
Шаблоны тестирования xUnit

*Рефакторинг
кода тестов*

xUnit Test Patterns

*Refactoring
Test Code*

Gerard Meszaros



ADDISON-WESLEY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokio • Singapore • Mexico City

Шаблоны тестирования xUnit

Рефакторинг кода тестов

Джерард Месарош



Москва • Санкт-Петербург • Киев
2009

ББК 32.973.26-018.2.75

M53

УДК 681.3.07

Издательский дом “Вильямс”

Главный редактор *С.Н. Тригуб*

Зав. редакцией *В.Р. Гинзбург*

Перевод с английского и редакция *О.А. Лещинского*

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:

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

Месарош, Джерард.

М53 Шаблоны тестирования xUnit: рефакторинг кода тестов. : Пер. с англ. — М. : ООО
“И.Д. Вильямс”, 2009. — 832 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1448-4 (рус.)

В данной книге показано, как применять принципы разработки программного обеспечения, в частности шаблоны проектирования, инкапсуляцию, исключение повторений и описательные имена, к написанию кода тестов. В части I рассматриваются теоретические основы методов разработки тестов и описываются концепции шаблонов и “запахов” тестов (признаков существующей проблемы). В частях II и III приводится каталог шаблонов проектирования тестов, “запахов” и других средств обеспечения большей прозрачности кода тестов. Кроме этого, в части III сделана попытка обобщить и привести к единому знаменателю терминологию тестовых двойников и подставных объектов, а также рассмотрены некоторые принципы их применения при проектировании как тестов, так и самого программного обеспечения.

Книга ориентирована на разработчиков программного обеспечения, практикующих гибкие процессы разработки. В основном здесь рассматриваются примеры для существующих реализаций инфраструктуры xUnit, но затронуты и более новые инфраструктуры тестирования на основе данных.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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

Authorized translation from the English language edition published by Addison-Wesley, Copyright © 2007.

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

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2009.

ISBN 978-5-8459-1448-4 (рус.)

ISBN 978-0-13-149505-0 (англ.)

© Издательский дом “Вильямс”, 2009

© Pearson Education, Inc., 2007

Оглавление

Визуальное представление языка шаблонов	19
Предисловие	21
Пролог	23
Благодарности	29
Введение	31
Рефакторинг тестов	43
Часть I. Общая информация	59
Глава 1. Краткий обзор	61
Глава 2. Запахи тестов	67
Глава 3. Цели автоматизации	75
Глава 4. Философия автоматизации тестов	85
Глава 5. Принципы автоматизации тестирования	93
Глава 6. Стратегия автоматизации тестирования	103
Глава 7. Основы xUnit	127
Глава 8. Управление временной тестовой конфигурацией	137
Глава 9. Управление постоянными тестовыми конфигурациями	147
Глава 10. Проверка результатов	159
Глава 11. Использование тестовых двойников	175
Глава 12. Организация тестов	201
Глава 13. Тестирование с использованием баз данных	213
Глава 14. План эффективной автоматизации тестирования	221
Часть II. Запахи тестов	227
Глава 15. Запахи кода	229
Глава 16. Запахи поведения	263
Глава 17. Запахи проектов	295

Часть III. Шаблоны	309
Глава 18. Шаблоны стратегии тестирования	311
Глава 19. Базовые шаблоны xUnit	377
Глава 20. Шаблоны настройки тестовой конфигурации	433
Глава 21. Шаблоны проверки результатов	483
Глава 22. Шаблоны очистки тестовой конфигурации	517
Глава 23. Шаблоны тестовых двойников	537
Глава 24. Шаблоны организации тестов	603
Глава 25. Шаблоны баз данных	657
Глава 26. Шаблоны проектирования с учетом тестов	683
Глава 27. Шаблоны значений	717
Часть IV. Приложения	735
Приложение А. Рефакторинг тестов	737
Приложение Б. Терминология xUnit	743
Приложение В. Пакеты семейства xUnit	749
Приложение Г. Инструментарий	755
Приложение Д. Цели и принципы	759
Приложение Е. Запахи, псевдонимы и причины	763
Приложение Ж. Шаблоны, псевдонимы и варианты	767
Словарь терминов	784
Источники информации	813
Предметный указатель	827

Содержание

Визуальное представление языка шаблонов	19
Предисловие	21
Пролог	23
Ценность самотестирующегося кода	23
Первый проект с использованием экстремального программирования	24
Мотивация	26
Для кого предназначена эта книга	26
О фотографии на обложке	27
Ждем ваших отзывов!	27
Благодарности	29
Введение	31
Обратная связь	31
Тестирование	31
Тестирование разработчиками	32
Автоматизированное тестирование	32
Снижение чувствительности	34
Использование автоматизированных тестов	34
Тест как спецификация	34
Разработка на основе тестов	35
Шаблоны	35
Шаблоны, принципы и запахи	36
Исторические шаблоны и запахи	38
Ссылки на шаблоны и запахи	38
Рефакторинг	39
Предположения	39
Терминология	40
Терминология тестирования	40
Зависящая от языка терминология xUnit	41
Примеры кода	41
Описание с помощью диаграмм	42
Ограничения	42
Рефакторинг тестов	43
Зачем нужен рефакторинг тестов	43
Сложный тест	43

Очистка теста	44
Очистка логики верификации	44
Очистка логики удаления тестовой конфигурации	47
Очистка кода инициализации	51
Тест после всех модификаций	54
Написание других тестов	55
Дальнейшее упрощение	56
Часть I. Общая информация	59
Глава 1. Краткий обзор	61
О чем идет речь в этой главе	61
Самая простая рабочая стратегия автоматизации тестирования	61
Процесс разработки	62
Приемочные тесты	62
Модульные тесты	63
Проектирование с учетом тестирования	64
Организация тестов	65
Что дальше	65
Глава 2. Запахи тестов	67
О чем идет речь в этой главе	67
Введение в запахи тестов	67
Что такое запах теста	67
Типы запахов тестов	68
Что делать с запахами	68
Каталог запахов	69
Запахи проектов	69
Запахи поведения	70
Запахи кода	73
Что дальше	74
Глава 3. Цели автоматизации	75
О чем идет речь в этой главе	75
Зачем нужны тесты	75
Экономическое обоснование автоматизации тестов	76
Цели автоматизации тестов	77
Тесты должны способствовать повышению качества	77
Тесты должны способствовать пониманию принципов работы тестируемой системы	78
Тесты должны снижать риск (не внося новых его источников)	79
Тесты должны легко запускаться	80
Тесты должны быть простыми в написании и обслуживании	82
Тесты должны требовать минимального обслуживания по мере развития системы	84
Что дальше	84

Глава 4. Философия автоматизации тестов	85
О чём идет речь в этой главе	85
Почему важна философия	85
Некоторые философские отличия	86
Тестировать до или после написания кода	86
Тесты как примеры	86
Тест за тестом или все тесты сразу	87
Извне вовнутрь или изнутри наружу	88
Проверка поведения или проверка состояния	89
Тестовая конфигурация для каждого теста или одна тестовая конфигурация для всех тестов	90
Почему возникают различия в философии	90
Философия автора	90
Что дальше	91
Глава 5. Принципы автоматизации тестирования	93
О чём идет речь в этой главе	93
Принципы	93
Принцип: сначала пишите тесты (Write the Tests First)	94
Принцип: проектируйте с учётом тестирования (Design for Testability)	94
Принцип: сначала используйте “главный” вход (Use the Front Door First)	94
Принцип: доносите намерение (Communicate Intent)	95
Принцип: не модифицируйте тестируемую систему (Don’t Modify the SUT)	95
Принцип: сохраняйте независимость тестов (Keep Tests Independent)	96
Принцип: изолируйте тестируемую систему (Isolate the SUT)	97
Принцип: минимизируйте пересечения тестов (Minimize Test Overlap)	98
Принцип: минимизируйте нетестируемый код (Minimize Untestable Code)	98
Принцип: не вносите логику тестов в код продукта (Keep Test Logic Out of Production Code)	99
Принцип: проверяйте одно условие за тест (Verify One Condition per Test)	99
Принцип: тестируйте аспекты по-отдельности (Test Concerns Separately)	101
Принцип: обеспечьте адекватные усилия и ответственность (Ensure Commensurate Effort and Responsibility)	101
Что дальше	101
Глава 6. Стратегия автоматизации тестирования	103
О чём идет речь в этой главе	103
Что значит “стратегический”	103
Какие тесты подвергать автоматизации	104
Тесты функциональности	104
Кроссфункциональные тесты	106
Инструментарий для автоматизации	107
Способы автоматизации тестирования и подходы к ней	108
Введение в xUnit	109
Сильные стороны xUnit	110
Управление тестовыми конфигурациями	111
Что такая тестовая конфигурация	111

Основные стратегии работы с тестовыми конфигурациями	112
Временная новая тестовая конфигурация	114
Постоянная новая тестовая конфигурация	115
Стратегии на основе общей тестовой конфигурации	116
Обеспечение простоты тестирования и взаимодействия с тестируемой системой	118
Тестируйте после, но не говорите, что вас не предупреждали	118
Готовый проект с учетом тестов — прыжок выше собственной головы	118
Возможность тестирования, обеспеченная тестами	118
Контрольные точки и точки наблюдения	119
Стили взаимодействия и шаблоны тестирования	120
Разделяй и тестируй	123
Что дальше	125
Глава 7. Основы xUnit	127
О чём идет речь в этой главе	127
Введение в xUnit	127
Общие функции	128
Базовый минимум	128
Определение тестов	128
Что такое тестовая конфигурация	130
Определение наборов тестов	130
Запуск тестов	131
Результаты выполнения теста	131
Что происходит “под капотом” xUnit	133
Команды тестов	134
Объекты наборов тестов	134
Реализации xUnit в процедурной парадигме	134
Что дальше	135
Глава 8. Управление временной тестовой конфигурацией	137
О чём идет речь в этой главе	137
Тестовые конфигурации	137
Что такое тестовая конфигурация	137
Что такое новая тестовая конфигурация	139
Что такое временная новая тестовая конфигурация	139
Создание новых тестовых конфигураций	140
Встроенная настройка тестовой конфигурации	140
Делегированная настройка тестовой конфигурации	141
Неявная настройка тестовой конфигурации	143
Гибридная настройка тестовой конфигурации	145
Очистка временной новой тестовой конфигурации	145
Что дальше	146
Глава 9. Управление постоянными тестовыми конфигурациями	147
О чём идет речь в этой главе	147
Управление постоянными новыми тестовыми конфигурациями	147
Что делает тестовую конфигурацию постоянной	148

Проблемы постоянных новых тестовых конфигураций	148
Очистка постоянных новых тестовых конфигураций	149
Как избежать очистки	152
Решение проблемы медленных тестов	153
Управление общими тестовыми конфигурациями	154
Доступ к общей тестовой конфигурации	154
Создание общей тестовой конфигурации	156
Что дальше	157
Глава 10. Проверка результатов	159
О чём идет речь в этой главе	159
Создание самопроверяющихся тестов	159
Что проверять: состояние или поведение?	160
Проверка состояния	161
Использование встроенных утверждений	162
Дельта-утверждения	163
Внешняя проверка результата	163
Проверка поведения	164
Процедурная проверка поведения	164
Спецификация ожидаемого поведения	165
Сокращение дублирования кода	166
Ожидаемый объект	166
Специальные утверждения	168
Метод проверки с описанием результата	168
Параметризованный и управляемый данными тест	169
Как избежать условной логики в тестах	170
Удаление операторов if	171
Исключение циклов	172
Другие способы	172
Разработка в порядке “извне вовнутрь”	172
Использование разработки на основе тестов для создания вспомогательных методов теста	173
Расположение повторно используемой логики проверки	173
Что дальше	174
Глава 11. Использование тестовых двойников	175
О чём идет речь в этой главе	175
Что такое опосредованные ввод и вывод	175
Назначение информации об опосредованном вводе	176
Назначение информации об опосредованном выводе	176
Управление опосредованным вводом	178
Проверка опосредованного вывода	179
Тестирование с помощью двойников	183
Типы тестовых двойников	183
Предоставление тестового двойника	189
Настройка тестового двойника	190
Установка тестового двойника	192

12 Содержание

Другие сферы применения тестовых двойников	197
Эндоскопическое тестирование	197
Разработка на основе потребностей	198
Ускорение создания тестовой конфигурации	198
Ускорение работы тестов	198
Другие аргументы	198
Что дальше	199
Глава 12. Организация тестов	201
О чём идет речь в этой главе	201
Базовые механизмы инструментария xUnit	201
Размер тестовых методов	202
Тестовые методы и классы тестов	203
Класс теста для каждого класса	203
Класс теста для каждой функции	203
Класс теста для каждой тестовой конфигурации	205
Выбор стратегии организации тестовых методов	205
Соглашения об именовании тестов	206
Организация наборов тестов	206
Запуск групп тестов	208
Запуск единственного теста	208
Повторное использование кода тестов	209
Наследование и повторное использование класса теста	210
Организация тестовых файлов	211
Встроенные автотесты	211
Пакеты тестов	211
Зависимости тестов	212
Что дальше	212
Глава 13. Тестирование с использованием баз данных	213
О чём идет речь в этой главе	213
Тестирование с использованием баз данных	213
Причины тестирования с базами данных	214
Проблемы, связанные с базами данных	214
Тестирование без баз данных	215
Тестирование базы данных	217
Тестирование хранимых процедур	217
Тестирование уровня доступа к данным	218
Обеспечение независимости разработчиков	219
Тестирование с базами данных (опять!)	219
Что дальше	219
Глава 14. План эффективной автоматизации тестирования	221
О чём идет речь в этой главе	221
Сложность автоматизации тестирования	221
План создания простых в обслуживании автоматизированных тестов	222
Выполните код на “счастливом маршруте”	223

Проверьте непосредственный вывод “счастливого маршрута”	223
Проверьте альтернативные ветви кода	224
Проверьте поведение опосредованного вывода	225
Оптимизируйте запуск и обслуживание тестов	225
Что дальше	226
Часть II. Запахи тестов	227
Глава 15. Запахи кода	229
Непонятный тест (Obscure Test)	230
Условная логика теста (Conditional Test Logic)	243
Сложный в тестировании код (Hard-to-Test Code)	251
Дублирование тестового кода (Test Code Duplication)	254
Логика теста в продукте (Test Logic in Production)	257
Глава 16. Запахи поведения	263
Рулетка утверждений (Assertion Roulette)	264
Нестабильный тест (Erratic Test)	267
“Хрупкий” тест (Fragile Test)	277
Частая отладка (Frequent Debugging)	285
Ручное вмешательство (Manual Intervention)	287
Медленные тесты (Slow Tests)	289
Глава 17. Запахи проектов	295
Тест с ошибками (Buggy Test)	296
Разработчики не пишут тесты (Developers Not Writing Tests)	298
Высокая стоимость обслуживания тестов (High Test Maintenance Cost)	300
Ошибки в продукте (Production Bugs)	303
Часть III. Шаблоны	309
Глава 18. Шаблоны стратегии тестирования	311
Записанный тест (Recorded Test)	312
Тест на основе сценария (Scripted Test)	319
Управляемый данными тест (Data-Driven Test)	322
Инфраструктура автоматизации тестов (Test Automation Framework)	332
Минимальная тестовая конфигурация (Minimal Fixture)	336
Стандартная тестовая конфигурация (Standard Fixture)	338
Новая тестовая конфигурация (Fresh Fixture)	344
Общая тестовая конфигурация (Shared Fixture)	350
Манипуляция через “черный ход” (Back Door Manipulation)	359
Тест уровня (Layer Test)	368
Глава 19. Базовые шаблоны xUnit	377
Тестовый метод (Test Method)	378
Четырехфазный тест (Four-Phase Test)	387
Метод с утверждением (Assertion Method)	390

Сообщение для утверждения (Assertion Message)	398
Класс теста (Testcase Class)	401
Программа запуска тестов (Test Runner)	405
Объект теста (Testcase Object)	410
Объект набора тестов (Test Suite Object)	414
Обнаружение тестов (Test Discovery)	420
Перечисление тестов (Test Enumeration)	425
Выбор тестов (Test Selection)	429
Глава 20. Шаблоны настройки тестовой конфигурации	433
Встроенная настройка (In-line Setup)	434
Делегированная настройка (Delegated Setup)	437
Метод создания (Creation Method)	441
Неявная настройка (Implicit Setup)	449
Предварительно созданная тестовая конфигурация (Prebuilt Fixture)	454
“Ленивая” настройка (Lazy Setup)	460
Настройка тестовой конфигурации набора (Suite Fixture Setup)	465
Декоратор настройки (Setup Decorator)	471
Цепочки тестов (Chained Tests)	477
Глава 21. Шаблоны проверки результатов	483
Проверка состояния (State Verification)	484
Проверка поведения (Behavior Verification)	489
Специальное утверждение (Custom Assertion)	495
Дельта-утверждение (Delta Assertion)	505
Сторожевое утверждение (Guard Assertion)	510
Утверждение незаконченного теста (Unfinished Test Assertion)	514
Глава 22. Шаблоны очистки тестовой конфигурации	517
Очистка со сборкой мусора (Garbage-Collected Teardown)	518
Автоматическая очистка (Automated Teardown)	521
Встроенная очистка (In-line Teardown)	527
Неявная очистка (Implicit Teardown)	533
Глава 23. Шаблоны тестовых двойников	537
Тестовый двойник (Test Double)	538
Тестовая заглушка (Test Stub)	544
Тестовый агент (Test Spy)	552
Подставной объект (Mock Object)	558
Поддельный объект (Fake Object)	565
Настраиваемый тестовый двойник (Configurable Test Double)	571
Фиксированный тестовый двойник (Hard-Coded Test Double)	581
Связанный с тестом подкласс (Test-Specific Subclass)	591
Глава 24. Шаблоны организации тестов	603
Именованный набор тестов (Named Test Suite)	604
Вспомогательный метод теста (Test Utility Method)	610

Параметризованный тест (Parameterized Test)	618
Класс теста для каждого класса (Testcase Class per Class)	627
Класс теста для каждой функции (Testcase Class per Feature)	633
Класс теста для каждой тестовой конфигурации (Testcase Class per Fixture)	639
Суперкласс теста (Testcase Superclass)	646
Вспомогательный класс теста (Test Helper)	651
Глава 25. Шаблоны баз данных	657
“Песочница” с базой данных (Database Sandbox)	658
Тест хранимой процедуры (Stored Procedure Test)	662
Очистка усечением таблиц (Table Truncation Teardown)	668
Очистка откатом транзакции (Transaction Rollback Teardown)	675
Глава 26. Шаблоны проектирования с учетом тестов	683
Вставка зависимости (Dependency Injection)	684
Поиск зависимости (Dependency Lookup)	692
Минимальный объект (Humble Object)	700
Ловушка для теста (Test Hook)	713
Глава 27. Шаблоны значений	717
Точное значение (Literal Value)	718
Вычисляемое значение (Derived Value)	722
Сгенерированное значение (Generated Value)	726
Объект-заглушка (Dummy Object)	730
Часть IV. Приложения	735
Приложение А. Рефакторинг тестов	737
Приложение Б. Терминология xUnit	743
Приложение В. Пакеты семейства xUnit	749
Приложение Г. Инструментарий	755
Приложение Д. Цели и принципы	759
Приложение Е. Запахи, псевдонимы и причины	763
Приложение Ж. Шаблоны, псевдонимы и варианты	767
Словарь терминов	784
Источники информации	813
Предметный указатель	827

Эта книга посвящается памяти Дениса Клилланда, который пригласил меня из компании Nortel в 1995 году для работы в ClearStream Consulting и, таким образом, предоставил мне возможность получить опыт, ставший источником этой книги. К сожалению, Денис умер 27 апреля 2006 года, когда завершалась работа над вторым черновиком книги.

