста. В листинге 37.12 показаны изменения, внесенные в класс** `SqlPayrollDatabase` **для того, чтобы этот тест прошел.**

Листинг 37.11. SqlPayrollDatabaseTest.cs (фрагмент)

```
[SetUp]
public void SetUp()
{
  ...
  CleanEmployeeTable();
  employee = new Employee(123, "George", "123 Baker St.");
  employee.Schedule = new MonthlySchedule();
  employee.Method = new DirectDepositMethod();
  employee.Classification= new SalariedClassification(1000.00);
}

private void ClearEmployeeTable()
{
  new SqlCommand("delete from Employee",
    this.connection).ExecuteNonQuery();
}

private DataTable LoadEmployeeTable()
{
  SqlCommand command = new SqlCommand(
    "select * from Employee", connection);
  SqlDataAdapter adapter = new SqlDataAdapter(command);
  DataSet dataset = new DataSet();
  adapter.Fill(dataset);
  return dataset.Tables["table"];
}

[Test]
public void ScheduleGetsSaved()
{
  CheckSavedScheduleCode(new MonthlySchedule(), "monthly");
  ClearEmployeeTable();
  CheckSavedScheduleCode(new WeeklySchedule(), "weekly");
  ClearEmployeeTable();
  CheckSavedScheduleCode(new BiWeeklySchedule(), "biweekly");
}

private void CheckSavedScheduleCode(
  PaymentSchedule schedule, string expectedCode)
{
  employee.Schedule = schedule;
  database.AddEmployee(123, employee);
  DataTable table = LoadEmployeeTable();
```

```
    DataRow row = table.Rows[0];
    Assert.AreEqual(expectedCode, row["ScheduleType"]);
  }
```

Листинг 37.12. SqlPayrollDatabase.cs (фрагмент)

```
  private static string ScheduleCode(PaymentSchedule schedule)
  {
    if(schedule is MonthlySchedule)
      return "monthly";
    if(schedule is WeeklySchedule)
      return "weekly";
    if(schedule is BiWeeklySchedule)
      return "biweekly";
    else
      return "unknown";
  }
```

В листинге 37.13 приведен новый тест для сохранения деталей метода платежа. Он написан по тому же образцу, что и тест сохранения графика. В листинге 37.14 приведен код, добавленный в класс базы данных.

Листинг 37.13. SqlPayrollDatabaseTest.cs (фрагмент)

```
  [Test]
  public void PaymentMethodGetsSaved()
  {
    CheckSavedPaymentMethodCode(new HoldMethod(), "hold");
    ClearEmployeeTable();
    CheckSavedPaymentMethodCode(
      new DirectDepositMethod("Bank -1", "0987654321"),
      "directdeposit");
    ClearEmployeeTable();
    CheckSavedPaymentMethodCode(
      new MailMethod("111 Maple Ct."), "mail");
  }

  private void CheckSavedPaymentMethodCode(
    PaymentMethod method, string expectedCode)
  {
    employee.Method = method;
    database.AddEmployee(employee);
    DataTable table = LoadTable("Employee");
    DataRow row = table.Rows[0];
    Assert.AreEqual(expectedCode, row["PaymentMethodType"]);
  }
```

Листинг 37.14. SqlPayrollDatabase.cs (фрагмент)

```
  public void AddEmployee(int id, Employee employee)
  {
    ...
    command.Parameters.Add("@PaymentMethodType",
      PaymentMethodCode(employee.Method));
```

```
    ...
  }

  private static string PaymentMethodCode(PaymentMethod method)
  {
    if(method is HoldMethod)
      return "hold";
    if(method is DirectDepositMethod)
      return "directdeposit";
    if(method is MailMethod)
      return "mail";
    else
      return "unknown";
  }
```

Все тесты проходят. Однако постойте: в классах `DirectDepositMethod` и `MailMethod` есть собственные данные, которые тоже нужно сохранить. При сохранении в таблице `Employee` любого из этих методов платежа необходимо заполнять также таблицы `DirectDepositAccount` и `PaycheckAddress`. В листинге 37.15 приведен тест для проверки сохранения объекта `DirectDepositMethod`.

Листинг 37.15. SqlPayrollDatabaseTest.cs (фрагмент)

```
[Test]
public void DirectDepositMethodGetsSaved()
{
  CheckSavedPaymentMethodCode(
    new DirectDepositMethod("Bank -1", "0987654321"),
    "directdeposit");

  SqlCommand command = new SqlCommand(
    "select * from DirectDepositAccount", connection);
  SqlDataAdapter adapter = new SqlDataAdapter(command);
  DataSet dataset = new DataSet();
  adapter.Fill(dataset);
  DataTable table = dataset.Tables["table"];

  Assert.AreEqual(1, table.Rows.Count);
  DataRow row = table.Rows[0];
  Assert.AreEqual("Bank -1", row["Bank"]);
  Assert.AreEqual("0987654321", row["Account"]);
  Assert.AreEqual(123, row["EmpId"]);
}
```

Глядя на этот код и думая, как сделать так, чтобы тест прошел, мы понимаем, что необходимо еще одно предложение `if/else`. Первое понадобилось для того, чтобы понять, какое значение записывать в столбец `PaymentMethodType`, – и это уже достаточно плохо. А второе нужно, чтобы определить таблицу, в которую записываются данные. Нарушения принципа OCP начинают громоздиться одно на другое. Нам срoч-

но требуется найти решение, в котором было бы только одно предложение `if/else`. Оно показано в листинге 37.16, где добавлено несколько переменных-членов, которые нас выручат.

Листинг 37.16. SqlPayrollDatabase.cs (фрагмент)

```
public void AddEmployee(int id, Employee employee)
{
  string sql = "insert into Employee values (" +
    "@EmpId, @Name, @Address, @ScheduleType, " +
    "@PaymentMethodType, @PaymentClassificationType)";
  SqlCommand command = new SqlCommand(sql, connection);
  command.Parameters.Add("@EmpId", id);
  command.Parameters.Add("@Name", employee.Name);
  command.Parameters.Add("@Address", employee.Address);
  command.Parameters.Add("@ScheduleType",
  ScheduleCode(employee.Schedule));
  SavePaymentMethod(employee);
  command.Parameters.Add("@PaymentMethodType", methodCode);
  command.Parameters.Add("@PaymentClassificationType",
  employee.Classification.GetType().ToString());
  command.ExecuteNonQuery();
}

private void SavePaymentMethod(Employee employee)
{
  PaymentMethod method = employee.Method;
  if(method is HoldMethod)
    methodCode = "hold";
  if(method is DirectDepositMethod)
  {
    methodCode = "directdeposit";
    DirectDepositMethod ddMethod =
      method as DirectDepositMethod;
    string sql = "insert into DirectDepositAccount" +
      "values (@Bank, @Account, @EmpId)";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@Bank", ddMethod.Bank);
    command.Parameters.Add("@Account", ddMethod.AccountNumber);
    command.Parameters.Add("@EmpId", employee.EmpId);
    command.ExecuteNonQuery();
  }
  if(method is MailMethod)
    methodCode = "mail";
  else
    methodCode = "unknown";
}
```

Опа! Тест *не проходит*! SQL Server возвращает ошибку, сообщая, что нельзя добавить запись в таблицу `DirectDepositAccount`, потому что не существует соответствующей записи в таблице `Employee`. То есть записы-

вать в таблицу DirectDepositAcount следует *после* того как в таблицу Employee добавлена запись о работнике. Но возникает интересный вопрос. Что если команда вставки работника завершится успешно, а команда вставки метода платежа – нет? Мы получим противоречивые данные – работника без метода платежа. Этого никак нельзя допустить.

Стандартное решение такой проблемы – воспользоваться *транзакциями*. Если хотя бы одна часть транзакции завершается с ошибкой, то отменяется вся транзакция и никакие данные не сохраняются. Ошибка при сохранении все равно огорчительна, но лучше не сохранить ничего, чем записать в базу данных противоречивую информацию. Однако прежде чем приступать к решению этой проблемы, позаботимся о том, чтобы прошли имеющиеся тесты. В листинге 37.17 показана очередная эволюция кода.

Листинг 37.17. SqlPayrollDatabase.cs (фрагмент)

```
public void AddEmployee(int id, Employee employee)
{
  PrepareToSavePaymentMethod(employee);

  string sql = "insert into Employee values (" +
    "@EmpId, @Name, @Address, @ScheduleType, " +
    "@PaymentMethodType, @PaymentClassificationType)";
  SqlCommand command = new SqlCommand(sql, connection);

  command.Parameters.Add("@EmpId", id);
  command.Parameters.Add("@Name", employee.Name);
  command.Parameters.Add("@Address", employee.Address);
  command.Parameters.Add("@ScheduleType",
    ScheduleCode(employee.Schedule));
  SavePaymentMethod(employee);
  command.Parameters.Add("@PaymentMethodType", methodCode);
  command.Parameters.Add("@PaymentClassificationType",
  employee.Classification.GetType().ToString());

  command.ExecuteNonQuery();

  if(insertPaymentMethodCommand != null)
    insertPaymentMethodCommand.ExecuteNonQuery();
}

private void PrepareToSavePaymentMethod(Employee employee)
{
  PaymentMethod method = employee.Method;
  if(method is HoldMethod)
    methodCode = "hold";
  else if(method is DirectDepositMethod)
  {
    methodCode = "directdeposit";
    DirectDepositMethod ddMethod =
```

```
            method as DirectDepositMethod;
        string sql = "insert into DirectDepositAccount" +
            "values (@Bank, @Account, @EmpId)";
        insertPaymentMethodCommand =
            new SqlCommand(sql, connection);
        insertPaymentMethodCommand.Parameters.Add(
          "@Bank", ddMethod.Bank);
        insertPaymentMethodCommand.Parameters.Add(
          "@Account", ddMethod.AccountNumber);
        insertPaymentMethodCommand.Parameters.Add(
          "@EmpId", employee.EmpId);
      }
      else if(method is MailMethod)
        methodCode = "mail";
      else
        methodCode = "unknown";
    }
```

Увы, тесты все равно не проходят. На этот раз СУБД ругается, когда мы очищаем таблицу `Employee`, поскольку это оставило бы в таблице `DirectDepositAccount` висящую ссылку. Следовательно, в методе `SetUp` нужно очищать обе таблицы. Я не забыл о том, что таблицу `DirectDepositAccount` нужно очищать первой, и был вознагражден зеленой полосой. Замечательно.

Еще осталось сохранить объект `MailMethod`. Покончим с этим, прежде чем перейти к транзакциям. Чтобы проверить правильность заполнения таблицы `PaycheckAddress`, нам необходимо загрузить из нее данные. Это будет уже третий случай загрузки данных из таблицы, пора заняться рефакторингом. Переименовав метод `LoadEmployeeTable` в `LoadTable` и добавив имя таблицы в качестве параметра, мы имеем возможность полюбоваться засверкавшим новыми гранями кода. В листинге 37.18 показано это изменение и новый тест. А в листинге 37.19 – код, с которым этот тест проходит, – после добавления предложения для очистки таблицы `PaycheckAddress` в методе `SetUp`.

Листинг 37.18. SqlPayrollDatabaseTest.cs (фрагмент)

```
private DataTable LoadTable(string tableName)
{
  SqlCommand command = new SqlCommand(
    "select * from " + tableName, connection);
  SqlDataAdapter adapter = new SqlDataAdapter(command);
  DataSet dataset = new DataSet();
  adapter.Fill(dataset);
  return dataset.Tables["table"];
}

[Test]
public void MailMethodGetsSaved()
{
```

```
    CheckSavedPaymentMethodCode(
      new MailMethod("111 Maple Ct."), "mail");
    DataTable table = LoadTable("PaycheckAddress");
    Assert.AreEqual(1, table.Rows.Count);
    DataRow row = table.Rows[0];
    Assert.AreEqual("111 Maple Ct.", row["Address"]);
    Assert.AreEqual(123, row["EmpId"]);
  }
```

Листинг 37.19. SqlPayrollDatabase.cs (фрагмент)

```
  private void PrepareToSavePaymentMethod(Employee employee)
  {
    ...
    else if(method is MailMethod)
    {
      methodCode = "mail";
      MailMethod mailMethod = method as MailMethod;
      string sql = "insert into PaycheckAddress " +
        "values (@Address, @EmpId)";
      insertPaymentMethodCommand =
        new SqlCommand(sql, connection);
      insertPaymentMethodCommand.Parameters.Add(
        "@Address", mailMethod.Address);
      insertPaymentMethodCommand.Parameters.Add(
        "@EmpId", employee.EmpId);
    }
    ...
  }
```

Транзакции

Теперь пришло время сделать операции с базой данных транзакционными. Выполнение транзакций SQL Server на платформе .NET не вызывает ни малейших сложностей. Вам понадобится только класс `System.Data.SqlClient.SqlTransaction`. Однако сначала нужно написать тест, который не проходит. Как убедиться, что операция базы данных действительно транзакционная?

Если удастся сделать так, чтобы первая команда транзакции прошла, а вторая завершилась неудачно, то потом можно проверить, что никакие данные не сохранены. Ну и как же этого добиться? Вспомним пример с сохранением объекта `DirectDepositMethod`. Мы знаем, что сначала сохраняются данные о работнике, а потом о перечислении на банковский счет. Если мы сможем устроить так, что вставка в таблицу `DirectDepositAccount` не пройдет, то задача будет решена. Попытка записать значение `null` в объект `DirectDepositMethod` должна привести к ошибке, потому что таблица `DirectDepositAccount` не допускает null-значений. В листинге 37.20 показано, что нужно сделать.

Листинг 37.20. SqlPayrollDatabaseTest.cs (фрагмент)

```
[Test]
public void SaveIsTransactional()
{
  // Null-значения не должны попасть в базу данных.
  DirectDepositMethod method =
    new DirectDepositMethod(null, null);
  employee.Method = method;
  try
  {
    database.AddEmployee(123, employee);
    Assert.Fail("Тут должно возникнуть исключение.");
  }
  catch(SqlException)
  {}

  DataTable table = LoadTable("Employee");
  Assert.AreEqual(0, table.Rows.Count);
}
```

И действительно, ошибка имеет место. Запись в таблицу `Employee` **добавлена, а в таблицу** `DirectDepositAccount` **– нет. Такой ситуации следует избегать. В листинге 37.21 показано, как с помощью класса** `SqlTransaction` **сделать работу с базой данных транзакционной.**

Листинг 37.21. SqlPayrollDatabase.cs (фрагмент)

```
public void AddEmployee(int id, Employee employee)
{
  SqlTransaction transaction =
    connection.BeginTransaction("Save Employee");
  try
  {
    PrepareToSavePaymentMethod(employee);
    string sql = "insert into Employee values (" +
      "@EmpId, @Name, @Address, @ScheduleType, " +
      "@PaymentMethodType, @PaymentClassificationType)";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@EmpId", id);
    command.Parameters.Add("@Name", employee.Name);
    command.Parameters.Add("@Address", employee.Address);
    command.Parameters.Add("@ScheduleType",
    ScheduleCode(employee.Schedule));
    command.Parameters.Add("@PaymentMethodType", methodCode);
    command.Parameters.Add("@PaymentClassificationType",
    employee.Classification.GetType().ToString());
    command.Transaction = transaction;
    command.ExecuteNonQuery();
    if(insertPaymentMethodCommand != null)
    {
```

```
      insertPaymentMethodCommand.Transaction = transaction;
      insertPaymentMethodCommand.ExecuteNonQuery();
    }
    transaction.Commit();
  }
  catch(Exception e)
  {
    transaction.Rollback();
    throw e;
  }
}
```

Все тесты проходят! Это оказалось совсем просто. Теперь надо почистить код. См. листинг 37.22.

Листинг 37.22. SqlPayrollDatabase.cs (фрагмент)

```
public void AddEmployee(int id, Employee employee)
{
  PrepareToSavePaymentMethod(employee);
  PrepareToSaveEmployee(employee);
  SqlTransaction transaction =
    connection.BeginTransaction("Save Employee");
  try
  {
    ExecuteCommand(insertEmployeeCommand, transaction);
    ExecuteCommand(insertPaymentMethodCommand, transaction);
    transaction.Commit();
  }
  catch(Exception e)
  {
    transaction.Rollback();
    throw e;
  }
}

private void ExecuteCommand(SqlCommand command,
  SqlTransaction transaction)
{
  if(command != null)
  {
    command.Connection = connection;
    command.Transaction = transaction;
    command.ExecuteNonQuery();
  }
}

private void PrepareToSaveEmployee(Employee employee)
{
  string sql = "insert into Employee values (" +
    "@EmpId, @Name, @Address, @ScheduleType, " +
    "@PaymentMethodType, @PaymentClassificationType)";
```

```
  insertEmployeeCommand = new SqlCommand(sql);
  insertEmployeeCommand.Parameters.Add(
    "@EmpId", employee.EmpId);
  insertEmployeeCommand.Parameters.Add(
    "@Name", employee.Name);
  insertEmployeeCommand.Parameters.Add(
    "@Address", employee.Address);
  insertEmployeeCommand.Parameters.Add(
    "@ScheduleType",ScheduleCode(employee.Schedule));
  insertEmployeeCommand.Parameters.Add(
    "@PaymentMethodType", methodCode);
  insertEmployeeCommand.Parameters.Add(
    "@PaymentClassificationType",
  employee.Classification.GetType().ToString());
}

private void PrepareToSavePaymentMethod(Employee employee)
{
  PaymentMethod method = employee.Method;
  if(method is HoldMethod)
    methodCode = "hold";
  else if(method is DirectDepositMethod)
  {
    methodCode = "directdeposit";
    DirectDepositMethod ddMethod =
    method as DirectDepositMethod;
    insertPaymentMethodCommand =
      CreateInsertDirectDepositCommand(ddMethod, employee);
  }
  else if(method is MailMethod)
  {
    methodCode = "mail";
    MailMethod mailMethod = method as MailMethod;
    insertPaymentMethodCommand =
      CreateInsertMailMethodCommand(mailMethod, employee);
  }
  else
    methodCode = "unknown";
}

private SqlCommand CreateInsertDirectDepositCommand(
  DirectDepositMethod ddMethod, Employee employee)
{
  string sql = "insert into DirectDepositAccount " +
    "values (@Bank, @Account, @EmpId)";
  SqlCommand command = new SqlCommand(sql);
  command.Parameters.Add("@Bank", ddMethod.Bank);
  command.Parameters.Add("@Account", ddMethod.AccountNumber);
  command.Parameters.Add("@EmpId", employee.EmpId);
  return command;
}
```

```
private SqlCommand CreateInsertMailMethodCommand(
  MailMethod mailMethod, Employee employee)
{
  string sql = "insert into PaycheckAddress " +
    "values (@Address, @EmpId)";
  SqlCommand command = new SqlCommand(sql);
  command.Parameters.Add("@Address", mailMethod.Address);
  command.Parameters.Add("@EmpId", employee.EmpId);
  return command;
}
```

В этот момент несохраненным остался объект `PaymentClassification`. Реализация этой части кода не несет в себе ничего нового и оставлена читателю в качестве упражнения.

Когда автор справился с последней задачей, на свет выплыл изъян в коде. Объект `SqlPayrollDatabase`, вероятно, создается на ранней стадии работы приложения и затем интенсивно используется. Памятуя об этом, взгляните на переменную-член `insertPaymentMethodCommand`. Ей присваивается значение при сохранении работника, для которого метод платежа – перечисление на банковский счет или отправка чека почтой, но не оставление у кассира. А переменная-то никогда не очищается. Что произойдет если мы сначала сохраним работника, которому чек посылается почтой, а потом другого – который сам забирает чек у кассира? В листинге 37.23 этот случай реализован в виде теста.

Листинг 37.23. SqlPayrollDatabaseTest.cs (фрагмент)

```
[Test]
public void SaveMailMethodThenHoldMethod()
{
  employee.Method = new MailMethod("123 Baker St.");
  database.AddEmployee(employee);
  Employee employee2 = new Employee(321, "Ed", "456 Elm St.");
  employee2.Method = new HoldMethod();
  database.AddEmployee(employee2);
  DataTable table = LoadTable("PaycheckAddress");
  Assert.AreEqual(1, table.Rows.Count);
}
```

Тест не проходит, потому что в таблицу `PaycheckAddress` добавлено две записи. При сохранении первого работника в переменную `insertPaymentMethodCommand` была записана команда добавления объекта `MailMethod`. А при сохранении второго эта команда так там и осталась, потому что для сохранения объекта `HoldMethod` дополнительной команды не требуется. Таким образом, команда `insertPaymentMethodCommand` была выполнена еще раз.

Исправить это упущение можно несколькими способами, но меня беспокоит еще кое-что. Мы начали с реализации метода `SqlPayrollDatabase.AddEmployee` и попутно создали целую кучу вспомогательных закрытых методов. Это здорово замусорило бедный класс `SqlPayrollDatabase`. На-

стало время ввести специальный класс для сохранения работника: SaveEmployeeOperation. Метод AddEmployee() будет создавать экземпляр SaveEmployeeOperation при каждом обращении. Таким образом, нам не придется обнулять команды и класс SqlPayrollDatabase станет гораздо чаще. Поскольку это просто рефакторинг и никакая функциональность не изменяется, то новые тесты не нужны.

Сначала я создам класс SaveEmployeeOperation и скопирую в него код сохранения работника. Мне придется добавить конструктор и новый метод Execute(), который и выполняет сохранение. Этот служебный класс показан в листинге 37.24.

Листинг 37.24. SaveEmployeeOperation.cs (фрагмент)

```
public class SaveEmployeeOperation
{
  private readonly Employee employee;
  private readonly SqlConnection connection;
  private string methodCode;
  private string classificationCode;
  private SqlCommand insertPaymentMethodCommand;
  private SqlCommand insertEmployeeCommand;
  private SqlCommand insertClassificationCommand;

  public SaveEmployeeOperation(
    Employee employee, SqlConnection connection)
  {
    this.employee = employee;
    this.connection = connection;
  }

  public void Execute()
  {
    /*
    Весь код, необходимый для сохранения Employee
    */
}
```

Затем я изменю метод SqlPayrollDatabase.AddEmplyee() так, чтобы он создавал объект SaveEmployeeOperation и вызывал его метод Execute (листинг 37.25). Все тесты проходят, включая и SaveMailMethodThenHoldMethod. После удаления скопированного кода класс SqlPayrollDatabase становится гораздо чище.

Листинг 37.25. SqlPayrollDatabase.AddEmployee()

```
public void AddEmployee(Employee employee)
{
  SaveEmployeeOperation operation =
    new SaveEmployeeOperation(employee, connection);
  operation.Execute();
}
```

Выборка работника

Настало время убедиться, что мы можем загрузить объекты `Employee` из базы данных. В листинге 37.26 приведен первый тест. Как видите, при его написании я не нарушил никаких принципов. Сначала с помощью уже протестированного метода `SqlPayrollDatabase.AddEmployee()` сохраняется объект `Employee`. Затем мы пытаемся загрузить этот объект методом `SqlPayrollDatabase.GetEmployee()`. Проверяются все аспекты загруженного объекта, включая график выплат, метод платежа и тарификацию. Понятно, что при первом выполнении тест не проходит и, чтобы добиться результата, нам предстоит еще много работы.

Листинг 37.26. SqlPayrollDatabaseTest.cs (фрагмент)

```
public void LoadEmployee()
{
  employee.Schedule = new BiWeeklySchedule();
  employee.Method =
    new DirectDepositMethod("1st Bank", "0123456");
  employee.Classification =
    new SalariedClassification(5432.10);
  database.AddEmployee(employee);

  Employee loadedEmployee = database.GetEmployee(123);
  Assert.AreEqual(123, loadedEmployee.EmpId);
  Assert.AreEqual(employee.Name, loadedEmployee.Name);
  Assert.AreEqual(employee.Address, loadedEmployee.Address);
  PaymentSchedule schedule = loadedEmployee.Schedule;
  Assert.IsTrue(schedule is BiWeeklySchedule);

  PaymentMethod method = loadedEmployee.Method;
  Assert.IsTrue(method is DirectDepositMethod);
  DirectDepositMethod ddMethod = method as DirectDepositMethod;
  Assert.AreEqual("1st Bank", ddMethod.Bank);
  Assert.AreEqual("0123456", ddMethod.AccountNumber);

  PaymentClassification classification =
  loadedEmployee.Classification;
  Assert.IsTrue(classification is SalariedClassification);
  SalariedClassification salariedClassification =
  classification as SalariedClassification;
  Assert.AreEqual(5432.10, salariedClassification.Salary);
}
```

Во время последнего рефакторинга метода `AddEmployee()` мы выделили класс `SaveEmployeeOperation`, у которого есть только одна задача: сохранять работника. Такой же прием мы применим для реализации кода загрузки работника. И, разумеется, первым делом напишем тест. Однако имеется одно принципиальное различие. При тестировании загрузки работника мы не будем обращаться к базе, как в предыдущем тесте.