Визуальное представление языка шаблонов

Цели, принципы и запахи

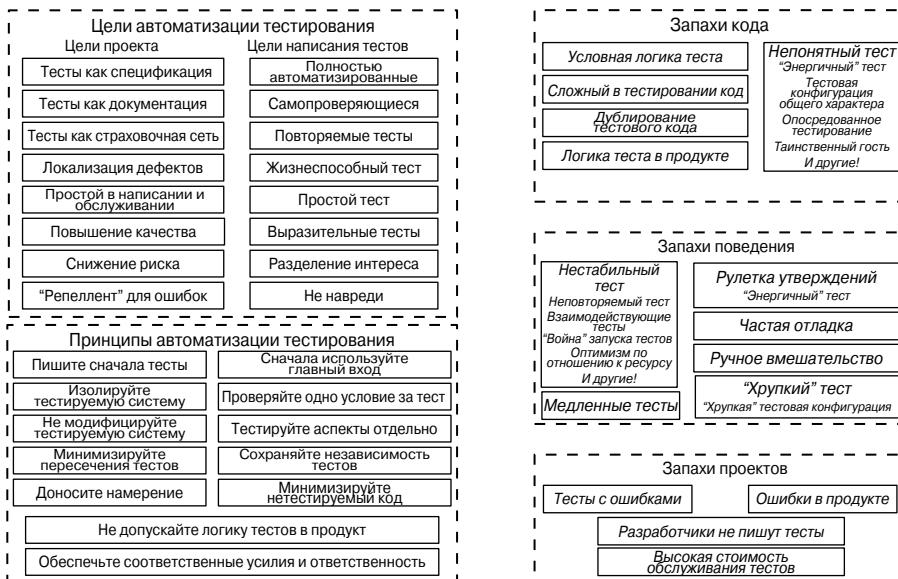
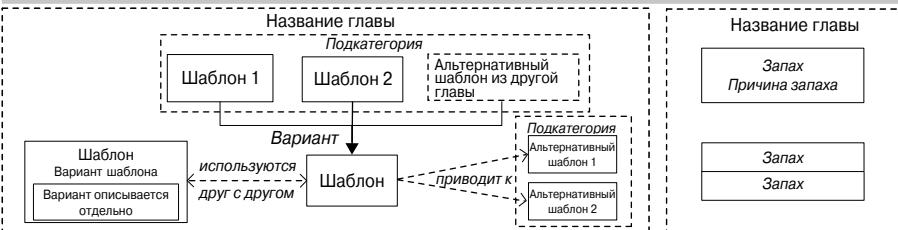
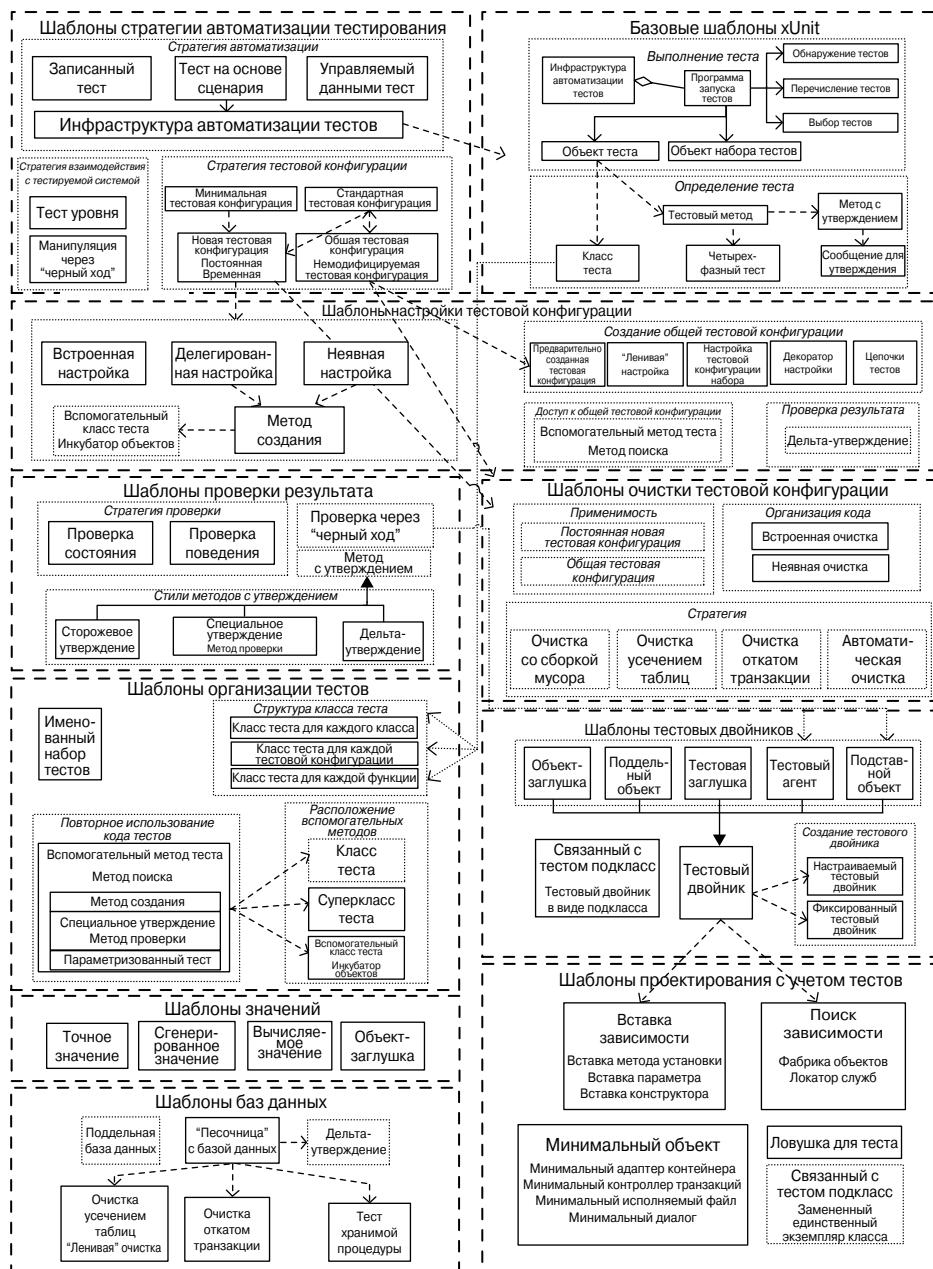


Схема визуального описания языка шаблона



Шаблоны



Предисловие

Если зайти на сайт junit.org, можно заметить мою цитату: “Никогда в отрасли разработки программного обеспечения столь многие не были так благодарны столь малому объему кода”. Проект JUnit рассматривался как хобби-проект, который любому опытному разработчику под силу написать за выходные. Это так, но подобный взгляд отвлекает от главного. Важность проекта JUnit заключается в фундаментальном смещении фокуса для многих разработчиков: тестирование превратилось в главную и основополагающую часть программирования. Многие предлагали такой подход к разработке и раньше, но именно JUnit сделал основной вклад в этот сдвиг.

Конечно, дело не только в JUnit. Идея была перенесена на большинство языков программирования. В результате целое семейство утилит стало называться xUnit, так как идея давно переросла реализацию для языка Java. (Конечно, все начиналось не с Java: Кент Бек написал этот код для языка Smalltalk значительно раньше.)

Утилиты xUnit, а также их философия, предоставляют огромные возможности командам разработчиков: мощные наборы регрессивных тестов позволяют вносить значительные изменения в код с минимальным риском, а при разработке на основе тестов — на ходу пересматривать дизайн продукта.

Но эти возможности сопровождаются новыми проблемами и новыми методиками. Как и любой инструмент, семейство xUnit может использоваться неправильно. Опытные разработчики накопили различные способы использования xUnit для эффективной организации тестов и данных. Как и на заре объектно-ориентированного программирования, большая часть знаний по практическому применению инструментария скрыта в головах опытных пользователей. Без этих “тайных знаний” очень тяжело использовать возможности пакета максимально эффективно.

Около двадцати лет назад была эта проблема была осознана сообществом объектно-ориентированной разработки. Решением проблемы стало описание “тайных знаний” в виде шаблонов. Джерард Месарош был одним из пионеров этого процесса. Когда я начал знакомиться с шаблонами, Джерард был одним из лидеров, у которого мне довелось учиться. Как и многие в мире шаблонов, Джерард стал одним из апологетов технологии экстремального программирования (eXtreme Programming), а значит, работал с утилитами xUnit с самого начала. Логично, что ему пришла в голову идея оформить свои знания эксперта в виде шаблонов.

Мне понравился этот проект с того момента, когда я о нем услышал. (Мне пришлось провести спецоперацию по похищению данной книги у Боба Мартина, так как я хотел включить ее в свою серию.) Как и любая хорошая книга о шаблонах, она содержит информацию, полезную новичкам в данной отрасли. Кроме того, книга

может служить словарем и источником базовой информации для опытных разработчиков, которые пытаются передать свои знания коллегам. Широко известная книга “банды четырех” *Design Patterns* (*Приемы объектно-ориентированного проектирования*, “Питер”, 2005 г.) раскрыла секреты объектно-ориентированного проектирования. Предлагаемая вашему вниманию книга имеет такую же ценность с точки зрения применения пакета xUnit.

Мартин Фаулер,
редактор серии,
главный научный сотрудник, ThoughtWorks

Пролог

Ценность самотестирующегося кода

В главе 4 книги *Рефакторинг* [Ref] Мартин Фаулер написал следующее.

Если обратить внимание на потраченное разработчиками время, можно заметить, что написание кода на самом деле составляет небольшую его часть. Некоторое время тратится на постановку задачи, некоторое — на проектирование, но большая его часть уходит на отладку. Каждый читатель может вспомнить долгие часы отладки (часто до глубокой ночи). Любой разработчик может рассказать историю об ошибке, исправление которой потребовало целого дня (или даже больше). На самом деле для исправления ошибки много времени не нужно. А вот найти ошибку — совсем другое дело. Не забывайте, что после исправления одной ошибки всегда существует вероятность появления другой, которая остается незаметной очень долго. И еще больше времени потребуется на ее обнаружение.

Некоторые приложения слишком сложны для тестирования вручную. Часто в таких случаях приходится писать тестовые программы.

В 1996 году автор участвовал в проекте, в котором его задача заключалась в создании инфраструктуры событий, позволяющей клиентскому программному обеспечению регистрироваться в системе и получать уведомление, когда другое приложение генерировало соответствующее событие (шаблон *наблюдатель*, Observer). Я не смог придумать лучшего способа тестирования инфраструктуры, чем создание макета клиентского программного обеспечения. Необходимо было проверить около двадцати различных сценариев, поэтому каждый сценарий был запрограммирован с использованием соответствующего числа наблюдателей, событий и генераторов событий. Поначалу за происходящим приходилось следить, рассматривая вывод на консоль. Очень быстро такой способ контроля оказался слишком утомительным.

Определенная степень лени стимулировала поиск другого решения, связанного с контролем над процессом тестирования. Для каждого теста был создан *словарь* (Dictionary), проиндексированный ожидаемыми событием и получателем и содержащий имя получателя в качестве значения. Как только конкретному получателю отправлялось уведомление о событии, получатель проверял *словарь* на наличие собственной записи с полученным событием. Если такая запись существовала, получатель ее удалял. Если запись не существовала, получатель добавлял запись с сообщением об ошибке, указывающим на уведомление о неожиданном событии.

После запуска всех тестов тестовая программа просто просматривала словарь и выводила непустые записи. В результате запуск всех тестов практически не был связан с накладными расходами. Тесты или успешно завершались без сообщений, или выводили

список сообщений о неудачах. Как потом оказалось, неожиданно для себя автор открыл концепции *подставного объекта* (Mock Object, с. 558) и *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332).

Первый проект с использованием экстремального программирования

В 1999 году автор был участником конференции OOPSLA, на которой распространялась новая книга Кента Бека *Экстремальное программирование* [ХРЕ]. К этому моменту автор уже применял итеративно-инкрементный подход к разработке и уже понимал ценность автоматизированного модульного тестирования, хотя и не имел опыта его универсального применения. К Кенту Беку автор питал уважение еще со времен первой конференции PLoP (Pattern Languages of Programs) в 1994 году. Совокупность этих факторов убедила автора в желательности применения методологии экстремального программирования в проекте ClearStream Consulting. Почти сразу же после конференции OOPSLA подвернулась возможность применить такой подход в реальном проекте — это было вспомогательное приложение, взаимодействующее с существующей базой данных, но не имеющее собственного пользовательского интерфейса. При этом клиент был готов к применению нетрадиционных методик разработки.

С самого начала применялись все описанные в литературе принципы экстремального программирования, включая программирование в парах, общее владение кодом, а также разработку на основе тестов. Конечно, возникли и сложности при создании тестов для некоторых аспектов поведения приложения, но для большей части кода тесты все же были написаны. После этого в процессе развития проекта стала наблюдаться настороживающая тенденция: реализация одних и тех же решений требовала все больше и больше времени.

После коротких объяснений разработчики стали записывать на каждой карточке задачи время, которое требовалось для создания новых тестов, модификации существующих тестов и создания фактического кода. Очень быстро была выявлена и изолирована интересная тенденция. Как оказалось, время создания новых тестов и написания кода осталось практически неизменным. Но значительно возросло время модификации существующих тестов. Когда разработчик предложил поработать в паре, оказалось, что 90% рабочего времени ушло на модификацию существующих тестов, соответствующих относительно небольшому изменению в функциональности.

Анализ ошибок компиляции и неудачных результатов тестов при добавлении новой функциональности показал, что многие тесты зависели от изменений в методах тестируемой системы. Это никого не удивило. А удивило то, что больше всего проблем возникало на этапе создания тестовой конфигурации и изменения не затрагивали основную логику тестов.

Это стало важным открытием, так как показало, что знания о создании объектов тестируемой системы были равномерно распределены между большинством тестов. Другими словами, тесты слишком много знали о не самых важных аспектах поведения тестируемой системы. Большинство затронутых тестов на самом деле не должны были зависеть от того, как создавались объекты в пределах тестовой конфигурации; их задачей была проверка правильности состояния объектов. При последующей проверке оказалось, что многие тесты создавали одинаковые или почти одинаковые объекты на этапе подготовки.

Очевидным решением проблемы было выделение этой логики в небольшой набор *вспомогательных методов теста* (Test Utility Method, с. 610). На тот момент существовало несколько вариантов.

- Если несколько тестов требовали идентичных объектов, просто использовался метод, который возвращал необходимый объект. Такое решение теперь называется *методом создания* (Creation Method, с. 441).
- В некоторых тестах требовались различные значения атрибутов объекта. В таком случае значение атрибута передавалось в качестве значения параметра для *параметризованного метода создания* (Parameterized Creation Method).
- Некоторым тестам для нормальной работы требовался некорректно сформированный объект, чтобы убедиться в том, что тестируемая система откажется его обрабатывать. Генерация отдельного *параметризованного метода создания* (Parameterized Creation Method) для каждого атрибута излишне раздувала сигнатуру *вспомогательного класса теста* (Test Helper, с. 651), поэтому создавался нормальный объект, значение атрибутов которого модифицировалось с помощью *единственного дефектного атрибута* (One Bad Attribute).

Результатом стал набор тестов и методов, в дальнейшем превратившихся в первые шаблоны автоматизации тестов¹.

Позднее, когда тесты стали завершаться неудачно из-за отказа базы данных принимать объект с уже существующим идентификатором, был добавлен код для генерации уникального ключа. Этот фрагмент был назван *анонимным методом создания* (Anonymous Creation Method, с. 443), что указывало на дополнительную функциональность.

Идентификация проблемы, которая сейчас называется “хрупким” тестом (Fragile Test, с. 277), была важным этапом проекта. Последующее определение шаблонов для ее решения спасло проект от возможного неудачного завершения. Без этого открытия пришлось бы как минимум отказаться от уже написанных автоматизированных модульных тестов. В худшем случае тесты снизили бы производительность разработчиков настолько, что клиенты не получили бы необходимый результат. Но в итоге удалось выдать необходимый продукт очень хорошего качества. Да, тестеры (функция тестирования иногда называется контролем качества; такое название является некорректным) находили ошибки в коде, так как некоторые тесты так и не были написаны. Но после того как были написаны отсутствующие тесты, внесение изменений для исправления обнаруженных ошибок практически не требовало усилий.

Тот проект показал, что автоматическое модульное тестирование и разработка на основе тестов действительно работают. С тех пор данные методики используются постоянно.

Хотя разработанные практики и шаблоны применялись в последующих проектах, приходилось сталкиваться с новыми проблемами и задачами. В каждом случае приходилось слой за слоем откапывать корень проблемы и изобретать подходящие решения. Спустя некоторое время развития открытые техники добавлялись в набор подходов к автоматизированному модульному тестированию.

Часть из этих шаблонов была описана в выступлении для конференции XP2001. Обсуждение с коллегами на этой и последующих конференциях показало, что многие

¹ Технически решение не является шаблоном, пока не будет найдено тремя независимыми группами разработчиков.

пользуются такими же или подобными методиками. В результате методы из “практик” превратились в “шаблоны” (повторяющееся решение повторяющейся проблемы). Первое описание **запахов теста** (*test smell*) было опубликовано на той же конференции. В основе понятия “запах теста” лежало понятие “запах кода”, описанное в книге *Рефакторинг* [Ref]².

Мотивация

Я глубоко убежден в важности автоматизированного модульного тестирования. Я разрабатывал программное обеспечение без этой технологии около двадцати лет. Инфраструктура xUnit и созданные на ее основе автоматизированные тесты являются одним из заметных шагов в развитии отрасли разработки программного обеспечения. Каждый раз очень неприятно наблюдать, как компания пытается применить автоматизированное модульное тестирование, но терпит неудачу из-за недостатка ключевой информации и необходимых навыков.

Как консультант по разработке ПО в компании ClearStream Consulting я по долгу службы сталкиваюсь с большим количеством проектов. Иногда клиент вызывает консультанта в начале проекта, чтобы “все было правильно”. Но чаще приходится подключаться к проекту, когда он начинает свой полет под откос. В результате пришлось познакомиться с “худшими практиками”, лежащими в основе большинства запахов тестов. Если повезло и меня позвали не слишком поздно, клиент сможет восстановиться после совершенных ошибок. В противном случае клиенту придется выпутываться самому, считая, что разработка на основе тестов и автоматизированное модульное тестирование дают не очень хорошие результаты — и клиент начнет распространять слухи об автоматизированном тестировании как о напрасной трате времени.

Оглядываясь в прошлое, приходится признаться, что все эти проблемы можно было обойти, имея необходимые знания в нужный момент. Но как получить знание, самостоятельно не совершив все ошибки? С точки зрения затраченного на изучение новых технологий времени выгоднее всего пригласить специалиста, который уже обладает необходимыми знаниями. Просвещение специализированных курсов или чтение книги является менее эффективной (хотя и более дешевой) альтернативой. Надеюсь, что, записав эти ошибки и рекомендованные методы их обхода, я смог помочь многим разработчикам.

Для кого предназначена эта книга

Эта книга написана в основном для разработчиков программного обеспечения (программистов, проектировщиков и архитекторов), которые хотят научиться лучше писать тесты, а также для руководителей и преподавателей, которые хотят понимать, что делают разработчики и зачем им необходимо выделять время на повышение эффективности этой деятельности. Основное внимание уделяется модульным тестам и приемочным тестам, которые автоматизируются с помощью пакета xUnit. Кроме того, ряд шаблонов высокого уровня относится к тестам, автоматизированным на основе отличных от

² В книге Мартина Фаулера [Ref] использовались термины “дурной запах” и “неприятный запах”. — Примеч. ред.

xUnit технологий. Рик Магридж и Уорд Каннингем написали отличную книгу по инфраструктуре Fit [FitB] и являются сторонниками многих из описанных здесь практик.

Скорее всего, разработчики захотят прочитать книгу от корки до корки, но справочные главы лучше сначала просмотреть, а не вчитываться в каждое слово. Основное внимание нужно уделить общему представлению о существующих шаблонах и принципах их функционирования. К конкретным шаблонам можно будет вернуться, когда в них возникнет необходимость. Обычно описание приводится в нескольких первых подразделах (до раздела “Когда это использовать” включительно) того раздела, который посвящен конкретному шаблону.

Руководителям и преподавателям рекомендуется прочитать часть I, “Общая информация”, и, возможно, часть II, “Запахи тестов”. Кроме того, желательно прочитать главу 18, “Шаблоны стратегии тестирования”, так как в ней описаны решения, смысл которых руководители должны понимать, чтобы поддерживать разработчиков в использовании этих шаблонов. Как минимум руководители должны прочитать главу 3, “Цели автоматизации”.

О фотографии на обложке

На обложке каждой книги данной серии размещена фотография моста. Когда Мартин Фаулер предложил включить книгу в свою серию, я задался вопросом: “Какой мост разместить на обложке?” Я подумал о способности тестирования предотвращать катастрофы в работе программного обеспечения и о связи тестирования ПО с мостами. Сразу в голову пришло несколько неудачных мостов, включая “Galloping Gertie” (мост Тэкома-Нэрроуз) и мост “Iron Workers Memorial Bridge” в Ванкувере (мост назван в память о рабочих, погибших при обрушении части конструкции).

После дальнейшего обдумывания мне показалось неоправданным утверждать, что тестирование позволило бы избежать этих ошибок, поэтому мост был выбран, исходя из личных побуждений. На обложке изображен мост “New River Gorge” в Западной Виргинии. Впервые мне довелось перейти его и проплыть под ним в конце 1980-х годов. Архитектура моста также связана с содержимым этой книги: сложная арочная структура под мостом в большинстве случаев скрыта от проезжающих по мосту. Дорога не имеет стыков, и ночью можно проехать мост, даже не заметив его тысячефутовой высоты. Хорошая инфраструктура автоматизации тестов имеет то же свойство: писать тесты легко, так как почти вся сложность скрыта под “дорожным полотном”.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@williamspublishing.com
WWW: <http://www.williamspublishing.com>

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
в Украине: 03150, Киев, а/я 152

Благодарности

Хотя в основном книга создана силами одного автора, многие люди сделали в нее свой вклад. Хотелось бы сразу извиниться перед теми, кто не упомянут здесь.

Те, кто хорошо меня знают, всегда задаются вопросом, где я взял время для написания этой книги. В нерабочее время я обычно занимаюсь различными (кто-то может сказать “экстремальными”) видами спорта, например горными лыжами, рафтингом и катанием на горном велосипеде. Лицо я не считаю эти виды деятельности более экстремальными, чем итеративное и инкрементное программирование. Несмотря на это вопрос о времени, затраченном на написание книги, остается вполне оправданным. Особая благодарность предназначена для моей подруги Хизер Армитидж (Heather Armitage), с которой я занимался большинством перечисленных видов спорта. Она много часов вела машину туда и обратно, пока я сидел с портативным компьютером на заднем сиденье, работая над книгой. Кроме того, хочу выразить признательность Альфу Скрастиньшу (Alf Skrastins), который приглашает своих друзей на лыжные прогулки к западу от Калгари. Отдельная благодарность — администраторам различных лыжных баз, которые разрешали перезаряжать портативный компьютер от их генераторов, что позволяло работать над книгой во время отпуска: Грании Дивайн (Grania Devine) из Selkirk Lodge, Таннис Дакин (Tannis Dakin) из Sorcerer Lodge, а также Дейву Флиру (Dave Flear) и Арону Куперману (Aaron Cooperman) из Sol Mountain Touring. Без их помощи книга создавалась бы намного дольше!

Как обычно, хотелось бы поблагодарить всех рецензентов, официальных и неофициальных. Роберт “Дядя Боб” Мартин (Robert C. “Uncle Bob” Martin) рецензировал ранний черновик. Официальными рецензентами первого “официального” черновика были Лиза Криспин (Lisa Crispin) и Рик Магридж (Rick Mugridge). Джереми Миллер (Jeremy Miller), Алистэр Дьюигид (Alistair Duguid), Майкл Хеджпет (Michael Hedgpeth) и Эндрю Стопфорд (Andrew Stopford) рецензировали второй черновик.

Хотелось бы также поблагодарить моих “пастырей” с различных конференций PLoP, которые высказали свои мнения по поводу описанных здесь шаблонов: Майкла Стала (Michael Stahl), Дэнни Диго (Danny Dig) и особенно Джо Йодера (Joe Yoder). Они предоставили экспертные комментарии по экспериментам с шаблонами. Так же отдельная благодарность — группе по языкам шаблонов на конференции PLoP 2004, особенно Юджину Уоллингфорду (Eugene Wallingford), Ральфу Джонсону (Ralph Johnson) и Джозефу Бергину (Joseph Bergin). Брайан Фут (Brian Foote) и группа SAG опубликовали несколько гигабайтов файлов MP3 с рецензиями на ранний черновик книги. Благодаря их комментариям я переписал одну из глав.

За время работы над книгой пришло много сообщений электронной почты с комментариями к материалам, опубликованным на сайте <http://xunitpatterns.com>. Каждый комментарий оказался очень к месту, так как часть опубликованного материала бы-

ла крайне сырой. Среди таких неофициальных рецензентов можно выделить Джавида Джамаи (Javid Jamae), Филипа Нельсона (Philip Nelson), Томаша Гаджевски (Tomasz Gajewski), Джона Херста (John Hurst), Свена Гортса (Sven Gorts), Брэдли Ландиса (Bradley T. Landis), Седрика Беста (Cedric Beust), Джозефа Пелрине (Joseph Pelrine), Себастьяна Бергмана (Sebastian Bergmann), Кевина Резерфорда (Kevin Rutherford), Скотта Амблера (Scott W. Ambler), Джей Би Рейнсбергера (J. B. Rainsberger), Оли Бая (Oli Bye), Дейла Эмери (Dale Emery), Дэйвида Нанна (David Nunn), Алекса Чиффи (Alex Chaffee), Буркхардта Хуфнагеля (Burkhardt Hufnagel), Йоханнеса Бродуолла (Johannes Brodwall), Брета Петтикорда (Bret Pettichord), Клинта Шанка (Clint Shank), Сунила Джоглекара (Sunil Joglekar), Рейчел Дейвис (Rachel Davies), Ната Прайса (Nat Pryce), Пола Ходжеттса (Paul Hodgetts), Оуэна Роджерса (Owen Rogers), Амира Кольски (Amir Kolsky), Кевина Лоуренса (Kevin Lawrence), Алистэра Коуберна (Alistair Cockburn), Майкла Фезерса (Michael Feathers) и Джо Шметцера (Joe Schmetzer). Отдельная благодарность выражается Нилю Норвицу (Neal Norwitz), Маркусу Гэлли (Markus Gaelli), Стефану Диокассе (Stephane Ducasse) и Стефану Райххарту (Stefan Reichhart).

Кое-кто отправлял мне электронные письма с описаниями часто применяемого шаблона или специальной функции для используемого пакета xUnit. Большинство из них были вариациями уже документированных шаблонов. В книге они приводятся как псевдонимы или варианты реализации. Было несколько более эзотерических шаблонов, которые не включены в книгу по соображениям объема. За это приношу извинения.

Многие из описанных в этой книге идей взяты из проектов, над которыми мне довелось работать в компании ClearStream Consulting. Нам постоянно создавались предпосылки к поиску более оптимальных способов решения поставленных задач при наличии минимальных ресурсов. Такой подход дал начало многим методикам, описанным в данной книге. Коллегами автора по этому непростому процессу были Дженнита Андреа (Jennitta Andrea), Ральф Боне (Ralph Bohnet), Дэйв Браат (Dave Braat), Рассел Брайант (Russel Bryant), Грэг Куик (Greg Cook), Джейф Харди (Geoff Hardy), Шон Смит (Shaun Smith) и Томас Таннахилл (Thomas Tannahill). Многие из них стали рецензентами ранних вариантов некоторых глав. Кроме того, Грэг является автором большинства примеров кода в главе 25, а Ральф настроил хранилище CVS и автоматизированный процесс компиляции для сайта. Хотелось бы также поблагодарить руководство ClearStream Consulting за разрешение отказаться от консультаций и сконцентрироваться на написании книги и за разрешение на использование упражнений из двухдневного курса “Тестирование для разработчиков” в качестве основы для большинства примеров кода. Итак, благодарности уходят Денису Клилланду (Denis Clelland) и Люку Макфарлану (Luke McFarlane)!

Несколько человек помогали мне работать над книгой, когда все становилось совсем сложно. Они всегда были готовы обсудить по телефону сложный вопрос. Особо хотелось бы отметить Джошуа Кериевски (Joshua Kerievsky) и Мартина Фаулера.

Я выражают искреннюю признательность Шону Смиту (Shaun Smith) за помощь и за техническую поддержку на начальных этапах работы над книгой. Он обеспечивал работу сайта, создал первые таблицы стилей CSS, научил меня применять Ruby, настроил Wiki-ресурс для обсуждения шаблонов и даже внес свой вклад в их наполнение, пока личные и рабочие факторы не заставили его отойти от писательской составляющей проекта. Каждый раз, когда в описании полученного опыта говорится “мы”, подразумеваемся я и Шон.

Введение

Как уже было сказано, разработка программного обеспечения без дефектов — исключительно сложный вид деятельности. Доказательство корректности реальных систем до сих пор остается за пределами возможностей, а описание поведения оказывается настолько же сложным. Любое предсказание будущего может оказаться ложным. Если бы этот навык давал гарантированный результат, мы стали бы игроками на бирже, а не разработчиками.

Автоматизированная проверка корректности поведения программного обеспечения является одним из важнейших улучшений методов разработки за последние несколько десятков лет. Данная практика оказала значительное влияние на повышение производительности разработчиков, повышение качества и защиту программного обеспечения от сбоев. Сам факт, что разработчики применяют эту методику по собственной воле, говорит об эффективности технологии.

Во введении описывается концепция автоматизации тестов с помощью ряда инструментов (включая xUnit), перечисляются причины применения этой методики и демонстрируются сложности на пути к правильной автоматизации тестирования.

Обратная связь

Обратная связь является важным элементом многих видов деятельности. Она позволяет убедиться, что выполняемые действия дают желаемый эффект. Чем быстрее работает обратная связь, тем быстрее реакция. Хорошим примером такой обратной связи являются дорожные отбойники между полотном и обочиной многих шоссе. Если съехать на обочину, сразу понятно, что колеса находятся не на дороге. Но, получив эту информацию раньше (когда колесо выходит на отбойник), можно изменить направление движения и снизить вероятность съезда с дороги.

Тестирование обеспечивает обратную связь при разработке программного обеспечения. Такая связь является неотъемлемым элементом “гибкого” процесса разработки. Циклы обратной связи позволяют сохранить уверенность в создаваемом программном обеспечении, ускорить разработку и уменьшить ненужные переживания о качестве. Тесты предоставляют возможность спокойно добавлять новую функциональность, показывая состояние старой.

Тестирование

Традиционное определение “тестирования” происходит из мира контроля качества. Тестировать программное обеспечение необходимо, так как в нем содержатся ошибки!

Поэтому ПО тестируется, тестируется, тестируется и тестируется еще, до тех пор пока становится невозможно доказать существование ошибок в коде. Традиционно данный процесс происходит уже после завершения разработки. В результате он становится методом контроля, а не обеспечения качества продукта. Во многих организациях тестирование выполняется не разработчиками. Обратная связь с таким процессом является очень ценной, но на поздних этапах разработки эта ценность значительно снижается. Кроме того, неприятным эффектом такой организации процесса является увеличение цикла разработки, так как найденные проблемы передаются разработчикам для исправления, требуя повторного цикла тестирования. Возникает вопрос: “Какая методика тестирования позволит разработчикам получать более быструю обратную связь?”

Тестирование разработчиками

Редкий разработчик верит, что может сразу написать полностью работающий код. На самом деле многие приятно удивляются, если что-то заработает сразу. (Надеюсь, ничьи иллюзии не были разбиты жестокой реальностью?)

Поэтому разработчики также занимаются тестированием. В процессе написания кода необходимо доказательство корректности работы программного обеспечения. Некоторые разработчики тестируют тем же способом, что и тестеры, рассматривая целую систему как единую сущность. Но большинство предпочитают модульное тестирование. “Модули” могут быть большими компонентами или отдельными классами, методами или функциями. Ключевым отличием этих тестов является выбор модулей в соответствии с дизайном программного продукта, а не на основе прямой трансляции требований. (Некоторая часть модульных тестов может непосредственно соответствовать бизнес-логике или пользовательским тестам, но большая часть тестов связана с кодом, окружающим бизнес-логику.)

Автоматизированное тестирование

Автоматизированное тестирование существует уже несколько десятков лет. Работая в начале 1980-х годов над системой коммутации телефонных звонков в компании Bell-Northern Research, которая вела разработки для компании Nortel, автору приходилось заниматься автоматизированным регрессионным и стресс-тестированием создаваемых программных и аппаратных компонентов. Тестирование выполнялось в контексте “системного теста” с помощью аппаратных и программных средств, которые программировались с помощью тестовых сценариев. Тестовые устройства подключались к коммутатору и играли роль нескольких телефонов и телефонных коммутаторов, генерируя телефонные звонки и запрашивая другие функции телефонии. Конечно, такая инфраструктура автоматизированного тестирования не была предназначена для модульного тестирования, а также была недоступна разработчикам из-за огромного количества аппаратных компонентов.

За последние десять лет появились более универсальные инструменты автоматизации тестирования, позволяющие взаимодействовать с приложениями через пользовательский интерфейс. Некоторые из них применяют языки сценариев для определения тестов. Более развитые продукты основаны на метафоре “робота” или “записи и воспроизведения” для автоматизации тестирования. К сожалению, опыт использования таких инструмен-

тов оказался не очень положительным. Причиной этому стала повышенная стоимость обслуживания тестов, связанная с проблемой “теста хрупкого”.

Проблема “хрупкого теста”

Автоматизация тестирования на основе коммерческих приложений с “записью и воспроизведением” или “роботом” заработала не очень хорошую репутацию среди пользователей. Такие тесты часто завершались неудачно по, на первый взгляд, тривиальным причинам. Важно понимать ограничения такого способа автоматизации тестов, чтобы не наткнуться на известные “подводные камни” — чувствительность к поведению, чувствительность к интерфейсу, чувствительность к данным и чувствительность к контексту.

Чувствительность к поведению

Если поведение системы изменилось (например, если изменились требования и система была модифицирована), любой тест модифицированной функциональности завершится неудачно при повторном воспроизведении (изменение в поведении может быть связано с тем, что система решает совсем другую задачу или ту же задачу, но в другой последовательности и с другими задержками). Это обычная реальность тестирования вне зависимости от подхода к автоматизации. Реальная проблема заключается в необходимости использования этой функциональности для перевода системы в подходящее для начала теста состояние. Следовательно, изменения в поведении имеют намного большее влияние на процесс тестирования, чем может показаться.

Чувствительность к интерфейсу

Нежелательно тестировать бизнес-логику системы через пользовательский интерфейс. Минимальные модификации интерфейса приведут к неудачному завершению теста, даже если человеку тест может показаться успешным. Такая неожиданная чувствительность к интерфейсу оказалась одной из причин отрицательного отношения к инструментам автоматизации в последнее десятилетие. Хотя такая проблема возникает вне зависимости от используемой технологии интерфейса, с некоторыми технологиями она усугубляется. Наиболее сложными с точки зрения взаимодействия с бизнес-логикой внутри системы являются графические интерфейсы пользователя. Наблюдаемое в последнее время смещение в сторону Web-интерфейса упростило некоторые аспекты автоматизации, но создало еще одну проблему, связанную с выполняемым кодом внутри кода HTML, который используется для более сложного взаимодействия с пользователем.

Чувствительность к данным

Каждый тест предполагает существование отправной точки, которая называется **тестовая конфигурация** (test fixture). Такой **контекст теста** (test context) иногда называется предусловием или предварительной картиной теста. Чаще всего он определяется в терминах данных, которые уже присутствуют в системе. Если данные меняются, тест может завершиться неудачно. Для решения этой проблемы могут потребоваться дополнительные усилия по обеспечению нечувствительности теста к данным.

Чувствительность к контексту

На поведение системы может оказывать влияние состояние компонентов за пределами системы. Среди таких внешних факторов можно перечислить состояние устройств (например, принтеров или серверов), других приложений и даже системных часов (например, время и/или дата запуска теста могут отличаться). Любой зависящий от контекста тест нельзя будет гарантированно повторить без получения контроля над контекстом.

Снижение чувствительности

Перечисленные типы чувствительности существуют вне зависимости от применяемой технологии автоматизации тестов. Конечно, одни технологии позволяют обойти эту проблему, а другие навязывают невыгодный способ решения. Инфраструктура автоматизации xUnit обеспечивает большую степень контроля и при необходимости квалификации может использоваться с достаточной эффективностью.

Использование автоматизированных тестов

В данной книге основное внимание уделяется регрессионному тестированию приложений. Это очень полезный тип обратной связи в процессе модификации существующего приложения, так как он позволяет обнаруживать случайно внесенные ошибки.

Тест как спецификация

Разработка на основе тестов (test driven development — TDD), являющаяся одной из ключевых практик таких гибких методов разработки, как, например, экстремальное программирование, включает в себя совершенно другое применение автоматизации тестов. В этом случае тесты используются, как спецификация поведения еще ненаписанного программного обеспечения. Эффективность разработки на основе тестов основана на разделении процесса разработки на две фазы: определение, что должна делать программа, и реализация задуманной функциональности.

Но разве сторонники гибких методов разработки не избегают каскадного процесса разработки? Да, конечно, избегают. Они предпочитают проектировать и создавать систему функция за функцией с получением работающей системы на каждом этапе с доказательством работоспособности каждой функции перед переходом к реализации следующей. Это не значит, что проектирование отсутствует; просто оно происходит непрерывно! Следование такому подходу приводит к “криSTALLизации дизайна” практически без предварительного проектирования. Это не единственный возможный метод разработки. Ничто не мешает комбинировать проектирование на высоком уровне (для создания архитектуры) с дизайном на уровне отдельных функций. В любом случае желательно отложить обдумывание реализации поведения конкретного класса или метода и описать это поведение в форме выполняемой спецификации. В конце концов, многим тяжело сконцентрироваться даже на чем-то одном, так что тем более не получится концентрироваться на нескольких задачах сразу.

После написания тестов и проверки ожидаемого неудачного результата можно переходить к реализации функций, обеспечивающих успешное завершение тестов. В результате тесты позволяют оценивать оставшийся объем работ. При последовательной реали-

зации функциональности тесты один за другим будут успешно завершаться. В процессе работы успешно завершенные тесты запускаются еще раз для регрессионного тестирования, подтверждая, что внесенные изменения не имели нежелательного эффекта. В этом и заключается истинная ценность автоматизированного модульного тестирования: в возможности “фиксировать” функциональность тестируемой системы, контролируя ее неизменность.

Разработка на основе тестов

В последнее время появилось много книг по данной теме, поэтому здесь разработке на основе тестов уделяется не очень много внимания. В данной книге основное внимание уделяется коду существующих тестов, а не подходам к их написанию. Ближе всего к разработке тестов будут разделы глав книги, посвященные **рефакторингу** (refactoring) тестов, при котором тесты на основе одного шаблона превращаются в тесты на основе шаблона с другими характеристиками.

В настоящей книге предпринята попытка отделить обсуждение от конкретного процесса разработки, так как любая группа разработчиков может применять автоматизированное тестирование как в начале разработки, так и по ее завершении. Кроме того, как только разработчики научатся автоматизировать тесты уже готового кода, они с большей вероятностью предпочтут экспериментировать с созданием тестов до написания тестируемого кода. Некоторые части процесса разработки будут рассмотрены в контексте упрощения автоматизации тестирования. Относительно процессов разработки нас интересуют два аспекта.

1. Взаимодействие *полностью автоматизированных тестов* (Fully Automated Tests, с. 81), инструментария и процесса интеграции при разработке.
2. Влияние процесса разработки на простоту тестирования.

Шаблоны

При подготовке этой книги применялось множество отчетов с конференций и книг по автоматизации тестов с помощью пакета xUnit. Не удивительно, что у каждого автора есть определенная область интересов и часто применяемые техники. Хотя не со всеми описанными подходами можно согласиться, всегда стоит разобраться, почему другие авторы применяют конкретные техники и когда их решения могут оказаться более оправданными.

Такой уровень представления материала отличает его от простых примеров использования методик и шаблонов. Шаблоны помогают понять, почему применяется данная методика, позволяя принять взвешенное решение о выборе одного из альтернативных шаблонов. Правильное решение позволяет обойти нежелательные последствия в будущем.

Шаблоны в программном обеспечении существуют уже около десяти лет, поэтому большинство читателей как минимум знакомы с самим понятием. Шаблон является решением повторяющейся проблемы. Одни проблемы серьезнее других, а значит, слишком велики для решения с помощью одного шаблона. В такой ситуации может пригодиться язык шаблонов. Набор (или грамматика) шаблонов описывает пошаговый переход от общей постановки проблемы к подробному решению. В языке шаблонов одни из шабло-

нов находятся на более высоком уровне абстракции, в то время как другие сконцентрированы на проблемах более низкого уровня. Между шаблонами должны существовать связи, чтобы можно было осуществлять переход от высокоуровневых стратегий к более детальным шаблонам проектирования и далее к еще более подробным идиомам кода.

Шаблоны, принципы и запахи

В этой книге рассматриваются три типа шаблонов. Наиболее традиционным типом является повторяющееся решение для распространенной проблемы. Большинство шаблонов из этой книги попадают в данную категорию, но и они делятся на три группы.

- **Стратегии.** Это шаблоны, применение которых имеет далеко идущие последствия. Решение об использовании *общей тестовой конфигурации* (Shared Fixture, с. 350) вместо *новой тестовой конфигурации* (Fresh Fixture, с. 344) приводит к применению совершенно другого набора шаблонов тестов. Каждому стратегическому шаблону посвящен отдельный раздел в главе 6, “Стратегия автоматизации тестирования”.
- **Шаблоны проектирования.** Используются при разработке тестов конкретной функциональности. Основное внимание уделяется организации логики теста. Большинству читателей будет понятен в качестве примера шаблон *подставной объект* (Mock Object, с. 558). Каждому шаблону проектирования посвящен отдельный раздел в соответствующей главе.
- **Идиомы кода.** Описывают разные способы кодирования конкретного теста. Многие идиомы имеют смысл только в пределах конкретного языка. В качестве примера можно привести блочную конструкцию (block closure) в языке Smalltalk при использовании шаблона *тест на ожидаемое исключение* (Expected Exception Test) и анонимные внутренние классы при реализации шаблона *подставной объект* (Mock Object) на языке Java. Некоторые идиомы, например *простой тест успешности* (Simple Success Test), являются настолько универсальными, что имеют соответствующие аналоги в каждом языке программирования. Обычно такие идиомы рассматриваются как варианты реализации или примеры в описании шаблона проектирования.

Часто на каждом уровне может использоваться несколько альтернативных шаблонов. Конечно, всегда существует более предпочтительный вариант, но антишаблон для одного может быть лучшим решением для другого. Поэтому в данную книгу включены шаблоны, которые не всегда рекомендуются к использованию. Здесь приводятся положительные и отрицательные стороны каждого шаблона, что позволяет принять осознанное решение об их применении. В большинстве случаев ссылка на альтернативы приводится в описании шаблона.

Выбор шаблона зависит от цели, которая должна быть достигнута в результате автоматизации теста. Такие цели поддерживаются рядом принципов, которые описывают систему признаков “хороших” автоматизированных тестов. В данной книге цели автоматизации тестирования рассматриваются в главе 3, “Цели автоматизации”, а принципы описываются в главе 5, “Принципы автоматизации тестирования”.

Последним типом шаблонов является антишаблон [AP]. Такие **запахи тестов** (test smells) соответствуют повторяющимся проблемам. Запахи описываются в виде наблюдаемых симптомов и реальных причин таких симптомов. **Запахи кода** (code smells)

впервые были описаны в книге Мартина Фаулера [Ref], а в отношении тестов на базе xUnit они были впервые описаны в докладе на конференции XP2001 [RTC]. Вместе с запахами тестов приводятся ссылки на шаблоны, которые позволяют от них избавиться, а также шаблоны (кто-то может назвать их антишаблонами), которые с большей вероятностью приведут к появлению такого запаха (некоторые шаблоны и запахи даже имеют одинаковые названия). Кроме того, запахи подробно рассматриваются в части II, “Запахи тестов”.

Формат шаблона

Здесь приводится авторское описание шаблонов. Сами шаблоны существовали еще до написания книги, так как перед превращением в шаблон они были изобретены как минимум тремя независимыми разработчиками автоматизированных тестов. Книга написана с целью сделать знание более удобным для распространения, и для этого пришлось выбрать формат описания шаблонов.

Описания могут иметь несколько вариантов. Одни имеют жесткую структуру с большим количеством заголовков, помогающих читателю ориентироваться в разделах. Другие имеют более литературный формат, но более сложны в использовании в виде справочника.

Мой формат шаблонов

Мне действительно понравились работы Мартина Фаулера, в основном тем, что связано с форматом описания шаблонов. Как говорится, “имитация — искренняя форма лести”, поэтому данный формат шаблонов был скопирован с минимальными модификациями.

Описание начинается с постановки задачи, общей информации и диаграммы. В выделенном курсивом абзаце описывается конкретная проблема, решаемая с помощью шаблона. Ее можно описать вопросом: “Как сделать...?” В абзаце, выделенном полужирным, описывается суть шаблона в одном-двух предложениях, а диаграмма обеспечивает визуальное представление. Текст после диаграммы содержит причины применения шаблона и разделы “Проблема” и “Контекст” из традиционного формата шаблона. Ознакомившись с этим разделом, читатель может принять решение о необходимости более детального изучения шаблона.