Мы тщательно проверим саму возможность загрузить объект Employee, но обойдемся без соединения с базой данных.

В листинге 37.27 приведено начало теста LoadEmployeeOperationTest. Первый тест, LoadEmployeeDataCommand, создает новый объект LoadEmployeeOperation, передавая конструктору идентификатор работника и null вместо соединения с базой данных. Затем тест получает объект SqlCommand для загрузки данных из таблицы Employee и проверяет структуру команды. Мы могли бы выполнить ее, но что это даст? Во-первых, тест усложнится, потому что перед выполнением запроса надо будет поместить в базу данные. Во-вторых, мы и так уже тестируем возможность соединиться с базой в методе SqlPayrollDatabaseTest.LoadEmployee(). Нет смысла тестировать одно и то же снова и снова. В листинге 37.28 приведено начало метода LoadEmployeeOperation вместе с кодом, необходимым для успешного завершения первого теста.

Листинг 37.27. LoadEmployeeOperationTest.cs

```
using System.Data;
using System.Data.SqlClient;
using NUnit.Framework;
using Payroll;

namespace PayrollDB
{
  [TestFixture]
  public class LoadEmployeeOperationTest
  {
    private LoadEmployeeOperation operation;
    private Employee employee;

    [SetUp]
    public void SetUp()
    {
      employee = new Employee(123, "Jean", "10 Rue de Roi");
      operation = new LoadEmployeeOperation(123, null);
      operation.Employee = employee;
    }

    [Test]
    public void LoadingEmployeeDataCommand()
    {
      operation = new LoadEmployeeOperation(123, null);
      SqlCommand command = operation.LoadEmployeeCommand;
      Assert.AreEqual("select * from Employee " +
        "where EmpId=@EmpId", command.CommandText);
      Assert.AreEqual(123, command.Parameters["@EmpId"].Value);
    }
  }
}
```

Листинг 37.28. LoadEmployeeOperation.cs

```
using System.Data.SqlClient;
using Payroll;

namespace PayrollDB
{
  public class LoadEmployeeOperation
  {
    private readonly int empId;
    private readonly SqlConnection connection;
    private Employee employee;

    public LoadEmployeeOperation(
      int empId, SqlConnection connection)
    {
      this.empId = empId;
      this.connection = connection;
    }

    public SqlCommand LoadEmployeeCommand
    {
      get
      {
        string sql = "select * from Employee " +
          "where EmpId=@EmpId";
        SqlCommand command = new SqlCommand(sql, connection);
        command.Parameters.Add("@EmpId", empId);
        return command;
      }
    }
  }
}
```

В этот момент тесты проходят, так что начало положено. Но одной лишь команды явно недостаточно; мы должны еще создать объект `Employee` по выбранным из базы данным. Один из способов получить данные из базы – загрузить их в объект `DataSet`, как мы делали в предшествующих тестах. Это удобно, потому что в тестах мы можем создать `DataSet`, который будет выглядеть в точности так же, как объект, построенный в результате настоящего запроса к базе данных. Тест в листинге 37.29 показывает, как это делается, а в листинге 37.30 приведен соответствующий код.

Листинг 37.29. LoadEmployeeOperationTest.LoadEmployeeData()

```
[Test]
public void LoadEmployeeData()
{
  DataTable table = new DataTable();
  table.Columns.Add("Name");
```

```
    table.Columns.Add("Address");
    DataRow row = table.Rows.Add(
      new object[]{"Jean", "10 Rue de Roi"});

    operation.CreateEmplyee(row);

    Assert.IsNotNull(operation.Employee);
    Assert.AreEqual(«Jean», operation.Employee.Name);
    Assert.AreEqual("10 Rue de Roi",
    operation.Employee.Address);
  }
```

Листинг 37.30. LoadEmployeeOperation.cs (фрагмент)

```
public void CreateEmplyee(DataRow row)
{
  string name = row["Name"].ToString();
  string address = row["Address"].ToString();
  employee = new Employee(empId, name, address);
}
```

Успешно завершив этот тест, мы можем перейти к загрузке графика выплат. В листингах 37.31 и 37.32 приведены тест и код, который загружает первый из подклассов `PaymentSchedule`: `WeeklySchedule`.

Листинг 37.31. LoadEmployeeOperationTest.LoadingSchedules()

```
[Test]
public void LoadingSchedules()
{
  DataTable table = new DataTable();
  table.Columns.Add("ScheduleType");
  DataRow row = table.NewRow();
  row.ItemArray = new object[] {"weekly"};

  operation.AddSchedule(row);

  Assert.IsNotNull(employee.Schedule);
  Assert.IsTrue(employee.Schedule is WeeklySchedule);
}
```

Листинг 37.32. LoadEmployeeOperation.cs (фрагмент)

```
public void AddSchedule(DataRow row)
{
  string scheduleType = row["ScheduleType"].ToString();
  if(scheduleType.Equals("weekly"))
    employee.Schedule = new WeeklySchedule();
}
```

Немного подработав этот метод, мы сможем столь же легко проверить все остальные подклассы `PaymentSchedule`. Поскольку в некоторых тестах мы уже создавали объекты `DataTable` и собираемся делать это впредь, то

имеет смысл оформить это рутинное действие в виде отдельного метода. Изменения показаны в листингах 37.33 и 37.34.

Листинг 37.33. LoadEmployeeOperationTest.LoadingSchedules() (после рефакторинга)

```
[Test]
public void LoadingSchedules()
{
  DataRow row = ShuntRow("ScheduleType", "weekly");
  operation.AddSchedule(row);
  Assert.IsTrue(employee.Schedule is WeeklySchedule);

  row = ShuntRow("ScheduleType", "biweekly");
  operation.AddSchedule(row);
  Assert.IsTrue(employee.Schedule is BiWeeklySchedule);

  row = ShuntRow("ScheduleType", "monthly");
  operation.AddSchedule(row);
  Assert.IsTrue(employee.Schedule is MonthlySchedule);
}

private static DataRow ShuntRow(
  string columns, params object[] values)
{
  DataTable table = new DataTable();
  foreach(string columnName in columns.Split(','))
    table.Columns.Add(columnName);
  return table.Rows.Add(values);
}
```

Листинг 37.34. LoadEmployeeOperation.cs (фрагмент)

```
public void AddSchedule(DataRow row)
{
  string scheduleType = row["ScheduleType"].ToString();
  if(scheduleType.Equals("weekly"))
    employee.Schedule = new WeeklySchedule();
  else if(scheduleType.Equals("biweekly"))
    employee.Schedule = new BiWeeklySchedule();
  else if(scheduleType.Equals("monthly"))
    employee.Schedule = new MonthlySchedule();
}
```

Теперь можно переходить к загрузке метода платежа. См. листинги 37.35 и 37.36.

Листинг 37.35. LoadEmployeeOperationTest.LoadingHoldMethod()

```
[Test]
public void LoadingHoldMethod()
{
  DataRow row = ShuntRow("PaymentMethodType", "hold");
```

```
    operation.AddPaymentMethod(row);
    Assert.IsTrue(employee.Method is HoldMethod);
  }
```

Листинг 37.36. LoadEmployeeOperation.cs (фрагмент)

```
  public void AddPaymentMethod(DataRow row)
  {
    string methodCode = row["PaymentMethodType"].ToString();
    if(methodCode.Equals("hold"))
      employee.Method = new HoldMethod();
  }
```

Тут ничего сложного не было. Однако загрузка прочих методов платежа уже не так проста. Обратимся к загрузке объекта `Employee` с методом платежа `DirectDepositMethod`. Сначала мы читаем данные из таблицы `Employee`. Значение `directdeposit` в столбце `PaymentMethodType` говорит, что мы должны создать для работника объект типа `DirectDepositMethod`. Но, чтобы это сделать, нам необходимы данные о банковском счете, хранящиеся в таблице `DirectDepositAccount`. Следовательно, метод `LoadEmployeeOperation.AddPaymentMethod()` должен будет создать новую sql-команду для выборки этих данных. Чтобы протестировать эту операцию, сначала нужно поместить данные в таблицу `DirectDepositAccount`.

Чтобы проверить правильность загрузки метода платежа без обращения к базе данных, нам придется создать новый класс: `LoadPaymentMethodOperation`. Он будет отвечать за определение того, какой подкласс `PaymentMethod` создавать, и за подготовку данных для его создания. В листинге 37.37 показана новая тестовая фикстура `LoadPaymentMethodOperationTest` с тестом для загрузки объектов типа `HoldMethod`. В листинге 37.38 показан первый набросок класса `LoadPaymentMethod`, а в листинге 37.39 – как этот класс используется в `LoadEmployeeOperation`.

Листинг 37.37. LoadPaymentMethodOperationTest.cs

```
using NUnit.Framework;
using Payroll;

namespace PayrollDB
{
  [TestFixture]
  public class LoadPaymentMethodOperationTest
  {
    private Employee employee;
    private LoadPaymentMethodOperation operation;

    [SetUp]
    public void SetUp()
    {
      employee = new Employee(567, "Bill", "23 Pine Ct");
    }
```

```
    [Test]
    public void LoadHoldMethod()
    {
      operation = new LoadPaymentMethodOperation(
        employee, "hold", null);
      operation.Execute();
      PaymentMethod method = this.operation.Method;
      Assert.IsTrue(method is HoldMethod);
    }
  }
}
```

Листинг 37.38. LoadPaymentMethodOperation.cs

```
using System;
using System.Data;
using System.Data.SqlClient;
using Payroll;

namespace PayrollDB
{
  public class LoadPaymentMethodOperation
  {
    private readonly Employee employee;
    private readonly string methodCode;
    private PaymentMethod method;

    public LoadPaymentMethodOperation(
      Employee employee, string methodCode)
    {
      this.employee = employee;
      this.methodCode = methodCode;
    }

    public void Execute()
    {
      if(methodCode.Equals("hold"))
        method = new HoldMethod();
    }

    public PaymentMethod Method
    {
      get { return method; }
    }
  }
}
```

Листинг 37.39. LoadEmployeeOperation.cs (фрагмент)

```
public void AddPaymentMethod(DataRow row)
{
  string methodCode = row["PaymentMethodType"].ToString();
  LoadPaymentMethodOperation operation =
```

```
      new LoadPaymentMethodOperation(employee, methodCode);
    operation.Execute();
    employee.Method = operation.Method;
  }
```

И на этот раз загрузка метода платежа `HoldMethod` **оказалась простым делом. Чтобы загрузить объект** `DirectDepositMethod`**, мы должны будем построить команду** `SqlCommand` **для выборки данных, а затем на основе этих данных сконструировать объект** `DirectDepositMethod`**. В листингах 37.40 и 37.41 показаны соответствующий тест и код. Отметим, что тест** `CreateDirectDepositMethodFromRow` **заимствует метод** `ShuntRow` **из класса** `LoadEmployeeOperationTest`**. Это удобный метод, поэтому пока оставим все, как есть. Но позже мы должны будем найти для** `ShuntRow` **более подходящее место.**

Листинг 37.40. LoadPaymentMethodOperationTest.cs (фрагмент)

```
[Test]
public void LoadDirectDepositMethodCommand()
{
  operation = new LoadPaymentMethodOperation(
    employee, "directdeposit") ;
  SqlCommand command = operation.Command;
  Assert.AreEqual("select * from DirectDepositAccount " +
    "where EmpId=@EmpId", command.CommandText);
  Assert.AreEqual(employee.EmpId,
  command.Parameters["@EmpId"].Value);
}

[Test]
public void CreateDirectDepositMethodFromRow()
{
  operation = new LoadPaymentMethodOperation(
    employee, "directdeposit");
  DataRow row = LoadEmployeeOperationTest.ShuntRow(
    "Bank,Account", "1st Bank", "0123456");
  operation.CreatePaymentMethod(row);

  PaymentMethod method = this.operation.Method;
  Assert.IsTrue(method is DirectDepositMethod);
  DirectDepositMethod ddMethod =
    method as DirectDepositMethod;
  Assert.AreEqual("1st Bank", ddMethod.Bank);
  Assert.AreEqual("0123456", ddMethod.AccountNumber);
}
```

Листинг 37.41. LoadPaymentMethodOperation.cs (фрагмент)

```
public SqlCommand Command
{
  get
  {
```

```
    string sql = "select * from DirectDepositAccount" +
      "where EmpId=@EmpId";
    SqlCommand command = new SqlCommand(sql);
    command.Parameters.Add("@EmpId", employee.EmpId);
    return command;
  }
}

public void CreatePaymentMethod(DataRow row)
{
  string bank = row["Bank"].ToString();
  string account = row["Account"].ToString();
  method = new DirectDepositMethod(bank, account);
}
```

Осталось протестировать загрузку объектов `MailMethod`. В листинге 37.42 показан тест создания SQL-запроса. При попытке написать код выясняются интересные вещи. Для присваивания значения свойству `Command` нам необходимо предложение `if/else`, в котором определяется имя таблицы, подставляемое в запрос. В методе `Execute()` нужно еще одно предложение `if/else` для определения подкласса `PaymentMethod`, которому принадлежит создаваемый объект. Где-то мы уже это встречали. И, как и раньше, дублирования предложений `if/else` лучше бы избежать.

Класс `LoadPaymentMethodOperation` следует реструктурировать так, чтобы осталось только одно предложение `if/else`. Проявив смекалку и воспользовавшись делегатами, мы решим эту проблему – см. листинг 37.43.

Листинг 37.42. LoadPaymentMethodOperationTest. LoadMailMethodCommand()

```
[Test]
public void LoadMailMethodCommand()
{
  operation = new LoadPaymentMethodOperation(employee, "mail");
  SqlCommand command = operation.Command;
  Assert.AreEqual("select * from PaycheckAddress " +
    "where EmpId=@EmpId", command.CommandText);
  Assert.AreEqual(employee.EmpId,
    command.Parameters["@EmpId"].Value);
}
```

Листинг 37.43. LoadPaymentMethodOperationTest. LoadMailMethodCommand()

```
public class LoadPaymentMethodOperation
{
  private readonly Employee employee;
  private readonly string methodCode;
  private PaymentMethod method;
  private delegate void PaymentMethodCreator(DataRow row);
  private PaymentMethodCreator paymentMethodCreator;
```

```
private string tableName;

public LoadPaymentMethodOperation(
  Employee employee, string methodCode)
{
  this.employee = employee;
  this.methodCode = methodCode;
}

public void Execute()
{
  Prepare();
  DataRow row = LoadData();
  CreatePaymentMethod(row);
}

public void CreatePaymentMethod(DataRow row)
{
  paymentMethodCreator(row);
}

public void Prepare()
{
  if(methodCode.Equals("hold"))
  paymentMethodCreator =
    new PaymentMethodCreator(CreateHoldMethod);
  else if(methodCode.Equals("directdeposit"))
  {
    tableName = "DirectDepositAccount";
    paymentMethodCreator = new PaymentMethodCreator(
      CreateDirectDepositMethod);
  }
  else if(methodCode.Equals("mail"))
  {
    tableName = "PaycheckAddress";
  }
}

private DataRow LoadData()
{
  if(tableName != null)
    return LoadEmployeeOperation.LoadDataFromCommand(Command);
  else
    return null;
}

public PaymentMethod Method
{
  get { return method; }
}
```

```
  public SqlCommand Command
  {
    get
    {
      string sql = String.Format(
        "select * from {0} where EmpId=@EmpId", tableName);
      SqlCommand command = new SqlCommand(sql);
      command.Parameters.Add("@EmpId", employee.EmpId);
      return command;
    }
  }

  public void CreateDirectDepositMethod(DataRow row)
  {
    string bank = row["Bank"].ToString();
    string account = row["Account"].ToString();
    method = new DirectDepositMethod(bank, account);
  }

  private void CreateHoldMethod(DataRow row)
  {
    method = new HoldMethod();
  }
}
```

Этот рефакторинг потребовал больше работы, чем обычно. Пришлось модифицировать тесты. Теперь из теста требуется вызывать метод `Prepare()` **перед тем, как получить команду для загрузки объекта** `PaymentMethod`**. В листинге 37.44 показано это изменение и окончательный тест для создания объекта** `MailMethod`**. А листинг 37.45 содержит остаток класса** `LoadPaymentMethodOperation`**.**

Листинг 37.44. LoadPaymentMethodOperationTest.cs (фрагмент)

```
[Test]
public void LoadMailMethodCommand()
{
  operation = new LoadPaymentMethodOperation(employee, "mail");
  operation.Prepare();
  SqlCommand command = operation.Command;
  Assert.AreEqual("select * from PaycheckAddress " +
    "where EmpId=@EmpId", command.CommandText);
  Assert.AreEqual(employee.EmpId,
    command.Parameters["@EmpId"].Value);
}

[Test]
public void CreateMailMethodFromRow()
{
  operation = new LoadPaymentMethodOperation(employee, "mail");
  operation.Prepare();
```

```
    DataRow row = LoadEmployeeOperationTest.ShuntRow(
      "Address", "23 Pine Ct");
    operation.CreatePaymentMethod(row);
    PaymentMethod method = this.operation.Method;
    Assert.IsTrue(method is MailMethod);
    MailMethod mailMethod = method as MailMethod;
    Assert.AreEqual("23 Pine Ct", mailMethod.Address);
  }
```

Листинг 37.45. LoadPaymentMethodOperation.cs (фрагмент)

```
  public void Prepare()
  {
    if(methodCode.Equals("hold"))
      paymentMethodCreator =
        new PaymentMethodCreator(CreateHoldMethod);
    else if(methodCode.Equals("directdeposit"))
    {
      tableName = "DirectDepositAccount";
      paymentMethodCreator =
        new PaymentMethodCreator(CreateDirectDepositMethod);
    }
    else if(methodCode.Equals("mail"))
    {
      tableName = "PaycheckAddress";
      paymentMethodCreator =
        new PaymentMethodCreator(CreateMailMethod);
    }
  }

  private void CreateMailMethod(DataRow row)
  {
    string address = row["Address"].ToString();
    method = new MailMethod(address);
  }
```

Итак, мы протестировали загрузку всех методов платежа и осталась только тарификация – объекты типа `PaymentClassification`. Чтобы проверить, как они загружаются, мы создадим новый класс `LoadPaymentClassificationOperation` и соответствующую тестовую фикстуру. Все это очень похоже на уже проделанное, поэтому оставлено читателю в качестве упражнения.

По завершении мы можем вернуться к тесту `SqlPayrollDatabaseTest.LoadEmployee`. Гм. Он по-прежнему не проходит. Похоже, мы что-то забыли. В листинге 37.46 показано, какие изменения следует внести, чтобы тест прошел.

Листинг 37.46. LoadEmployeeOperation.cs (фрагмент)

```
  public void Execute()
  {
```

```
    string sql = "select * from Employee where EmpId = @EmpId";
    SqlCommand command = new SqlCommand(sql, connection);
    command.Parameters.Add("@EmpId", empId);

    DataRow row = LoadDataFromCommand(command);

    CreateEmplyee(row);
    AddSchedule(row);
    AddPaymentMethod(row);
    AddClassification(row);
  }

  public void AddSchedule(DataRow row)
  {
    string scheduleType = row[«ScheduleType»].ToString();
    if(scheduleType.Equals(«weekly»))
      employee.Schedule = new WeeklySchedule();
    else if(scheduleType.Equals(«biweekly»))
      employee.Schedule = new BiWeeklySchedule();
    else if(scheduleType.Equals(«monthly»))
      employee.Schedule = new MonthlySchedule();
  }

  private void AddPaymentMethod(DataRow row)
  {
    string methodCode = row[«PaymentMethodType»].ToString();
    LoadPaymentMethodOperation operation =
      new LoadPaymentMethodOperation(employee, methodCode);
    operation.Execute();
    employee.Method = operation.Method;
  }

  private void AddClassification(DataRow row)
  {
    string classificationCode =
      row[«PaymentClassificationType»].ToString();
    LoadPaymentClassificationOperation operation =
      new LoadPaymentClassificationOperation(employee,
        classificationCode);
    operation.Execute();
    employee.Classification = operation.Classification;
  }
```

Вы, наверное, заметили, что в классах `LoadOperation` много повторяющегося кода. А кроме того, само желание назвать эту группу классов собирательным именем `LoadOperations` наводит на мысль о том, что у них должен быть общий базовый класс. Такой класс стал бы естественным местом для всего повторяющегося кода, который сейчас присутствует в его будущих подклассах. Оставляю этот рефакторинг вам.

Что осталось?

Класс `SqlPayrollDatabase` умеет сохранять новые объекты `Employee` и загружать объекты `Employee` из базы данных. Но это еще не все. Что произойдет если мы попытаемся сохранить объект `Employee`, который ранее уже был сохранен? Этот случай еще предстоит обработать. Кроме того, мы еще даже не приступали к карточкам табельного учета, справкам о продажах и членству в профсоюзе. Но после того что мы уже сделали, добавление такой функциональности не вызовет трудностей, займитесь этим самостоятельно.

38

Система расчета заработной платы: Модель–Вид–Презентатор



С точки зрения пользователя интерфейс – это продукт.

Джеф Раскин

Ну что ж, наше приложение для расчета зарплаты обретает форму. Оно поддерживает работников, получающих твердый оклад, почасовой заработок и комиссионные. Начисленная зарплата может доставляться почтой в виде чека, перечисляться на банковский счет или сохраняться у кассира до востребования. Система умеет начислять зарплату в соот-

ветствии с заданным для каждого работника графиком. Кроме того, данные, создаваемые и используемые системой, хранятся в реляционной базе.

В своем нынешнем состоянии система поддерживает все потребности заказчика. Более того, на прошлой неделе она была запущена в эксплуатацию. Приложение установлено на компьютере в отделе кадров, и Джо прошел курс обучения работе с системой. Все запросы на добавление новых и изменение существующих пользователей стекаются к Джо. Он вводит соответствующие операции в текстовый файл, который обрабатывается каждую ночь. Еще недавно Джо брюзжал по этому поводу, но здорово обрадовался, узнав, что мы собираемся снабдить систему графическим интерфейсом. Интерфейс упростит работу. А радуется Джо потому, что теперь каждый сможет сам вводить операции, не перекладывая эту обязанность на него.

Вопрос о характере интерфейса долго обсуждался с заказчиком. Предлагалось, в частности, разработать текстовый интерфейс на базе системы меню; для перехода от одного меню к другому и для ввода данных пользователь использовал бы клавиатуру. Но, хотя текстовые интерфейсы легко писать, работать с ними совсем не просто. К тому же в наши дни большинство пользователей считает их устаревшими.

Рассматривался также веб-интерфейс. Веб-приложения хороши тем, что обычно не требуют установки на компьютеры пользователей, а работать с ними можно с любого компьютера, подключенного к корпоративной интрасети. Однако построение веб-интерфейса затрудняется тем, что они привязывают приложение к большой и сложной инфраструктуре веб-серверов, серверов приложений и многоуровневых архитектур.[1] Эту инфраструктуру необходимо покупать, устанавливать, конфигурировать и администрировать. К тому же веб-системы неотъемлемы от таких технологий, как HTML, CSS и JavaScript, а получающийся интерфейс напоминает программы конца 1970-х годов на черно-зеленом терминале IBM 3270.[2]

И пользователи, и наша компания хотели бы сделать что-нибудь простое с точки зрения разработки, использования, установки и администрирования. Поэтому мы остановились на персональном приложении с графическим интерфейсом, поскольку оно предлагает более развитые средства взаимодействия и проще для разработки, чем веб-интерфейс. Первоначальную реализацию не предполагается развертывать в сети, так что нам не понадобится сложная инфраструктура, без которой, похоже, не обойтись в случае веб-системы.

---

[1] Или так кажется непросвещенному архитектору ПО. Во многих случаях эта дополнительная инфраструктура выгодна скорее поставщикам, чем пользователям.

[2] Это утверждение, пожалуй, уже неактуально. Достаточно вспомнить карты Google. – *Прим. перев.*

Разумеется, у приложений с ГИП есть недостатки. Они не переносимы и плохо адаптируются к распределенной архитектуре. Но, поскольку все пользователи системы расчета зарплаты работают в одном офисе и пользуются компьютерами компании, было решено, что эти недостатки обойдутся дешевле, чем создание веб-архитектуры. Итак, для построения ГИП мы воспользуемся каркасом Windows Forms.

Поскольку разработка ГИП – дело нетривиальное, в начальной версии мы ограничимся добавлением работников. Первый же выпуск даст нам ценную информацию. Во-первых, мы узнаем, насколько сложно будет построить ГИП. Во-вторых, Джо сможет поработать с новым ГИП и скажет, стала ли его жизнь легче, – на что мы очень надеемся. Вооружившись этими знаниями, мы сможем лучше понять, как работать дальше. Не исключено также, что отзывы о первом выпуске покажут, что текстовый или веб-интерфейс был бы лучше. Если такое случится, то лучше узнать об этом как можно раньше, пока в разработку всего приложения не вложено слишком много сил.

Внешний вид ГИП не так важен, как его внутренняя архитектура. Неважно, идет ли речь о персональном или веб-приложении, ГИП обычно склонен к более частым изменениям, чем скрывающиеся за ним бизнес-правила. Поэтому требуется тщательно отделить бизнес-логику от пользовательского интерфейса. Имея это в виду, мы постараемся написать как можно меньше кода в Windows Forms, а вместо этого разместим код в обычных классах на C#, которые будут работать совместно с Windows Forms. Такая стратегия защитит бизнес-правила от изменчивости ГИП. Изменение кода ГИП не должно отражаться на бизнес-правилах. Более того, если в будущем мы решим переключиться на веб-интерфейс, то код бизнес-правил будет уже выделен.

Интерфейс

На рис. 38.1 показана общая идея того ГИП, который мы собираемся создать. Меню Action содержит список всех поддерживаемых действий. При выборе действия открывается форма для ввода соответствующих данных. Так, на рис. 38.2 изображена форма, появляющаяся при выборе действия Add Employee (Добавить работника). Пока что нас только это действие и интересует.

В верхней части окна Payroll имеется текстовое поле Pending Transactions (Ожидающие операции). Расчет зарплаты – пакетное приложение. Операции вводятся в течение дня, но выполняются только ночью единым пакетом. В верхнем поле находится список введенных, но еще не выполненных операций. На рис. 38.1 показана одна ожидающая операция – добавление работника с почасовой оплатой. Формат списка вполне понятен, но, возможно, впоследствии мы захотим его улучшить. А пока сойдет и так.

Нижнее текстовое поле Employees содержит список уже хранящихся в системе работников. После выполнения операции `AddEmployeeTrans-`

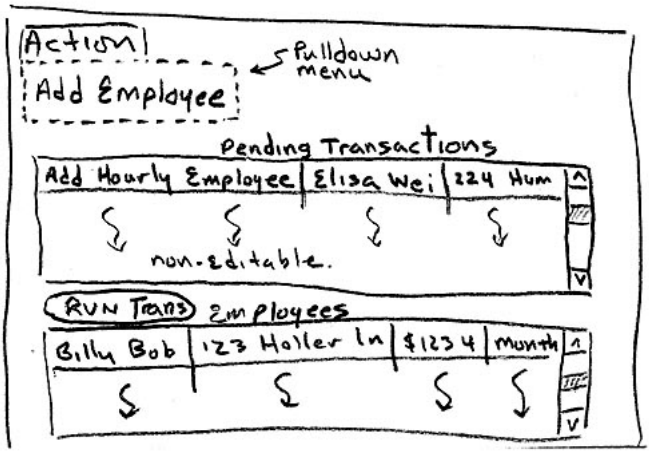


Рис. 38.1. *Первый вариант пользовательского интерфейса к системе расчета зарплаты*

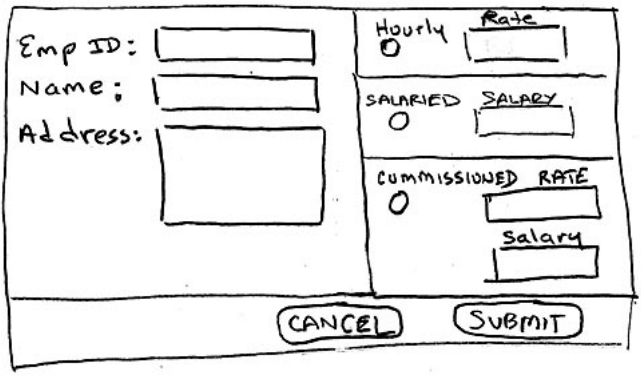


Рис. 38.2. *Форма для ввода операции Add Employee*

`actions` в этом списке появятся новые работники. Опять-таки можно представить гораздо более удобный способ отображения работников. Например, они красиво смотрелись бы в виде таблицы. Для каждого поля данных можно было бы ввести отдельный столбец и дополнительно добавить столбцы для даты последнего выписанного чека, итоговой суммы, начисленной с момента принятия на работу, и т. д. В строки, описывающие работников с почасовой и комиссионной оплатой, можно было бы включить ссылку на новое окно, в котором отображались бы карточки табельного учета и справки о продажах соответственно. Но с этим тоже можно повременить.

В середине находится кнопка Run Transactions. При ее нажатии запускается обработка пакета ожидающих операций и обновляется список работников. К сожалению, эту кнопку кто-то должен нажать, иначе

операции никогда не обработаются. Это временное решение – до того, как мы реализуем автоматический запуск задания по расписанию.

Реализация

Мы не можем продолжать работать с окном Payroll, пока не обеспечим возможность вводить операции, поэтому начнем с формы для ввода операции добавления работника (рис. 38.2). Давайте подумаем, какие бизнес-правила должно поддерживать это окно. Нам необходимо собрать всю информацию для создания операции. Для этого пользователь должен заполнить форму. Исходя из введенной информации, мы должны решить, какого типа операцию создавать, а затем поместить ее в список для последующей обработки. Эти действия инициируются нажатием кнопки Submit (Сохранить).

На этом бизнес-правила исчерпываются, но нам необходимы и другие механизмы, делающие ГИП более удобным для работы. Например, кнопка Submit должна оставаться неактивной, пока не будет введена вся необходимая информация. Кроме того, текстовое поле для ввода почасовой ставки должно быть недоступно если не включен переключатель Hourly. Аналогично поля Salary (Оклад), Base Salary (Базовый оклад) и Commission (Комиссионные) должны быть неактивны если не включен соответствующий переключатель.

Необходимо строго разделять поведение, относящееся к бизнес-правилам и к ГИП. Для этого мы воспользуемся паттерном проектирования Модель-Вид-Презентатор (Model View Presenter). На рис. 38.3 изображена UML-диаграмма, показывающая, как мы будем использовать этот паттерн в рассматриваемой задаче. Как видите, структура решения состоит из трех компонентов: модель (Model), вид (View) и презентатор (Presenter). Моделью в данном случае является класс `AddEmployeeTransaction` и производные от него. Вид – это форма Windows Form, названная `AddEmployeeWindow` и показанная на рис. 38.2. Презентатор – класс `AddEmployeePresenter` – связывает ГИП с моделью. Класс `AddEmployeePresenter` содержит всю бизнес-логику, относящуюся к этой части приложения. В классе же `AddEmployeeWindow`, напротив, бизнес-логики вообще нет; он отвечает только за поведение ГИП, делегируя все бизнес-решения презентатору.

Альтернатива применению паттерна Модель-Вид-Презентатор – втиснуть всю бизнес-логику в форму Windows Form. Такой подход очень распространен, но сулит немало проблем. Включая бизнес-правила в код ГИП, вы, конечно, нарушаете принцип SRP. Но это полбеды, хуже, что в таком случае бизнес-правила с трудом поддаются автоматизированному тестированию. Для выполнения теста пришлось бы нажимать кнопки, читать метки, выбирать пункты из комбинированного списка и манипулировать другими элементами управления. Иначе говоря, чтобы протестировать бизнес-правила, пришлось бы *использовать* ГИП. Те-

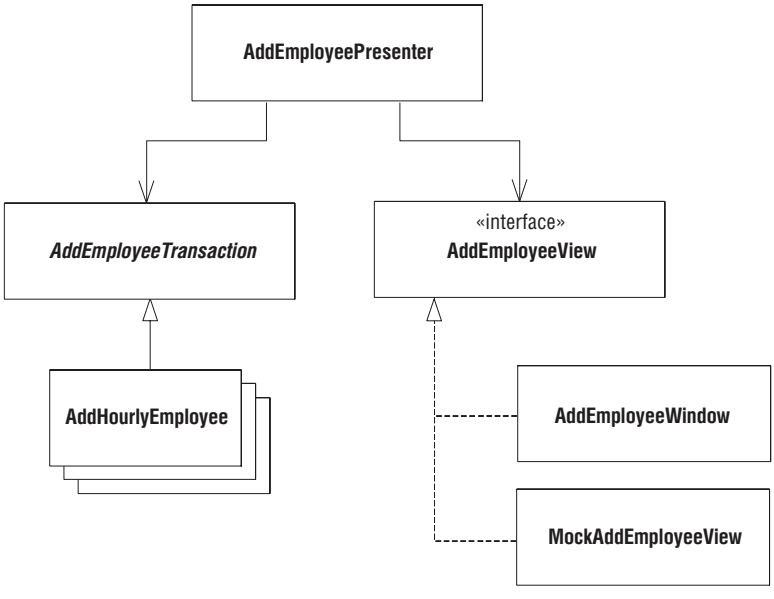


Рис. 38.3. *Применение паттерна Модель-Вид-Презентатор для ввода операции добавления работника*

сты, использующие ГИП, хрупки, потому что даже незначительное изменение в структуре ГИП оказывает на них большое влияние. Вдобавок к этому их трудно писать, поскольку включение ГИП в тестовую оснастку – само по себе нетривиальная задача. Кроме того, впоследствии может быть принято решение о переходе на веб-интерфейс, и тогда бизнес-логику, встроенную в Windows-форму, придется дублировать в ASP.NET.

Внимательный читатель заметит, что класс `AddEmployeePresenter` напрямую не зависит от `AddEmployeeWindow`. Интерфейс `AddEmployeeView` инвертирует зависимость. Зачем? Да просто для того, чтобы упростить тестирование. Получить доступ к тестируемому ГИП сложно. Если бы `AddEmployeePresenter` напрямую зависел от `AddEmployeeWindow`, то и `AddEmployeePresenterTest` должен был бы зависеть от `AddEmployeeWindow`, а это крайне нежелательно. Применение интерфейса и класса `MockAddEmployeeView` существенно упрощает тестирование.

В листингах 38.1 и 38.2 показаны классы `AddEmployeePresenterTest` и `AddEmployeePresenter` соответственно. С этого все и начинается.

Листинг 38.1. AddEmployeePresenterTest.cs

```
using NUnit.Framework;
using Payroll;
```

```
namespace PayrollUI
{
  [TestFixture]
  public class AddEmployeePresenterTest
  {
    private AddEmployeePresenter presenter;
    private TransactionContainer container;
    private InMemoryPayrollDatabase database;
    private MockAddEmployeeView view;

    [SetUp]
    public void SetUp()
    {
      view = new MockAddEmployeeView();
      container = new TransactionContainer(null);
      database = new InMemoryPayrollDatabase();
      presenter = new AddEmployeePresenter(
        view, container, database);
    }

    [Test]
    public void Creation()
    {
      Assert.AreSame(container,
        presenter.TransactionContainer);
    }

    [Test]
    public void AllInfoIsCollected()
    {
      Assert.IsFalse(presenter.AllInformationIsCollected());
      presenter.EmpId = 1;
      Assert.IsFalse(presenter.AllInformationIsCollected());
      presenter.Name = "Bill";
      Assert.IsFalse(presenter.AllInformationIsCollected());
      presenter.Address = "123 abc";
      Assert.IsFalse(presenter.AllInformationIsCollected());
      presenter.IsHourly = true;
      Assert.IsFalse(presenter.AllInformationIsCollected());
      presenter.HourlyRate = 1.23;
      Assert.IsTrue(presenter.AllInformationIsCollected());
      presenter.IsHourly = false;
      Assert.IsFalse(presenter.AllInformationIsCollected());
      presenter.IsSalary = true;
      Assert.IsFalse(presenter.AllInformationIsCollected());
      presenter.Salary = 1234;
      Assert.IsTrue(presenter.AllInformationIsCollected());
      presenter.IsSalary = false;
      Assert.IsFalse(presenter.AllInformationIsCollected());
      presenter.IsCommission = true;
      Assert.IsFalse(presenter.AllInformationIsCollected());
```

```
  presenter.CommissionSalary = 123;
  Assert.IsFalse(presenter.AllInformationIsCollected());
  presenter.Commission = 12;
  Assert.IsTrue(presenter.AllInformationIsCollected());
}

[Test]
public void ViewGetsUpdated()
{
  presenter.EmpId = 1;
  CheckSubmitEnabled(false, 1);
  presenter.Name = "Bill";
  CheckSubmitEnabled(false, 2);
  presenter.Address = "123 abc";
  CheckSubmitEnabled(false, 3);
  presenter.IsHourly = true;
  CheckSubmitEnabled(false, 4);
  presenter.HourlyRate = 1.23;
  CheckSubmitEnabled(true, 5);
}

private void CheckSubmitEnabled(bool expected, int count)
{
  Assert.AreEqual(expected, view.submitEnabled);
  Assert.AreEqual(count, view.submitEnabledCount);
  view.submitEnabled = false;
}

[Test]
public void CreatingTransaction()
{
  presenter.EmpId = 123;
  presenter.Name = "Joe";
  presenter.Address = "314 Elm";
  presenter.IsHourly = true;
  presenter.HourlyRate = 10;
  Assert.IsTrue(presenter.CreateTransaction()
    is AddHourlyEmployee);
  presenter.IsHourly = false;
  presenter.IsSalary = true;
  presenter.Salary = 3000;
  Assert.IsTrue(presenter.CreateTransaction()
    is AddSalariedEmployee);
  presenter.IsSalary = false;
  presenter.IsCommission = true;
  presenter.CommissionSalary = 1000;
  presenter.Commission = 25;
  Assert.IsTrue(presenter.CreateTransaction()
    is AddCommissionedEmployee);
}
```

```
    [Test]
    public void AddEmployee()
    {
      presenter.EmpId = 123;
      presenter.Name = "Joe";
      presenter.Address = "314 Elm";
      presenter.IsHourly = true;
      presenter.HourlyRate = 25;
      presenter.AddEmployee();
      Assert.AreEqual(1, container.Transactions.Count);
      Assert.IsTrue(container.Transactions[0]
        is AddHourlyEmployee);
    }
  }
}
```

Листинг 38.2. AddEmployeePresenter.cs

```
using Payroll;

namespace PayrollUI
{
  public class AddEmployeePresenter
  {
    private TransactionContainer transactionContainer;
    private AddEmployeeView view;
    private PayrollDatabase database;
    private int empId;
    private string name;
    private string address;
    private bool isHourly;
    private double hourlyRate;
    private bool isSalary;
    private double salary;
    private bool isCommission;
    private double commissionSalary;
    private double commission;

    public AddEmployeePresenter(AddEmployeeView view,
      TransactionContainer container,
      PayrollDatabase database)
    {
      this.view = view;
      this.transactionContainer = container;
      this.database = database;
    }

    public int EmpId
    {
      get { return empId; }
      set
      {
```

```
    empId = value;
    UpdateView();
  }
}

public string Name
{
  get { return name; }
  set
  {
    name = value;
    UpdateView();
  }
}

public string Address
{
  get { return address; }
  set
  {
    address = value;
    UpdateView();
  }
}

public bool IsHourly
{
  get { return isHourly; }
  set
  {
    isHourly = value;
    UpdateView();
  }
}

public double HourlyRate
{
  get { return hourlyRate; }
  set
  {
    hourlyRate = value;
    UpdateView();
  }
}

public bool IsSalary
{
  get { return isSalary; }
  set
  {
    isSalary = value;
```

```
    UpdateView();
  }
}

public double Salary
{
  get { return salary; }
  set
  {
    salary = value;
    UpdateView();
  }
}

public bool IsCommission
{
  get { return isCommission; }
  set
  {
    isCommission = value;
    UpdateView();
  }
}

public double CommissionSalary
{
  get { return commissionSalary; }
  set
  {
    commissionSalary = value;
    UpdateView();
  }
}

public double Commission
{
  get { return commission; }
  set
  {
    commission = value;
    UpdateView();
  }
}

private void UpdateView()
{
  if(AllInformationIsCollected())
    view.SubmitEnabled = true;
  else
    view.SubmitEnabled = false;
}
```

```
    public bool AllInformationIsCollected()
    {
      bool result = true;
      result &= empId > 0;
      result &= name != null && name.Length > 0;
      result &= address != null && address.Length > 0;
      result &= isHourly || isSalary || isCommission;
      if(isHourly)
        result &= hourlyRate > 0;
      else if(isSalary)
        result &= salary > 0;
      else if(isCommission)
      {
        result &= commission > 0;
        result &= commissionSalary > 0;
      }
      return result;
    }

    public TransactionContainer TransactionContainer
    {
      get { return transactionContainer; }
    }

    public virtual void AddEmployee()
    {
      transactionContainer.Add(CreateTransaction());
    }

    public Transaction CreateTransaction()
    {
      if(isHourly)
        return new AddHourlyEmployee(
          empId, name, address, hourlyRate, database);
      else if(isSalary)
        return new AddSalariedEmployee(
            empId, name, address, salary, database);
      else
        return new AddCommissionedEmployee(
            empId, name, address, commissionSalary,
            commission, database);
    }
  }
}
```

Начиная с метода `SetUp`**, мы видим, к чему сводится создание экземпляра класса** `AddEmployeePresenter`**. Его конструктор принимает три параметра. Первый – объект** `AddEmployeeView`**, в качестве которого в тесте используется** `MockAddEmployeeView`**. Второй – объект** `TransactionContainer`**, то есть контейнер, в который можно поместить созданный объект** `AddEmployeeTransaction`**. Третий – экземпляр** `PayrollDatabase`**, который напря-**

мую не используется, а передается конструкторам класса AddEmployeeTransaction.

Первый тест, Creation, совершенно нелепый. Когда принимаешься писать код, трудно решить, что протестировать сначала. Часто лучше всего начинать с тестирования самой простой вещи. Тогда дело сдвинется с мертвой точки, и последующие тесты будут складываться сами собой. Тест Creation – как раз свидетельство такого подхода. В нем проверяется, что параметр container сохранен; впоследствии его, пожалуй, следовало бы удалить.

Следующий тест, AllInfoIsCollected, гораздо интереснее. Одна из обязанностей AddEmployeePresenter – собрать всю информацию, необходимую для создания операции. Частичные данные не годятся, поэтому презентатор должен знать, когда введены *все* требуемые данные. Этот тест говорит, что презентатору нужен метод AllInformationIsCollected, возвращающий булевское значение. Кроме того, тест показывает, что данные передаются презентатору через свойства – один элемент за другим. После каждого шага тест спрашивает у презентатора, получил ли тот все требуемые данные, и проверяет ответ с помощью утверждения Assert. В коде класса AddEmployeePresenter мы видим, что свойства просто сохраняют переданные значения в соответствующих полях. Метод AllInformationIsCollected выполняет логические операции, проверяя, все ли поля инициализированы.

После того как презентатор получит всю необходимую информацию, пользователь может добавить операцию, нажав кнопку Submit. Но эта кнопка не должна быть доступна, пока презентатор не будет доволен полученными данными. Следовательно, уведомлять пользователя о том, что форму можно сохранить, должен презентатор. Метод ViewGetsUpdated проверяет текущее положение дел. Он передает данные презентатору по одному элементу и всякий раз проверяет, получил ли вид уведомление от презентатора о том, что кнопку Submit можно активировать.

Глядя на код презентатора, мы видим, что каждое свойство вызывает метод UpdateView, который, в свою очередь, устанавливает свойство вида SubmitEnabled. В листинге 38.3 показан интерфейс AddEmployeeView, в котором объявлено свойство SubmitEnabled. Объект AddEmployeePresenter уведомляет о том, что кнопку сохранения следует активировать, устанавливая это свойство. В данный момент нас не очень интересует, как реализовано свойство SubmitEnabled. Мы просто хотим быть уверены, что ему присваивается правильное значение. Именно здесь и оказывается полезен интерфейс AddEmployeeView. Он позволяет создать имитацию вида, чтобы упростить тестирование. В классе MockAddEmployeeView, показанном в листинге 38.4, есть два поля: submitEnabled, в котором сохраняется последнее переданное значение, и submitEnabledCount, где хранится счетчик обращений к свойству SubmitEnabled. Наличия этих тривиальных полей достаточно для написания теста. Тесту нужно лишь проверить поле submitEnabled, дабы удостовериться в том, что презента-

тор вызывал свойство SubmitEnabled с правильным значением, а также проверить submitEnabledCount, чтобы убедиться, что это свойство вызывалось ожидаемое число раз. Представьте, как тяжело было бы писать тест если бы нам пришлось забраться в дебри форм и оконных элементов управления.

Листинг 38.3. AddEmployeeView.cs

```
namespace PayrollUI
{
  public interface AddEmployeeView
  {
    bool SubmitEnabled { set; }
  }
}
```

Листинг 38.4. MockAddEmployeeView.xs

```
namespace PayrollUI
{
  public class MockAddEmployeeView : AddEmployeeView
  {
    public bool submitEnabled;
    public int submitEnabledCount;

    public bool SubmitEnabled
    {
      set
      {
        submitEnabled = value;
        submitEnabledCount++;
      }
    }
  }
}
```

В этом тесте происходит кое-что интересное. Мы тестировали, как ведет себя AddEmployeePresenter во время ввода данных через вид, а не то, что происходит при вводе данных. В готовой программе после ввода всех данных кнопка Submit становится активной. Мы могли бы это проверить, но взамен тестировали поведение презентатора. Мы убедились, что после ввода всех данных презентатор отправит сообщение виду, уведомляя о том, что теперь сохранение возможно.

Такой стиль тестирования называется разработкой через поведение (Behavior-Driven Development). Идея в том, что на тесты следует смотреть не как на утверждения о состоянии и результатах, а как на *спецификации поведения*, в которых описывается правильное поведение программы.

Следующий тест, CreatingTransaction, демонстрирует, что AddEmployeePresenter правильно создает операцию по предоставленным ему данным.

В классе AddEmployeePresenter для определения типа операции используется ветвление по методу платежа.

Остался последний тест – AddEmployee. После того как все данные собраны и операция создана, презентатор должен сохранить ее в объекте TransactionContainer для последующего использования. Тест проверяет, что это сделано.

Реализовав класс AddEmployeePresenter, мы получили все бизнес-правила, необходимые для создания объектов AddEmployeeTransaction. Теперь нужен пользовательский интерфейс.

Конструирование окна

Конструирование окна формы Add Employee – легкая задача. Со встроенным в Visual Studio конструктором нам остается только перетащить мышью элементы управления в нужные места. Автоматически сгенерированный код в следующих ниже листингах не показан. Но на конструировании окна наша работа не заканчивается. Нужно реализовать в ГИП определенное поведение и связать его с презентатором. И тест тоже понадобится. В листинге 38.5 приведен тест AddEmployeeWindowTest, а в листинге 38.6 – класс AddEmployeeWindow.