В следующих трех разделах формата приводится подробная информация о шаблоне. В разделе “Как это работает” описываются структура шаблона и принципы работы. Кроме того, приводится описание результирующего контекста, если существует несколько способов реализации важного аспекта шаблона. Этот раздел соответствует разделам “Решение” и “Следовательно” традиционного формата описания. В разделе “Когда это использовать” перечислены ситуации, когда имеет смысл применять шаблон. Этот раздел соответствует разделам “Проблема”, “Вызывает”, “Контекст” и “Похожие шаблоны” традиционного формата. Кроме того, раздел включает в себя описание результирующего контекста, если он может повлиять на решение о применении шаблона. Также перечислены запахи тестов, при которых рекомендуется применение шаблона. В разделе “Замечания по реализации” описываются особенности реализации конкретного шаблона. Подразделы в этом разделе соответствуют ключевым компонентам шаблона или вариантам реализации.

Большинство конкретных шаблонов имеют по три дополнительных раздела. Раздел “Мотивирующий пример” содержит пример тестового кода до применения шаблона. В разделе “Пример: (Имя шаблона)” показан код теста после применения шаблона. В разделе “Замечания по рефакторингу” приводится более подробная информация по переходу от кода из раздела “Мотивирующий пример” к коду из раздела “Пример: (Имя шаблона)”.

Если существует дополнительная информация о данном шаблоне, описание может содержать раздел “Источники дополнительной информации”. Раздел “Известные применения” содержит примечательные моменты конкретного теста. Конечно, большинство шаблонов использовались в множестве систем, поэтому указывать для каждого шаблона примеры использования будет лишней тратой времени.

Если существует несколько связанных одна с другой методик, обычно они описываются в виде шаблона с вариациями. Если вариации представляют собой различные способы реализации одного и того же фундаментального шаблона (т.е. решения одной проблемы одним общим способом), отличия в реализациях описываются в разделе “Замечания по реализации”. Если отличие является основной причиной использования шаблона, оно рассматривается в разделе “Когда это использовать”.

Исторические шаблоны и запахи

Значительных усилий стоило создать достаточно выразительный список пакетов и запахов, по возможности сохраняя исторические названия. Часто историческое название указывается в качестве псевдонима шаблона или запаха. В некоторых случаях историческое название имеет смысл рассматривать как вариант большего шаблона. В таком случае название указывается в разделе “Замечания по реализации”.

Многие исторические запахи не прошли “проверку нюхом”, т.е. они описывали главную причину, а не симптом. (“Проверка нюхом” основана на истории из книги *Рефакторинг*, в которой Кент Бек спрашивает бабушку: “Как узнать, что пришло время менять пеленку?” “Если воняет, меняй!” — ответила она. Названия запахов происходят от “вони”, а не от ее причины.) Если историческое название описывает причину, а не симптом, оно перемещено в соответствующий симптоматический запах, как специальная вариация под названием “Причина”. Хорошим примером кажется *тайный гость* (*Mystery Guest*).

Ссылки на шаблоны и запахи

Определенных усилий потребовало создание ссылок на шаблоны и запахи, особенно создание исторические названия. Хотелось использовать как исторические, так и новые агрегированные названия, в зависимости от контекста. В электронной версии книги для этой цели применяются гиперссылки. Но в печатной версии в качестве ссылки используется номер страницы. Если ссылка указывает на вариант шаблона или причину запаха, в первый раз указывается имя агрегированного шаблона или запаха. Обратите внимание, что вторая ссылка на причину (*тайный гость*, *Mystery Guest*) запаха *непонятного теста* (*Obscure Test*) указывается без имени запаха, а ссылка на другие причины *непонятного теста* (*Obscure Test*), например *неуместная информация* (*Irrelevant Information*), включает в себя имя агрегированного запаха, но не включает номер страницы.

Рефакторинг

Рефакторинг является относительно новой концепцией в разработке программного обеспечения. Хотя модификация существующего кода требовалась всегда, рефакторинг является строго формализованным подходом к модификации дизайна без изменения поведения кода. Рефакторинг тесно связан с автоматизированным тестированием, так как очень сложно выполнять рефакторинг без страховочной сети тестов, которая позволяет доказать целостность кода.

Многие интегрированные среды разработки имеют встроенную поддержку рефакторинга. Большинство из них обеспечивают автоматизацию хотя бы нескольких операций, описанных в книге Мартина Фаулера [Ref]. К сожалению, эти инструменты не сообщают, когда и зачем выполнять рефакторинг. Для этого придется приобрести саму книгу. Также обязательно нужно прочитать книгу Джошуа Кериевски [RtP].

Рефакторинг тестов несколько отличается от рефакторинга работающего кода, так как для автоматизированных тестов не существует автоматизированных тестов! Если тест завершается неудачно после рефакторинга, может ли причиной быть ошибка во время рефакторинга? Если тест проходит успешно после рефакторинга, можно ли быть уверенными, что тест завершится неудачно в соответствующей ситуации? Для решения этой проблемы используются очень консервативные подходы к рефакторингу тестов. Кроме того, применяя подходящую стратегию тестирования (как показано в главе 6, “Стратегия автоматизации тестирования”), можно обойтись без внесения значительных модификаций в тесты.

Основное внимание здесь уделяется цели рефакторинга, а не механизмам. Краткое описание рефакторинга приводится в приложении А, но не это основная тема. Шаблоны сами по себе достаточно новы, чтобы не успело сформироваться общее мнение об их именовании, содержимом или применимости, тем более нет единого мнения о лучших способах рефакторинга. Наличие нескольких отправных точек для каждого объекта рефакторинга (шаблона) еще больше усложняет ситуацию. Попытка привести подробные инструкции по рефакторингу сделает эту, и так большую, книгу еще большей.

Предположения

При написании книги предполагалось, что читатель знаком с объектной технологией (так называемым объектно-ориентированным программированием). Объектная технология стала необходимым условием для роста популярности автоматизированного модульного тестирования. Это не значит, что тесты в процедурных или функциональных языках невозможны, но использование таких языков может усложнить тестирование (по крайней мере, все будет происходить немного по-другому).

Каждый человек имеет свой стиль обучения. Одним нужна общая картина с последующим спуском к достаточно подробным деталям. Другие могут понять детали и не нуждаются в общей картине. Третьим достаточно услышать или прочитать слова. Четвертым нужны визуальные представления концепций. Пятым изучение идей программирования лучше всего удастся при чтении кода. Была сделана попытка приспособиться ко всем стилям обучения, поэтому везде, где это возможно, приводятся общие описания, подробные описания, примеры кода и иллюстрации. Эти элементы можно пропустить, если они не соответствуют выбранному вами стилю изучения.

Терминология

В этой книге собрана терминология из двух проблемных областей: разработки и тестирования программного обеспечения. В результате часть терминологии будет незнакома некоторым читателям. Встретив непонятный термин, желательно найти его в словаре терминов. Но один или два термина будут приведены здесь, так как их понимание является обязательным условием для понимания большей части материала данной книги.

Терминология тестирования

Разработчикам программного обеспечения термин *тестируемая система* (system under test — SUT) может показаться незнакомым. Означает он то, что в данный момент тестируется. При создании модульных тестов тестируемой системой является тестируемый класс или метод (методы). При создании клиентских тестов тестируемой системой, скорее всего, является целое приложение (или одна из его основных подсистем).

Любая часть приложения или создаваемой системы, не включенная в тестируемую систему, все равно может потребоваться для работы тестов, так как она вызывается тестируемой системой или предоставляет данные, необходимые тестируемой системе во время тестирования. Вызываемая часть называется *вызываемым компонентом* (depended-on component — DOC). И вызываемый компонент, и предоставляющая данные часть являются элементами тестовой конфигурации (рис. 1).

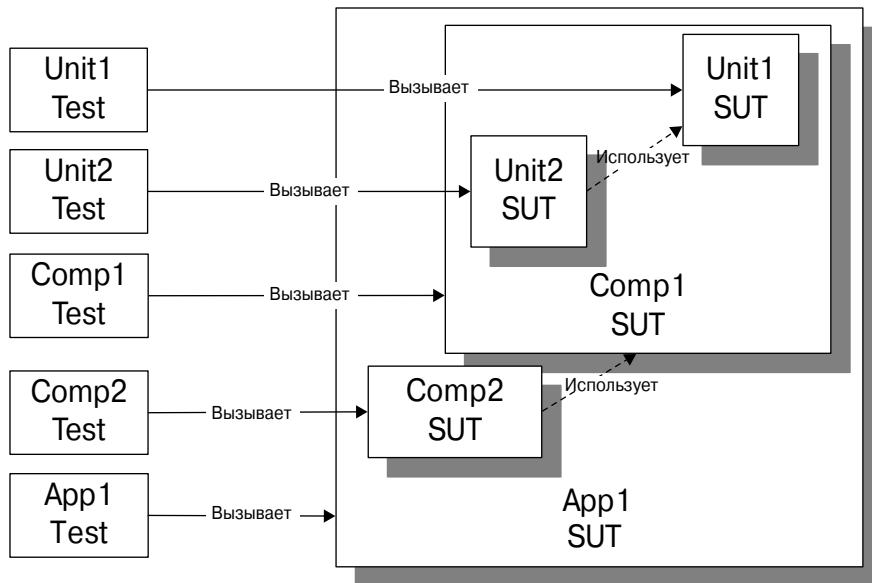


Рис. 1. Набор тестов. Каждому тесту соответствует своя тестируемая система. Приложение, компонент или модуль является тестируемой системой только относительно конкретного набора тестов. Некоторые тестируемые системы играют роль вызываемого компонента для других тестируемых систем

Зависящая от языка терминология xUnit

Хотя в этой книге содержатся примеры на основе различных языков и различных продуктов из семейства xUnit, очень часто используется продукт JUnit, так как многие знакомы с этой инфраструктурой. Существует несколько практически полных переводов JUnit на другие языки с минимальными модификациями имен классов и методов (в соответствии с особенностями используемого языка). Не всегда реализации оказываются полными. В таком случае обратитесь к приложению Б, “Терминология xUnit”, в котором приводится соответствие между терминами в разных реализациях xUnit.

Использование Java как основного языка для примеров означает, что в некоторых случаях будут использоваться названия методов JUnit и не будут приводиться соответствующие названия из других инфраструктур xUnit. Например, в обсуждении может указываться имя метода `assertTrue` (JUnit) без указания, что эквивалентом в NUnit является `Assert.IsTrue`, в SUnit — `should:`, а в VbUnit — `verify`. Предполагается самостоятельное выполнение замены названий методов для SUnit, VbUnit, Test::Unit и других эквивалентов. Указывающих намерение названий методов JUnit должно быть достаточно для обсуждения интересующей тематики.

Примеры кода

Примеры кода всегда являются проблемой. Примеры из реальных проектов чаще всего слишком велики и связаны соглашением о неразглашении, запрещающим публикацию. “Игрушечные программы” не вызывают уважения, так как “они не настоящие”. В данной книге практически нет выбора, кроме как использовать “игрушечные программы”, но автором были приложены все усилия, чтобы сделать их максимально похожими на реальные проекты.

Практически все приведенные здесь примеры кода взяты из настоящего компилируемого и выполняемого кода, а значит, они не должны (стучим по дереву) содержать ошибок компиляции, если их не внесли в процессе редактирования. Большинство примеров на языке Ruby взято из системы публикации на основе XML, которая использовалась для подготовки данной книги. Примеры на языках Java и C# взяты из учебных материалов, которые компания ClearStream использует для обучения клиентов.

Несколько языков использовались для иллюстрации универсальной применимости инфраструктуры xUnit. В одних случаях конкретный шаблон продиктовал использование конкретного языка. Это может быть связано как с особенностями языка, так и с особенностями конкретной реализации инфраструктуры xUnit. В других случаях выбор языка продиктован доступностью сторонних расширений для конкретной реализации xUnit. Если особенностей нет, по умолчанию используется язык Java или C#, так как большинство разработчиков знакомы с ними как минимум на уровне чтения уже написанного кода.

Форматирование кода для книги оказалось отдельной сложной задачей, так как ширина строки составляет всего 65 символов. Некоторые имена переменных и классов сокращены для уменьшения количества переносов строк. Кроме того, были использованы определенные соглашения по поводу переноса строк для уменьшения вертикального размера примеров. Утешением должно служить то, что реальный тестовый код будет выглядеть намного более коротким, чем здесь, из-за меньшего количества переносов строк!

Описание с помощью диаграмм

“Лучше один раз увидеть, чем сто раз услышать”. По возможности вместе с каждым тестом приводится диаграмма шаблона или запаха. В основном диаграммы базируются на языке UML (Unified Modeling Language), но несколько отступлений от стандарта позволили сделать их более выразительными. Например, используются символы агрегации (ромб) и наследования (треугольник) диаграммы классов UML, но на одной диаграмме одновременно используются классы и объекты, а также показаны ассоциации и взаимодействие объектов. Большая часть формата диаграмм описывается в главе 19, “Базовые шаблоны xUnit”. Просмотрите эту главу хотя бы из-за рисунков.

Хотя предполагалось, что формат диаграмм “интуитивно понятен”, стоит обратить внимание на несколько соглашений. Объекты имеют тени. Классы и методы их не имеют. Классы имеют прямые углы в соответствии с UML, методы имеют скругленные углы. Большие восклицательные знаки представляют *утверждение* (assertion) — потенциальное *неудачное завершение теста* (test failure). Символ звезды описывает ошибку или исключение. Облако обозначает тестовую конфигурацию. Центральный элемент диаграммы всегда выделен более жирными линиями и более темными тенями. В результате, сравнив две диаграммы, можно определить, что означает каждая из них.

Ограничения

При использовании шаблонов не забывайте, что автор не мог столкнуться со всеми проблемами автоматизации тестов и найти все решения этих проблем. Возможно, существуют лучшие способы их решения. Приведенные здесь решения просто работают для автора и его коллег. Любой совет принимайте с определенной долей сомнения!

Надеюсь, эти шаблоны станут отправной точкой для написания хороших, качественных автоматизированных тестов. При удачном стечении обстоятельств можно избежать многих ошибок, которые были допущены нами при первых попытках. Возможно, вы сумеете изобрести еще лучшие способы автоматизации тестов.

Рефакторинг тестов

Зачем нужен рефакторинг тестов

Тесты могут очень быстро превратиться в узкое место гибкого процесса разработки. Это не всегда очевидно для тех, кто не имеет опыта работы со сложными в написании и обслуживании тестами. Разница в производительности разработчика может стать разительной!

Этот раздел может рассматриваться как “мотивирующий” пример для целой книги. Здесь показано, какую пользу может принести рефакторинг теста. Пример содержит сложный тест, который поэтапно превращается в тест, простой для понимания. Будут показаны некоторые ключевые запахи и шаблоны, которые позволят избавиться от этих запахов.

Сложный тест

Ниже приведен тест, который достаточно часто встречается в различных проектах.

```
public void testAddItemQuantity_severalQuantity_v1() {
    Address billingAddress = null;
    Address shippingAddress = null;
    Customer customer = null;
    Product product = null;
    Invoice invoice = null;
    try {
        // Настройка тестовой конфигурации
        billingAddress = new Address("1222 1st St SW",
            "Calgary", "Alberta", "T2N 2V2", "Canada");
        shippingAddress = new Address("1333 1st St SW",
            "Calgary", "Alberta", "T2N 2V2", "Canada");
        customer = new Customer(99, "John", "Doe",
            new BigDecimal("30"),
            billingAddress,
            shippingAddress);
        product = new Product(88, "SomeWidget",
            new BigDecimal("19.99"));
        invoice = new Invoice(customer);
        // Вызов тестируемой системы
        invoice.addItemQuantity(product, 5);
        // Проверка результата
        List lineItems = invoice.getLineItems();
        if (lineItems.size() == 1) {
            LineItem actItem = (LineItem) lineItems.get(0);
            assertEquals("inv", invoice, actItem.getInv());
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```

        assertEquals("prod", product, actItem.getProd());
        assertEquals("quant", 5, actItem.getQuantity());
        assertEquals("discount", new BigDecimal("30"),
                    actItem.getPercentDiscount());
        assertEquals("unit price", new BigDecimal("19.99"),
                    actItem.getUnitPrice());
        assertEquals("extended", new BigDecimal("69.96"),
                    actItem.getExtendedPrice());
    } else {
        assertTrue("Invoice should have 1 item", false);
    }
} finally {
    // Очистка
    deleteObject(invoice);
    deleteObject(product);
    deleteObject(customer);
    deleteObject(billingAddress);
    deleteObject(shippingAddress);
}
}

```

Это достаточно длинный тест¹, и он намного сложнее, чем нужно. Этот *непонятный тест* (Obscure Test, с. 230) сложно понять, так как большое количество строк не позволяет увидеть общую картину. Кроме того, тест содержит в себе и другие проблемы, которые будут рассматриваться одна за другой.

Очистка теста

Рассмотрим каждую часть теста в отдельности.

Очистка логики верификации

Сначала стоит сфокусироваться на ожидаемом результате. Можно исходить из утверждений, в которых тест пытается проверять тестовые условия.

```

List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actItem = (LineItem) lineItems.get(0);
    assertEquals("inv", invoice, actItem.getInv());
    assertEquals("prod", product, actItem.getProd());
    assertEquals("quant", 5, actItem.getQuantity());
    assertEquals("discount", new BigDecimal("30"),
                actItem.getPercentDiscount());
    assertEquals("unit price", new BigDecimal("19.99"),
                actItem.getUnitPrice());
    assertEquals("extended", new BigDecimal("69.96"),
                actItem.getExtendedPrice());
} else {
}

```

¹ Хотя перенос строк по границе в 65 символов делает тест визуально еще более длинным, он на самом деле длиннее, чем нужно. Он содержит 25 выполняемых инструкций, включая инициализацию, 6 операторов управления, 4 комментария и 2 строки декларации тестового метода. В результате получается 37 строк кода без переносов строк.

```

    assertTrue("Invoice should have 1 item", false);
}

```

Достаточно просто будет исправить тривиальное утверждение в последней строке. Вызов `assertTrue` с аргументом `false` всегда приводит к неудачному завершению теста. Почему бы не указать на это прямо? Изменим вызов на `fail`.

```

List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actItem = (LineItem) lineItems.get(0);
    assertEquals("inv", invoice, actItem.getInv());
    assertEquals("prod", product, actItem.getProd());
    assertEquals("quant", 5, actItem.getQuantity());
    assertEquals("discount", new BigDecimal("30"),
                actItem.getPercentDiscount());
    assertEquals("unit price", new BigDecimal("19.99"),
                actItem.getUnitPrice());
    assertEquals("extended", new BigDecimal("69.96"),
                actItem.getExtendedPrice());
} else {
    fail("Invoice should have exactly one line item");
}

```

Этот ход можно рассматривать как рефакторинг *выделение метода* (Extract Method), так как происходит замена *утверждения с заявленным результатом* (Stated Outcome Assertion; см. *Метод с утверждением*, Assertion Method, с. 390) с жестко установленным параметром на более интуитивно понятный вызов метода *утверждения с единственным результатом* (Single Outcome Assertion; см. *Метод с утверждением*, Assertion Method).

Конечно, данный набор утверждений содержит еще несколько вопросов. Например, почему их так много? Оказывается, многие утверждения проверяют поля, устанавливаемые конструктором объекта `LineItem`, который проверяется другим модульным тестом. Зачем повторять эти утверждения здесь, если позднее придется обслуживать больше кода при изменении в логике работы.

Одно из решений — использовать единственное утверждение по отношению к *ожидаемому объекту* (Expected Object; см. *Проверка состояния*, State Verification, с. 484) вместо отдельных утверждений для каждого поля объекта. Сначала определяется объект, выглядящий точно так же, как должен выглядеть результат. В данном случае создается ожидаемый объект `LineItem`, полям которого присваиваются ожидаемые значения, включая `unitPrice` и `extendedPrice`, инициализируемые из объекта `product`.

```

List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem expected =
        new LineItem(invoice, product, 5,
                     new BigDecimal("30"),
                     new BigDecimal("69.96"));
    LineItem actItem = (LineItem) lineItems.get(0);
    assertEquals("invoice", expected.getInv(),
                actItem.getInv());
    assertEquals("product", expected.getProd(),
                actItem.getProd());
    assertEquals("quantity", expected.getQuantity(),
                actItem.getQuantity());
}

```

46 Рефакторинг тестов

```
assertEquals("discount",
            expected.getPercentDiscount(),
            actItem.getPercentDiscount());
assertEquals("unit pr", new BigDecimal("19.99"),
            actItem.getUnitPrice());
assertEquals("extend pr", new BigDecimal("69.96"),
            actItem.getExtendedPrice());
} else {
    fail("Invoice should have exactly one line item");
}
```

После создания *ожидаемого объекта* (Expected Object) утверждение относительно него можно проверить с помощью вызова `assertEquals`.

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem expected =
        new LineItem(invoice, product, 5,
                     new BigDecimal("30"),
                     new BigDecimal("69.96"));
    LineItem actItem = (LineItem) lineItems.get(0);
    assertEquals("invoice", expected, actItem);
} else {
    fail("Invoice should have exactly one line item");
}
```

Очевидно, что рефакторинг *сохранения целого объекта* (Preserve Whole Object) делает код намного проще и значительно очевиднее. Но осталось еще кое-что! Что делает оператор `if` в коде теста? Если тест имеет несколько ветвей выполнения, как определить, какая из них выполняется в данный момент? Желательно избавиться от *условной логики места* (Conditional Test Logic, с. 243). К счастью, шаблон *сторожевое утверждение* (Guard Assertion, с. 510) предназначен для решения именно такой проблемы. Достаточно просто воспользоваться *заменой условия на сторожевой код* (Replace Conditional with Guard Clause) для замены последовательности `if... else fail()...` утверждением с тем же условием. В таком случае *сторожевое утверждение* (Guard Assertion) останавливает выполнение при невыполнении условия, но при этом код не содержит *условной логики места* (Conditional Test Logic).