Листинг 38.5. AddEmployeeWindowTest.cs

```
using NUnit.Framework;

namespace PayrollUI
{
  [TestFixture]
  public class AddEmployeeWindowTest
  {
    private AddEmployeeWindow window;
    private AddEmployeePresenter presenter;
    private TransactionContainer transactionContainer;

    [SetUp]
    public void SetUp()
    {
      window = new AddEmployeeWindow();
      transactionContainer = new TransactionContainer(null);
      presenter = new AddEmployeePresenter(
        window, transactionContainer, null);

      window.Presenter = presenter;
      window.Show();
    }

    [Test]
    public void StartingState()
    {
```

```
  Assert.AreSame(presenter, window.Presenter);
  Assert.IsFalse(window.submitButton.Enabled);
  Assert.IsFalse(window.hourlyRateTextBox.Enabled);
  Assert.IsFalse(window.salaryTextBox.Enabled);
  Assert.IsFalse(window.commissionSalaryTextBox.Enabled);
  Assert.IsFalse(window.commissionTextBox.Enabled);
}

[Test]
public void PresenterValuesAreSet()
{
  window.empIdTextBox.Text = "123";
  Assert.AreEqual(123, presenter.EmpId);

  window.nameTextBox.Text = "John";
  Assert.AreEqual("John", presenter.Name);

  window.addressTextBox.Text = "321 Somewhere";
  Assert.AreEqual("321 Somewhere", presenter.Address);

  window.hourlyRateTextBox.Text = "123.45";
  Assert.AreEqual(123.45, presenter.HourlyRate, 0.01);

  window.salaryTextBox.Text = "1234";
  Assert.AreEqual(1234, presenter.Salary, 0.01);

  window.commissionSalaryTextBox.Text = "123";
  Assert.AreEqual(123, presenter.CommissionSalary, 0.01);

  window.commissionTextBox.Text = "12.3";
  Assert.AreEqual(12.3, presenter.Commission, 0.01);

  window.hourlyRadioButton.PerformClick();
  Assert.IsTrue(presenter.IsHourly);

  window.salaryRadioButton.PerformClick();
  Assert.IsTrue(presenter.IsSalary);
  Assert.IsFalse(presenter.IsHourly);

  window.commissionRadioButton.PerformClick();
  Assert.IsTrue(presenter.IsCommission);
  Assert.IsFalse(presenter.IsSalary);
}
[Test]
public void EnablingHourlyFields()
{
  window.hourlyRadioButton.Checked = true;
  Assert.IsTrue(window.hourlyRateTextBox.Enabled);

  window.hourlyRadioButton.Checked = false;
  Assert.IsFalse(window.hourlyRateTextBox.Enabled);
}
```

```
    [Test]
    public void EnablingSalaryFields()
    {
      window.salaryRadioButton.Checked = true;
      Assert.IsTrue(window.salaryTextBox.Enabled);

      window.salaryRadioButton.Checked = false;
      Assert.IsFalse(window.salaryTextBox.Enabled);
    }

    [Test]
    public void EnablingCommissionFields()
    {
      window.commissionRadioButton.Checked = true;
      Assert.IsTrue(window.commissionTextBox.Enabled);
      Assert.IsTrue(window.commissionSalaryTextBox.Enabled);

      window.commissionRadioButton.Checked = false;
      Assert.IsFalse(window.commissionTextBox.Enabled);
      Assert.IsFalse(window.commissionSalaryTextBox.Enabled);
    }

    [Test]
    public void EnablingAddEmployeeButton()
    {
      Assert.IsFalse(window.submitButton.Enabled);

      window.SubmitEnabled = true;
      Assert.IsTrue(window.submitButton.Enabled);

      window.SubmitEnabled = false;
      Assert.IsFalse(window.submitButton.Enabled);
    }

    [Test]
    public void AddEmployee()
    {
      window.empIdTextBox.Text = "123";
      window.nameTextBox.Text = "John";
      window.addressTextBox.Text = "321 Somewhere";
      window.hourlyRadioButton.Checked = true;
      window.hourlyRateTextBox.Text = "123.45";

      window.submitButton.PerformClick();
      Assert.IsFalse(window.Visible);
      Assert.AreEqual(1,
      transactionContainer.Transactions.Count);
    }
  }
}
```

Листинг 38.6. AddEmployeeWindow.cs

```
using System;
using System.Windows.Forms;

namespace PayrollUI
{
  public class AddEmployeeWindow : Form, AddEmployeeView
  {
    public System.Windows.Forms.TextBox empIdTextBox;
    private System.Windows.Forms.Label empIdLabel;
    private System.Windows.Forms.Label nameLabel;
    public System.Windows.Forms.TextBox nameTextBox;
    private System.Windows.Forms.Label addressLabel;
    public System.Windows.Forms.TextBox addressTextBox;
    public System.Windows.Forms.RadioButton hourlyRadioButton;
    public System.Windows.Forms.RadioButton salaryRadioButton;
    public System.Windows.Forms.RadioButton commissionRadioButton;
    private System.Windows.Forms.Label hourlyRateLabel;
    public System.Windows.Forms.TextBox hourlyRateTextBox;
    private System.Windows.Forms.Label salaryLabel;
    public System.Windows.Forms.TextBox salaryTextBox;
    private System.Windows.Forms.Label commissionSalaryLabel;
    public System.Windows.Forms.TextBox commissionSalaryTextBox;
    private System.Windows.Forms.Label commissionLabel;
    public System.Windows.Forms.TextBox commissionTextBox;
    private System.Windows.Forms.TextBox textBox2;
    private System.Windows.Forms.Label label1;
    private System.ComponentModel.Container components = null;
    public System.Windows.Forms.Button submitButton;
    private AddEmployeePresenter presenter;

    public AddEmployeeWindow()
    {
      InitializeComponent();
    }

    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
        if(components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose( disposing );
    }

    #region Windows Form Designer generated code
    // опущено
```

```
#endregion

public AddEmployeePresenter Presenter
{
  get { return presenter; }
  set { presenter = value; }
}

private void hourlyRadioButton_CheckedChanged(
  object sender, System.EventArgs e)
{
  hourlyRateTextBox.Enabled = hourlyRadioButton.Checked;
  presenter.IsHourly = hourlyRadioButton.Checked;
}

private void salaryRadioButton_CheckedChanged(
  object sender, System.EventArgs e)
{
  salaryTextBox.Enabled = salaryRadioButton.Checked;
  presenter.IsSalary = salaryRadioButton.Checked;
}

private void commissionRadioButton_CheckedChanged(
  object sender, System.EventArgs e)
{
  commissionSalaryTextBox.Enabled =
    commissionRadioButton.Checked;
  commissionTextBox.Enabled =
    commissionRadioButton.Checked;
  presenter.IsCommission =
    commissionRadioButton.Checked;
}

private void empIdTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.EmpId = AsInt(empIdTextBox.Text);
}

private void nameTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.Name = nameTextBox.Text;
}

private void addressTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.Address = addressTextBox.Text;
}
```

```
private void hourlyRateTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.HourlyRate = AsDouble(hourlyRateTextBox.Text);
}

private void salaryTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.Salary = AsDouble(salaryTextBox.Text);
}

private void commissionSalaryTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.CommissionSalary =
    AsDouble(commissionSalaryTextBox.Text);
}

private void commissionTextBox_TextChanged(
  object sender, System.EventArgs e)
{
  presenter.Commission = AsDouble(commissionTextBox.Text);
}

private void addEmployeeButton_Click(
  object sender, System.EventArgs e)
{
  presenter.AddEmployee();
  this.Close();
}

private double AsDouble(string text)
{
  try
  {
    return Double.Parse(text);
  }
  catch (Exception)
  {
    return 0.0;
  }
}

private int AsInt(string text)
{
  try
  {
    return Int32.Parse(text);
  }
  catch (Exception)
```

```
        {
          return 0;
        }
      }

      public bool SubmitEnabled
      {
        set { submitButton.Enabled = value; }
      }
    }
  }
```

Несмотря на все мои стенания по поводу того, как мучительно тестирование кода ГИП, тестировать код в Windows Form относительно просто. Однако есть и подводные камни. По какой-то странной причине, известной лишь программистам в Microsoft, половина функциональности элементов управления не работает если они не *отображены* на экране. Именно по этой причине в метод `SetUp` тестовой фикстуры включен вызов `window.Show()`. Во время выполнения каждого теста вы видите, как окно на мгновение появляется и тут же исчезает. Это раздражает, но терпеть можно. Все, что замедляет работу тестов или вызывает какие-то неудобства при их выполнении, повышает вероятность того, что тесты не будут успешно завершаться.

Еще одно ограничение заключается в том, что трудно имитировать событие, генерируемое элементом управления. В случае кнопок и им подобных элементов можно вызвать метод `PerformClick`, но для таких событий, как `MouseOver`, `Leave`, `Validate` и др., столь же простого решения не существует. Однако имеется расширение каркаса NUnit – NUnitForms, позволяющее решить эту и ряд других проблем. Впрочем, наши тесты настолько просты, что мы обошлись без дополнительной помощи.

В методе `SetUp` создается экземпляр класса `AddEmployeeWindow`, который передается конструктору класса `AddEmployeePresenter`. Затем в первом тесте, `StartingState`, мы проверяем, что элементы управления `hourlyRateTextBox`, `salaryTextBox`, `commissionSalaryTextBox`, `commissionTextBox` не активны. Нам понадобятся только одно или два из этих полей, но какие именно, неизвестно, пока пользователь не выбрал метод платежа. Чтобы не смущать пользователя, мы запрещаем ввод во все поля, пока не станет ясно, какие должны быть открытыми. Правила активации этих элементов управления описаны в следующих трех тестах: `EnablingHourlyFields`, `EnablingSalaryField` и `EnablingCommissionFields`. Например, `EnablingHourlyFields` показывает, что поле `hourlyRateTextBox` активно когда переключатель `hourlyRadioButton` включен, и неактивно когда он выключен. Достигается это путем регистрации обработчика событий `EventHandler` от каждого элемента `RadioButton`. Обработчик активирует и деактивирует соответствующие текстовые поля.

Тест `PresenterValuesAreSet` особенно важен. Презентатор знает, что делать с данными, но предоставить данные – задача вида. Поэтому всякий

раз, как какое-нибудь поле формы изменяется, вид обращается к соответствующему полю презентатора. Для каждого текстового поля TextBox мы изменяем свойство Text, а затем проверяем, что презентатор действительно обновлен. В классе AddEmployeeWindow для каждого элемента типа TextBox зарегистрирован обработчик события TextChanged. Для элементов RadioButton мы вызываем в тесте метод PerformClick и опять-таки проверяем, что презентатор получил уведомление. Об этом заботятся обработчики событий CheckChanged от элементов RadioButton.

Тест EnablingAddEmployeeButton определяет, что кнопка submitButton должна стать активной когда свойство SubmitEnabled установлено в true, и наоборот. Напомним: в классе AddEmployeePresenterTest нам было безразлично, что делает это свойство. А теперь это существенно. Вид должен правильно реагировать на изменение свойства SubmitEnabled, однако же проверять это поведение в AddEmployeePresenterTest было неуместно. А вот класс AddEmployeeWindowTest как раз посвящен проверке поведения класса AddEmployeeWindow, так что в нем самое место для тестирования этот аспекта.

Последний тест, AddEmployee, заполняет допустимый набор полей, нажимает кнопку Submit и проверяет, что окно теперь невидимо, а операция добавлена в transactionContainer. Для этого мы регистрируем обработчик события от кнопки submitButton, в котором вызывается метод AddEmployee презентатора, и закрываем окно. Вообще-то этот тест производит довольно много действий только для того, чтобы проверить, был ли вызван метод AddEmployee: сначала нужно заполнить все поля, а потом проанализировать содержимое transactionContainer.

Тут можно возразить, поскольку для проверки того, что метод вызывался, было бы проще воспользоваться имитацией презентатора. Честно говоря, я не стал бы возражать если бы мой партнер так и поступил. Но и та реализация, что есть, меня не очень смущает. Нет ничего дурного в том, чтобы включать парочку таких высокоуровневых тестов. Они помогают убедиться в том, что все части правильно сопрягаются и система в целом ведет себя, как положено. Обычно для контроля таких вещей на еще более высоком уровне мы готовим приемочные тесты, но вставить проверку-другую в автономные тесты тоже не помешает.

Теперь у нас есть работоспособная форма для создания объектов AddEmployeeTransactions. Но ее невозможно использовать, пока не будет готово главное окно Payroll и код для открытия формы AddEmployeeWindow.

Окно Payroll

При проектировании вида Payroll, показанного на рис. 38.4, мы применяем тот же самый паттерн Модель-Вид-Презентатор.

В листингах 38.7 и 38.8 показан весь код, относящийся к этой части программы. Процесс разработки этого вида очень похож на то, как раз-

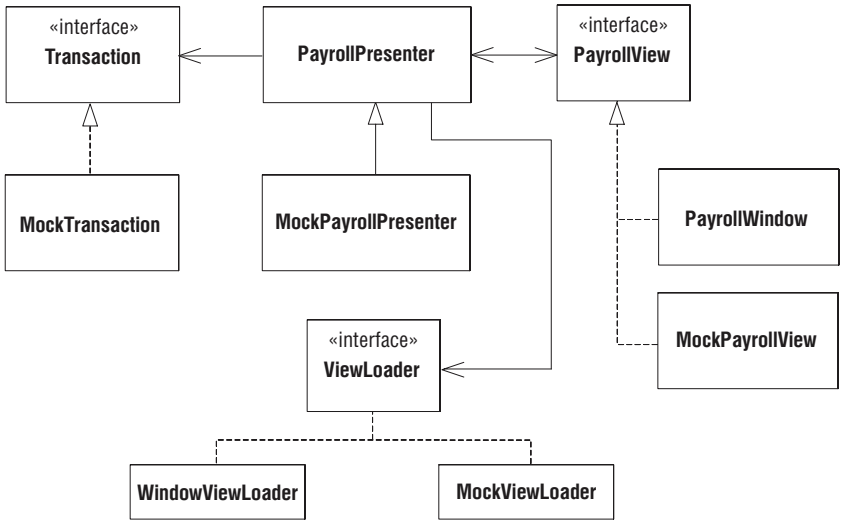


Рис. 38.4. *Дизайн вида Payroll*

рабатывали вид Add Employee. Поэтому мы не станем разбирать его так же подробно. Но иерархии `ViewLoader` все же стоит уделить внимание.

При разработке этого окна мы рано или поздно должны будем заняться реализацией обработчика события от пункта меню (элемент типа `MenuItem`) Add Employee (Добавить работника). Этот обработчик будет вызывать метод `AddEmployeeActionInvoked` объекта `PayrollPresenter`. В этот момент и должно появиться окно `AddEmployeeWindow`. Кто должен создать объект `AddEmployeeWindow` – презентатор? До сих пор нам удавалось отделять ГИП от приложения. Если объекту `PayrollPresenter` поручить создание `AddEmployeeWindow`, то мы нарушим принцип DIP. Но кто-то же должен этот объект создать.

На помощь приходит паттерн Фабрика! Это именно та задача, для которой Фабрика и создана. Интерфейс `ViewLoader` вместе с производными от него классами, по существу, реализует паттерн Фабрика. В нем объявлено два метода: `LoadPayrollView` и `LoadAddEmployeeView`. В классе `WindowsViewLoader` они реализованы так, что создают и отображают формы Windows Forms. А класс `MockViewLoader`, который легко подставляется вместо `WindowsViewLoader`, существенно упрощает тестирование.

При наличии класса `ViewLoader` класс `PayrollPresenter` уже не должен зависеть от классов, реализующих формы. Он просто обращается к методу `LoadAddEmployeeView` того экземпляра `ViewLoader`, который ему передан. Если возникнет необходимость, мы сможем полностью изменить пользовательский интерфейс системы, подставив другую реализацию `ViewLoader`. Никакой код изменять не придется! Вот это сила! Вот это OCP!

Листинг 38.7. PayrollPresenterTest.cs

```
using System;
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
  [TestFixture]
  public class PayrollPresenterTest
  {
    private MockPayrollView view;
    private PayrollPresenter presenter;
    private PayrollDatabase database;
    private MockViewLoader viewLoader;

    [SetUp]
    public void SetUp()
    {
      view = new MockPayrollView();
      database = new InMemoryPayrollDatabase();
      viewLoader = new MockViewLoader();
      presenter = new PayrollPresenter(database, viewLoader);
      presenter.View = view;
    }

    [Test]
    public void Creation()
    {
      Assert.AreSame(view, presenter.View);
      Assert.AreSame(database, presenter.Database);
      Assert.IsNotNull(presenter.TransactionContainer);
    }

    [Test]
    public void AddAction()
    {
      TransactionContainer container =
        presenter.TransactionContainer;
      Transaction transaction = new MockTransaction();
      container.Add(transaction);
      string expected = transaction.ToString()
        + Environment.NewLine;
      Assert.AreEqual(expected, view.transactionsText);
    }

    [Test]
    public void AddEmployeeAction()
    {
      presenter.AddEmployeeActionInvoked();
```

```
      Assert.IsTrue(viewLoader.addEmployeeViewWasLoaded);
    }

    [Test]
    public void RunTransactions()
    {
      MockTransaction transaction = new MockTransaction();
      presenter.TransactionContainer.Add(transaction);
      Employee employee =
        new Employee(123, "John", "123 Baker St.");
      database.AddEmployee(employee);
      presenter.RunTransactions();
      Assert.IsTrue(transaction.wasExecuted);
      Assert.AreEqual("", view.transactionsText);
      string expectedEmployeeTest = employee.ToString()
        + Environment.NewLine;
      Assert.AreEqual(expectedEmployeeTest, view.employeesText);
    }
  }
}
```

Листинг 38.8. PayrollPresenter.cs

```
using System;
using System.Text;
using Payroll;

namespace PayrollUI
{
  public class PayrollPresenter
  {
    private PayrollView view;
    private readonly PayrollDatabase database;
    private readonly ViewLoader viewLoader;
    private TransactionContainer transactionContainer;

    public PayrollPresenter(PayrollDatabase database,
      ViewLoader viewLoader)
    {
      this.view = view;
      this.database = database;
      this.viewLoader = viewLoader;
      TransactionContainer.AddAction addAction =
        new TransactionContainer.AddAction(TransactionAdded);
      transactionContainer = new TransactionContainer(addAction);
    }
    public PayrollView View
    {
      get { return view; }
      set { view = value; }
    }
```

```
public TransactionContainer TransactionContainer
{
  get { return transactionContainer; }
}

public void TransactionAdded()
{
  UpdateTransactionsTextBox();
}

private void UpdateTransactionsTextBox()
{
  StringBuilder builder = new StringBuilder();
  foreach(Transaction transaction in
  transactionContainer.Transactions)
  {
    builder.Append(transaction.ToString());
    builder.Append(Environment.NewLine);
  }
  view.TransactionsText = builder.ToString();
}

public PayrollDatabase Database
{
  get { return database; }
}

public virtual void AddEmployeeActionInvoked()
{
  viewLoader.LoadAddEmployeeView(transactionContainer);
}

public virtual void RunTransactions()
{
  foreach(Transaction transaction in
    transactionContainer.Transactions)
    transaction.Execute();

  transactionContainer.Clear();
  UpdateTransactionsTextBox();
  UpdateEmployeesTextBox();
}

private void UpdateEmployeesTextBox()
{
  StringBuilder builder = new StringBuilder();
  foreach(Employee employee in database.GetAllEmployees())
  {
    builder.Append(employee.ToString());
    builder.Append(Environment.NewLine);
  }
```

```
        view.EmployeesText = builder.ToString();
      }
    }
  }
```

Листинг 38.9. PayrollView.cs

```
namespace PayrollUI
{
  public interface PayrollView
  {
    string TransactionsText { set; }
    string EmployeesText { set; }
    PayrollPresenter Presenter { set; }
  }
}
```

Листинг 38.10. MockPayrollView.cs

```
namespace PayrollUI
{
  public class MockPayrollView : PayrollView
  {
    public string transactionsText;
    public string employeesText;
    public PayrollPresenter presenter;

    public string TransactionsText
    {
      set { transactionsText = value; }
    }

    public string EmployeesText
    {
      set { employeesText = value; }
    }

    public PayrollPresenter Presenter
    {
      set { presenter = value; }
    }
  }
}
```

Листинг 38.11. ViewLoader.cs

```
namespace PayrollUI
{
  public interface ViewLoader
  {
    void LoadPayrollView();
    void LoadAddEmployeeView(
    TransactionContainer transactionContainer);
```

```
  }
}
```

Листинг 38.12. MockViewLoader.cs

```
namespace PayrollUI
{
  public class MockViewLoader : ViewLoader
  {
    public bool addEmployeeViewWasLoaded;
    private bool payrollViewWasLoaded;

    public void LoadPayrollView()
    {
      payrollViewWasLoaded = true;
    }

    public void LoadAddEmployeeView(
      TransactionContainer transactionContainer)
    {
      addEmployeeViewWasLoaded = true;
   }
  }
}
```

Листинг 38.13. WindowViewLoaderTest.cs

```
using System.Windows.Forms;
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
  [TestFixture]
  public class WindowViewLoaderTest
  {
    private PayrollDatabase database;
    private WindowViewLoader viewLoader;

    [SetUp]
    public void SetUp()
    {
      database = new InMemoryPayrollDatabase();
      viewLoader = new WindowViewLoader(database);
    }

    [Test]
    public void LoadPayrollView()
    {
      viewLoader.LoadPayrollView();
      Form form = viewLoader.LastLoadedView;
      Assert.IsTrue(form is PayrollWindow);
      Assert.IsTrue(form.Visible);
```

```
      PayrollWindow payrollWindow = form as PayrollWindow;
      PayrollPresenter presenter = payrollWindow.Presenter;
      Assert.IsNotNull(presenter);
      Assert.AreSame(form, presenter.View);
    }

    [Test]
    public void LoadAddEmployeeView()
    {
      viewLoader.LoadAddEmployeeView(
        new TransactionContainer(null));
      Form form = viewLoader.LastLoadedView;
      Assert.IsTrue(form is AddEmployeeWindow);
      Assert.IsTrue(form.Visible);
      AddEmployeeWindow addEmployeeWindow =
        form as AddEmployeeWindow;
      Assert.IsNotNull(addEmployeeWindow.Presenter);
    }
  }
}
```

Листинг 38.14. WindowViewLoader.cs

```
using Payroll;

namespace PayrollUI
{
  public class WindowViewLoader : ViewLoader
  {
    private readonly PayrollDatabase database;
    private Form lastLoadedView;

    public WindowViewLoader(PayrollDatabase database)
    {
      this.database = database;
    }

    public void LoadPayrollView()
    {
      PayrollWindow view = new PayrollWindow();
      PayrollPresenter presenter =
        new PayrollPresenter(database, this);
      view.Presenter = presenter;
      presenter.View = view;
      LoadView(view);
    }

    public void LoadAddEmployeeView(
      TransactionContainer transactionContainer)
    {
      AddEmployeeWindow view = new AddEmployeeWindow();
      AddEmployeePresenter presenter =
```

```
            new AddEmployeePresenter(view,
            transactionContainer, database);
          view.Presenter = presenter;
          LoadView(view);
        }

        private void LoadView(Form view)
        {
          view.Show();
          lastLoadedView = view;
        }

        public Form LastLoadedView
        {
          get { return lastLoadedView; }
        }
      }
    }
```

Листинг 38.15. PayrollWindowTest.cs

```
using NUnit.Framework;

namespace PayrollUI
{
  [TestFixture]
  public class PayrollWindowTest
  {
    private PayrollWindow window;
    private MockPayrollPresenter presenter;

    [SetUp]
    public void SetUp()
    {
      window = new PayrollWindow();
      presenter = new MockPayrollPresenter();
      window.Presenter = this.presenter;
      window.Show();
    }

    [TearDown]
    public void TearDown()
    {
      window.Dispose();
    }

    [Test]
    public void TransactionsText()
    {
      window.TransactionsText = “abc 123”;
      Assert.AreEqual(“abc 123”,
      window.transactionsTextBox.Text);
```

```
    }

    [Test]
    public void EmployeesText()
    {
      window.EmployeesText = "some employee";
      Assert.AreEqual("some employee",
      window.employeesTextBox.Text);
    }

    [Test]
    public void AddEmployeeAction()
    {
      window.addEmployeeMenuItem.PerformClick();
      Assert.IsTrue(presenter.addEmployeeActionInvoked);
    }

    [Test]
    public void RunTransactions()
    {
      window.runButton.PerformClick();
      Assert.IsTrue(presenter.runTransactionCalled);
    }
  }
}
```

Листинг 38.16. PayrollWindow.cs

```
namespace PayrollUI
{
  public class PayrollWindow : System.Windows.Forms.Form,
    PayrollView
  {
    private System.Windows.Forms.MainMenu mainMenu1;
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Label employeeLabel;
    public System.Windows.Forms.TextBox employeesTextBox;
    public System.Windows.Forms.TextBox transactionsTextBox;
    public System.Windows.Forms.Button runButton;
    private System.ComponentModel.Container components = null;
    private System.Windows.Forms.MenuItem actionMenuItem;
    public System.Windows.Forms.MenuItem addEmployeeMenuItem;
    private PayrollPresenter presenter;

    public PayrollWindow()
    {
      InitializeComponent();
    }

    protected override void Dispose( bool disposing )
    {
      if( disposing )
```

```
      {
        if(components != null)
        {
          components.Dispose();
        }
      }
      base.Dispose( disposing );
    }

    #region Windows Form Designer generated code
    // опущено
    #endregion

    private void addEmployeeMenuItem_Click(
      object sender, System.EventArgs e)
    {
      presenter.AddEmployeeActionInvoked();
    }

    private void runButton_Click(
      object sender, System.EventArgs e)
    {
      presenter.RunTransactions();
    }

    public string TransactionsText
    {
      set { transactionsTextBox.Text = value; }
    }

    public string EmployeesText
    {
      set { employeesTextBox.Text = value; }
    }

    public PayrollPresenter Presenter
    {
      get { return presenter; }
      set { presenter = value; }
    }
  }
}
```

Листинг 38.17. TransactionContainerTest.cs

```
using System.Collections;
using NUnit.Framework;
using Payroll;

namespace PayrollUI
{
  [TestFixture]
```

```
  public class TransactionContainerTest
  {
    private TransactionContainer container;
    private bool addActionCalled;
    private Transaction transaction;

    [SetUp]
    public void SetUp()
    {
      TransactionContainer.AddAction action =
        new TransactionContainer.AddAction(SillyAddAction);
      container = new TransactionContainer(action);
      transaction = new MockTransaction();
    }

    [Test]
    public void Construction()
    {
      Assert.AreEqual(0, container.Transactions.Count);
    }

    [Test]
    public void AddingTransaction()
    {
      container.Add(transaction);
      IList transactions = container.Transactions;
      Assert.AreEqual(1, transactions.Count);
      Assert.AreSame(transaction, transactions[0]);
    }

    [Test]
    public void AddingTransactionTriggersDelegate()
    {
      container.Add(transaction);
      Assert.IsTrue(addActionCalled);
    }

    private void SillyAddAction()
    {
      addActionCalled = true;
    }
  }
}
```

Листинг 38.18. TransactionContainer.cs

```
using Payroll;

namespace PayrollUI
{
  public class TransactionContainer
  {
```

```
    public delegate void AddAction();
    private IList transactions = new ArrayList();
    private AddAction addAction;

    public TransactionContainer(AddAction action)
    {
      addAction = action;
    }

    public IList Transactions
    {
      get { return transactions; }
    }

    public void Add(Transaction transaction)
    {
      transactions.Add(transaction);
      if(addAction != null)
        addAction();
    }

    public void Clear()
    {
      transactions.Clear();
    }
  }
}
```

Снимаем покрывало

Мы славно потрудились над системой расчета зарплаты, и наконец-то пришла пора посмотреть, как она будет выглядеть с новым графическим интерфейсом. В листинге 38.19 приведен класс `PayrollMain` – точка входа в приложение. Прежде чем загружать вид `Payroll`, нам необходим экземпляр базы данных. В данном случае мы создаем объект `InMemoryPayrollDatabase` – для демонстрации. В промышленной программе мы создали бы объект `SqlPayrollDatabase`, работающий с реальной базой данных под управлением SQL Server. Но приложение прекрасно работает и с объектом `InMemoryPayrollDatabase`, который хранит все данные в памяти.

Затем создается экземпляр `WindowViewLoader`, вызывается его метод `LoadPayrollView` – и работа начинается. Теперь мы можем откомпилировать приложение, запустить его и ввести столько работников, сколько душе угодно.

Листинг 38.19. PayrollMain.cs

```
using System.Windows.Forms;
using Payroll;
```

```
namespace PayrollUI
{
  public class PayrollMain
  {
    public static void Main(string[] args)
    {
      PayrollDatabase database =
        new InMemoryPayrollDatabase();
      WindowViewLoader viewLoader =
        new WindowViewLoader(database);
      viewLoader.LoadPayrollView();
      Application.Run(viewLoader.LastLoadedView);
    }
  }
}
```

Заключение

Джо будет счастлив, увидев, что мы для него сделали. Мы соберем промышленную версию и дадим ему поработать с ней. Разумеется, он скажет, что интерфейс убогий и неотшлифованный. Какие-то его особенности замедляют работу, что-то кажется неясным. Пользовательские интерфейсы вообще трудно сразу сделать правильно. Поэтому мы со всем вниманием отнесемся к его замечаниям и отправимся на следующий круг. На следующем этапе мы добавим действия для изменения данных о работнике, потом – для ввода карточек табельного учета и справок о продажах. И напоследок оставим собственно расчет зарплаты. Все это, конечно, вы сделаете самостоятельно.

Библиография

http://daveastels.com/index.php?p=5

www.martinfowler.com/eaaDev/ModelViewPresenter.html

http://nunitforms.sourceforge.net/

www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf

A

Сказ о двух компаниях

Я намереваюсь приобрести клюшку
и надавать тебе по башке.

Руфус Файрфлай (персонаж фильма «Утиный суп»)

Rufus Inc.
Проект Kickoff

Вас зовут Боб. Сегодня 3 января 2001 года, и у вас все еще болит голова после недавней пирушки по поводу наступления нового тысячелетия. Вы сидите в конференц-зале вместе с несколькими менеджерами и своими коллегами. Вы руководитель группы. Здесь же присутствует ваш шеф, который привел всех руководителей групп. Совещание созвал его начальник.

«Мы приступаем к работе над новым проектом», – говорит начальник вашего шефа. Назовем его НШ. Его пьедестал начальственности настолько высок, что он задевает головой потолок. Ваш шеф только начал строить свой пьедестал и с нетерпением ждет того времени, когда тоже

Rupert Industries:
Проект Alpha

Вас зовут Роберт. Сегодня 3 января 2001 года. Спокойно отпраздновав знаменательную дату в семейном кругу, вы чувствуете себя отдохнувшим и готовым к работе. Вы сидите в конференц-зале со своей командой профессионалов. Совещание созвал руководитель подразделения.

«У меня есть кое-какие мысли по поводу нового проекта», – начинает руководитель подразделения. Назовем его Расс. Это подтянутый британец, у которого энергии больше, чем у ядерного реактора. Он амбициозен и честолюбив, но при этом понимает ценность команды.

Расс рассказывает о сути новой рыночной ниши, выявленной компанией, и представляет вам Джейн,

сможет оставлять пятна бриолина на потолке.

НШ рассказывает о недавно обнаружившемся рынке сбыта и продукте, который компания хочет для него разработать.

«Проект должен быть готов к четвертому кварталу, 1 октября, – требует НШ. – Это высший приоритет, поэтому ваш текущий проект отменяется».

Все присутствующие ошарашенно молчат. Несколько месяцев работы просто выбрасываются в корзину. Но вот вокруг стола проносится шепоток возражений.

Глаза НШ загораются зловещим зеленым огнем. Он обводит взглядом присутствующих. Каждый, на ком останавливается этот устрашающий взгляд, превращается в дрожащий комок протоплазмы. Ясно, что никаких дискуссий не будет.

Дождавшись тишины, НШ говорит: «Начинать нужно немедленно. Сколько времени займет анализ?»

Вы тянете руку. Шеф пытается вас остановить, но брошенный им бумажный шарик летит мимо и вы не замечаете его попытку.

«Сэр, мы не можем сказать, сколько времени займет анализ, пока не получим требований».

«Техническое задание будет готово через 3–4 недели, – говорит НШ с явными нотками раздражения в голосе. – Но вы *представьте*, что требования уже есть. Сколько нужно на анализ?»

Все перестали дышать. Каждый оглядывается в надежде, что кого-то осенит.

менеджера по маркетингу, которая будет отвечать за продукты, нацеленные на этот рынок.

Обращаясь к вам, Джейн говорит: «Мы хотели бы определить характер продукта как можно скорее. Когда я могу встретиться с вашей командой?»

Вы отвечаете: «Мы закончим текущую итерацию нашего проекта к пятнице. А в промежутке можем уделить вам несколько часов. Затем мы заберем из команды несколько человек и отдадим их в ваше распоряжение. Набирать новых людей вместо них и для работы в вашей команде мы начнем немедленно».

«Отлично, – говорит Расс, – но я хотел бы, чтобы вы четко понимали, что на июльской выставке мы должны что-то продемонстрировать. Если там мы не сможем показать что-то значимое, то возможность будет упущена».

«Понимаю, – отвечаете вы. – Пока не знаю, что у вас на уме, но уверен, что к июлю что-нибудь будет готово. Только пока не могу сказать, что конкретно это будет. В любом случае вы с Джейн будете полностью контролировать все, чем заняты разработчики, так что не беспокойтесь – к июлю все важные функции, которые в принципе возможно реализовать к этому сроку, будут реализованы».

Расс довольно кивает. Он знает, как все будет происходить. Ваша команда всегда держала его в курсе дел и позволяла направлять разработку. Он абсолютно уверен, что команда в первую очередь сосредоточится на самом главном и выдаст продукт высочайшего качества.

* * *

«Если анализ затянется дольше 1 апреля, у нас будут проблемы. Вы сможете закончить анализ к этому сроку?»

Ваш шеф набрался мужества: «Мы что-нибудь придумаем, сэр!» Его пьедестал подрос на три миллиметра, а ваша голова стала болеть сильнее на две таблетки тайленола.

«Хорошо, – довольно улыбается НШ. – Ну а на проектирование сколько времени вам потребуется?»

«Сэр, – вступаете вы. Ваш шеф заметно бледнеет. Очевидно, он обеспокоен тем, что набранные миллиметры можно легко потерять. – Без анализа невозможно сказать, сколько времени займет проектирование».

НШ суровеет. «*ПРЕДСТАВЬТЕ*, что анализ уже выполнен! – говорит он, не сводя с вас взгляда крохотных, как бусинки, невыразительных глаз. – Сколько тогда времени уйдет на проектирование?»

Два тайленола уже не помогут. Ваш шеф в отчаянной попытке сохранить достигнутое мямлит: «Если на весь проект отведено всего шесть месяцев, то с проектированием нужно уложиться самое большее в три месяца».

«Рад, что вы согласны, Смизерс!» – говорит НШ, лучась от восторга. Ваш шеф расслабляется. Он понимает, что его три миллиметра в безопасности, и чуть погодя даже начинает тихонько мурлыкать рекламную песенку о бриолине.

НШ продолжает: «Итак, анализ будет готов к 1 апреля, проект – к 1 июля, и у вас еще остается три месяца на реализацию. Это совещание – яркий пример того, как работает наша новая политика консенсуса и доверия исполнителям. Ну а те-

«Итак, Роберт, – говорит Джейн на первом совещании, – как команда относится к разбиению на две части?»

«Нам будет не хватать друг друга, – отвечаете вы. – Но некоторые уже устали от последнего проекта и хотят переключиться на что-то другое. Так что вы нам там приготовили?»

Джейн лучится улыбкой. «Вы знаете, сколько проблем сейчас у наших заказчиков...» И в течение примерно получаса она описывает проблему и возможное решение.

«Секундочку, – отвечаете вы. – Тут надо кое-что прояснить». И вместе с Джейн вы обсуждаете, как могла бы работать система. Некоторые ее идеи не вполне оформились. Вы предлагаете возможные решения. Кое-что ей нравится. Обсуждение продолжается.

Всякий раз, как в обсуждении затрагивается новая тема, Джейн записывает на карточке пользовательскую историю. На каждой карточке изложен фрагмент того, что должна делать новая система. Количество карточек растет, она раскладывает их перед вами на столе. Вместе с Джейн вы выбираете то одну, то другую и делаете на них пометки. Карточки – это действенный механизм запоминания, позволяющий представить сложные, едва оформившиеся идеи.

В конце совещания вы говорите: «Хорошо, общее впечатление о том, что вы хотите, я получил. Теперь я переговорю с командой. Полагаю, они захотят немного поэкспериментировать с различными структурами базы данных и формата-

перь за работу. На следующей неделе у меня на столе должен лежать план по управлению качеством и персональное распределение обязанностей по улучшению качества. Да, не забудьте – для аудита качества в следующем месяце надо будет провести межотдельские совещания и представить отчеты».

«К черту тайленол, – думаете вы, возвращаясь в свой бокс. – Бурбон мне нужен».

Тут влетает взволнованный шеф со словами: «Слушай, какое совещание было! Думаю, мы потрясем мир этим проектом». Вы согласно киваете, испытывая чересчур сильное раздражение, чтобы что-то отвечать.

«Кстати, – продолжает шеф, – чуть не забыл». Он вручает вам 30-страничный документ. «Помнишь, на следующей неделе у нас оценка по методу SEI?[1] Так вот, это шпаргалка. Ее нужно прочитать, выучить наизусть, а потом уничтожить. Тут говорится, как отвечать на вопросы, которые будет задавать аудитор. А также о том, в какие помещения их можно водить, а от каких держать подальше. Мы твердо намерены заработать к июню уровень CMM 3[2]».

* * *

Вместе с коллегами вы приступаете к анализу нового проекта. Это трудно, так как требований нет. Но из 10-минутной вводной речи НШ этим злосчастным утром вы примерно *ми представления. На следующую встречу мы придем всей группой и начнем идентифицировать наиболее важные функции системы».*

Спустя неделю вновь образованная команда встречается с Джейн. Они раскладывают на столе карточки и начинают обсуждать некоторые детали системы.

Совещание протекает очень динамично. Джейн излагает истории в порядке их важности. Каждая подробно обсуждается. Разработчики хотят, чтобы истории были относительно небольшими, чтобы можно было оценить время на разработку и тестирование. Поэтому они все время просят Джейн разбивать одну историю на несколько меньших. Джейн важно, чтобы у каждой истории была ясная ценность для бизнеса, поэтому, разбивая их на части, она следит за соблюдением этого условия.

Истории постепенно накапливаются. Записывает их Джейн, но и разработчики оставляют свои замечания, когда считают нужным. Никто не пытается зафиксировать все сказанное; карточки для этого и не предназначены, они всего лишь средство оставить память о беседе.

По мере того как разработчики вникают в суть историй, они начинают записывать некоторые оценки. Пока что оценки грубые, но позво-

---

[1] Метод количественной оценки зрелости процесса разработки ПО, разработанный Институтом программной инженерии, входящим в состав университета Карнеги-Меллон. – *Прим. перев.*

[2] Capability Maturity Model – модель зрелости процесса разработки ПО. – *Прим. перев.*

представляете, что должен делать продукт.

Корпоративный процесс требует начинать с создания документа, в котором описываются прецеденты. Вы начинаете перечислять все прецеденты и рисовать овалы и человечков.

В команде разгораются философские дебаты. Есть разные мнения о том, какими отношениями соединять прецеденты: <<extends>> или <<includes>>. Создаются конкурирующие модели, но никто не знает, как их оценивать. Споры не затихают, полностью парализуя процесс.

Через неделю кто-то натыкается на сайт iceberg.com, где рекомендуется вообще отказаться от отношений <<extends>> и <<includes>> и заменить их на <<precedes>> и <<uses>>. В выложенных на этом сайте статьях Дона Сенгруа описывается некий метод под названием stalwart-analysis, который якобы обеспечивает пошаговое преобразование прецедентов в проектные диаграммы.

По этой новой схеме создаются новые конкурирующие модели прецедентов, но критериев их оценки по-прежнему нет. Продолжаем буксовать на месте.

Совещания по поводу прецедентов становятся все более эмоциональными, а не рациональными. Если бы не тот факт, что требований так и нет, вы были бы крайне огорчены отсутствием хоть какого-то прогресса.

Техзадание поступает 15 февраля. А потом еще 20 и 25 февраля и далее каждую неделю. Каждый новый вариант противоречит предыдущему. Очевидно, что исполнители из отдела маркетинга, которые писали этот *ляют Джейн представить, во что обойдется каждая история.*

В конце совещания становится ясно, что предстоит обсудить еще очень много историй. Но ясно и то, что самые важные истории уже затронуты и на их реализацию уйдет несколько месяцев. Джейн закрывает совещание, берет с собой карточки и обещает утром дать предложения по первому выпуску.

* * *

На следующее утро совещание возобновляется. Джейн отбирает пять карточек и кладет их на стол.

«По вашим оценкам на эти карточки требуется примерно одна неделя работы идеальной команды. На последней итерации предыдущего проекта одна неделя работы идеальной команды составила три реальных недели. Если мы сумеем реализовать эти пять историй за три недели, то сможем продемонстрировать их Рассу. Тогда он будет удовлетворен ходом работы».

Джейн давит. По неуверенности, написанной на ее лице, вы понимаете, что она сама это осознает. Вы отвечаете: «Джейн, это новая команда, работающая над новым проектом. Было бы самонадеянно ожидать, что наша скорость сравняется со скоростью предыдущей команды. Однако вчера днем я поговорил с командой и мы согласились, что начальная скорость, скорее всего, действительно составит одну идеальную неделю к трем реальным. Тут вам повезло».

«Но не забывайте, – продолжаете вы, – что оценки историй и скорость на данном этапе очень приблизительны. Мы будем знать боль-

документ, хотя и удостоились доверия, но к консенсусу не пришли.

А тем временем члены команды предложили еще несколько конкурирующих шаблонов для описания прецедентов. И каждый из них предлагает свой собственный чрезвычайно креативный способ затруднить хоть какое-нибудь продвижение вперед. Споры набирают силу.

1 марта Пруденс Патридженс, инспектору процесса, удается собрать все конкурирующие формы и шаблоны прецедентов в одну всеобъемлющую форму. Занимает она ни много ни мало 15 страниц. Зато Пруденс сумела втиснуть туда абсолютно все поля из всех предлагавшихся шаблонов. К форме прилагается инструкция по заполнению на 159 страницах. Все уже имеющиеся прецеденты надлежит переделать в соответствии с новым стандартом.

Вы мысленно изумляетесь тому, что теперь для ответа на вопрос «Что должна делать система, когда пользователь нажимает Return?» нужно заполнить 15-страничный бланк.

Корпоративный процесс (придуманный Л. Э. Оттом, знаменитым автором работы «Глобальный анализ: прогрессивная диалектика для разработчиков ПО») настаивает на том, чтобы выявить все первичные прецеденты, 87% вторичных и 36,274 % третичных, и только после этого можно считать анализ законченным и переходить к этапу проектирования. Вы понятия не имеете, что такое третичные прецеденты. Поэтому, стремясь не отступать от требований, вы пытаетесь отдать документ с описанием прецедентов на рецензию в отдел маркетинга

ше, когда спланируем итерацию, и еще больше, когда реализуем ее».

Джейн смотри на вас поверх очков, как будто хочет сказать: «Кто тут начальник, в самом-то деле?», но потом улыбается и говорит: «Ладно, не переживайте. Теперь я знаю, что нужно делать».

Затем Джейн выкладывает на стол еще 15 карточек и говорит: «Если у нас получится сделать вот это к концу марта, то мы сможем передать систему заказчикам для бета-тестирования. И получим от них отзывы».

Вы отвечаете: «Хорошо, итак, первая итерация определена и есть истории для следующих трех. Вот эти четыре итерации и составят первый выпуск».

«Но вы действительно сделаете эти пять историй за три недели?» – спрашивает Джейн.

«Точно сказать не могу – сознаетесь вы. – Давайте разобьем их на задачи и посмотрим, что мы имеем».

Итак, Джейн вместе с вами и всей командой следующие несколько часов занята разбиением пяти отобранных историй на более мелкие задачи. Разработчики быстро приходят к выводу, что некоторые задачи встречаются в нескольких историях, а у других задач есть общие черты, из чего, возможно, удастся извлечь выгоду. Видно, что в головах разработчиков уже зреет определенный дизайн. Время от времени завязываются короткие дискуссии и ребята рисуют на некоторых карточках UML-диаграммы.

Вскоре вся доска заполнена задачами, после завершения которых все

в надежде, что уж они-то знают все о третичных прецедентах.

Увы, маркетологи заняты переговорами с ребятами из поддержки продаж, на вас у них нет времени. Собственно, с момента запуска проекта вы так ни разу и не встретились с маркетологами, от которых исходит нескончаемый поток постоянно изменяющихся и противоречивых требований.

Пока одна команда бесплодно трудится над прецедентами, вторая разрабатывает модель предметной области. Она производит на свет бесконечные варианты UML-диаграмм. Каждую неделю модель перерабатывается. Члены команды не могут решить, какие стереотипы использовать: <<interfaces>> или <<types>>. Нет никакого согласия и по поводу применения и синтаксиса языка объектных ограничений OCL. Часть членов команды только что вернулась с 5-дневных курсов по катаболизму и теперь рисует невероятно подробные и загадочные диаграммы, которые больше никто постичь не в состоянии.

27 марта, за неделю до предполагаемого завершения анализа, у вас готова куча документов и диаграмм, но понимание задачи осталось на том же уровне, что и 3 января.

* * *

И тут происходит чудо.

* * *

В субботу 1 апреля вы из дома просматриваете свою почту. И обнаруживаете докладную записку, которую ваш шеф отправил НШ. В ней ясно говорится о том, что этап анализа завершен!

пять историй будут реализованы. Вы говорите: «А теперь давайте распределим эти задачи».

«Я займусь генерацией начальной базы данных, – говорит Пит. – Я это уже делал в последнем проекте, а этот не сильно отличается. Оцениваю в два своих идеальных рабочих дня».

«Ну тогда я беру вход в систему», – говорит Джо.

«Эх, где наша не пропадала, – говорит Элейн, самый младший член команды. – Я никогда не занималась ГИПами и хотела бы попробовать».

«О, нетерпеливая юность, – говорит мудрый Джо, незаметно подмигивая вам. – Ты могла бы помочь мне с этим, девушка-джедай». И обращаясь к Джейн: «Полагаю, на это уйдет три моих идеальных рабочих дня».

Так, один за другим, разработчики разбирают задачи и оценивают их в терминах своих идеальных рабочих дней. И вы, и Джейн знаете, что лучше позволить разработчикам выбирать себе задачи добровольно, а не навязывать их сверху. Вы также точно знаете, что не станете оспаривать сделанных оценок. Вы знаете своих людей и доверяете им. Вы уверены, что они будут работать с полной отдачей.

Разработчики знают, что не могут подписаться на больше идеальных рабочих дней, чем достигли на последней итерации, над которой работали. После того как кто-то исчерпал свою квоту, он уже не принимает участия в распределении задач.

Вы звоните шефу и укоризненно спрашиваете: «Как же ты мог сказать НШ, что мы закончили анализ?»

«А ты на календарь когда в последний раз смотрел? – отвечает он. – Сегодня 1 апреля!»

Скрытая в этой дате ирония не ускользнула от вас. «Но нам еще столько нужно продумать. Столько проанализировать! Мы даже не решили, на чем остановиться: на <<extends>> или <<precedes>>!»

«А где доказательства, что вы еще не готовы?» – нетерпеливо интересуется шеф.

«Ну...»

Он он дает договорить. «Анализировать можно бесконечно, но когда-то надо поставить точку. И поскольку точка была запланирована на сегодняшний день, то сегодня мы ее и поставим. Итак, в понедельник все материалы по анализу должны быть собраны и помещены в открытую папку. И сообщите об этой папке Пруденс, чтобы она могла зарегистрировать ее в системе СМ к полудню в понедельник. А потом приступайте к проектированию».

Повесив трубку, вы понимаете, что неплохо было бы завести в нижнем ящике стола бутылочку бурбона.

* * *

Чтобы отметить своевременное завершение фазы анализа, руководство устроило вечеринку. НШ произнес воодушевляющую речь о том, как важно доверять людям. А ваш шеф со своего пьедестала, подросшего еще на три миллиметра, поздравил всю команду с проявлением необычайного единства и коллективизма. А в конце вышел директор по

В конечном итоге каждый взял все, на что имел право. Однако на столе остались задачи.

«Я боялся, что это случится – говорите вы. – Что ж, Джейн, это означает лишь то, что мы взяли слишком много для первой итерации. Какие истории или задачи можно исключить?»

Джейн вздыхает. Она знает, что ничего другого не остается. Внеурочная работа в начале проекта – безумие; все проекты, где она пыталась на этом настаивать, оказались провальными.

Поэтому Джейн начинает исключать наименее важные функции. «Ладно, окно входа в систему нам пока не нужно. Можно просто считать, что пользователь уже вошел».

«Вот гадство! – восклицает Элейн. – А я так хотела это сделать».

«Терпение, попрыгунья, – говорит Джо. – Кто дождется, пока пчелы улетят из улья, тот не будет пробовать мед распухшими губами».

Элейн выглядит сконфуженной.

Все выглядят сконфуженными.

«Так... – продолжает Джейн. – Думаю, что мы можем обойтись без...»

Потихоньку список задач тает. Разработчики, лишившиеся задач, подписываются на те, что остались.

Переговоры не обходятся без трений. Несколько раз Джейн выказывает недовольство и нетерпение. В какой-то момент, когда страсти накалились особенно сильно, Элейн прорывает: «Я буду работать что есть сил, чтобы сэкономить побольше времени». Вы собираетесь возра-

информации и обрадовал всех сообщением о том, что аудит SEI прошел без сучка без задоринки и он благодарит всех за изучение и последующее уничтожение розданных руководств. Уровень 3 нам теперь гарантирован и будет оформлен к июню. (Ходят слухи, что менеджеры уровня НШ и выше получат солидные премии, как только SEI присвоит нам уровень 3.)

Недели летят, команда занята проектированием системы. Конечно, выясняется, что анализ, на который предположительно должен опираться проект, небезупречен. Нет, бесполезен. А если уж совсем откровенно, то хуже чем бесполезен. Но когда вы намекаете шефу, что неплохо было бы сделать шаг назад и еще поработать над анализом, чтобы подпереть слабые места, он отрубает: «Фаза анализа завершена. Сейчас разрешено только проектирование. Вот им и занимайся».

Так что вы со своей командой усердно трудитесь над проектом, сомневаясь в правильности анализа требований. Впрочем, все это не имеет значения, так как каждую неделю по-прежнему выпускаются новые версии технического задания, а маркетологи как не хотели, так и не хотят с вами встречаться.

Работа над проектом превращается в кошмар. К несчастью, ваш шеф недавно прочел книгу «Финишная прямая», в которой автор, Марк Детомазо, беспечно предложил доводить проектные документы до уровня кода.

«Но если работать на таком уровне детализации, – спрашиваете вы, – то почему бы сразу не писать код?»

зить ей, но тут, к счастью, Джо, глядя ей прямо в глаза, говорит: «Вступивший на стезю греха никогда не сойдет с нее».

В конечном итоге удается составить итерацию, приемлемую для Джейн. Это не совсем то, на что Джейн рассчитывала. На самом деле существенно меньше. Но это то, что команда, по собственному мнению, сумеет сделать за три недели. И наиболее важное из того, что Джейн намеревалась включить в эту итерацию, все-таки сохранено.