```
List lineItems = invoice.getLineItems();
assertEquals("number of items", 1, lineItems.size());
LineItem expected =
    new LineItem(invoice, product, 5,
                 new BigDecimal("30"),
                 new BigDecimal("69.96"));
LineItem actItem = (LineItem) lineItems.get(0);
assertEquals("invoice", expected, actItem);
```

На этом этапе проверочный код сократился с 11 до 4 строк, а оставшиеся строки намного проще². Кое-кто может счесть этот результат достаточно хорошим. Но можно ли сделать тест еще более очевидным? Что в данном случае проверяется на самом деле? Тест

² Хорошо, что в нашем случае оплата не зависит от количества написанных строк кода! Это еще один пример неуклюжести количества строк кода в качестве метрики производительности.

пытается доказать, что должен существовать только один линейный элемент и он должен выглядеть точно так, как созданный `expectedLineItem`. На это можно явно указать, воспользовавшись рефакторингом *выделение метода* (Extract Method) для определения *специального утверждения* (Custom Assertion, с. 495).

```
LineItem expected =
    new LineItem(invoice, product, 5,
        new BigDecimal("30"
        new BigDecimal("69.96"
assertContainsExactlyOneLineItem(invoice, expected);
```

Так намного лучше! Теперь, когда проверочная часть теста сведена до двух строк, можно еще раз посмотреть на тест в целом.

```
public void testAddItemQuantity_severalQuantity_v6() {
    Address billingAddress = null;
    Address shippingAddress = null;
    Customer customer = null;
    Product product = null;
    Invoice invoice = null;
    try {
        // Настройка тестовой конфигурации
        billingAddress = new Address("1222 1st St SW",
            "Calgary", "Alberta", "T2N 2V2", "Canada");
        shippingAddress = new Address("1333 1st St SW",
            "Calgary", "Alberta", "T2N 2V2", "Canada");
        customer = new Customer(99, "John", "Doe",
            new BigDecimal("30"),
            billingAddress,
            shippingAddress);
        product = new Product(88, "SomeWidget",
            new BigDecimal("19.99"));
        invoice = new Invoice(customer);
        // Вызов тестируемой системы
        invoice.addItemQuantity(product, 5);
        // Проверка результата
        LineItem expected =
            new LineItem(invoice, product, 5,
                new BigDecimal("30"
                new BigDecimal("69.96"));
        assertContainsExactlyOneLineItem(invoice, expected);
    } finally {
        // Очистка
        deleteObject(invoice);
        deleteObject(product);
        deleteObject(customer);
        deleteObject(billingAddress);
        deleteObject(shippingAddress);
    }
}
```

Очистка логики удаления тестовой конфигурации

Теперь, когда логика проверки результата очищена, можно обратить внимание на блок `finally` в конце теста. Что делает этот код?

```

} finally {
    // Очистка
    deleteObject(invoice);
    deleteObject(product);
    deleteObject(customer);
    deleteObject(billingAddress);
    deleteObject(shippingAddress);
}

```

Большинство современных языков имеют конструкции, эквивалентные блоку `try/finally`. Они используются для гарантированного выполнения кода даже при появлении ошибки или исключения. В *тестовом методе* (Test Method, с. 378) блок `finally` обеспечивает запуск кода очистки вне зависимости от успешного или неудачного завершения теста. Ложное утверждение приводит к генерации исключения, которое передает управление коду обработки исключений в *инфраструктуре автоматизации тестов* (Test Automation Framework, с. 332). Поэтому блок `finally` используется для очистки результатов работы теста. Такой подход позволяет обойтись без перехвата и повторной генерации исключения.

В этом тесте блок `finally` вызывает метод `deleteObject` для каждого созданного тестом объекта. К сожалению, этот код имеет один очень серьезный недостаток. Медаль тому, кто его уже заметил!

Ошибка может произойти в процессе очистки. Что произойдет, если первый вызов метода `deleteObject` сгенерирует исключение? В соответствии с приведенным кодом остальные вызовы `deleteObject` не будут выполнены. Решением может стать помещение первого вызова в блок `try/finally`, чтобы обеспечить гарантированный второй вызов метода `deleteObject`. Но что если неудачно завершится второй вызов? В таком случае потребуется шесть вложенных блоков `try/finally` для полной реализации такого маневра. А это может привести к удвоению размеров теста. Затраты на написание и обслуживание такого кода в каждом тесте могут оказаться слишком большими.

```

} finally {
    // Очистка
    try {
        deleteObject(invoice);
    } finally {
        try {
            deleteObject(product);
        } finally {
            try {
                deleteObject(customer);
            } finally {
                try {
                    deleteObject(billingAddress);
                } finally {
                    deleteObject(shippingAddress);
                }
            }
        }
    }
}

```

В результате мы получили *сложную очистку* (Complex Teardown; см. *Непонятный тест*, Obscure Test). Какова вероятность правильного написания такого кода? Как проверить его работоспособность? Очевидно, что текущий подход неэффективен.

Конечно, этот код можно перенести в метод `tearDown`. Тогда он исчезнет из *тестового метода* (Test Method). Кроме того, поскольку метод `tearDown` будет работать как блок `finally`, от внешнего блока `try/finally` можно будет избавиться. К сожалению, эта стратегия не решает основной проблемы: необходимости писать сложный код удаления в каждом teste.

Можно попытаться избежать создания объектов в начале теста, воспользовавшись *общей тестовой конфигурацией* (Shared Fixture, с. 350), которая не уничтожается по завершении теста. К сожалению, этот подход с большой вероятностью приводит к появлению различных запахов, включая *неповторяемый тест* (Unrepeatable Test; см. *Нестабильный тест*, Erratic Test) и *взаимодействующие тесты* (Interacting Tests), связанных с использованием *общей тестовой конфигурации* (Shared Fixture). Еще одной проблемой может стать превращение ссылок на *общую тестовую конфигурацию* (Shared Fixture) в *мнимых гостей* (Mystery Guest; см. *Непонятный тест*, Obscure Test)³.

Лучшим решением является использование *новой тестовой конфигурации* (Fresh Fixture, с. 344), но отказ от создания кода удаления объектов в каждом teste. Для этого можно оставить тестовую конфигурацию в памяти, что приведет к ее автоматическому удалению сборщиком мусора. Однако такой подход не сработает, если создаваемые объекты предназначены для длительного хранения (например, если они сохраняются в базе данных). Хотя имеет смысл проектировать архитектуру системы таким образом, чтобы тесты могли работать и без базы данных, практически всегда существуют тесты, требующие доступа к базе данных.

В таких случаях можно расширить *инфраструктуру автоматизации тестов* (Test Automation Framework) всей необходимой функциональностью. Можно реализовать возможность регистрировать каждый созданный объект, чтобы инфраструктура удаляла эти объекты самостоятельно.

Сначала каждый объект необходимо зарегистрировать при создании.

```
// Настройка тестовой конфигурации
billingAddress = new Address("1222 1st St SW", "Calgary",
    "Alberta", "T2N 2V2", "Canada");
registerTestObject(billingAddress);
shippingAddress = new Address("1333 1st St SW", "Calgary",
    "Alberta", "T2N 2V2", "Canada");
registerTestObject(shippingAddress);
customer = new Customer(99, "John", "Doe",
    new BigDecimal("30"),
    billingAddress,
    shippingAddress);
registerTestObject(shippingAddress);
product = new Product(88, "SomeWidget",
    new BigDecimal("19.99"));
registerTestObject(shippingAddress);
invoice = new Invoice(customer);
registerTestObject(shippingAddress);
```

³ При чтении теста не видны объекты, которые используются тестом.

50 Рефакторинг тестов

Регистрация предусматривает добавление объекта в коллекцию тестовых объектов.

```
List testObjects;
protected void setUp() throws Exception {
    super.setUp();
    testObjects = new ArrayList();
}
protected void registerTestObject(Object testObject) {
    testObjects.add(testObject);
}
```

В методе `testObject` происходит итерация по списку тестовых объектов и удаление каждого из них.

```
public void tearDown() {
    Iterator i = testObjects.iterator();
    while (i.hasNext()) {
        try {
            deleteObject(i.next());
        } catch (RuntimeException e) {
            // Ничего не делаем. Нужно просто убедиться
            // что мы переходим к следующему объекту в списке.
        }
    }
}
```

Теперь тест выглядит так.

```
public void testAddItemQuantity_severalQuantity_v8() {
    Address billingAddress = null;
    Address shippingAddress = null;
    Customer customer = null;
    Product product = null;
    Invoice invoice = null;
    // Настройка тестовой конфигурации
    billingAddress = new Address("1222 1st St SW", "Calgary",
        "Alberta", "T2N 2V2", "Canada");
    registerTestObject(billingAddress);
    shippingAddress = new Address("1333 1st St SW", "Calgary",
        "Alberta", "T2N 2V2", "Canada");
    registerTestObject(shippingAddress);
    customer = new Customer(99, "John", "Doe",
        new BigDecimal("30"),
        billingAddress,
        shippingAddress);
    registerTestObject(shippingAddress);
    product = new Product(88, "SomeWidget",
        new BigDecimal("19.99"));
    registerTestObject(shippingAddress);
    invoice = new Invoice(customer);
    registerTestObject(shippingAddress);
    // Вызовируемой системы
    invoice.addItemQuantity(product, 5);
    // Проверка результата
    LineItem expected =
        new LineItem(invoice, product, 5,
        new BigDecimal("30")),
        new BigDecimal("19.99"),
        new BigDecimal("159.95"));
```

```

        new BigDecimal("69.96"));
assertContainsExactlyOneLineItem(invoice, expected);
}

```

В результате удалось избавиться от блока `try/finally`, и, кроме вызова `registerTestObject`, код стал намного проще. Но и этот код можно немного упростить. Например, зачем объявлять переменные и инициализировать их значением `null`, если позднее им будут присвоены другие значения? В исходном тесте эти операции были нужны, так как они должны были быть видны в блоке `finally`. Теперь, когда этот блок удален, можно объединить декларацию и инициализацию.

```

public void testAddItemQuantity_severalQuantity_v9() {
    // Настройка тестовой конфигурации
    Address billingAddress = new Address("1222 1st St SW",
                                         "Calgary", "Alberta", "T2N 2V2", "Canada");
    registerTestObject(billingAddress);
    Address shippingAddress = new Address("1333 1st St SW",
                                         "Calgary", "Alberta", "T2N 2V2", "Canada");
    registerTestObject(shippingAddress);
    Customer customer = new Customer(99, "John", "Doe",
                                      new BigDecimal("30"),
                                      billingAddress,
                                      shippingAddress);
    registerTestObject(shippingAddress);
    Product product = new Product(88, "SomeWidget",
                                   new BigDecimal("19.99"));
    registerTestObject(shippingAddress);
    Invoice invoice = new Invoice(customer);
    registerTestObject(shippingAddress);
    // Вызов тестируемой системы
    invoice.addItemQuantity(product, 5);
    // Проверка результата
    LineItem expected =
        new LineItem(invoice, product, 5,
                     new BigDecimal("30"),
                     new BigDecimal("69.95"));
    assertContainsExactlyOneLineItem(invoice, expected);
}

```

Очистка кода инициализации

Очистив список утверждений и процедуру удаления тестовой конфигурации, можно переходить к процедуре создания тестовой конфигурации. Очевидным “быстрым решением” будет использование рефакторинга *выделение метода* (Extract Method) по отношению к каждому вызову конструктора и последующему вызову `registerTestObject` для получения *метода создания* (Creation Method, с. 441). В результате тест станет проще для чтения и модификации. Использование *метода создания* (Creation Method) имеет еще одно преимущество: он обеспечивает скрытие программного интерфейса тестируемой системы и сокращает затраты на обслуживание теста в результате модификации конструкторов объектов (изменения придется вносить только в одном месте, а не в каждом teste).

52 Рефакторинг тестов

```
public void testAddItemQuantity_severalQuantity_v10() {
    // Настройка тестовой конфигурации
    Address billingAddress =
        createAddress( "1222 1st St SW", "Calgary", "Alberta",
                      "T2N 2V2", "Canada");
    Address shippingAddress =
        createAddress( "1333 1st St SW", "Calgary", "Alberta",
                      "T2N 2V2", "Canada");
    Customer customer =
        createCustomer( 99, "John", "Doe", new BigDecimal("30"),
                        billingAddress, shippingAddress);
    Product product =
        createProduct( 88, "SomeWidget", new BigDecimal("19.99"));
    Invoice invoice = createInvoice(customer);
    // Вызов тестируемой системы
    invoice.addItemQuantity(product, 5);
    // Проверка результата
    LineItem expected =
        new LineItem(invoice, product, 5, new BigDecimal("30"),
                     new BigDecimal("69.96"));
    assertContainsExactlyOneLineItem(invoice, expected);
}
```

Такая логика создания конфигурации все еще содержит несколько проблем. Первая связана со сложностью прогнозирования результата теста на основе тестовой конфигурации. Зависит ли результат от особенностей конкретного потребителя? Зависит ли результат от адреса потребителя? Что на самом деле проверяет данный тест?

Также для этого теста характерна проблема *фиксированных данных теста* (Hard-Coded Test Data; см. *Непонятный тест*, Obscure Test). Учитывая, что тестируемая система сохраняет все создаваемые объекты в базе данных, использование *фиксированных данных теста* (Hard-Coded Test Data) может привести к появлению *неповторяемых тестов* (Unrepeatable Test), *взаимодействующих тестов* (Interacting Tests) или возникновению *войны* запуска тестов (Test Run War; см. *Нестабильный тест*, Erratic Test), если любое из полей потребителя, продукта или счета должно быть уникальным.

Для решения этой проблемы можно генерировать уникальное значение для каждого теста и с помощью этого значения заполнять атрибуты создаваемых объектов. Такой подход обеспечивает создание уникальных объектов при каждом запуске теста. Так как логика создания объектов уже вынесена в *метод создания* (Creation Method), выполнить эту операцию достаточно легко. Необходимая логика просто помещается в *метод создания* (Creation Method), и соответствующие параметры удаляются. Это еще одно приложение для рефакторинга *выделение метода* (Extract Method), в котором создается новая версия *метода создания* (Creation Method) без параметров.

```
public void testAddItemQuantity_severalQuantity_v11() {
    final int QUANTITY = 5;
    // Настройка тестовой конфигурации
    Address billingAddress = createAnAddress();
    Address shippingAddress = createAnAddress();
    Customer customer = createACustomer(new BigDecimal("30"),
                                          billingAddress, shippingAddress);
    Product product = createAProduct(new BigDecimal("19.99"));
    Invoice invoice = createInvoice(customer);
    // Вызов тестируемой системы
```

```

        invoice.addItemQuantity(product, QUANTITY);
        // Проверка результата
        LineItem expected =
            new LineItem(invoice, product, 5, new BigDecimal("30"),
                         new BigDecimal("69.96"));
        assertContainsExactlyOneLineItem(invoice, expected);
    }
private Product createAProduct(BigDecimal unitPrice) {
    BigDecimal uniqueId = getUniqueNumber();
    String uniqueString = uniqueId.toString();
    return new Product(uniqueId.toBigInteger().intValue(),
                       uniqueString, unitPrice);
}
}

```

Этот шаблон называется *анонимный метод создания* (Anonymous Creation Method; см. *Метод создания*, Creation Method), так как явно указывается, что значения свойств объекта метод не интересуют. Если ожидаемое поведение тестируемой системы зависит от конкретного значения, это значение можно передать в качестве параметра или указать на него неявно в названии метода создания объекта.

Теперь тест выглядит намного лучше, но работа еще не полностью завершена. Зависит ли ожидаемый результат от адреса потребителя? Если не зависит, создание объекта можно полностью скрыть с помощью рефакторинга *выделение метода* (Extract Method) — опять! — для написания версии метода `createACustomer`, создающей объект.

```

public void testAddItemQuantity_severalQuantity_v12() {
    // Настройка тестовой конфигурации
    Customer cust = createACustomer(new BigDecimal("30"));
    Product prod = createAProduct(new BigDecimal("19.99"));
    Invoice invoice = createInvoice(cust);
    // Вызов тестируемой системы
    invoice.addItemQuantity(prod, 5);
    // Проверка результата
    LineItem expected = new LineItem(invoice, prod, 5,
                                      new BigDecimal("30"), new BigDecimal("69.96"));
    assertContainsExactlyOneLineItem(invoice, expected);
}

```

Перенос создающих адрес вызовов в метод, который генерирует потребителя, явно указывает на то, что адрес не влияет на проверяемую тестом логику. Но результат зависит от скидки потребителя, поэтому процент скидки передается в качестве параметра методу создания клиента.

Осталось внести еще пару правок. Например, необходимо убрать дважды повторяющиеся *фиксированные данные теста* (Hard-Coded Test Data) для цены единицы, количества и скидки. Назначение этих чисел можно подчеркнуть с помощью рефакторинга *замена магического числа символьной константой* (Replace Magic Number with Symbolic Constant). Кроме этого, используемый для создания объекта `LineItem` конструктор не применяется в тестируемой системе, так как обычно объект `LineItem` вычисляется значение `extendCost` в момент создания. Этот фрагмент теста имеет смысл превратить во *внешний метод* (Foreign Method), реализованный в пределах тестовой конфигурации. Примером такой модификации могут служить объекты `Customer` и `Product`: для возврата основанного на интересующих значениях ожидаемого объекта `LineItem` исполь-

зуется *параметризованный метод создания* (Parameterized Creation Method; см. *Метод создания*, Creation Method).

```
public void testAddItemQuantity_severalQuantity_v13() {
    final int QUANTITY = 5;
    final BigDecimal UNIT_PRICE = new BigDecimal("19.99");
    final BigDecimal CUST_DISCOUNT_PC = new BigDecimal("30");
    // Настройка тестовой конфигурации
    Customer customer = createACustomer(CUST_DISCOUNT_PC);
    Product product = createAProduct(UNIT_PRICE);
    Invoice invoice = createInvoice(customer);
    // Вызов тестируемой системы
    invoice.addItemQuantity(product, QUANTITY);
    // Проверка результата
    final BigDecimal EXTENDED_PRICE = new BigDecimal("69.96");
    LineItem expected =
        new LineItem(invoice, product, QUANTITY,
                     CUST_DISCOUNT_PC, EXTENDED_PRICE);
    assertContainsExactlyOneLineItem(invoice, expected);
}
```

И последнее замечание: откуда взялось значение “69.96”? Если оно берется из вывода другой системы, это необходимо указать явно. Так как данное значение вычислялось вручную и было просто вписано в код теста, для удобства сопровождения теста процесс расчета можно реализовать в пределах теста.

Тест после всех модификаций

Ниже приведена последняя версия теста.

```
public void testAddItemQuantity_severalQuantity_v14() {
    final int QUANTITY = 5;
    final BigDecimal UNIT_PRICE = new BigDecimal("19.99");
    final BigDecimal CUST_DISCOUNT_PC = new BigDecimal("30");
    // Настройка тестовой конфигурации
    Customer customer = createACustomer(CUST_DISCOUNT_PC);
    Product product = createAProduct(UNIT_PRICE);
    Invoice invoice = createInvoice(customer);
    // Вызов тестируемой системы
    invoice.addItemQuantity(product, QUANTITY);
    // Проверка результата
    final BigDecimal BASE_PRICE =
        UNIT_PRICE.multiply(new BigDecimal(QUANTITY));
    final BigDecimal EXTENDED_PRICE =
        BASE_PRICE.subtract(BASE_PRICE.multiply(
            CUST_DISCOUNT_PC.movePointLeft(2)));
    LineItem expected =
        createLineItem(QUANTITY, CUST_DISCOUNT_PC,
                      EXTENDED_PRICE, product, invoice);
    assertContainsExactlyOneLineItem(invoice, expected);
}
```

В целях документирования вычисления значения BASE_PRICE (ценахколичество) и EXTENDED_PRICE (цена со скидкой) использовался рефакторинг *введение описываю-*

щей переменной (Introduce Explaining Variable). Теперь модифицированный тест выглядит намного меньшим и более понятным, чем изначальный громоздкий код. Данный тест хорошо играет роль документации (Tests as Documentation, с. 79). Так что же проверяет этот тест? Тест подтверждает, что добавляемые в счет строки действительно добавлены и расширенная стоимость основана на цене продукта, скидке потребителя и заказанном количестве.

Написание других тестов

Складывается впечатление, что на рефакторинг предыдущего теста потребовалось достаточно много усилий. Неужели придется так работать над каждым тестом?

Хотелось бы надеяться, что нет! Большая часть усилий была направлена на выбор вспомогательных методов теста (Test Utility Method, с. 610), необходимых для написания теста. Для тестирования приложения был определен язык высокого уровня (Higher-Level Language, с. 95). После создания этих методов написание новых тестов будет происходить значительно проще. Например, если необходимо написать тест, проверяющий пересчет расширенной стоимости при изменении количества объектов LineItem, можно повторно воспользоваться большинством вспомогательных методов теста (Test Utility Method).