«Итак, Джейн, – говорите вы после того как все немного успокоились, – когда ждать от вас приемочных тестов?»

Джейн вздыхает. Это обратная сторона медали. Для каждой истории, реализованной разработчиками, Джейн должна предоставить комплект приемочных тестов, доказывающих, что все сделано правильно. И они потребуются команде задолго до окончания итерации, так как в них наверняка проявится различие в понимании системы Джейн и разработчиками.

«Я подготовлю несколько примеров тестовых сценариев сегодня, – обещает Джейн. – И потом каждый день буду приносить новые. Полный комплект будет у вас к середине итерации».

* * *

Итерация начинается в понедельник утром со споров по поводу Классов, Обязанностей и Кооперации. Часам к десяти все разработчики разбились на пары и с энтузиазмом принялись за кодирование.

«Да потому, что тогда это будет не проектирование, разве не понятно? А на фазе проектирования разрешено заниматься только проектированием!»

«Кроме того, – продолжает шеф, – мы только что купили для всей компании лицензию на Dandelion! Эта штука поддерживает "конструирование от начала до конца"! Вы должны будете загрузить туда все свои диаграммы. И она автоматически сгенерирует код! Да еще будет синхронизировать диаграммы с кодом!»

Шеф вручает вам яркую, завернутую в целлофан коробку с дистрибутивом Dandelion. Онемевшими руками вы берете ее и медленно бредете в свой бокс. Спустя 12 часов, пережив восемь падений системы, одно переформатирование диска и приняв восемь стопок «Бакарди 151», вы наконец установили программу на свой сервер. Вы грустно размышляете о неделе, которую всей команде придется потерять на курсах по обучению работе с Dandelion. Но потом улыбаетесь, приходя к выводу: «Любая неделя подальше отсюда – отличная неделя».

Команда выдает на гора одну проектную диаграмму за другой. Рисовать диаграммы в Dandelion очень трудно. Нужно правильно заполнить текстовые поля и безошибочно разбросать флажки по десяткам диалоговых окон, скрытых глубоко в лабиринтах интерфейса. А тут еще и проблема перемещения классов из одного пакета в другой...

Сначала диаграммы рисовались на основе прецедентов. Но требования меняются так часто, что прецеденты очень быстро теряют всякий смысл.

«А теперь, моя юная ученица, – обращается Джо к Элейн, – ты постигнешь великую тайну: как проектировать, начиная с тестов!»

«Ух ты, звучит заманчиво, – отвечает Элейн. – Ну и как вы это делаете?»

Джо сияет. Ясно, что он предвидел этот момент. «Ответствуй, что сейчас делает программа?»

«Как это? – недоумевает Элейн. – Ничего не делает, нет никакой программы».

«Ну, подумай о нашей задаче; можешь ты назвать что-нибудь, что программа должна делать?»

«Конечно, – говорит Элейн с уверенностью, присущей юности. – Первым делом она должна соединиться с базой данных».

«Ну а что потребно, дабы с базой соединишася?»

«Как вы странно говорите, – смеется Элейн. – Думаю, что нужно получить из какого-то реестра объект базы данных и вызвать его метод Connect()».

«Ага, проницательная юная волшебница. Истинно глаголешь ты, что объект нам потребен, коий кэшануть мог бы объект базы данных».

«А так разве можно сказать – кэшануть?»

«Можно, раз я говорю! Ну так какой мы могли бы написать тест для реестра объектов базы данных?»

Элейн вздыхает. Она понимает, что игру надо продолжать. «Мы должны создать объект базы данных и поместить его в реестр методом Store(). А потом достать из реестра мето-

Разгораются споры о том, какой паттерн использовать: Посетитель или Декоратор. Кто-то из разработчиков наотрез отказывается использовать Посетитель в любом виде, заявляя, что эту конструкцию нельзя считать по-настоящему объектно-ориентированной. Другой открещивается от множественного наследования, именуя его порождением дьявола.

Текущие совещания стремительно превращаются в дебаты о том, что называть объектной ориентацией, в чем разница между анализом и проектированием и когда нужно применять агрегирование, а когда ассоциацию.

Когда проходит половина срока, отведенного на проектирование, ребята из отдела маркетинга заявляют, что теперь видят систему по-другому. Новое техническое задание полностью переработано. Несколько крупных функций исключено, а взамен добавлена функциональность, которая, как им кажется, больше понравится пользователям.

Вы сообщаете шефу, что после таких изменений нужно заново анализировать и проектировать значительную часть системы. Но в ответ слышите: «Фаза анализа завершена. Сейчас разрешено только проектирование. Вот им и занимайся».

Вы предлагаете создать простенький прототип и показать его маркетологам и, быть может, даже потенциальным заказчикам. Но шеф твердит: «Фаза анализа завершена. Сейчас разрешено только проектирование. Вот им и занимайся».

Черт, черт, черт! Вы пытаетесь написать хоть какой-нибудь документ,

дом Get() и убедиться, что это тот же самый объект».

«О, хорошо сказано, младая фея!»

«А то!»

«Ну а теперь давай-ка напишем тест, который подтвердит твою идею».

«А разве не надо сначала написать объект базы данных и объект реестра?»

«О, сколь многое предстоит тебе еще познать, моя юная нетерпеливица. Давай все-таки сначала напишем тест».

«Но он даже не откомпилируется!»

«Ты уверена? А вдруг?»

«Гм...»

«Пиши тест, Элейн. Доверься мне». Вот так Джо, Элейн и все остальные занялись кодированием своих задач, по одному тесту за раз. В комнате слышны разговоры между партнерами. Бормотание иногда прерывается громким возгласом, когда пара заканчивает задачу или особенно трудный тест.

Разработчики меняют партнеров один или два раза в день. Каждый разработчик должен видеть, что делают все остальные, поэтому детали кода становятся известны всей команде.

Закончив что-то существенное – целую задачу или ее важную часть, – пара интегрирует сделанное с остальной системой. Поэтому объем готового кода ежедневно растет, а трудности интеграции сведены к минимуму.

Разработчики общаются с Джейн ежедневно. Они обращаются к ней со всеми вопросами о функциональ-

отражающий новые требования. Но произошедшая революция отнюдь не положила конец потоку изменений. Напротив, частота и амплитуда колебаний ТЗ только увеличились. Вы упорно пробиваетесь сквозь них.

15 июня слетает база данных Dandelion. Очевидно, к этому дело шло уже давно. Мелкие ошибки в базе накапливались месяцами, становясь все более и более серьезными. И в конце концов CASE-средство просто перестало работать. Разумеется, эта медленно развивавшаяся коррозия присутствует и во всех резервных копиях.

На обращение в техподдержку Dandelion несколько дней никто не реагирует. Наконец, вы получаете от них краткое послание с извещением о том, что это известная проблема и решение только одно – приобрести новую версию, которая, по их обещаниям, будет готова где-то в следующем квартале, а потом заново ввести все диаграммы вручную.

* * *

Однако 1 июля происходит еще одно чудо! Оказывается, проектирование завершено!

Вместо того чтобы идти к шефу с претензиями, вы затариваете средний ящик стола водкой.

* * *

Руководство отметило вечеринкой своевременное завершение фазы проектирования и присвоение уровня CMM 3. Вы догадываетесь, что на этот раз речь НШ будет настолько воодушевляющей, что лучше до ее начала посетить туалет.

На вашем рабочем месте развешаны новые флажки и плакаты с изображением горных орлов и покорителей

ности системы и интерпретации приемочных тестов.

Джейн, верная своему слову, снабжает команду неиссякаемым потоком приемочных тестов. Команда внимательно изучает их и, как следствие, лучше понимает, чего Джейн ожидает от системы.

К началу второй недели набирается достаточно материала, чтобы продемонстрировать Джейн. Она с интересом смотрит, как один тест сменяется другим.

«Здорово, правда, – говорит Джейн, когда демонстрация подходит к концу. – Но не похоже, чтобы тут была треть задачи. Что, скорость оказалась ниже, чем вы рассчитывали?»

Вы морщитесь. Вы хотели выбрать момент, чтобы потактичнее сказать об этом Джейн, но она сама подняла вопрос.

«Да, к сожалению, мы продвигаемся медленнее, чем ожидали. Новый сервер приложений, который мы используем, оказалось очень трудно сконфигурировать. Кроме того, он перегружается целую вечность, а это нужно делать при любом изменении конфигурации».

Джейн подозрительно глядит на вас. Напряжение понедельничных споров еще не улеглось. Она говорит: «А как это скажется на нашем графике? Мы не можем снова сорвать его, не можем, и все тут. Расса хватит удар! Он нас на части порвет».

Вы смотрите Джейн прямо в глаза. Преподносить такие новости не очень-то приятно. Но приходится: «Послушай, если все будет идти, как сейчас, то к следующей пятнице мы не успеем. Вполне возможно,

вершин и афоризмами о коллективной работе и доверии исполнителям. Читать эти тексты лучше после нескольких рюмок шотландского виски. Вы вспоминаете, что собирались вынести из шкафа лишнюю бумагу, чтобы освободить место для коньяка.

Команда приступает к кодированию. Но очень скоро обнаруживается, что в нескольких важных местах система недопроектирована. Вообще-то, она недопроектирована везде. Вы собираете совещание по проектированию в одном из конференц-залов, пытаясь решить хотя бы самые насущные проблемы. Однако шеф застает вас за этим занятием и разгоняет собрание со словами: «Фаза проектирования завершена. Сейчас разрешено только кодирование. Вот им и занимайся».

Сгенерированный Dandelion код ужасен. Выясняется, что вы все-таки ухитрились неправильно использовать ассоциацию и агрегирование. И теперь весь сгенерированный код приходится исправлять вручную. А это очень трудно, потому что код испещрен уродливыми комментариями со специальным синтаксисом, и избавиться от них нельзя, потому что Dandelion использует их для синхронизации диаграмм с кодом. Если, не дай бог, изменить хотя бы один комментарий, то диаграммы будут генерироваться неправильно. Получается, что это «конструирование от начала до конца» требует массы усилий.

Чем старательнее вы пытаетесь сохранить совместимость кода с Dandelion, тем больше ошибок она выдает. В итоге вы сдаетесь и решаете поддерживать актуальность диаграмм вручную. А еще через секунду

что мы найдем способ ускорить процесс. Но, честно говоря, я бы не очень рассчитывал на это. Надо начинать думать о том, чтобы убрать одну-две задачи, но так, чтобы не расстроить всю демонстрацию для Расса. Случись конец света или всемирный потоп, мы все равно должны провести демонстрацию в пятницу, и я полагаю, тебе не понравится, если мы сами выберем, какие задачи исключить».

«О, боже мой!» – Джейн едва удается не сорваться на крик, и она покидает комнату, качая головой.

Уже в который раз вы говорите себе: «А никто и не обещал, что руководить проектом легко». И вы уверены, что этот раз не последний.

* * *

На самом деле все складывается немного лучше, чем вы думали. Команде, правда, все же пришлось исключить из итерации одну задачу, но Джейн подошла к ее выбору с умом и демонстрация для Расса прошла без сучка без задоринки.

Расс не был особенно впечатлен, но и разочарования не выказал. Он просто сказал: «Неплохо. Но не забывайте, что мы должны продемонстрировать систему на июльской выставке, а при таком темпе вряд ли у нас будет что показать».

Джейн, чье отношение к проекту заметно улучшилось к концу итерации, вступилась за нас, сказав: «Расс, эта команда трудится усердно и работает хорошо. Я уверена, что в июле у нас найдется что продемонстрировать. Конечно, это будет не все, а кое-что окажется трюками с напусканием тумана, но что-то мы будем иметь».

приходите к выводу, что ну ее к черту, эту актуальность. Да и у кого на это есть время?

Шеф нанимает консультанта, поручая ему написать инструменты для подсчета строк создаваемого кода. На стену вешается здоровенный график с изображением шкалы термометра и числом 1 000 000 наверху. Каждый день красная полоса ползет вверх, отмечая, сколько строк добавлено.

Через три дня после появления термометра шеф останавливает вас в коридоре. «Что-то график растет недостаточно быстро. К 1 октября у нас должен быть миллион строк».

«Мы, эт-т-та, не уверены, ик, что в проекте ваще будет ми-ли-ми-он строк», – не вполне внятно произносите вы.

«К 1 октября миллион строк должен быть, – повторяет шеф. Его пьедестал еще подрос, и краска для волос, которой он пользуется, создает ауру властности и компетентности. – Ты уверен, что комментарии достаточно обширны?»

И тут же в припадке руководящего озарения восклицает: «Есть! Я хочу внедрить новую политику. Не должно быть строк кода длиннее 20 символов. Если строка длиннее, ее надо разбивать на две или больше – лучше больше. Весь написанный код следует переработать в соответствии с этим стандартом. Вот тогда счетчик строк подскочит вверх!»

Вы решаете не говорить ему, что на это уйдет два месяца незапланированной работы. Вы решаете вообще ничего не говорить. Вы решаете, что единственный выход – внутривенная инъекция чистого этилового

Как ни болезненна была последняя итерация, она помогла откалибровать скорости. На следующей итерации дела пошли намного лучше. Не потому, что команда сделала больше, чем на предыдущей итерации, а просто потому, что не пришлось исключать ни одной истории, ни одной задачи.

К концу четвертой итерации установился стабильный ритм. Джейн, вы и команда знают, чего ожидать друг от друга. Команда работает с полной отдачей, но взятый темп поддерживается без напряжения. Вы уверены, что в таком темпе сможете работать и год, и дольше.

Число неожиданностей в плане выполнения графика уменьшилось почти до нуля, чего не скажешь о сюрпризах в части требований. Джейн и Расс часто исследуют растущую систему и дают рекомендации или просят внести изменения в уже реализованную функциональность. Но все стороны понимают, что изменения требуют времени и их нужно включать в график. Поэтому изменения не обманывают ничьих ожиданий.

В марте проходит большая демонстрация для совета директоров. Система еще очень ограниченна и не готова для выставки, но прогресс налицо, и правление очень довольно.

Второй выпуск проходит даже более гладко, чем первый. К этому моменту команда нашла способ автоматизировать приемочные тесты Джейн. Кроме того, команда подвергла рефакторингу дизайн системы, так что добавлять новые функции и изменять старые стало гораздо проще.

спирта. И делаете соответствующие приготовления.

Черт, черт, черт и еще раз черт! Команда кодирует в бешеном темпе. К 1 августа шеф, с неудовольствием глядя на термометр, вводит обязательную 50-часовую рабочую неделю.

Черт, черт, черт и еще раз черт! 1 сентября термометр показывает 1,2 миллиона строк, и шеф предлагает письменно объяснить, почему бюджет кодирования превышен на 20 процентов. Он вводит черные субботы и требует уменьшить количество строк до миллиона. Вы начинаете кампанию по слиянию строк.

Черт, черт, черт и еще раз черт! Атмосфера накаляется, люди увольняются, ОТК засыпает вас извещениями об ошибках. Заказчики требуют руководства по установке и эксплуатации. Продавцы требуют проведения расширенных демонстраций для особо важных заказчиков. Техническое задание все еще окончательно не определено. Ребята из отдела маркетинга жалуются, что продукт ничуть не похож на то, что они специфицировали. А винный магазин больше не принимает вашу кредитку. Что-то должно произойти. На 15 сентября НШ назначает совещание.

Когда он входит в зал, его пьедестал испускает клубы пара. Стоит ему заговорить, как грозные нотки в тщательно контролируемом голосе начинают выворачивать ваш желудок наизнанку. «Начальник ОТК сообщил мне, что в проекте реализовано меньше половины требуемых функций. Он также проинформировал меня о том, что система постоянно сбоит, выдает странные результаты

Вторая версия выпущена к концу июня и поехала на выставку. В нее вошло меньше, чем хотелось бы Джейн и Рассу, но большинство наиболее важных функций системы продемонстрировано. Хотя побывавшие на выставке заказчики отметили отсутствие кое-каких частей, в целом впечатление было весьма благоприятным. Вы, Расс и Джейн покидаете выставку с улыбками на лице. Все чувствуют себя так, будто проект завоевал первый приз.

А много месяцев спустя к вам обратились представители компании Rufus Inc. Она работала над похожей системой для внутренних целей. Rufus прекратила разработку, не выдержав гонки, и ведет переговоры о лицензировании вашей технологии в применении к своим условиям.

Жизнь прекрасна!

и работает недопустимо медленно. Он также пожаловался, что не может далее мириться с бесконечным потоком ежедневных выпусков, каждый из которых содержит больше ошибок, чем предыдущий!»

Он делает секундную паузу, очевидно пытаясь взять себя в руки. «Начальник ОТК полагает, что при таком темпе разработки мы не сможем поставить продукт до декабря!»

Вы-то думаете, что не раньше марта, но предпочитаете помалкивать.

«Декабрь! – в крике НШ слышится такая издевка, что присутствующие втягивают голову в плечи, как будто он целится в них из ружья. – О декабре не может быть и речи. Руководители групп, утром у меня на столе должны лежать новые графики. Начиная с сегодняшнего дня вводится 65-часовая рабочая неделя вплоть до завершения проекта. И очень советую вам закончить к 1 ноября».

Слышно, как, выходя из зала, он бормочет себе под нос: «Вот тебе и доверие!»

* * *

Ваш шеф облысел; обломки его разлетевшегося пьедестала впечатались в стены кабинета НШ. Свет люминесцентных ламп, отражающийся от его лысины, на мгновение ослепляет вас.

«Выпить есть?» – спрашивает он. Поскольку вы как раз прикончили последнюю бутылочку «Бунс Фарм», то достаете из книжного шкафа бутылку «Сандербед» и наливаете в кофейную чашку. «Что нужно, чтобы доделать этот проект?» – интересуется он.

«Нужно заморозить требования, проанализировать их, составить

проект и реализовать его», – холодно отвечаете вы.

«К первому ноября? – не верит шеф. – Это невозможно! Давай возвращайся к кодированию этой хреновины». Он бушует, яростно скребя облысевшую голову.

Через несколько дней выясняется, что ваш шеф переведен в исследовательское подразделение. Все мгновенно меняется. Заказчики, которых в последнюю минуту проинформировали, что продукт не будет поставлен в срок, отменяют заказы. Ребята из отдела маркетинга задумались о том, а отвечает ли продукт долгосрочным целям компании. Летают докладные записки, катятся головы, изменяются политики, и вообще все выглядит довольно мрачно.

Наконец, к марту, после многих 65-часовых рабочих недель кряду, готова крайне нестабильная версия программы. Частота ошибок, обнаруженных во время эксплуатации, очень высока, техподдержка лезет из кожи вон, пытаясь справиться с потоком претензий и требований от раздраженных заказчиков. Всем несладко.

В апреле НШ решает положить конец этой проблеме, купив лицензию на продукт компании Rupert Industries с правом редистрибуции. Заказчики угомонились, маркетологи умолкли, а вас уволили.

B

Что такое проектирование программного обеспечения[1]

Я до сих пор помню, где именно меня посетило вдохновение, которое в итоге привело к написанию этой статьи. Летом 1986 года я работал временным консультантом в Центре вооружения ВМФ в Чайна Лейк, штат Калифорния. Там я воспользовался возможностью посетить круглый стол, посвященный языку Ada. В какой-то момент кто-то из присутствующих задал типичный вопрос: «А разработчики программного обеспечения инженеры?» Не помню, что конкретно ему ответили, но точно помню, что ответ был не по существу. Поэтому я откинулся на списку стула и стал думать, как бы я ответил на этот вопрос. Сейчас уже не могу сказать, что именно, но что-то в последовавшем обсуждении навело меня на мысль о статье в журнале *Datamation*, которую я читал лет за 10 до того. В статье обосновывалось мнение о том, что инженеры обязаны быть хорошими писателями (кажется, речь шла об этом, но времени-то прошло много), однако ключевой запомнившийся мне момент – отстаиваемая автором точка зрения, что конечным результатом процесса инженерного проектирования является документ. Иными словами, инженеры производят документы, а не вещи. Вещи производят другие люди на основе этих документов. И тогда в моем беспорядочном мозгу возник такой вопрос: «Во всей документации, обычно создаваемой в ходе работы над программным проектом, есть ли что-то такое, что можно было бы назвать инженерным документом?» И я решил, что да, такой документ есть, причем всего один – исходный код .

Взгляд на исходный код как на инженерный документ – проект – полностью перевернул мои представления о выбранной профессии. Я по-новому взглянул на все. И чем больше я размышлял об этом, тем силь-

[1] Jack Reeves, “What Is Software Design?” *C++ Journal*, 2(2), 1992: Перепечатывается с разрешения. ©Jack W. Reeves 1992.

нее убеждался в том, что это объясняет многочисленные проблемы, типичные для программных проектов. Точнее, я ощущал, что объяснение напрямую связано с непониманием или активным неприятием этой особенности со стороны большинства. Лишь через несколько лет мне представилась возможность публично изложить свои аргументы. Статья в журнале *The C++ Journal* навела меня на мысль написать редактору письмо на эту тему. Последовала переписка, и редактор, Ливлин Сингх, согласился опубликовать мои размышления по этому поводу в виде статьи. Результат предлагается вашему вниманию.

***

Джек Ривз, 22 декабря 2001

Похоже, объектно-ориентированные методики, и язык C++ в частности, покоряют мир программного обеспечения. Появились многочисленные статьи и книги о применении новых технологий. Вопрос о том, не является ли ООП очередной рекламной шумихой, сменился обсуждением, как извлечь из этой технологии максимум выгоды с минимальными усилиями. Объектно-ориентированные методики появились не вчера, но этот взрыв популярности представляется не совсем обычным. Откуда такой внезапный интерес? Предлагались самые разные объяснения. По правде говоря, нельзя назвать какую-то единственную причину. Возможно, сочетание различных факторов достигло критической массы и началась реакция. Тем не менее похоже, что самым значимым фактором на этом последнем этапе революции в мире ПО стал сам язык C++. У этого явления, наверное, тоже есть много причин, но я хочу взглянуть на него под несколько иным углом зрения: C++ приобрел популярность потому, что облегчает как проектирование ПО, так и программирование.

Возможно, мое замечание показалось вам странным, и это неслучайно. В этой статье я как раз и хочу рассмотреть связь между программированием и проектированием ПО. Уже почти 10 лет меня не оставляет ощущение, что индустрия программного обеспечения в целом не улавливает тонкого различия между разработкой проекта программы и тем, что на самом деле представляет собой такой проект. Я полагаю, что из растущей популярности C++ можно извлечь важный урок о том, что нам нужно сделать, чтобы стать лучшими инженерами-программистами. А заключается он в том, что суть программирования – не изготовление программного обеспечения, а его проектирование.

Несколько лет назад я присутствовал на семинаре, где обсуждался вопрос, является ли разработка ПО инженерной дисциплиной. Я не помню последовавшей дискуссии, но помню, что она стала катализатором для моих собственных размышлений о том, что в индустрии разработки ПО было создано много ложных параллелей с конструированием ап-

паратуры, тогда как некоторые совершенно законные параллели упущены из виду. По существу, я пришел к выводу, что нас нельзя назвать инженерами-программистами, потому что мы не осознаем, что же такое проектирование ПО. И сегодня это мое убеждение только окрепло.

Конечной целью любой инженерной деятельности является та или иная документация. По завершении проектирования вся проектная документация передается на производство. Там работают совершенно другие люди, обладающие принципиально иными навыками, чем проектировщики. Если проектная документация действительно полна, то производственники могут приступить к изготовлению изделия. И даже целых партий изделий – вмешательства проектировщиков уже не требуется. Проанализировав жизненный цикл разработки программного обеспечения так, как я его понимал, я пришел к выводу, что единственная документация, которая хоть как-то удовлетворяет критериям инженерного проектирования, – это листинги с исходным кодом.

Вероятно, аргументов за и против этого тезиса наберется на много статей. В данной же статье предполагается, что окончательный исходный код и есть настоящий программный проект, а затем рассматриваются некоторые следствия этого предположения. Возможно, я не сумею доказать правильность своей точки зрения, но надеюсь продемонстрировать, что она объясняет ряд наблюдаемых в индустрии программного обеспечения фактов, в том числе популярность C++.

У взгляда на код как на проект программы есть одно следствие, затмевающее все остальные. Оно настолько важно и очевидно, что для подавляющего большинства организаций, занимающихся программным обеспечением, оказывается мертвой зоной. Речь идет о том, что изготовление ПО стоит очень дешево. Его даже нельзя назвать недорогим; оно дешево настолько, что уже почти бесплатно. Если исходный код – это проект программы, то ее изготовлением занимаются компиляторы и компоновщики. Мы часто называем процесс компиляции и компоновки полной программной системы «сборкой». Капитальные вложения в оборудование для изготовления ПО низки – требуется лишь компьютер, редактор текстов, компилятор и компоновщик. Если все необходимое имеется, то для сборки программы требуется лишь немного времени. Может показаться, что компиляция программы на C++ из 50 000 строк занимает вечность, но подумайте, сколько времени ушло бы на изготовление аппаратной системы такой же сложности, как программа на C++ из 50 000 строк?

Еще одно следствие взгляда на код как на проект программы – тот факт, что создать проект программы относительно легко, по крайней мере, если говорить о механическом труде. На написание (то есть проектирование) типичного программного модуля длиной от 50 до 100 строк обычно уходит всего пара дней (его отладка – совсем другое дело, но об этом ниже). Так и хочется спросить, есть ли еще какая-нибудь инженерная дисциплина, позволяющая порождать проекты такой сложности, как

программы, в такое же короткое время. Однако сначала надо понять, как измерять и сравнивать сложность. Но в любом случае очевидно, что программные проекты довольно быстро становятся очень большими.

Учитывая, что программные проекты относительно легко выпускать, а изготовление ПО практически бесплатно, неудивительно, что проекты оказываются невообразимо большими и сложными. На первый взгляд факт тривиальный, но при этом часто игнорируют размер задачи. Учебные проекты часто насчитывают несколько тысяч строк кода. Есть открытые программные продукты размером 10 000 строк. Но мы давно уже прошли тот этап, когда простые программы могли представлять значительный интерес. Проекты типичных коммерческих продуктов состоят из сотен тысяч строк, а есть немало таких, где строк миллионы. К тому же проекты программ почти всегда постоянно изменяются. Даже если текущий проект насчитывает всего несколько тысяч строк кода, на протяжении жизни продукта могло быть написано во много раз больше.

Конечно, есть примеры проектов оборудования, не менее сложных, чем проекты программ, но отметим два факта, касающихся современного оборудования. Во-первых, сложные проекты оборудования вовсе не всегда свободны от ошибок – вопреки уверениям критиков ПО. Основные микропроцессоры поставлялись с логическими ошибками, мосты рушились, плотины прорывало, самолеты падали, а уж количество отзывов автомобилей и других потребительских продуктов и вовсе исчисляется сотнями; и все эти факты, еще не изгладившиеся из памяти, – результат ошибок проектирования. Во-вторых, сложным проектам оборудования соответствует не менее сложное и дорогостоящее производство. В результате количество компаний, способных производить такие системы, ограничено располагаемыми ресурсами. В области программного обеспечения таких ограничений нет. Существуют сотни организаций, занимающихся выпуском программ, и тысячи очень сложных программных систем. Их количество и сложность ежедневно возрастают. Это означает, что индустрия программного обеспечения вряд ли сможет отыскать решения своих проблем, подражая разработчикам оборудования. Скорее уж наоборот, поскольку системы автоматизированного проектирования и производства помогают конструкторам оборудования создавать все более сложные проекты, то это проектирование оборудования становится все более похожим на проектирование ПО.

Проектирование ПО – это задача на управление сложностью. Сложность свойственна самому проекту ПО, организации отдельной компании и индустрии в целом. Проектирование ПО очень напоминает проектирование систем. В проекте могут применяться различные технологии, а зачастую и различные подотрасли знаний. Для спецификаций ПО характерна подвижность, они быстро и часто изменяются, обычно еще до завершения процесса проектирования. Команды разработчиков ПО также текучи, люди нередко сменяются в ходе процесса. Все это де-

лает проектирование ПО трудной и подверженной ошибкам процедурой. Эти мысли не новы, за те без малого 30 лет, что продолжается революция в деле инженерного проектирования ПО, разработка программ по-прежнему видится недисциплинированным искусством по сравнению с другими инженерными профессиями.

Общее мнение таково, что, когда настоящий инженер завершает проект – любой сложности, – он уверен, что изделие будет работать. Кроме того, он уверен, что изделие можно изготовить, применяя общепринятые методы производства. Для достижения такой уверенности проектировщики оборудования тратят много времени на перепроверку и доводку проекта. Возьмем, к примеру, проект моста. Перед тем как его возводить, инженеры рассчитывают прочность конструкции, строят компьютерные модели, создают выполненные в масштабе макеты и продувают их в аэродинамической трубе или испытывают иным способом. Короче говоря, проектировщики делают все возможное, чтобы удостовериться в качестве проекта еще до того, как приступить к его изготовлению. Конструирование нового самолета еще более трудоемко; тут нужно построить прототип в натуральную величину и испытать его в воздухе, подтвердив тем самым заложенные в проект характеристики.

Большинству людей кажется очевидным, что проекты программ не проходят такого же строгого контроля. Однако если рассматривать исходный код как проект, то мы увидим, что инженеры-программисты на самом деле прикладывают немало сил к проверке и уточнению своих проектов. Правда, мы называем это не инженерным проектированием, а тестированием и отладкой. Люди по большей части не склонны считать тестирование и отладку настоящей «инженерной деятельностью», уж во всяком случае не в индустрии программного обеспечения. Причина скорее в отказе индустрии ПО согласиться с тем, что код есть проект, нежели в каких-то реальных инженерных различиях. Макеты, прототипы и лабораторные образцы – общепринятая составная часть всех прочих инженерных дисциплин. Проектировщики программ не пользуются более формальными методами проверки проектов просто из-за экономики цикла изготовления ПО.

Открытие номер два: дешевле и проще всего воплотить проект в жизнь и протестировать его. Неважно, сколь раз придется повторить процесс изготовления, – с точки зрения времени он почти ничего не стоит, а затраченные на неудачный проект ресурсы полностью восстановимы. Отметим, что тестирование – это не только доказательство правильности проекта, но и часть его уточнения. Инженеры, конструирующие сложные аппаратные системы, часто строят модели (или хотя бы визуально воспроизводят их средствами машинной графики). Это позволяет им «почувствовать» те аспекты проекта, которые невозможно уловить путем одного лишь анализа документации. При проектировании программ такое моделирование невозможно и не нужно. Мы просто изготавливаем само изделие. Даже если бы системы формального доказательства правильности программ были так же автоматизированы, как

компиляторы, мы все равно не отказались бы от цикла сборки и тестирования. Следовательно, формальные доказательства никогда не представляли особого практического интереса для индустрии программного обеспечения.

Это реалии сегодняшнего процесса разработки ПО. Все более сложные программные проекты создаются все большим числом людей и организаций. Проекты кодируются на каком-то языке программирования, а затем проверяются и уточняются в цикле сборки/тестирования. Этот процесс изначально подвержен ошибкам и не отличается особой строгостью. Тот факт, что очень многие разработчики программ не желают поверить, что именно так он и работает, безмерно осложняет задачу.

В большинстве современных процессов разработки ПО делается попытка разложить различные фазы проектирования программы по полочкам. Прежде чем приступать к написанию кода, необходимо завершить и заморозить проект верхнего уровня. Для искоренения ошибок конструирования необходимы тестирование и отладка. Посередине находятся программисты, рабочие-сборщики индустрии ПО. Многие полагают, что если бы удалось заставить программистов покончить с «трюкачеством» и «наладить производство» в соответствии с полученным проектом (и при этом делать меньше ошибок), то разработка ПО могла бы превратиться в настоящую зрелую инженерную дисциплину. Но это вряд ли произойдет, поскольку такой процесс полностью игнорирует инженерные и экономические реалии.

Например, в какой еще современной отрасли промышленности терпима стопроцентная переделка в процессе изготовления? Рабочий-сборщик, не способный собрать изделие с первого раза, скоро потеряет работу. В производстве ПО даже мельчайший фрагмент кода вполне может быть подвергнут ревизии и полному переписыванию в процессе тестирования и отладки. Мы готовы смириться с такого рода уточнением в ходе творческого процесса проектирования, но не в процессе производства. Никто не ждет, что инженер с первого раза создаст идеальный проект. А даже если такое и случится, все равно проект должен пройти стадию доводки, дабы доказать, что он действительно идеален.

Если уж заимствовать что-то из японских методик управления, то прежде всего то, что наказание рабочих за ошибки приводит к обратному результату. Вместо того чтобы силой заставлять разработчиков придерживаться неверной модели процесса, мы должны пересмотреть сам процесс и сделать так, чтобы он помогал, а не препятствовал созданию более качественного ПО. Это лакмусовая бумажка в инженерном проектировании ПО. Когда мы говорим «инженерный», то имеем в виду организацию процесса, а не то, что для получения окончательного проектного документа необходима САПР.

Основная проблема разработки ПО заключается в том, что все является частью процесса проектирования. Кодирование – это проектирование; тестирование и отладка – тоже часть проектирования. И то, что

мы привычно именуем проектированием ПО, – всего лишь часть проектирования. Да, изготовление программного обеспечения обходится очень дешево, зато его проектирование невероятно дорого. ПО настолько сложно, что существует множество аспектов проектирования, а значит, и способов взглянуть на него. Проблема в том, что все эти аспекты взаимосвязаны (как, впрочем, и в проектировании оборудования). Как было бы здорово, если бы проектировщики верхнего уровня могли игнорировать детали алгоритмов модулей. Или если бы программисты могли забыть о проекте верхнего уровня, когда проектируют внутренние алгоритмы модуля. Увы, аспекты одного уровня проекта проникают в другие. Выбор алгоритмов работы отдельного модуля может оказаться для общего успеха системы столь же важным, сколь и любая из сторон проекта верхнего уровня. Не существует иерархии важности аспектов проекта программы. Ошибка в проекте модуля самого нижнего уровня может оказаться такой же фатальной, как и на самом верхнем уровне. Проект программы должен быть полным и правильным во всех отношениях, иначе программа, построенная по этому проекту, будет содержать ошибки.

Чтобы справиться со сложностью, программы проектируют по уровням. Занимаясь детальным проектированием одного модуля, программист не может одновременно держать в поле зрения сотни других модулей и тысячи разнообразных деталей. Например, существуют важные аспекты проекта ПО, не укладывающиеся в категории алгоритмов и структур данных. В идеале программист не должен думать об этих аспектах, когда проектирует код.

Но в действительности все работает не так, и теперь мы начинаем понимать, почему. Проект программы нельзя считать законченным, пока он не закодирован и не протестирован. Тестирование – это неотъемлемая часть процесса проверки и уточнения проекта. Проект структуры верхнего уровня – еще не полный проект программы, а лишь каркас, на который насаживается детальный проект. Наши возможности строго подтвердить правильность проекта верхнего уровня крайне ограничены. Детальный проект в конечном счете влияет (по крайней мере, это не следует запрещать) на проект верхнего уровня точно так же, как все остальные факторы. Уточнение всех аспектов проекта – это процесс, который должен пронизывать весь цикл проектирования. Если какой-то аспект проекта замораживается и исключается из процесса уточнения, то следует ли удивляться, что окончательный проект получился плохим или даже неработоспособным?

Было бы прекрасно, если бы проектирование верхнего уровня ПО можно было сделать более точным инженерным процессом, но реальность мира программных систем далека от точности. Программное обеспечение слишком сложно и зависит от слишком многих внешних факторов. Бывает, что оборудование работает не совсем так, как предполагали проектировщики. Бывает, что в библиотечной процедуре есть недокументированное ограничение. С такого рода проблемами рано или

поздно сталкивается любой программный проект. И выявляются они только при тестировании, просто потому, что выявить их раньше негде. А когда проблема обнаружена, приходится вносить изменения в проект. Если нам повезло, то изменения будут локальны. Но чаще рябь от них расходится по значительной части всего проекта (закон Мэрфи). Если какую-то часть проекта по той или иной причине изменить невозможно, то прочие части приходится подстраивать. Нередко это приводит к тому, что руководители воспринимают как «трюкачество», но таковы реалии разработки программ.

Например, недавно я работал над проектом, где между модулями A и B была обнаружена временная зависимость. К сожалению, внутренний механизм модуля A был скрыт абстракцией, не позволявшей вызывать модуль B в нужной последовательности. Естественно, к моменту обнаружения этой проблемы было уже поздно изменять абстракцию модуля A. Как и следовало ожидать, далее последовал сложный набор «исправлений» внутренней структуры A. Еще до того, как мы закончили работу над версией 1, у всех появилось чувство, что проект разваливается. Каждое новое исправление грозило разрушить предыдущие. Это нормальное явление в проекте разработки ПО. В конце концов мы с коллегами решились изменить проект, но, чтобы уговорить руководство, пришлось согласиться на неоплачиваемую переработку.

Такие проблемы гарантированно возникают в любом программном проекте типичного размера. Как ни старайся, что-то существенное обязательно упустишь. Здесь мы видим разницу между ремеслом и инженерным подходом. Опыт может подсказать нужное направление. Это ремесло. Но идти, полагаясь на опыт, можно лишь до тех пор, пока не вступишь на неизведанную территорию. А дальше нужно взять то, с чего мы начали, и улучшить его путем контролируемых уточнений. Это инженерный подход.

Небольшое замечание – все программисты знают, что проектная документация, составленная после, а не до написания кода, гораздо точнее. Теперь понятна и причина этого. Лишь окончательный проект, будучи отраженным в коде, является уточненным в ходе цикла сборка/тестирование. Вероятность того, что первоначальный проект останется неизменным на протяжении этого цикла, обратно пропорциональна количеству модулей и программистов, участвующих в проекте. И она очень быстро становится неотличимой от нуля.

В инженерном проектировании ПО нам отчаянно нужен хороший проект на всех уровнях. И в особенности – качественный проект верхнего уровня. Чем он лучше, тем проще будет заниматься детальным проектированием. Проектировщики должны использовать все, что может помочь. Структурные диаграммы, диаграммы Буча, таблицы состояний, языки описания проекта и т. д. – если это помогает, пользуйтесь. Но нужно иметь в виду, что сами эти инструменты и нотационные системы не есть проект программы. Рано или поздно нам предстоит создать ис-

тинный проект, и он будет выражен на языке программирования. Поэтому не бойтесь кодировать проект, когда он начинает вырисовываться. Просто нужно быть готовым к уточнению по мере необходимости.

Пока что не изобретено проектной нотации, которая была бы одинаково применима к проектам верхнего и детального уровней. В конечном итоге проект всегда сводится к какому-то языку программирования. Это означает, что нотация проекта верхнего уровня должна быть переведена на язык программирования еще до начала детального проектирования. Перевод может потребовать времени и внести ошибки. Вместо того чтобы транслировать нотацию, которая плохо ложится на выбранный язык, программисты часто возвращаются к требованиям и переделывают проект верхнего уровня, попутно кодируя его. И это тоже реалии разработки ПО.

Вероятно, лучше изначально дать проектировщикам возможность писать код, чем нанимать кого-то, кто переведет независимый от языка проект на язык программирования. Что нам нужно, так это унифицированная нотационная система, пригодная для всех уровней проектирования. Другими словами, нам нужен язык программирования, способный передавать также проектные идеи. Вот тут-то и вступает в игру C++. Это язык программирования, пригодный для создания реальных программ, но также и выразительный язык проектирования ПО. На C++ можно непосредственно передать высокоуровневую информацию о компонентах проекта. Это упрощает как первоначальное составление проекта, так и его дальнейшее уточнение. Наличие строгой проверки типов также помогает выявлять ошибки проектирования. Тем самым мы получаем более надежный проект, или, если хотите, лучше инженерно проработанный проект.

Рано или поздно проект программы должен быть представлен на некоем языке программирования, а затем проверен и уточнен с помощью цикла сборки/тестирования. Притворяться, что это не так, просто глупо. Подумайте сами, какие средства и методы разработки программ получили наибольшее распространение. В свое время прорывом считалось структурное программирование. Его популяризовал язык Pascal, который и сам приобрел популярность. Объектно-ориентированное проектирование – новое повальное увлечение, и в центре него находится C++. А что не сработало? CASE-средства? Популярны, но не универсальны. Структурные диаграммы? То же самое. А также и диаграммы Уорнера–Орра, диаграммы Буча, диаграммы объектов – список можете продолжить сами. У каждой технологии есть свои сильные стороны и один общий недостаток – это не настоящий проект программы. На самом деле единственной нотацией для проектирования ПО, которую можно назвать распространенной, является язык PDL и все, что на него похоже.

Это говорит о том, что коллективное подсознательное индустрии ПО инстинктивно сознает, что совершенствование техники программиро-

вания и особенно языков программирования для реальных задач неизмеримо важнее всего остального в отрасли. Это также говорит о том, что программисты заинтересованы в проектировании. Когда появятся более выразительные языки программирования, разработчики примут их с радостью.

Посмотрим также на изменение процесса разработки ПО. Когда-то был популярен процесс водопада. Теперь мы говорим о спиральной разработке и быстром прототипировании. Хотя для обоснования этих методик в ходу такие выражения, как «снижение рисков» или «сокращение сроков поставки продукта», на самом деле в их основе лежит завуалированное желание начать кодирование на ранних стадиях жизненного цикла. И это хорошо. Это позволяет быстрее приступить к проверке и уточнению проекта с помощью цикла сборки/тестирования. Заодно повышается вероятность того, что проектировщики, разработавшие проект верхнего уровня, никуда не делись и могут принять участие в детальном проектировании.

Выше уже отмечалось, что инженерный подход больше относится к организации процесса, а не к тому, как выглядит конечный продукт. Все мы, занимающиеся программным обеспечением, близки к инженерам, но нам необходимо несколько изменить собственное восприятие. Программирование и цикл сборки/тестирования – это основа процесса инженерного проектирования ПО. И управлять ими следует соответственно. Экономика цикла сборки/тестирования в сочетании с тем фактом, что программная система может представить практически все что угодно, делают крайне маловероятной возможность отыскания универсального метода проверки программного проекта. Этот процесс можно усовершенствовать, но уйти от него не удастся.

И последнее: цель любого инженерного проектирования – получение некоторой документации. Понятно, что наиболее важны сами проектные документы, но ими дело не должно ограничиваться. Ведь кто-то же будет пользоваться программой. И весьма вероятно, что впоследствии система будет модифицироваться и совершенствоваться. И, следовательно, для программного проекта вспомогательная документация не менее важна, чем для «железного». Даже если забыть пока о руководстве пользователя, инструкции по установке и прочих документах, напрямую не связанных с процессом проектирования, остается еще две важных задачи, которые должны быть решены с помощью вспомогательных проектных документов.

Во-первых, во вспомогательных документах отражается важная информация о предметной области, которая не вошла непосредственно в проект. Проектирование ПО – это создание программных моделей для описания предметной области. При этом вырабатывается понимание концепций, присущих предметной области. Обычно такое понимание включает сбор информации, которая напрямую не отражается в программной модели, но тем не менее помогает проектировщику выделить

наиболее важные концепции и решить, как лучше представить их в модели. Эту информацию необходимо где-то сохранить на случай, если впоследствии модель придется изменить.

Во-вторых, вспомогательные документы нужны для того, чтобы документировать те аспекты проекта – как верхнего, так и нижнего уровня, – которые трудно извлечь из самого проекта. Многие такие аспекты лучше всего изображать графически, что затрудняет их включение в виде комментариев к исходному коду. Это не значит, что графическая проектная нотация лучше языка программирования. Относитесь к графическим комментариям как к пояснительным надписям на чертежах оборудования. Никогда не забывайте, что именно исходный код, а не вспомогательная документация, определяет, что в действительности делает проект. В идеале хотелось бы иметь программные средства, способные генерировать вспомогательные документы на основе исходного кода, но не будем требовать слишком многого. Пусть будут хотя бы инструменты, позволяющие программисту (или техническому писателю) извлекать из исходного кода определенную информацию для дальнейшего документирования. Без сомнения, поддерживать такую документацию в актуальном виде вручную нелегко. И это еще один аргумент в пользу более выразительных языков программирования, а также в пользу сведения вспомогательной документации к минимуму и окончательного оформления ее на как можно более поздней стадии проекта. И конечно же, хорошо бы располагать более совершенными инструментами, чтобы не скатиться к карандашу, бумаге и грифельной доске.

Подведем итоги:

- Реальная программа выполняется на компьютере. Это последовательность нулей и единиц, записанная на каком-то магнитном носителе. Это не листинг программы на C++ (или каком-то другом языке программирования).
- Листинг программы – это проектный документ. А сам проект изготавливается компилятором и компоновщиком.
- Изготовить реальную программу очень дешево и по мере увеличения быстродействия компьютеров становится все дешевле.
- Спроектировать реальную программу безумно дорого, поскольку программы невероятно сложны и практически все этапы создания программного продукта – это части процесса проектирования.
- Программирование – это разновидность проектирования. Хорошие проектировщики поняли это и, не колеблясь, кодируют, когда необходимо.
- Кодировать имеет смысл чаще, чем многим кажется. Зачастую процесс представления проекта в коде выявляет упущения и влечет за собой дополнительное проектирование. Чем раньше это случится, тем лучше окажется проект.

- Коль скоро изготавливать ПО так дешево, формальные инженерные методы проверки работоспособности проектов не слишком востребованы. Гораздо легче и дешевле изготовить, а потом протестировать проект, чем тратить усилия на доказательство его корректности.
- Тестирование и отладка – эквивалент проверки и уточнения проекта в других инженерных дисциплинах. В правильном процессе разработки важность этих этапов не принижается.
- Есть и другие виды проектирования ПО – можете называть их проектированием верхнего уровня, проектированием модулей, структурным проектированием, проектированием архитектуры или както еще. В правильном процессе проектирования они учитываются и осознанно включаются соответствующие шаги реализации.
- Все виды проектирования взаимосвязаны. Правильный процесс разработки предусматривает – порой радикальные – изменения проекта, если на каком-то этапе проектирования выясняется, что это необходимо.
- В процессе проектирования потенциально полезны разнообразные нотационные системы – в качестве вспомогательной документации или средства для упрощения проектирования. Однако они не являются частью проекта программы.
- Разработка ПО по-прежнему является скорее ремеслом, нежели инженерной дисциплиной. В основном это объясняется недостатком строгости в критически важных процессах проверки и улучшения проекта.
- В конечном счете достижения в области разработки ПО зависят от успешного развития методов программирования, что, в свою очередь, означает дальнейшее совершенствование языков программирования. C++ можно считать таким совершенствованием. Он завоевал популярность потому, что оказался основным языком программирования, непосредственно поддерживающим проектирование ПО.
- C++ – шаг в правильном направлении, но этого недостаточно.

***

Послесловие

Перечитывая написанное мной почти 10 лет назад, я хотел бы отметить несколько моментов. Во-первых (и это имеет самое непосредственное отношение к данной книге), сегодня я даже сильнее, чем прежде, убежден в фундаментальной справедливости высказанных тогда основных положений. Мое убеждение подкрепляется рядом завоевавших широкую популярность разработок, которые появились за прошедшие с тех

пор годы и подтверждают многие мои мысли. Самое очевидное (и, пожалуй, наименее важное) явление – это распространение объектно-ориентированных языков программирования. Ныне есть много ОО-языков помимо C++. И появились различные нотации для объектно-ориентированного проектирования, например UML. Мое полемическое утверждение о том, что объектно-ориентированные языки завоевали признание, поскольку предложили более выразительные средства представления проекта непосредственно в коде, теперь кажется не слишком актуальным.

Идея рефакторинга – изменения структуры кода с целью сделать его более надежным, повторно используемым и т. д. – также стоит в одном ряду с моим мнением о том, что все аспекты проекта должны быть гибкими и допускать изменение в процессе его проверки. Рефакторинг – это просто некоторый процесс и набор рекомендаций о том, как улучшить проект, в котором обнаружились слабые места.

Наконец, появилась новая концепция гибкой разработки. Наиболее известная в этом ряду методика – экстремальное программирование (eXtreme Programming), но у всех подходов есть общая особенность: понимание того, что исходный код – самый главный продукт разработки ПО.

С другой стороны, есть ряд моментов (некоторые из них были затронуты в статье), значимость которых в моих глазах возросла. Во-первых, это важность архитектуры, или проекта верхнего уровня. В своей статье я отмечал, что архитектура – лишь одна из сторон проекта, которая должна оставаться подвижной по мере того, как проект поверяется циклом сборки/тестирования. По существу, это положение остается справедливым, но, оглядываясь назад, я полагаю, что был несколько наивен. Хотя цикл сборки/тестирования и может выявить архитектурные проблемы, большая их часть обычно обнаруживается в результате изменения требований. Проектировать ПО «в целом» трудно, и ни новые языки программирования типа Java или C#, ни графическая нотация типа UML не помогут человеку, который не знает, как за это взяться. Более того, когда уже написано достаточно много кода, опирающегося на определенную архитектуру, ее кардинальное изменение зачастую равносильно переписыванию всего проекта заново, а на это, конечно, рассчитывать не приходится. Даже организации, которые в принципе поддерживают идею рефакторинга, с большой неохотой идут на изменения, напоминающие полное переписывание. Отсюда следует, что важно делать правильно (или хотя бы почти правильно) с первого раза, причем тем важнее, чем крупнее задача. К счастью, в этом плане немалую помощь могут оказать паттерны проектирования.