```
public void testAddLineItem_quantityOne() {
    final BigDecimal BASE_PRICE = UNIT_PRICE;
    final BigDecimal EXTENDED_PRICE = BASE_PRICE;
    // Настройка тестовой конфигурации
    Customer customer = createACustomer(NO_CUST_DISCOUNT);
    Invoice invoice = createInvoice(customer);
    // Вызов тестируемой системы
    invoice.addItemQuantity(PRODUCT, QUAN_ONE);
    // Проверка результата
    LineItem expected =
        createLineItem(QUAN_ONE, NO_CUST_DISCOUNT,
                      EXTENDED_PRICE, PRODUCT, invoice);
    assertContainsExactlyOneLineItem(invoice, expected);
}
public void testChangeQuantity_severalQuantity(){
    final int ORIGINAL_QUANTITY = 3;
    final int NEW_QUANTITY = 5;
    final BigDecimal BASE_PRICE =
        UNIT_PRICE.multiply(new BigDecimal(NEW_QUANTITY));
    final BigDecimal EXTENDED_PRICE =
        BASE_PRICE.subtract(BASE_PRICE.multiply(
            CUST_DISCOUNT_PC.movePointLeft(2)));
    // Настройка тестовой конфигурации
    Customer customer = createACustomer(CUST_DISCOUNT_PC);
    Invoice invoice = createInvoice(customer);
    Product product = createAProduct(UNIT_PRICE);
    invoice.addItemQuantity(product, ORIGINAL_QUANTITY);
    // Вызов тестируемой системы
    invoice.changeQuantityForProduct(product, NEW_QUANTITY);
    // Проверка результата
    LineItem expected = createLineItem(NEW_QUANTITY,
        CUST_DISCOUNT_PC, EXTENDED_PRICE, PRODUCT, invoice);
    assertContainsExactlyOneLineItem(invoice, expected);
}
```

Этот тест был написан примерно за две минуты и не потребовал создания дополнительных *вспомогательных методов теста* (Test Utility Method). Сравните это время со временем, которое потребовалось бы для написания полностью нового теста в исходном стиле. Обратите внимание, что сохранение времени на написание теста не является единственной выгодой. Учтите и время, которое требуется для понимания особенностей каждого теста при его модификации. В процессе разработки проекта и при последующем обслуживании общая экономия времени окажется достаточно заметной.

Дальнейшее упрощение

Новые тесты показали, что код содержит еще несколько источников *дублирования тестового кода* (Test Code Duplication, с. 254). Например, каждый раз создаются объекты *Customer* и *Invoice*. Почему бы не объединить эти две строки? Точно так определяются и инициализируются константы *QUANTITY* и *CUSTOMER_DISCOUNT_PC*. Нельзя ли все эти операции выполнить не последовательно, а одновременно. Объект *Product* не играет никакой роли в этих тестах и всегда создается одинаково. Можно ли вынести эту функциональность за пределы теста? Конечно! Для этого достаточно применить рефакторинг *выделение метода* (Extract Method) к каждому фрагменту дублирующегося кода. В результате будут получены более мощные *методы создания* (Creation Method).

```
public void testAddItemQuantity_severalQuantity_v15() {
    // Настройка тестовой конфигурации
    Invoice invoice = createCustomerInvoice(CUST_DISCOUNT_PC);
    // Вызов тестируемой системы
    invoice.addItemQuantity(PRODUCT, SEVERAL);
    // Проверка результата
    final BigDecimal BASE_PRICE =
        UNIT_PRICE.multiply(new BigDecimal(SEVERAL));
    final BigDecimal EXTENDED_PRICE =
        BASE_PRICE.subtract(BASE_PRICE.multiply(
            CUST_DISCOUNT_PC.movePointLeft(2)));
    LineItem expected = createLineItem(SEVERAL,
        CUST_DISCOUNT_PC, EXTENDED_PRICE, PRODUCT, invoice);
    assertContainsExactlyOneLineItem(invoice, expected);
}
public void testAddLineItem_quantityOne_v2() {
    final BigDecimal BASE_PRICE = UNIT_PRICE;
    final BigDecimal EXTENDED_PRICE = BASE_PRICE;
    // Настройка тестовой конфигурации
    Invoice invoice = createCustomerInvoice(NO_CUST_DISCOUNT);
    // Вызов тестируемой системы
    invoice.addItemQuantity(PRODUCT, QUAN_ONE);
    // Проверка результата
    LineItem expected = createLineItem(SEVERAL,
        CUST_DISCOUNT_PC, EXTENDED_PRICE, PRODUCT, invoice);
    assertContainsExactlyOneLineItem(invoice, expected);
}
public void testChangeQuantity_severalQuantity_V2() {
    final int NEW_QUANTITY = SEVERAL + 2;
    final BigDecimal BASE_PRICE =
        UNIT_PRICE.multiply(new BigDecimal(NEW_QUANTITY));
    final BigDecimal EXTENDED_PRICE =
```