Из других вопросов, которым следовало бы уделить больше внимания, отмечу вспомогательную документацию, особенно относящуюся к архитектуре. Пусть даже исходный код и является проектом, вычленить из него архитектуру – занятие почти непосильное. В статье я выражал

надежду на появление программных средств для автоматической генерации вспомогательных документов из исходного кода. С тех пор я практически отказался от этой идеи. Хорошую объектно-ориентированную архитектуру обычно можно выразить с помощью немногих диаграмм и нескольких десятков страниц текста. Но эти диаграммы и текст должны описывать лишь основные классы и отношения между ними. К сожалению, я не питаю надежд на то, что программные инструменты когда-либо станут настолько интеллектуальными, что смогут вычленить наиболее важные аспекты из массы деталей, скрытых в исходном коде. А значит, составлять и поддерживать такую документацию предстоит людям. Но я по-прежнему считаю, что готовить ее лучше после написания исходного кода или, по крайней мере, параллельно с кодированием, но никак не заранее.

Наконец, в заключительной части статьи я отметил, что язык C++ усовершенствовал искусство программирования, а значит, и проектирования, но его одного недостаточно. Поскольку новые языки, бросившие вызов популярности C++, на мой взгляд, не обеспечивают сколько-нибудь существенных продвижений в искусстве программирования, то это замечание сегодня даже более актуально, чем тогда.

Джек Ривз, 1 января 2002 года

Алфавитный указатель

A

Agile Alliance, 38
API сторонние, 571
A (абстрактность), метрика, 475, 499

C

CASE-средства, 237
Ca (входящие связи), 470
Ce (исходящие связи), 470, 499
CoffeeMaker, класс, 298
  абстракции, 303, 310
  объектно-ориентированное решение, 318
  плохое решение, 301
  пользовательский интерфейс, 306
  спецификации, 298
  улучшенное решение, 305
Copy, программа, 146

D

D, метрика удаленности от главной последовательности, 478, 500

E

Eiffel, язык, 181
enterSub, действие, 244
enterSuper, действие, 244
exitSub2, событие, 244
exitSuper, событие, 244

F

FitNesse, инструмент, 74

G

GeneratePrimes, программа
  автономное тестирование, 79
  версия 1, 77
  версия 2, 80
  версия 3, 82
  версия 4, 83
  версия 5, 84
  окончательная версия, 85
  рефакторинг, 80
  тестирование, 88

H

H (сцепленность по связям), метрика, 499

I

I (неустойчивость), метрика, 500

N

Null-объект, паттерн
  для начисления платы за услуги, 419
  описание, 385

P

PayrollDatabase, класс
  изъян в дизайне, 649
  и класс DeleteEmployeeTransaction, 412
  сохранение объектов, 451
  транзакции, 662
  членство в профсоюзе, 431

S

Shape, приложение, 162
SMC (State Machine Compiler), 634
SQL Server, 648

U

UML (унифицированный язык моделирования), 214
  CASE-средства, 237
  как средство коммуникации, 225
  как финальная документация, 228

A

абстрактные классы
  и принцип открытости/закрытости, 474
  на диаграммах классов, 288
Абстрактный сервер, паттерн, 540

абстракции
в задаче о кофеварке, 303, 310
в системе расчета зарплаты, 399
и принцип инверсии зависимости, 191, 194
и принцип открытости/закрытости, 161, 167, 475
и принцип устойчивых абстракций, 475
метрики, 475
устранение ненужных повторов, 145
автономные тесты, 79
для программы GeneratePrimes, 79
ограничения, 72
Автошунтирование, паттерн, 486
агрегирование на диаграммах классов
ассоциации, 289
композиция, 290
кратность, 292
Адаптер, паттерн, 542
в задаче о модеме, 543
класса, 543
объекта, 543
активации на диаграммах последовательности, 220, 262
активные объекты
на диаграммах объектов, 250
на диаграммах последовательности, 276
Активный объект, паттерн, 345
архитектура в придачу, 74
асинхронные сообщения на диаграммах последовательности, 270
ассоциации
в диаграммах классов, 218, 282
агрегирование, 289
квалификаторы, 295
классы ассоциациий, 294
расположение по горизонтали, 285
стереотипы, 292
в задаче о кофеварке, 305
один-ко-многим, 282
ациклический ориентированный граф, 464
Ациклический посетитель, паттерн, 592

Б

базы данных
в системе расчета зарплаты, 647
как деталь реализации, 390
паттерны, 584
сторонние, 571
шлюзы к, 580
банкомат, задача о
диаграмма классов, 285
пример пользовательского интерфейса, 206
безопасность на этапе компиляции, 484
белый ящик, способ тестирования, 72
бизнес-правила
и механизмы сохранения, 157
и пользовательский интерфейс, 686
бизнес-решения, 60
боулинг, игра, 93
бюджет разработчика, 62

В

веб-интерфейсы, 683
вершина графа, 464
видимые события, в прецедентах, 256
владение
в экстремальном программировании, 52
инверсия, 192
вложенные классы на диаграммах классов, 294
входящие связи (Ca), 470
выгорания диаграмма, 63
вязкость дизайна, 143
вязкость окружения, 143

Г

генерация отчетов
паттерн Объект расширения, 610
паттерн Посетитель, 597
гибкое проектирование, 141
и загнивающая программа, 142
программа Copy, 146
главная последовательность
и абстракции, 475
расстояние от, 478, 500
главный цикл приложения, структура, 352
Голливуда принцип, 192
«готово», определение понятия, 61
граф зависимостей между компонентами, 462
графический интерфейс пользователя (ГИП)
задача о банкомате, 206
задача о кофеварке, 306
и высокоуровневые политики приложения, 642
контроллеры взаимодействия с, 644
персональные приложения, 683

результат, 715
система расчета зарплаты
конструирование окна, 696
окно Payroll, 703
проектирование, 685
реализация, 686

Д

двоичная единица развертывания, 457
двойная диспетчеризация, 588, 592
действия на диаграммах состояний, 221
действующие лица в прецедентах, 258
Декоратор, паттерн
в задаче о модеме, 604
и базы данных, 585
и стереотипы ассоциаций, 293
делегирование
как способ разделения, 204
детали
в принципе общей закрытости, 494
диаграммы, 214
выбрасывание, 229
выгорания, 63
динамические, 216
для игры в боулинг, 94
итеративное уточнение, 230
как дорожные карты, 227
как средство коммуникации, 225
как финальная документация, 228
концептуального уровня, 215
назначение, 223
объектов, 219, 248
активные объекты, 250
назначение, 249
прецедентов, 258
системных границ в прецедентах, 258
скорости, 63
статические, 216
уместность, 236
уровня реализации, 214
уровня спецификаций, 214
физические, 216
эффективное использование, 225
диаграммы классов, 218
ассоциации в, 218, 282
агрегирование, 289
квалификаторы, 295
классы ассоциаций, 294
расположение по горизонтали, 285
стереотипы, 292
в задаче о банкомате, 285
в задаче о мобильном телефоне, 232
закрытые классы, 281
защищенные классы, 281
знаки -, # и +, 281
изображение классов, 281
абстрактных, 288
ассоциаций, 294
вложенных, 294
композиция, 290
кратность, 292
наследование, 282
отделения, 281
открытые классы, 281
свойства, 288
стереотипы, 286
диаграммы компонентов, 464
диаграммы кооперации
в задаче о мобильном телефоне, 231
изображение связей, 220
порядковые номера сообщений, 220
диаграммы последовательности, 220, 260
активные объекты, 276
изображение длительности сообщения, 269
объекты, 261
отправка сообщений интерфейсам, 277
потоки, 276
создание и уничтожение объектов, 262
сообщения, 262, 269
сторожевые условия, 268
сценарии, 264
условия, 268
циклы, 264, 268
диаграммы состояний, 221, 240
переходов, 245
подсостояния, 244
псевдосостояния, 245
специальные события, 242
суперсостояния, 243
конечные автоматы, 245
начальное и конечное псевдосостояния, 242, 245
основные понятия, 241
рефлексивные переходы, 242
дизайн верхнего уровня, место, 474
добавление работника
в базу данных, 651
в системе расчета зарплаты, 342, 391, 405

документация
диаграммы как, 228
исчерпывающая, 40
листинги с исходным кодом как, 141
объем, 238
приемочные тесты как, 72
существенная, 40
дублирование кода, 55

Ж

жесткость дизайна, 143

З

зависимости
в гибком проектировании, 151
в задаче о кофеварке, 317
в задаче о модеме, 546
и диаграммы классов, 280, 283
и паттерн Фабрика, 483
и статическая типизация, 484
между классами, 457
метрики управления, 458
зависимость по именам, 483
загрязнение интерфейса, 200
задачи
в экстремальном программировании, 49
планирование, 62
задачи, исполняемыми до завершения, 348
заказчики
в экстремальном программировании, 48
сотрудничество с, 41
Заместитель, паттерн
в задаче об электронном магазине, 552
для сторонних API, 573
и рефакторинг, 157
и фабрики, 485, 487
реализация, 557
стереотипы ассоциаций, 293
зона бесполезности, 477
зона неприятностей, 477
зоны исключения, 476

И

иерархия вызовов на диаграммах кооперации, 220
изменение требований
загнивание программ, 146
отношение к, 43
изменения
жесткость и хрупкость дизайна, 143
и принцип открытости/закрытости, 167
как критерий качества модуля, 77
как причина загнивания программ, 145
программы Copy, 147
реагирование на, 42
сведений о работнике, 395, 420
требований, 43
изолирующие слои, 572
изоляция тестов, 69
иллюзорная абстракция, 303
имитация, 486
инверсия владения, 192
инкапсуляция, 497
интеграция, проблемы, 463
интерфейсы
без состояния, 642
имена, 341
исходящие связи (Ce), 470, 499
исчерпывающая документация, 40
итеративное уточнение на диаграммах, 230
итерации
планирование, 49, 61

К

карточки табельного учета
реализация, 413
регистрация, 393
кассовый терминал, 256
классы
абстрактные, 288, 474
и принцип общей закрытости, 461
боги, 304
вырожденные, 592
зависимости, 457
и принцип совместного повторного использования, 460
и принцип эквивалентности повторного использования и выпуска, 458
контейнерные, 181
миражи, 303
служебные, 287
клиенты
и принцип открытости/закрытости, 161
и принцип разделения интерфейсов, 202

Команда, паттерн, 339
вариации на тему Undo, 344
простые команды, 339
разрыв связей, 343
реализация класса DeleteEmployeeTransaction, 411
транзакции, 341
команды
в экстремальном программировании, 48
самоорганизующиеся, 45
компоненты, 457
зависимые, 471
принципы устойчивости, 462
принцип ацикличности зависимостей, 462
принцип устойчивых абстракций, 475
принцип устойчивых зависимостей, 469
структура и обозначения, 490
сцепленность, 462
принцип общей закрытости, 461
принцип совместного повторного использования, 460
принцип эквивалентности повторного использования и выпуска, 458
Компоновщик, паттерн, 511
и стереотипы ассоциаций, 294
кратность, 514
составные команды, 513
конечные автоматы, 622
UML-нотация, 221
в задаче о кофеварке, 318
вложенные switch/case, 623
высокоуровневые политики приложения для ГИП, 642
описание входа в систему, 241
реализация в виде моносостояния, 377
конструирование в диаграммах последовательности, 220
контракты в принципе подстановки Лисков, 180
конфигурационные данные, 604
копирование и вставка как источник повторов, 144
косность дизайна, 143
кратность
в паттерне Компоновщик, 514
на диаграммах классов, 292

Л

лампа, задача о, 539
абстракции, 195
и принцип инверсии зависимости, 195
паттерн Абстрактный сервер, 540
паттерн Адаптер, 542
линии жизни на диаграммах последовательности, 262

М

манифест гибкой разработки программ, 38
маркеры данных в диаграммах последовательности, 220, 262
матрицы функций, 592, 596
ментальное программирование, 68
метрики
в анализе пакетов, 498
в задаче о расчете зарплаты, 500
измерения абстрактности, 475, 499
измерения устойчивости, 470
мобильные телефоны, задача о
диаграмма кооперации, 231
диаграмма последовательности, 230
код, 234
эволюция диаграмм, 235
модели
в паттерне Наблюдатель, 536
назначение, 223
Модель-Вид-Презентатор, паттерн, 682
окна
Payroll, 703
конструирование, 696
применение к вводу данных о работнике, 686
модем, задача о
паттерн Адаптер, 543
паттерн Ациклический посетитель, 592
паттерн Декоратор, 604
паттерн Мост, 547
паттерн Посетитель, 588
Моносостояние, паттерн, 375
в классе DeleteEmployeeTransaction, 413
достоинства и недостатки, 376
пример, 377
Мост, паттерн, 547

Н

Наблюдатель, паттерн
  модель проталкивания и вытягивания, 535
  постепенное приближение, 515
направление зависимости, 151
наследование, 215, 541
  изображение по вертикали, 283
  и паттерн Одиночка, 373
  использование для разделения интерфейсов, 205
  множественное, 205
  на диаграммах классов, 282
наследование и факторизация, 186
независимые компоненты, 470, 493
неответственные компоненты, 471, 492
непрерывная интеграция, 52
непрозрачность дизайна, 145
неустойчивость (I), метрика, 471, 500

К

обобщение, 283
общность, метрика, 499
объект доступа к данным (DAO), 574
Объект-имитация, паттерн, 70
объектно-ориентированное проектирование
  в задаче о кофеварке, 318
  ограничения, 137
объектно-ориентированные системы управления базами данных (ООСУБД), 408, 451
Объект расширения, паттерн, 584, 610
ограничения, 38
Одиночка, паттерн, 371
  в классе DeleteEmployeeTransaction, 413
  достоинства и недостатки, 373
  пример, 373
ориентированные графы, 464
ориентированные ребра, 464
оси изменения в принципе единственной обязанности, 156
ответственные компоненты, 470, 492
Охота на Вампуса, программа, 68

П

пакетирование, 457, 490
  и принцип общей закрытости, 492
  метрики, 498
  окончательная структура, 507
  принцип эквивалентности повторного использования и выпуска, 494
  связанность и инкапсуляция, 497
парное программирование
  в экстремальном программировании, 50
  задачи об игре в боулинг, 93
первый закон документирования, 41
перепроектирование, 142
перепутанные провода, 306
персональные приложения, 683
печь, задача о, 197
планирование
  выпусков, 49, 60
  в экстремальном программировании, 59
  гибкость, 42
  задач, 62
  итераций, 49, 61
  первичное обследование, 59
  подведение итогов, 63
поведение на диаграммах, 230
повторное использование
  принцип совместного повторного использования, 460
  принцип эквивалентности повторного использования и выпуска, 458
повторы в дизайне, 144
поддержка сторонних компонентов, 458
подмена, 486
позиционная устойчивость компонента, 470
полиморфизм
  в классе Shape, 174
  в паттерне Моносостояние, 377
  и принцип открытости/закрытости, 172
полная сборка, 466
пользовательские истории
  в экстремальном программировании, 48
  и планирование, 59
  разбиение, 59
  скорость, 60
ПО промежуточного уровня
  и паттерн Одиночка, 373
  стороннее, 571
Посетитель, паттерн, 587
  в задаче о модеме, 588
  и базы данных, 585
  применение для генерации отчетов, 597

Посредник, паттерн, 367
постусловия, 180
потоки
  на диаграммах объектов, 250
  на диаграммах последовательности, 276
правила игры в боулинг, 92
правильность, в принципе подстановки Лисков, 179
предусловия, 180
преобразования между шкалами Цельсия и Фаренгейта, 353, 361
прецеденты, 255
  альтернативные потоки событий, 257
  для задачи о кофеварке, 306
  для системы расчета зарплаты, 390
  записывание, 256
  основной поток событий, 256
  представление на диаграммах, 258
приемочные тесты
  в экстремальном программировании, 50
  назначение, 72
принцип ацикличности зависимостей (ADP), 462
принцип единственной обязанности (SRP), 154
  в задаче о добавлении работника, 405
  в задаче о кофеварке, 312
  в задаче о настольной лампе, 542
  в системе расчета зарплаты, 400
  генерация отчетов, 597
  для компонентов, 461
  обеспечение сохранности, 157
  определение обязанности, 155
  разделение связанных обязанностей, 157
  циклические зависимости, 468
принципиальная модель системы расчета зарплаты, 397
принцип инверсии зависимости (DIP), 191, 466
  абстракции, 194
  в задаче об управлении печью, 197
  в задаче о кофеварке, 310
  в задаче о модеме, 543
  в задаче о настольной лампе, 540
  в программе Copy, 151
  инверсия владения, 192
  и паттерн Наблюдатель, 537
  конкретные классы, 194
  модули нижнего уровня, 191
  нарушения, 481
  простой пример, 195
  разбиение на слои, 191
принцип общей закрытости (CCP), 461
  и паттерн Декоратор, 606
  и устойчивость, 469
  и циклы зависимостей, 468
  применение, 492
принцип открытости/закрытости (OCP)
  абстракции, 161, 167, 475
  в задаче о добавлении работника, 405
  в задаче о кругах и квадратах, 165
  в задаче о модеме, 543
  в паттерне Наблюдатель, 536
  в программе Copy, 151
  в системе расчета зарплаты, 399
  для компонентов, 461
  естественная структура, 167
  и устойчивость, 474
  нарушение, 163
  описание, 160
  предвидение, 167
  применение таблицы данных, 170
принцип подстановки Лисков (LSP), 172
  в задаче о модеме, 543
  нарушения, 173
  реальный пример, 181
  факторизация, 186
  эвристика и соглашения, 188
принцип разделения интерфейсов (ISP), 200
  в задаче о модеме, 544
  в паттерне Наблюдатель, 537
  загрязнение интерфейса, 200
  интерфейсы классов и объектов, 204
  и разделение клиентов, 202
  пример ГИП в задаче о банкомате, 206
  разделение путем делегирования, 204
принцип совместного повторного использования, 460
принцип эквивалентности повторного использования и выпуска (REP)
  описание, 458
  применение, 494
программирование
  наугад, 108
  различий, 351
проектирование
  больших систем, 457
  в экстремальном программировании, 54

гибкое, 139
загнивающая программа, 142
программа Copy, 146
до начала кодирования, 224
начиная с тестов, 68
по контракту, 180
сверху вниз и снизу вверх, 467
простота
важность, 45
в экстремальном программировании, 54
при записи прецедентов, 255
прямая зависимость, 192
пузырьковая сортировка, 356

P

работники в системе расчета зарплаты
добавление, 342, 391, 405
изменение, 395, 420
начисление зарплаты, 434
удаление, 392, 411
разделение
путем делегирования, 204
связанных обязанностей, 157
разработка через тестирование, 51, 67
разработчики
бюджет, 62
в экстремальном программировании, 48
и бизнес-решения, 436
разреженные матрицы, 596
разрыв связей
в придачу, 71
физических и темпоральных, 343
распределенная обработка, 645
расстояние от главной последовательности, 478, 500
«реализует», отношение, 284
реляционные СУБД, 408, 451
рефакторинг, 76
в экстремальном программировании, 51, 55
рисование прямоугольников, 644

C

сборки, 457
еженедельные, 463
свойства
виртуальные, 177
на диаграммах классов, 288
связанность
и анализ пакетов, 497
метрики, 470, 498
разделение обязанностей, 157
типы связей, 470
фабрики, 504
«связан с», отношение, 282
связи на диаграммах кооперации, 220
синдром следующего утра, 462
синхронные сообщения на диаграммах последовательности, 270
система расчета зарплаты, 388, 404
головная программа, 449
изменение тарификации, 425
карточки табельного учета, 393, 413
метрики, 500
операции, 405
паттерн Null-объект, 385, 419
паттерн Команда, 341
паттерн Фабрика, 485
плата за услуги профсоюза, 394, 416
пользовательский интерфейс, 684
принадлежность в внешним организациям, 402
работники
добавление, 341, 391, 405
изменение, 395, 411, 420
начисление зарплаты, 434
удаление, 392
спецификация, 389
справки о продажах, 393, 416
транзакции, 341
системы управления версиями неблокирующие, 52
сиюминутные требования, 256
скорость, 60
диаграмма, 64
обратная связь по, 61
собираемость системы, 468
события
в диаграммах состояний, 221, 242
в прецедентах, 256
элементов управления, 702
соглашения в принципе подстановки Лисков, 188
Состояние, паттерн
в задаче о турникете, 630
достоинства и недостатки, 634
и Стратегия, 633
состязания на диаграммах последовательности, 270
сотрудничество
и формальные договоренности по контракту, 41
с заказчиками, 44

спецификация изделия, задача о, 597
справки о продажах
  реализация, 416
  регистрация, 393
статистический анализ дизайна, 478
сторожевые условия в диаграммах последовательности, 220
Стратегия, паттерн, 151, 352
  в алгоритме Application, 360
  в системе расчета зарплаты, 401
  и принцип открытости/закрытости, 161
  и Состояние, 633
  и Шаблонный метод, 363
  начисление зарплаты, 439
структура
  документирование, 40
  изображении на диаграммах классов, 232
  компонентов, 490
сцепленность
  границы, 506
  компонентов, 462
    принцип общей закрытости, 461
    принцип совместного повторного использования, 460
    принцип эквивалентности повторного использования и выпуска, 458
  метрики, 498
сцепленность по связям (H), метрика, 499

Т

таблицы данных в принципе открытости/закрытости, 170
таблицы переходов состояний, 245
темпоральные связи, разрыв, 343
тестирование, 66
  GeneratePrimes, программы, 79
  архитектура в придачу, 74
  изоляция, 69
  и принцип открытости/закрытости, 168
  приемочные тесты, 50, 72
  проектирование начиная с, 68
  разработка через, 67
  разрыв связей в придачу, 71
  фикстуры, 485
тестопригодность
  моделей, 224
  программного обеспечения, 67
типизация статическая и динамическая, 484
точки подключения в принципе открытости/закрытости, 168
транзакции
  в задаче о расчете зарплаты, 660, 662
  и паттерн Команда, 341
требования
  в экстремальном программировании, 48
  приемочные тесты как, 72
турникет, задача о
  вложенные switch/case, 623
  конечный автомат, 623
  паттерн Моносостояние, 377
  паттерн Состояние, 630
  применение компилятора SMC, 634
  таблицы переходов, 627

У

удаление работников, 392, 411
удобно вызываемый код, 67
условия в диаграммах последовательности, 268
устойчивость
  и абстрактность, 475
  метрики, 470, 498
  определение, 469
  принципы, 462
    ацикличности зависимостей, 462
    устойчивых абстракций, 475
    устойчивых зависимостей, 469

Ф

Фабрика, паттерн, 480
  важность, 487
  взаимозаменяемость, 485
  зависимости, 483
  инициализация, 506
  и связанность, 504
  использование в тестовых фикстурах, 485
  применение в окне Payroll, 703
  статическая и динамическая типизация, 484
Фабричный метод, паттерн, 408
Фасад, паттерн, 365
  и базы данных, 585
  и паттерн Одиночка, 374
  и паттерн шлюз к табличным данным, 574
  и рефакторинг, 157
  класс PayrollDatabase, 407

физические связи, 541
фикстуры тестовые, 485
финальная документация, 228
формальные договоренности по контракту, 41
формы, 684
функциональная декомпозиция, 468

Х

хрупкость дизайна, 143

Ц

целеустремленные люди, 44
циклы
  в экстремальном программировании, 49
  зависимостей, 463
цифровые часы
  задача о, 516

Ч

часть/целое, отношение, изображение на диаграммах классов, 289

Ш

Шаблонный метод, паттерн
  в задаче Application, 352
  в задаче о модеме, 605
  добавление работников, 408
  злоупотребление, 355
  изменение сведений о работниках, 422, 427, 430, 432
  и принцип открытости/закрытости, 162
  и Стратегия, 363
  похожие функции, 55
  пузырьковая сортировка, 356
Шлюз к табличным данным, паттерн
  и Фасад, 366
  пример, 574
  тестирование, 580
шлюзы
  и паттерн Фасад, 366
  и тестирование, 580
  пример, 573

Э

эвристика
  в принципе подстановки Лисков, 188
экстремальное программирование, 47
  единая команда, 48
  коллективное владение, 52
  короткие циклы, 49
  метафора, 56
  непрерывная интеграция, 52
  открытое рабочее пространство, 53
  парное программирование, 50
  планирование, 53
  пользовательские истории, 48
  приемочные тесты, 50
  простота дизайна, 54
  разработка через тестирование, 51
  рефакторинг, 51, 55
  умеренный темп, 53
электронный магазин
  объектная модель, 552
  паттерн Заместитель, 552
  реализация, 557
  реляционная модель данных, 553
элементы управления, события, 702
эффективность
  паттерна Моносостояние, 377
  паттерна Одиночка, 373

Я

«является», отношение, 175, 180