```

    BASE_PRICE.subtract(BASE_PRICE.multiply(
        CUST_DISCOUNT_PC.movePointLeft(2)));
// Настройка тестовой конфигурации
Invoice invoice = createCustomerInvoice(CUST_DISCOUNT_PC);
invoice.addItemQuantity(PRODUCT, SEVERAL);
// Вызов testируемой системы
invoice.changeQuantityForProduct(PRODUCT, NEW_QUANTITY);
// Проверка результата
LineItem expected = createLineItem(NEW_QUANTITY,
    CUST_DISCOUNT_PC, EXTENDED_PRICE, PRODUCT, invoice);
assertContainsExactlyOneLineItem(invoice, expected);
}

```

На этом этапе объем кода уменьшился⁴ с 35 операторов до 6. В результате придется поддерживать только одну шестую исходного кода! Можно пойти еще дальше и вынести создание тестовой конфигурации в отдельный метод `setUp`, но это будет иметь смысл только в случае, если множество тестов будут использовать одинаковую конфигурацию объектов `Customer/Discount/Invoice`. Если эти *вспомогательные методы теста* (Test Utility Method) должны повторно использоваться другими *классами теста* (Testcase Class, с. 401), можно воспользоваться рефакторингом *выделение суперкласса* (Extract Superclass) для создания *суперкласса теста* (Testcase Superclass, с. 646), после чего использовать рефакторинг *повышения метода уровнем выше* (Pull Up Method) для переноса *вспомогательного метода теста* (Test Utility Method).

⁴ Если игнорировать переносы строк, тест содержит 6 выполняемых операторов, окруженных двумя строками декларации и завершения метода.

Часть I

Общая информация

Глава 1

Краткий обзор

О чём идет речь в этой главе

В этой книге описывается множество принципов, шаблонов и запахов. Однако существует большое количество шаблонов, которые в данной книге не рассматриваются. Нужно ли изучать все шаблоны? Придется ли все шаблоны использовать в своей работе? Скорее всего, нет! В этой главе приводится краткий обзор основных концепций книги. Ее можно использовать как краткое введение, прежде чем подробно знакомиться с конкретными шаблонами или запахами.

Самая простая рабочая стратегия автоматизации тестирования

Существует простая стратегия автоматизации тестирования, которая будет работать для очень большого количества проектов. Данная минимальная стратегия рассматривается в этом разделе. Упомянутые здесь принципы, шаблоны и запахи являются основными и будут использоваться часто и долго. Научившись эффективно их использовать, можно быть уверенными, что автоматизация тестирования также будет успешной. Если не удается заставить работать минимальную стратегию с применением этих шаблонов, можно воспользоваться альтернативными шаблонами, приведенными в полном описании каждого базового шаблона.

Простую стратегию можно разделить на пять частей.

- **Процесс разработки.** Насколько используемый процесс разработки кода затрагивает тесты.
- **Приемочные тесты.** Первые тесты должны быть главным определением внешнего вида готового продукта.
- **Модульные тесты.** Позволяют поэтапно разрабатывать дизайн и обеспечивать тестирование всего кода.
- **Проектирование с учетом тестирования.** Упрощают тестирование, а значит, снижают стоимость автоматизации тестирования.
- **Организация тестов.** Как организовать *тестовые методы* (Test Method, с. 378) и *классы теста* (Testcase Class, с. 401).

Процесс разработки

Сначала о самом главном: когда нужно писать тесты? Создание тестов до написания кода несет ряд преимуществ. В частности, можно заранее определить, как будет выглядеть успешное решение¹.

Начиная разработку нового программного продукта, мы практикуем **разработку на основе тестов историй** (storytest-driven development), автоматизируя набор **приемочных тестов** (customer tests), которые проверяют предоставляемую приложением функциональность. Для тестирования всех частей приложения приемочные тесты дополняются набором **модульных тестов** (unit tests), которые проверяют все ветки кода или хотя бы все ветки кода, не проверяемые приемочными тестами. Можно воспользоваться утилитами анализа кода для выявления непокрытых тестами фрагментов. По результатам анализа можно добавить модульные тесты для непокрытого тестами кода².

Организация модульных тестов и приемочных тестов в отдельные **наборы тестов** (test suites) позволяет обеспечить независимый запуск модульных или приемочных тестов. Модульные тесты всегда должны завершаться успешно перед включением разработанного кода в общее хранилище. Для обеспечения достаточной частоты запуска модульные тесты могут быть включены в **контрольный набор тестов** (Smoke Tests), которые запускаются в процессе **интеграционной компиляции** (Integration Build). Хотя большинство приемочных тестов будет завершаться неудачно до реализации соответствующей функциональности, желательно запускать все завершающиеся успешно приемочные тесты на этапе интеграции (если это не слишком замедляет процесс компиляции). В таком случае тесты можно вынести из компиляции кода в основном хранилище и запускать их только по ночам.

Разработка на основе тестов позволяет создавать поддающееся тестированию программное обеспечение. Таким образом, модульные тесты разрабатываются раньше кода. Они же служат основой для определения дизайна программного обеспечения. Такая стратегия позволяет сосредоточиться на **бизнес-логике** (business logic), которая требует проверки в полностью определенных объектах, тестируемых независимо от базы данных. Хотя, кроме этого, должны существовать модульные тесты для уровня доступа к данным и базы данных, для бизнес-логики зависимость от базы данных должна минимизироваться.

Приемочные тесты

Приемочные тесты должны описывать суть требований потребителя продукта. Перечисление тестов до их разработки — очень важный шаг, даже если они не будут автоматизированы. Это позволяет разработчикам понять, чего хочет заказчик. Именно эти тесты определяют, как будет выглядеть успешный результат разработки. Для автоматизации тестов можно воспользоваться *тестами на основе сценария* (Scripted Test, с. 319) или *тестами, управляемыми данными* (Data-Driven Test, с. 322). При использовании *тестов*,

¹ Если потребитель не может определить тесты до начала разработки, можно начинать по-настоящему волноваться!

² Разработка на основе тестов приводит к появлению меньшего количества *отсутствующих модульных тестов* (Missing Unit Test; см. *Ошибки в продукте*, Production Bugs, с. 303), чем при использовании политики “тестировать в конце”. Несмотря на это подобные утилиты анализа кода имеет смысл использовать и при разработке на основе тестов.

управляемых данными потребитель может принять участие в их автоматизации. Иногда можно даже воспользоваться *записанными тестами* (Recorded Test, с. 312) для регрессионного тестирования существующего приложения в процессе рефакторинга для упрощения создания тестов. Конечно, эти тесты не потребуются после того, как будут созданы другие тесты для проверки функциональности, поскольку записанные тесты часто превращаются в *“хрупкие” тесты* (Fragile Test, с. 277).

В процессе разработки приемочные тесты должны описывать, как будет использоваться система. К сожалению, данная цель часто конфликтует с попытками сократить длину тестов, так как длинные тесты часто превращаются в *непонятные тесты* (Obscure Test, с. 230) и не обеспечивают достаточного уровня *локализации дефектов* (Defect Localization, с. 78), если выполнение прерывается после выполнения части теста. Кроме этого, хорошо написанные *тесты можно использовать как документацию* (Tests as Documentation, с. 79) для описания предполагаемого поведения системы. Для сохранения простоты тестов можно отказаться от обращения к пользовательскому интерфейсу и сразу воспользоваться *“подкожными тестами”* (Subcutaneous Test; см. *Тест уровня*, Layer Test, с. 368) по отношению к одному или нескольким *фасадам служб* (Service Facade) [CJ2EEP]. Фасад служб скрывает всю бизнес-логику за простым интерфейсом, который используется презентационным уровнем.

Каждому тесту нужна отправная точка. В соответствии с нашим планом тестирования каждый тест после запуска устанавливает собственную отправную точку, известную также, как *тестовая конфигурация*. Шаблон *новая тестовая конфигурация* (Fresh Fixture, с. 344) позволяет избежать появления *взаимодействующих тестов* (Interacting Tests; см. *Нестабильный тест*, Erratic Test, с. 267), делая тест независимым от элементов, которые не были созданы самим тестом. Использование шаблона *общая тестовая конфигурация* (Shared Fixture, с. 350) является нежелательным, если это не шаблон *немодифицируемая общая тестовая конфигурация* (Immutable Shared Fixture). Это позволит избежать перехода к *нестабильным тестам* (Erratic Test).

Если разрабатываемое приложение взаимодействует с другими приложениями, может потребоваться изоляция от воздействий из-за пределов среды разработки с помощью той или иной формы шаблона *тестовый двойник* (Test Double, с. 538), реализованного в виде объекта, выступающего в качестве интерфейса к другому приложению. Если тест работает слишком медленно из-за доступа к базе данных или к другому медленному компоненту, такой компонент можно заменить функционально эквивалентным *поддельным объектом* (Fake Objects, с. 565) для ускорения работы теста и стимуляции разработчиков к их более частому запуску. По возможности рекомендуется отказаться от использования *цепочек тестов* (Chained Tests, с. 477) — это просто замаскированный запах *взаимодействующих тестов* (Interacting Tests).

Модульные тесты

Для обеспечения эффективности модульных тестов каждый из них должен быть *полностью автоматизированным* (Fully Automated Test, с. 81). Каждый из них проверяет класс через его открытый интерфейс. Желательно стремиться к *локализации дефектов* (Defect Localization), создавая каждый тест как *тест одного условия* (Single Condition Test, с. 99), вызывающий единственный метод или объект в каждом сценарии. Кроме того, тесты должны быть написаны таким образом, чтобы каждая часть *четырехфазного теста*

(Four-Phase Test, с. 387) легко распознавалась. В этом случае тесты могут использоваться как **документация** (Tests as Documentation).

Стратегия на основе новой тестовой конфигурации позволяет не беспокоиться о появлениях *взаимодействующих тестов* (Interacting Tests) или о необходимости **очистки тестовой конфигурации** (fixture teardown). Сначала создается *класс теста* (Testcase Class) для каждого тестируемого класса (см. *Класс теста для каждого класса*, Testcase Class per Class, с. 627). При этом каждый тест является отдельным *тестовым методом* (Test Method) в пределах этого класса. Каждый *тестовый метод* (Test Method) может использовать *делегированную настройку* (Delegated Setup, с. 437) для создания *минимальной тестовой конфигурации* (Minimal Fixture, с. 336). В таком случае тесты легко понять благодаря использованию создающих объекты для тестовой конфигурации *методов создания* (Creation Method, с. 441) с описательными названиями.

Для создания *самопроверяющихся тестов* (Self-Checking Test, с. 81) ожидаемый результат каждого теста описывается в терминах ожидаемых объектов (Expected Objects; см. *Проверка состояния*, State Verification, с. 484) и сравнивается с реальными объектами, возвращаемыми **тестируемой системой** (system under test — SUT). Для сравнения используются встроенные *утверждения равенства* (Equality Assertions; см. *Методы с утверждением*, Assertion Method, с. 390) и *специальное утверждение* (Custom Assertion, с. 495), в которых реализовано **обусловленное тестом равенство** (test-specific equality). Если несколько тестов должны возвращать одинаковые результаты, логика проверки может быть вынесена в описывающий результат *метод проверки* (Verification Method; см. *Специальное утверждение*, Custom Assertion), который достаточно просто распознать при чтении теста.

Если обнаружился фрагмент *не тестируемый код* (Untested Code; см. *Ошибки в продукте*, Production Bugs, с. 303) из-за невозможности выполнить эту ветвь кода, можно воспользоваться *тестовой заглушкой* (Test Stub, с. 544) для контроля над **опосредованным вводом** (indirect inputs) тестируемой системы. Если обнаружились *не тестируемые требования* (Untested Requirements; см. *Ошибки в продукте*, Production Bugs) из-за недоступности части поведения системы через открытый интерфейс, можно воспользоваться шаблоном *подставной объект* (Mock Object, с. 558) для перехвата и проверки **опосредованного вывода** (indirect outputs) тестируемой системы.

Проектирование с учетом тестирования

Автоматизировать тестирование значительно проще, если использовать многоуровневую архитектуру. Как минимум бизнес-логика должна быть отделена от базы данных и пользовательского интерфейса, обеспечивая возможность тестирования с помощью “*подкожного*” *теста* (Subcutaneous Test) или *теста служебного уровня* (Service Layer Test; см. *Тест уровня*, Layer Test). Для снижения зависимости от “песочницы” с базой данных (Database Sandbox, с. 658) большинство тестов (желательно все) необходимо выполнять только на основе объектов, находящихся в памяти. Такая схема позволяет среди реального времени реализовать автоматическую *очистку со сборкой мусора* (Garbage-Collected Teardown, с. 518), избавляя разработчиков от написания потенциально сложной логики удаления созданных объектов (такая логика является гарантированным источником *утечки ресурсов* (Resource Leakage)). Кроме того, такой подход позволит избежать появления *медленных тестов* (Slow Tests, с. 289) за счет сокращения обмена данными с диском (диск намного медленнее оперативной памяти).

При создании графического интерфейса желательно вынести сложную логику графического интерфейса из визуальных классов. Использование *минимального диалога* (Humble Dialog; см. *Минимальный объект*, Humble Object, с. 700), делегирующего все принятие решений в невизуальные классы, позволяет создавать модульные тесты для логики графического интерфейса (например, включение/отключение переключателей) без создания экземпляра графического объекта или инфраструктуры, от которой зависят эти объекты.

Если приложение является достаточно сложным или планируется создавать компоненты, которые будут повторно использоваться другими проектами, модульные тесты можно дополнить **тестами компонентов** (component tests), которые проверяют поведение каждого компонента в отдельности. Скорее всего, для компонентов, от которых зависит тестируемый компонент, придется создавать *тестовый двойник* (Test Double). Для его установки во время выполнения можно воспользоваться *вставкой зависимости* (Dependency Injection, с. 684), *поиском зависимости* (Dependency Lookup, с. 692) или единственным экземпляром подкласса (Subclassed Singleton; см. *Связанный с тестом подкласс*, Test-Specific Subclass, с. 591).

Организация тестов

Если в составе *класса теста* (Testcase Class) будет присутствовать слишком много *тестовых методов* (Test Method), класс можно разделить с учетом методов или с учетом требований к тестовой конфигурации. Такие схемы организации называются *класс теста для каждой функции* (Testcase Class per Feature, с. 633) и *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639) соответственно. Вариант *класс теста для каждой тестовой конфигурации* позволяет переместить весь код создания конфигурации в метод `setUp`. Такой подход называется *неявной настройкой* (Implicit Setup, с. 449). После этого *объекты набора тестов* (Test Suite Object, с. 414) можно агрегировать в единый *объект набора тестов* (Test Suite Object). В результате получится *набор наборов* (Suite of Suites), содержащий все тесты из исходного *класса теста* (Testcase Class). В свою очередь, такой *объект набора тестов* (Test Suite Object) может быть добавлен в объект набора тестов пакета верхнего уровня или пространства имен. После этого все тесты или подмножество тестов, относящихся к разрабатываемой в данный момент области программного продукта, можно запускать.

Что дальше

Этот водоворот самых важных целей, принципов, шаблонов и запахов является лишь кратким введением в автоматизацию тестирования. В главах 2–14 приводится более подробное описание каждой затронутой здесь области. Если после прочтения данной главы некоторые шаблоны или запахи вызвали у вас повышенный интерес, можете сразу перейти к их подробному описанию в частях II и III. В противном случае переходите к следующим главам, в которых перечисленные шаблоны и их альтернативы рассматриваются более подробно. На очереди — глава 2, “Запахи тестов”, в которой рассматриваются некоторые распространенные “запахи тестов”, ставшие причиной большинства процедур рефакторинга.

Глава 2

Запахи тестов

О чём идет речь в этой главе

В главе 1, “Краткий обзор”, очень сжато описывались основные шаблоны и запахи, рассматриваемые в данной книге. В этой главе более подробно рассматриваются “запахи тестов”, которые встречаются в реальных проектах. Сначала речь идет о базовой концепции запаха теста, а затем — о запахах из трех основных категорий: запахи кода тестов, запахи поведения автоматизированных тестов и запахи проекта, связанные с автоматизированным тестированием.

Введение в запахи тестов

В своей книге *Рефакторинг: улучшение существующего кода* Мартин Фаулер описал несколько способов модификации кода без изменения реализованной функциональности. Причиной такого рефакторинга был назван “неприятный запах”, часто возникающий в объектно-ориентированном коде.

Запахи кода были описаны в главе 3, “Код с душком”, соавтором в которой выступил Кент Бек. Глава начиналась знаменитой цитатой бабушки Бека: “Если воняет, менять!” Причиной появления этой цитаты был вопрос: “Как узнать, что пришло время менять пеленки?” В результате в лексикон разработчиков был добавлен новый термин.

Запахи кода, описанные в книге Фаулера, характерны для проблем в коде готовых программных продуктов. Многие давно подозревали, что существуют запахи, встречающиеся только в коде автоматизированных тестов. Публикация доклада *Рефакторинг тестового кода [RTC]* на конференции XP2001 подтвердила эти подозрения. В докладе был идентифицирован ряд “неприятных запахов”, возникающих только в тестовом коде. Авторы доклада предложили ряд рефакторингов, которые позволяют избавиться от ядовитых запахов.

В этой главе приводится обзор запахов тестов (test smells). Более подробные примеры каждого запаха приводятся в соответствующих главах.

Что такое запах теста

Запах является симптомом проблемы. Запах не обязательно указывает на конкретную ошибку, так как может исходить из нескольких источников. Большинство запахов в дан-

ной книге имеют несколько причин. Некоторые причины приводят к появлению нескольких запахов. Это связано с тем, что основная причина может проявляться в нескольких симптомах (т.е. запахах).

Не все проблемы считаются запахами, а некоторые проблемы могут даже быть причиной запахов. Тест “бритва Оккама” позволяет определить, запах это или проблема (т.е. запах должен заставить разработчика схватиться за нос, показывая, что “здесь что-то не так”). Как будет продемонстрировано в следующем разделе, запахи классифицированы по проявляемым симптомам (по способу “хватали за нос”).

На основании этого критерия некоторые запахи тестов из предыдущих публикаций и статей были понижены до уровня “причин”. Их названия в большинстве случаев не изменились, поэтому они упоминаются в описании побочных эффектов применения шаблонов. В таком случае проще рассматривать непосредственно причину, а не более общий, но заметный запах.

Типы запахов тестов

С течением времени было обнаружено, что существует как минимум два типа запахов: **запахи кода** (code smells), которые распознаются в ходе анализа кода, и **запахи поведения** (behavior smell), которые влияют на результат выполненного теста.

Запахи кода представляют собой **антишаблоны** на уровне создания кода. Разработчик, тестер или преподаватель может заметить их при чтении или написании кода тестов, т.е. код просто выглядит не совсем правильно или не доносит намерение достаточно ясно. Запахи кода сначала необходимо распознать и только потом действовать. При этом необходимость такого действия не всем может показаться очевидной. Запахи кода относятся ко всем тестам, включая *тесты на основе сценариев* (Scripted Test, с. 319) и *записанные тесты* (Recorded Test, с. 312). Особенно они характерны для записанных тестов, в которых приходится сопровождать записанный код. К сожалению, большинство записанных тестов превращается в *непонятные тесты* (Obscure Test, с. 230), так как они записаны утилитой, не рассчитанной на генерацию читаемого людьми кода.

С другой стороны, запахи поведения намного сложнее игнорировать, так как из-за них тесты завершаются неудачно (или вообще не компилируются) в самый неожиданный момент, например при попытке интеграции кода в важную версию; в таком случае придется обнаружить причину проблемы, прежде чем ее решать. Как и запахи кода, запахи поведения касаются тестов на основе сценариев и записанных тестов.

Обычно разработчики замечают запахи кода и поведения во время автоматизации, сопровождения и запуска тестов. В последнее время был обнаружен еще один тип запаха — обычно этот запах выявляет руководитель проекта или потребитель, который не имеет доступа к коду тестов и не запускает тесты. **Запахи проекта** (project smell) являются индикаторами общего состояния проекта.

Что делать с запахами

Появления некоторых запахов не избежать, так как они требуют слишком больших трудозатрат на устранение. Важно осознавать присутствие запахов и понимать причины их возникновения. После этого можно принять осознанное решение о выборе запахов, которые будут устраниены для нормального завершения проекта.

Решение о списке удаляемых запахов принимается на основе баланса между стоимостью трудозатрат и полученной выгодой. От одних запахов избавиться сложнее; другие вызывают больше проблем. Необходимо избавиться от запахов, которые имеют наибольший отрицательный эффект, так как они не позволят нормально завершить работу. Как уже было сказано, многих запахов можно избежать, если выбрать подходящую стратегию автоматизации и следовать хорошим стандартам оформления кода автоматизированных тестов.

Хотя здесь тщательно описываются различные типы запахов, важно обратить внимание, что часто одновременно наблюдаются симптомы запахов каждого типа. Например, запахи проекта представляют собой симптом проблемы на более низком уровне. Проблема может проявляться как запах поведения, но реальной причиной проблемы, скорее всего, является еще более низкоуровневый запах кода. Хорошая новость: существуют три различных способа идентификации проблемы. Плохая новость: очень легко сконцентрироваться на симптоме более высокого уровня и попытаться решить проблему, не разобравшись в реальной причине происходящего.

Эффективной методикой определения причины является *пять “почему”*. Сначала необходимо узнать, почему что-то происходит. После идентификации факторов, ставших причиной происходящего, нужно определить, почему возник каждый из этих факторов. Этот процесс повторяется до тех пор, пока не будет получена новая информация. На практике пяти итераций “почему” оказывается достаточно — отсюда и название методики *пять “почему”*¹.

В оставшейся части этой главы рассматриваются запахи, которые с большей вероятностью возникают при работе над проектами. Начнем с запахов проектов. После этого будут рассмотрены запахи поведения и запахи кода, которые являются их причиной.

Каталог запахов

Разобравшись, что такое запахи и какую роль они играют в проектах, использующих автоматизированное тестирование, рассмотрим примеры таких запахов. Описание отдельных запахов и причин их возникновения приводятся в части II.

Запахи проектов

Запахи проектов являются симптомами ошибок в самом проекте. Причиной таких запахов может быть запах поведения или кода. Но так как руководители проектов редко пишут или запускают тесты, запахи проектов будут для них первым сигналом о снижении качества части проекта, связанной с автоматизацией тестов.

Основное внимание руководители проектов уделяют функциональности, качеству, ресурсам и стоимости. Именно по этой причине запахи проекта, скорее всего, будут касаться этих характеристик. Наиболее очевидной для руководителя проекта метрикой, которая будет воспринята, как запах, является качество программного обеспечения, которое измеряется в виде количества дефектов, обнаруженных во время формального тестирования или при использовании потребителем. Если количество ошибок в продукте (Production Bugs, с. 303) оказывается большим, чем ожидалось, руководи-

¹ Эта методика также называется анализом причин или “очисткой лука”.

тель проекта должен задать вопрос: “Почему эти ошибки прошли через фильтр автоматизированных тестов?”

Руководитель проекта может следить за количеством неудачных ежедневных интеграционных компиляций, чтобы получить своевременную индикацию качества программного обеспечения и следования выбранному групповому процессу разработки. Руководитель должен начинать беспокоиться, если интеграция завершается неудачей слишком часто (особенно если проблема не решается за несколько минут). **Анализ причин** (root cause analysis) возникновения ошибок может показать, что многие проблемы не связаны с ошибками в самом программном обеспечении, а являются следствием *тестов с ошибками* (Buggy Test, с. 296). Это пример, когда тесты кричат “Караул!” и требуют ресурсов для исправления ситуации, но не повышают качества **кода продукта** (production code).

Тест с ошибками (Buggy Test) является только одним из компонентов более общей проблемы *высокой стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300), что может заметно повлиять на производительность команды разработчиков. Если тесты должны модифицироваться слишком часто (т.е. при каждой модификации тестируемой системы) или если стоимость модификации слишком высока из-за непонятных тестов, руководитель проекта может принять решение о перенаправлении ресурсов с написания автоматизированных тестов на создание большего объема кода продукта или тестирование вручную. Скорее всего, на этом этапе руководитель проекта запретит разработчикам писать тесты. (Достаточно сложно заставить руководителя проекта разрешить разработчикам писать автоматизированные тесты. Но если это все-таки удалось сделать, не стоит терять данную возможность из-за небрежности или неэффективности. Скорее всего, именно по этой причине и была написана данная книга: помочь разработчикам убедить руководителя проекта и не дать им повода отказаться от автоматизированного модульного тестирования.)

С другой стороны, руководитель проекта может решить, что *ошибки в продукте* (Production Bugs) стали следствием того, что *разработчики не пишут тесты* (Developers Not Writing Tests, с. 298). Такое заявление обычно делается во время ретроспективного анализа или анализа причин проблемы. Отказ от написания тестов может быть вызван слишком жестким графиком разработки, непосредственными руководителями, которые рекомендуют “не тратить время на тесты”, или разработчиками, не обладающими достаточной квалификацией для написания тестов. Среди других возможных причин можно назвать навязанный дизайн, плохо поддающийся тестированию, или среду тестирования, которая способствует появлению “хрупких” тестов (Fragile Test, с. 277). Наконец, эта проблема может быть следствием *потерянных тестов* (Lost Test) — необходимые тесты существуют, но не включены в *набор всех тестов* (AllTests Suite; см. *Именованный набор тестов*, Named Test Suite, с. 604), который используется разработчиками перед включением изменений в общее хранилище кода или вызывается инструментами автоматизированной компиляции.

Запахи поведения

Запахи поведения проявляются при компиляции и запуске тестов. Чтобы их заметить, сверхвнимательность не требуется, так как они проявляются в виде ошибок компиляции и неудачного завершения тестов.

Самым распространенным запахом поведения является “хрупкий” тест (Fragile Test). Его проявлением является тест, по какой-то причине завершающийся неудачно, но до

этого работавший нормально. Проблема “хрупких” тестов стала причиной неприятия автоматизированных тестов в некоторых кругах. Особенно это касается коммерческих инструментов тестирования, которые работают по принципу “записи и воспроизведения” и обещают простую автоматизацию тестов. Сразу после записи такие тесты очень уязвимы. Часто единственным способом решения проблемы является повторная запись теста, так как записанный код очень сложно читать и модифицировать вручную.

Основные причины возникновения этого запаха можно разделить на четыре широкие категории.

- *Чувствительность к интерфейсу* (Interface Sensitivity) возникает, когда работа теста нарушается из-за изменений в программном или пользовательском интерфейсе, который использовался для автоматизации теста. Коммерческие инструменты “записи и воспроизведения” обычно взаимодействуют с системой через пользовательский интерфейс. Минимальные модификации интерфейса могут привести к неудачному завершению тестов даже в ситуации, в которой пользователь-человек счел бы тест пройденным.
- *Чувствительность к поведению* (Behavior Sensitivity) возникает, когда тест завершается неудачно из-за изменений в поведении тестируемой системы. Конечно, если система меняется, тесты должны завершаться неудачно. Но проблема в том, что на небольшое изменение должно реагировать небольшое количество тестов. Проблема возникает, когда в ответ на небольшое изменение неудачно завершается большинство тестов или вообще все тесты.
- *Чувствительность к данным* (Data Sensitivity) возникает, когда тесты завершаются неудачно после изменения данных, уже присутствующих в тестируемой системе. Такая проблема особенно характерна для приложений, использующих базы данных. Чувствительность к данным является специальным случаем чувствительности к контексту, когда в качестве контекста выступает база данных.
- *Чувствительность к контексту* (Context Sensitivity) возникает, когда тесты завершаются неудачно из-за изменений в окружении тестируемой системы. Наиболее частым примером является зависимость тестов от даты и времени, но эта проблема может возникать и в результате зависимости от состояния устройств (серверов, принтеров или мониторов).

Чувствительность к данным и чувствительность к контексту являются примерами специального типа “хрупкого” теста (Fragile Test), известного как “хрупкая” тестовая конфигурация (Fragile Fixture). При этом модификация часто используемой тестовой конфигурации приводит к неудачному завершению нескольких существующих тестов. Такой сценарий увеличивает стоимость расширения стандартной тестовой конфигурации (Standard Fixture, с. 338) для поддержки новых тестов, а значит, делает невыгодным создание тестов для всех компонентов продукта. Хотя основной причиной “хрупкости” конфигурации является плохой дизайн тестов, проблема проявляется в результате изменения тестовой конфигурации, а не после изменений в тестируемой системе.

В большинстве проектов с гибкой разработкой применяется ежедневная или **непрерывная интеграция** (continuous integration), состоящая из двух этапов: компиляции последней версии исходного кода и запуска всех автоматизированных тестов над скомпилированным продуктом. *Рулетка утверждений* (Assertion Roulette, с. 264) может затруднить определение, как и почему тесты завершаются неудачно во время интеграции и

какие утверждения оказались ошибочными. Это связано с недостаточностью информации в журнале ошибок. Диагностика ошибок во время компиляции может потребовать значительного времени, так как ошибку необходимо воспроизвести в среде разработки и только после этого выполнять поиск причины ошибки.

Распространенным источником неприятностей является неудачное завершение тестов без видимых причин, т.е. ни код тестируемой системы, ни код тестов не модифицировались, а тест неожиданно начинает завершаться неудачно. При попытке воспроизвести результат в среде разработки тест может завершиться успешно или неудачно. Такие *нестабильные тесты* (Erratic Test, с. 267) очень раздражают разработчиков и требуют много времени для исправления, так как причин некорректного поведения может быть множество. Некоторые из них перечислены ниже.

- *Взаимодействующие тесты* (Interacting Tests) возникают, когда несколько тестов используют *общую тестовую конфигурацию* (Shared Fixture, с. 350). В таком случае очень сложно запустить тесты по отдельности или запустить несколько наборов тестов в пределах большего *набора наборов* (Suite of Suites). Также это может привести к каскадным отказам, когда неудачное завершение одного теста приводит *стандартную тестовую конфигурацию* (Standard Fixture) в состояние, заставляющее завершаться неудачно множество других тестов.
- “*Война*” запуска тестов (Test Run War) возникает, когда несколько *программ запуска тестов* (Test Runner, с. 405) одновременно запускают тесты с использованием *общей тестовой конфигурации* (Shared Fixture). Обычно это происходит в самое неподходящее время, например при попытке исправить последние несколько ошибок перед выпуском следующей версии.
- *Неповторяемые тесты* (Unrepeatable Test) возвращают разные результаты при каждом запуске теста. Это может вынудить разработчика прибегнуть к *ручному вмешательству* (Manual Intervention, с. 287).

Частая отладка (Frequent Debugging, с. 285) является еще одним запахом, снижающим производительность труда. Автоматизированные модульные тесты должны избавить от необходимости пользоваться отладчиком в большинстве случаев, так как список неудачно завершившихся тестов явно указывает на источник ошибки. Запах *частая отладка* (Frequent Debugging) указывает, что модульные тесты недостаточно покрывают существующий код или тест проверяет слишком большую часть функциональности.

Реальная ценность *полностью автоматизированных тестов* (Fully Automated Test, с. 81) заключается в максимальной частоте их запуска. При разработке на основе тестов разработчик может запускать весь набор или часть тестов один раз в несколько минут. Такое поведение стоит стимулировать, так как снижается время обратной связи, а значит, и стоимость устранения дефектов кода. Если тесты требуют *ручного вмешательства* (Manual Intervention) при каждом запуске, разработчики стараются запускать тесты реже. Такая практика увеличивает стоимость обнаружения всех дефектов, внесенных с момента последнего запуска теста.

Такое же отрицательное действие на производительность оказывает запах *медленные тесты* (Slow Tests, с. 289). Если запуск тестов требует более 30 секунд, разработчики не будут запускать тесты после внесения каждой модификации. Вместо этого будет выбираться “логичный момент” для запуска, например перед перерывом на кофе, перед обедом или перед совещанием. Отложенная обратная связь приводит к потере “непрерывности” и увели-

чению времени между внесением и обнаружением дефекта. Наиболее распространенное решение для избавления от этого запаха является и наиболее проблемным; применение *общей тестовой конфигурации* (Shared Fixture) может привести к появлению различных запахов поведения и должно рассматриваться как крайняя мера.

Запахи кода

Запахи кода являются “классическими” запахами, описанными в книге Мартина Фаулера *Рефакторинг* [Ref]. Большинство запахов, рассмотренных Фаулером, относились к коду. Они должны распознаваться авторами тестов в процессе обслуживания кода. Хотя запахи кода обычно влияют на стоимость обслуживания тестов, их можно рассматривать как ранние признаки появления запахов поведения в будущем.

При чтении тестов достаточно очевидным (и часто игнорируемым) запахом является *непонятный тест* (Obscure Test). Он может принимать разные формы, но все версии имеют один и тот же эффект: очень сложно определить, что же делает тест, так как тест не *доносит намерение* (Communicate Intent, с. 95). Такая неопределенность увеличивает стоимость обслуживания тестов и может привести к появлению *тестов с ошибками* (Buggy Test), когда **ответственный за тест** (test maintainer) вносит некорректные изменения в код теста.

Еще одним характерным запахом является *условная логика теста* (Conditional Test Logic, с. 243). Тесты должны быть простыми, линейными последовательностями операторов. Если тест имеет несколько ветвей выполнения, нельзя знать точно, по какому пути пойдет выполнение теста в конкретном случае.

Запах *фиксированные данные теста* (Hard-Coded Test Data) может стать очень опасным по ряду причин. Во-первых, тест сложнее понять: приходится рассматривать каждое значение и угадывать, связано ли оно с другими значениями, чтобы спрогнозировать поведение тестируемой системы. Во-вторых, возникает проблема, если тестируемая система включает в себя базу данных. Запах *фиксированные данные теста* (Hard-Coded Test Data) может привести к появлению *неустойчивых тестов* (Erratic Test), если тест использует уже существующий в базе данных идентификатор, или “хрупкой” *тестовой конфигурации* (Fragile Fixture), если значения ссылаются на уже модифицированные записи в базе данных.

Запах *сложный в тестировании код* (Hard-to-Test Code, с. 251) может оказаться фактором, способствующим появлению других запахов кода и поведения. Эта проблема наиболее очевидна для автора тестов, который не может найти способ создания тестовой конфигурации, вызова методов тестируемой системы или проверки ожидаемого результата. В итоге автору тестов приходится проверять больший объем кода (большую тестируемую систему с большим количеством классов), чем хотелось бы. При чтении кода теста данный запах проявляется как *непонятный тест* (Obscure Test), поскольку автору теста приходится обходить различные ограничения для взаимодействия с тестируемой системой.

Запах *дублирование тестового кода* (Test Code Duplication, с. 254) является плохой практикой, так как он увеличивает стоимость обслуживания тестов. В результате приходится обслуживать больше кода и код оказывается сложнее, так как зачастую этот запах сопровождается запахом *непонятный тест* (Obscure Test). Такая ситуация возникает, когда автор теста просто клонирует его и не задумывается о более осмысленном повторном использовании логики теста². С ростом потребности в тестировании от автора тестов тре-

² Обратите внимание, что написано “повторном использовании логики”, а не “повторном использовании методов”.

буется выделение часто используемых последовательностей операторов во *вспомогательные методы теста* (Test Utility Method, с. 610), которые можно использовать повторно в различных *тестовых методах* (Test Method, с. 378)³. Такой подход снижает стоимость обслуживания тестов.

Запах логики *теста в продукте* (Test Logic in Production, с. 257) является нежелательным, так как нет гарантии невозможности случайного запуска этого кода⁴. Кроме того, код готового продукта становится больше и сложнее. Наконец, такая ошибка может привести к включению в выполняемый файл дополнительных программных компонентов и библиотек.

Что дальше

В этой главе были описаны различные неприятности, происходящие при автоматизации тестов. В главе 3, “Цели автоматизации”, рассматриваются цели, о которых необходимо помнить при создании автоматизированных тестов. Понимание целей позволит подготовиться к пониманию принципов. А следование принципам позволит обойти большинство проблем, перечисленных в данной главе.

³ Очень важно не использовать *тестовые методы* (Test Method) повторно, так как в результате получается *гибкий тест* (Flexible Test; см. *Условная логика тестов*, Conditional Test Logic).

⁴ Обратите внимание на врезку о ракете Ariane на с. 257 с поучительной историей.

Глава 3

Цели автоматизации

О чём идет речь в этой главе

В главе 2, “Запахи тестов”, были показаны различные запахи, которые могут выступать в роли симптомов проблем в работе автоматизированных тестов. В этой главе рассматриваются цели, которых необходимо достичь для успешного использования автоматизированных модульных и приемочных тестов. В начале главы приводится описание причин автоматизации, которое переходит в рассмотрение общих целей автоматизации тестирования, включая снижение стоимости, повышение качества и создание более понятного кода. Каждая цель состоит из более конкретных подцелей, которые также рассматриваются в данной главе. Здесь не показано, как достигаются перечисленные цели; об этом речь пойдет в последующих главах, где описанные здесь цели рассматриваются в качестве основы многих принципов.

Зачем нужны тесты

О важности автоматизированных модульных и приемочных тестов как необходимого элемента гибкого процесса разработки написано много книг. Написать хороший тест сложно, а сопровождать плохой тест еще сложнее. Так как код теста не является обязательным (т.е. потребитель за него не платит), возникает очень сильный соблазн отказаться от тестирования, когда тесты становятся слишком сложными или дорогими в обслуживании. После отказа от принципа “все тесты должны завершаться успешно, чтобы код считался правильным” по большей части автоматизированные тесты теряют ценность.

В нескольких проектах, в разработке которых автор принимал участие, возникла ряд сложностей при автоматизации тестов. Стоимость написания и обслуживания наборов тестов становится заметным препятствием, особенно когда количество тестов исчисляется тысячами. К счастью, сработал принцип “Необходимость — мать изобретательности”. В таких ситуациях разработчикам удалось найти несколько решений для обхода возникших проблем. С тех пор было достаточно времени, чтобы обдумать, чем были хороши эти решения. В процессе обдумывания компоненты успешных решений были разделены на цели (чего необходимо достигнуть) и принципы (как именно достичь целей). Следование целям и принципам упрощает создание и обслуживание автоматизированных тестов.

Экономическое обоснование автоматизации тестов

Конечно, с созданием и обслуживанием набора автоматизированных тестов всегда связана определенная стоимость. Горячие поклонники автоматизации тестов утверждают, что стоит потратить больше сейчас, чтобы иметь возможность модифицировать программное обеспечение потом. К сожалению, подход “заплати сейчас, чтобы не платить потом” не всегда действует в жестких экономических условиях¹.

Таким образом, целью является автоматизация тестирования без дополнительных затрат на разработку. Для этого затраты на создание и обслуживание тестов должны компенсироваться сокращением ручного модульного тестирования и отладки, а также снижением количества дефектов, доживающих до фазы формального тестирования приложения или использования конечным потребителем. На рис. 3.1 показано, как стоимость автоматизации компенсируется полученной экономией затрат.

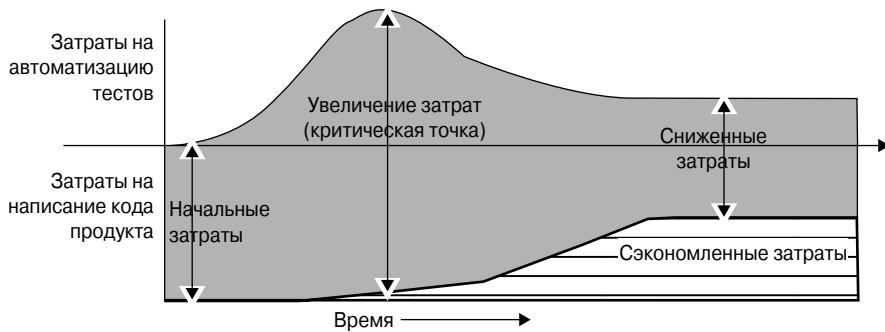


Рис. 3.1. Проект с автоматизированными тестами и заметной экономией затрат, когда затраты компенсируются правильным применением тестов

Изначально изучение новых технологий и практик требует дополнительных усилий. После прохода критической точки затраты должны вернуться к исходному уровню, так как дополнительные затраты (часть над линией) полностью компенсируются положительным эффектом от тестирования. Если тесты сложно писать или понимать или если они требуют частого и дорогостоящего обслуживания, общая стоимость разработки программного обеспечения (высота вертикальных стрелок) растет, как показано на рис. 3.2.

Обратите внимание, что дополнительный объем работы на рис. 3.2 превышает таковой на рис. 3.1 и продолжает увеличиваться со временем. Кроме того, экономия ниже основной линии оказывается меньше. Это результат увеличения общих трудозатрат, которые превышают исходные трудозатраты без использования автоматизированных тестов.

¹ Утверждение об улучшении качества за счет увеличения расходов также не имеет большой силы в современных реалиях разработки ПО с “достаточно хорошим” качеством.

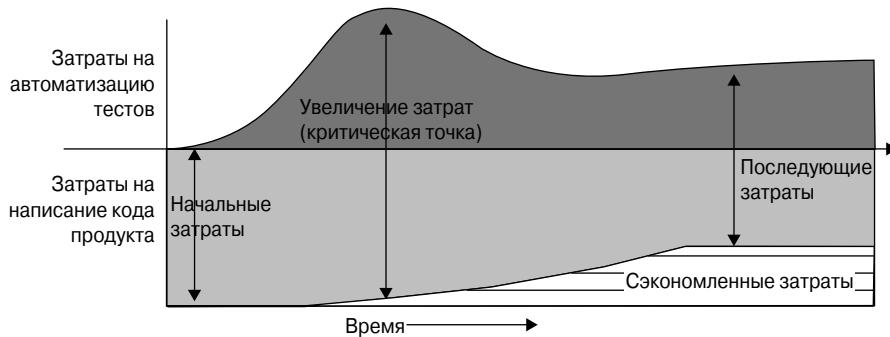


Рис. 3.2. Проект с автоматизированными модульными тестами и неэффективными затратами. В данной ситуации общая стоимость проекта увеличивается

Цели автоматизации тестов

Все разработчики подходят к идеи автоматизации тестов с определенной степенью понимания, почему это “хорошо”. Вот некоторые высокоуровневые цели, которые могут интересовать разработчиков при автоматизации тестов.

- Тесты должны способствовать повышению качества
- Тесты должны способствовать пониманию принципов работы тестируемой системы
- Тесты должны снижать риск (не внося новых его источников)
- Тесты должны просто запускаться
- Тесты должны быть просты в написании и обслуживании
- Тесты должны требовать минимального обслуживания с развитием системы

Первые три цели иллюстрируют ценность использования тестов. Остальные цели описывают характеристики тестов. Большинство этих целей можно разложить на более конкретные (и поддающиеся измерению) цели. Для них выбраны короткие запоминающиеся имена, чтобы на них можно было ссылаться при описании конкретных принципов или шаблонов.

Тесты должны способствовать повышению качества

Традиционные причины тестирования лежат в области **контроля качества** (quality assurance — QA). Что конкретно имеется в виду? Что такое качество? Традиционные определения выделяют две основные категории качества на основе следующих вопросов: “Правильно ли скомпилирован программный продукт?” и “Правильный ли программный продукт скомпилирован?”

Цель: тесты как спецификация (Tests as Specification)

При разработке на основе тестов действия тестируемой системы описываются еще до написания самого кода. При этом описание поведения в различных сценариях

Также известна как:
Выполняемая спецификация (Executable Specification)

может выполняться в будущем (получается так называемая “выполняемая спецификация”). Для обеспечения “компиляции правильного программного продукта” необходимо описать с помощью тестов реальное использование тестируемой системы. Для этого можно разработать имитацию пользовательского интерфейса, которая перехватывает данные о поведении и внешнем виде приложения, достаточные для написания тестов.

Сам процесс обдумывания тестируемой системы в различных ситуациях для написания тестов позволяет определить области приложения, в которых требования неоднозначны или противоречивы. Такой анализ повышает качество спецификации, а значит, повышает качество программного продукта.

Цель: “репеллент” для ошибок (Bug Repellent)

Да, тесты выявляют ошибки, но это не основная задача автоматизированных тестов. Автоматизированные тесты позволяют не вносить ошибок. Воспринимайте автоматизированные тесты как “репеллент”, который “отпугивает” мелкие ошибки от очищенного от ошибок кода. При наличии регрессионных тестов ошибки будут обнаруживаться еще до включения кода в общее хранилище. (Тесты ведь запускаются до включения кода в хранилище, правда?)

Цель: локализация дефектов (Defect Localization)

Ошибки случаются! Конечно, некоторые ошибки намного сложнее предотвратить, чем исправить. Предположим, ошибка каким-то образом вкрадась в код и проявилаась на этапе интеграции. Если модульные тесты достаточно малы (в каждом тесте проверяется одиночный аспект поведения), ошибка будет быстро обнаружена по результатам списка неудачно завершившихся тестов. Такая избирательность является одним из преимуществ модульных тестов по сравнению с приемочными тестами. Приемочные тесты показывают, что некоторое ожидаемое потребителем поведение не работает; модульный тест отвечает на вопрос: “Почему оно не работает?” Этот результат называется *локализацией дефектов* (Defect Localization). Если приемочный тест завершается неудачно, а все модульные тесты — успешно, найден *отсутствующий модульный тест* (Missing Unit Test; см. *Ошибки в продукте*, Production Bugs, с. 303).

Все эти преимущества окажутся нереализованными, если тесты не покрывают все возможные сценарии для каждого модуля программного продукта. Точно так же преимущества не имеют смысла, если сами тесты содержат ошибки. Очевидно, что тесты должны быть максимально простыми для быстрой проверки на отсутствие ошибок. Хотя написание модульных тестов для модульных тестов не кажется практическим решением, можно (и нужно) написать модульные тесты для всех *вспомогательных методов теста* (Test Utility Method, с. 610), которым делегированы сложные алгоритмы.

Тесты должны способствовать пониманию принципов работы тестируемой системы

“Отпугивание” ошибок — не единственная задача тестов. Кроме того, тесты показывают, как должен работать тестируемый код. Тесты **черного ящика** (black box) фактически описывают требования к программным компонентам.

Цель: тесты как документация (Tests as Documentation)

Без автоматизированных тестов пришлось бы копаться в коде тестируемой системы в попытке ответить на вопрос: “Каким должен быть результат, если...?” При наличии тестов можно воспользоваться *тестами как документацией* (Tests as Documentation). Они указывают на ожидаемый результат (вспомните, что *самопроверяющийся тест*, Self-Checking Test, указывает на ожидаемый результат с помощью одного или нескольких утверждений). Если интересно, как система что-то делает, можно включить отладчик, запустить тест и пройти код построчно, рассматривая реализацию. В этом смысле автоматизированные тесты служат документацией к тестируемой системе.

Тесты должны снижать риск (не внося новых его источников)

Как было показано, тесты должны повышать качество программного обеспечения за счет документирования требований и предотвращения ошибок в процессе инкрементной разработки. Это один из способов снижения риска. Другие способы включают в себя проверку поведения продукта в “невозможных” ситуациях, которые нельзя повторить средствами обычных приемочных тестов при проверке приложения как “черного ящика”. Это очень хороший подход к оценке рисков проекта и поиску путей избежания части из них через *полностью автоматизированные тесты* (Fully Automated Tests).

Цель: тесты как страховка (Tests as Safety Net)

При работе с унаследованным кодом часто приходится нервничать. По определению унаследованный код не имеет набора автоматизированных регрессионных тестов. Модификация такого кода сопряжена с риском, так как неизвестно, что сломается следующим. И никак не определить, что уже сломалось! Как следствие приходится работать очень осторожно, часто вручную анализируя код перед внесением изменений.

Также известна как:
Страховочная сеть (Safety Net)

С другой стороны, код с набором регрессионных тестов позволяет вносить изменения намного быстрее. Появляется свобода эксперимента. “Что будет, если изменить это? Какие тесты завершились неудачно? Интересно! Так вот зачем нужен этот параметр!” Таким образом, автоматизированные тесты выступают в роли страховки и дают разработчику определенную свободу действий².

Эффективность страховки дополняется и усиливается системами контроля версий в современных средах разработки программного обеспечения. Хранилище исходного кода на основе CVS, Subversion или SourceSafe позволяет откатить изменения до заведомо работающей версии, если тесты показывают, что текущий набор изменений слишком сильно влияет на работоспособность кода.

Также известна как:
Не вносить риски с помощью тестов (No Test Risk)

Цель: не навреди (Do No Harm)

У применения автоматизированных тестов есть и обратная сторона: они могут стать причиной дополнительного риска. Необходимо очень внимательно следить,

² Представьте обучение работе на трапеции в цирке без страховочного троса, позволяющего спокойно падать. Без него можно научиться только раскачиванию.

чтобы автоматизированные тесты не привели к появлению новых проблем в тестируемой системе. Принцип отсутствия тестового кода в самом продукте рекомендует не добавлять функции-“ловушки” для тестов в тестируемую систему. Конечно, необходимость тестиования должна учитываться при проектировании, но тестовый код должен подключаться только в составе тестов и только в специальной тестовой среде. В уже готовом продукте от него не должно остаться и следа.

Определенный риск появляется в ситуации, когда код считается тщательно протестированным, а значит, надежным, когда тестиирование было не настолько тщательным. Многие разработчики совершают ошибку, используя *тестовый двойник* (Test Double, с. 538) для замены большей части тестируемой системы. Из этого следует еще один важный принцип: не модифицировать тестируемую систему. Необходима предельная ясность в предмете тестиирования, поэтому не стоит заменять фрагменты тестируемой системы специально написанным кодом (рис. 3.3).

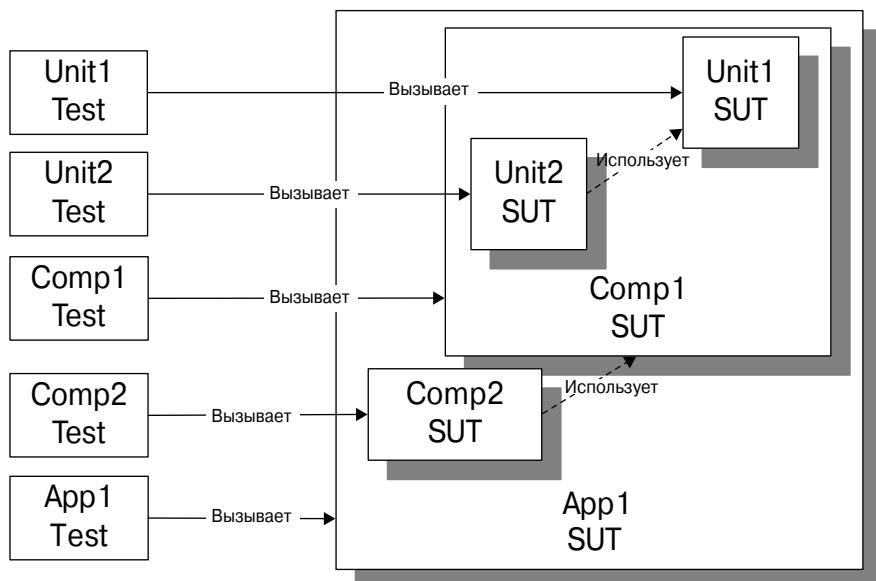


Рис. 3.3. Набор тестов. Каждый набор имеет свою тестируемую систему. Приложение, компонент и модуль являются тестируемой системой только относительно конкретного набора тестов. Модуль Unit1 является вызываемым компонентом (частью тестовой конфигурации) для теста Unit2 и частью компонента Comp1

Тесты должны легко запускаться

Большинство разработчиков заинтересованы только в написании кода. Тестиирование просто является меньшим из зол. Автоматизированные тесты обеспечивают страховку, позволяющую ускорить³ написание кода, но автоматизированные тесты будут запускаться часто, только если сделать это достаточно легко.

³ “Уменьшить стресс” — более правильное выражение.

Что делает тесты простыми в использовании? Четыре основные цели отвечают на этот вопрос.

- Тесты должны быть *полностью автоматизированы* (Fully Automated Tests) и не требовать вмешательства со стороны разработчика.
- Тесты должны быть *самопроверяющимися* (Self-Checking Test), т.е. обнаруживать все ошибки без ручного контроля и сообщать о них.
- Тесты должны быть *повторяемыми тестами* (Repeatable Tests) и давать один и тот же результат для одинакового кода вне зависимости от количества запусков.
- В идеальном случае тесты должны быть *независимыми тестами* (Independent Tests) и запускаться по-отдельности.

Удовлетворение этих целей позволит получить результат работы тестов одним нажатием клавиши (или щелчком мышью). Рассмотрим каждую из целей более подробно.

Цель: полностью автоматизированные тесты (Fully Automated Tests)

Если тест может работать без *ручного вмешательства* (Manual Intervention, с. 287), он является полностью автоматизированным. Удовлетворение данного критерия является необходимым условием для множества других целей. Да, можно написать полностью автоматизированные тесты, которые не проверяют результат и могут запускаться только один раз. Программа `main()`, которая запускает код и перенаправляет вывод команд `print` на консоль, является хорошим примером такого теста. Эти два аспекта автоматизации тестирования являются настолько важными в обеспечении простоты запуска тестов, что они были выделены в две отдельные цели: *самопроверяющийся тест* (Self-Checking Test) и *повторяемые тесты* (Repeatable Test).

Цель: самопроверяющийся тест (Self-Checking Test)

Самопроверяющийся тест содержит внутри себя все, что необходимо для проверки правильности результата работы. Такой тест использует голливудский принцип: “Не звоните нам, мы сами позвоним вам”. Таким образом, *программа запуска тестов* (Test Runner, с. 405) требует внимания разработчика, только если тест завершился неудачно. В результате при удачном завершении всех тестов ручное вмешательство совершенно не требуется. Многие реализации xUnit предлагают *программу запуска тестов с графическим интерфейсом* (Graphical Test Runner), использующим *зеленую полосу* (green bar) для индикации удачного завершения всех тестов. **Красная полоса** (red bar) показывает, что тест завершился неудачно и требует выяснения причин ситуации.

Цель: повторяемые тесты (Repeatable Tests)

Повторяемые тесты могут запускаться несколько раз подряд и каждый раз возвращать один и тот же результат без вмешательства разработчика между запусками. *Неповторяемый тест* (Unrepeatable Test; см. *Нестабильный тест*, Erratic Test, с. 267) значительно увеличивает накладные расходы по запуску тестов. Такой результат очень нежелателен, так как разработчики должны иметь возможность как можно чаще запускать тесты (как минимум после каждого “сохранения” внесенных изменений). *Неповторяемый тест* (Unrepeatable Test) может запускаться только один раз с последующим *ручным вмеша-*

тельством (Manual Intervention). Настолько же нежелательным является *неопределенный тест* (Nondeterministic Test; см. *Нестабильный тест*, Erratic Test), который при каждом запуске возвращает разные результаты; в такой ситуации приходится тратить очень много времени на поиск причин отказа теста. Сила красной полосы значительно ослабевает, когда она часто появляется без правильной причины. Рано или поздно красная полоса будет игнорироваться разработчиками в предположении, что она исчезнет сама по себе, если подождать достаточно времени. Если это произошло, значит, ценность автоматизированных тестов значительно снизилась, так как обратная связь, указывающая на внесение новой ошибки и необходимость ее исправить, теряется. Чем дольше длится такое ожидание, тем больше усилий потребуется при поиске причин отказа теста.

Если тест работает только с содержимым оперативной памяти и использует только локальные переменные, его легко повторить без дополнительных усилий. Обычно *неповторяемый тест* (Unrepeatable Test) возникает после использования одного из типов *общей тестовой конфигурации* (Shared Fixture, с. 350) — такое определение включает в себя и сохранение данных средствами тестируемой системы. В таком случае необходимо убедиться, что тест самостоятельно обеспечивает удаление всех служебных данных. Если требуется удаление данных, наиболее целостной стратегией является использование универсального механизма *автоматической очистки* (Automated Teardown, с. 521). Хотя код очистки может присутствовать в каждом тесте, такой подход может привести к появлению *нестабильных тестов* (Erratic Test), если в код очистки будут внесены ошибки.

Тесты должны быть простыми в написании и обслуживании

Написание кода само по себе является сложной деятельностью, так как разработчик должен держать в голове большой объем информации. При написании тестов основное внимание должно уделяться тестированию, а не написанию кода тестов. Это значит, что тесты должны быть простыми — простыми для чтения и написания. Тесты должны быть также простыми для понимания, так как тестирование автоматизированных тестов является исключительно сложным процессом. Полноценное тестирование возможно только через внесение тех ошибок, которые тесты должны обнаруживать в тестируемой системе. Данный процесс практически не поддается автоматизации и выполняется однократно (если вообще выполняется) при первоначальном написании тестов. Исходя из этих особенностей приходится полагаться на собственную внимательность при поиске ошибок в коде тестов, а значит, тесты должны быть максимально простыми.

Конечно, после модификации поведения одного из компонентов системы некоторое количество тестов должно среагировать на такое изменение. Необходимо *минимизировать пересечения тестов* (Minimize Test Overlap), чтобы на каждое изменение поведения реагировало минимальное количество тестов. Вопреки распространенному мнению рост количества тестов, проверяющих один и тот же код, не повышает качество кода, если все тесты делают одно и то же.

Сложность тестов растет по двум причинам:

- в пределах одного теста проверяется слишком большой объем функциональности;
- слишком большая “пропасть выразительности” разделяет язык написания тестов (например, Java) и предусловия (или постусловия) для концепций предметной области, которые описываются тестом.

Цель: простые тесты (Simple Tests)

Чтобы тесты не “откусывали больше, чем могут проглотить”, они должны быть небольшими и проверять один аспект функциональности. Сохранение простоты тестов особенно важно, если на тестах основан процесс разработки, так как написание кода нацелено на последовательное исправление неудачных результатов запуска тестов, а значит, каждый тест должен описывать собственный фрагмент функциональности тестируемой системы. Необходимо стремиться, чтобы *один тест проверял одно условие* (Verify One Condition per Test), создавая отдельные *тестовые методы* (Test Method, с. 378) для каждой уникальной комбинации предварительной конфигурации теста и входных данных. Каждый *тестовый метод* (Test Method) должен заставлять тестовую систему выполнять одну ветвь кода⁴.

Основным исключением требования к краткости *тестовых методов* (Test Method) являются приемочные тесты, которые описывают реальные сценарии использования приложения. Такие расширенные тесты могут быть полезны при документировании поведения реального пользователя программного продукта. Если такое взаимодействие подразумевает длинные последовательности действий, *тестовый метод* (Test Method) должен отражать эту ситуацию.

Цель: выразительные тесты (Expressive Tests)

“Пропасть выразительности” можно преодолеть за счет библиотеки *вспомогательных методов теста* (Test Utility Method), составляющих связанный с предметной областью язык тестов. Такая коллекция методов позволит автору теста выразить тестируемые концепции без их перевода в более низкоуровневый код. Хорошими примерами строительных блоков для создания *языка высокого уровня* (Higher-Level Language) являются *метод создания* (Creation Method, с. 441) и *специальное утверждение* (Custom Assertion, с. 495).

Избегайте излишних повторений внутри тестов. Принцип “не повторять себя” должен применяться к коду тестов так же, как и к коду основного продукта. Но в данной ситуации существует и противоположная сила. Так как тесты должны *доносить намерение* (Communicate Intent), лучше сохранить основную логику теста в каждом *тестовом методе* (Test Method), чтобы ее можно было охватить взглядом в одном месте. Но все-таки эта идея не противоречит переносу большого объема поддерживающего кода во *вспомогательные методы теста* (Test Utility Method). Таким образом, при изменении тестируемой системы модификации придется вносить только в одном месте.

Цель: разделение интереса (Separation of Concerns)

Эта цель имеет два измерения.

- Тестовый код не должен попадать в код продукта в соответствии с принципом *не вносить логику тестов в код продукта* (Keep Test Logic Out of Production Code).
- Каждый тест должен проверять единственный аспект системы в соответствии с принципом *тестировать аспекты отдельно* (Test Concerns Separately), чтобы избежать появления *непонятных тестов* (Obscure Test, с. 230).

⁴ Для каждой ветви кода тестовой системы должен существовать как минимум один *тестовый метод* (Test Method). Часто таких методов будет несколько: по одному для каждого **граничного значения** (boundary value) **класса эквивалентности** (equivalence class).

Наглядным примером того, как не стоит поступать, является тестирование бизнес-логики вместе с пользовательским интерфейсом, поскольку проверяются два аспекта одновременно. Если один из них будет модифицирован (например, в пользовательский интерфейс будут внесены изменения), придется менять все подобные тесты. Для достижения этой цели может потребоваться разнесение логики по разным компонентам. Это ключевой элемент проектирования с учетом тестов. Данная тема более подробно рассматривается в главе 11, “Использование тестовых двойников”.

Тесты должны требовать минимального обслуживания по мере развития системы

Изменений не избежать. На самом деле автоматизированные тесты для того и пишутся, чтобы проще было вносить изменения. Необходимо следить, чтобы тесты не сделали внесение изменений более сложным процессом.

Предположим, необходимо изменить сигнатуру одного из методов класса. Неожиданно после добавления параметра 50 тестов не компилируются. Стимулирует ли это к внесению изменений? Скорее всего, нет. Для решения этой проблемы можно ввести новый метод с дополнительным параметром и вызывать его из старого метода с установкой нового параметра в принятые по умолчанию значение. После этого все тесты компилируются, но 30 из них завершаются неудачно! Помогают ли тесты вносить изменения?

Цель: устойчивый к изменениям тест (Robust Test)

По мере развития проекта и требований к нему неизбежно придется вносить множество изменений в код. Из-за этого тесты должны быть написаны таким образом, чтобы каждое изменение затрагивало как можно меньше тестов. Это ведет к минимизации перекрытия тестов. Кроме того, изменения среды не должны оказывать влияния на сами тесты; для этого test-система максимально изолируется от среды. В результате получается более *устойчивый к изменениям тест* (Robust Test).

Необходимо стремиться к достижению состояния *один тест проверяет одно условие* (Verify One Condition per Test). В идеальном случае модификация тестов должна требоваться только после изменений одного типа: изменений системы, затрагивающих код формирования и очистки тестовой конфигурации. Этот код можно скрыть во *вспомогательных методах теста* (Test Utility Method), что позволит еще больше сократить количество тестов, непосредственно затрагиваемых модификацией.

Что дальше

В этой главе рассматривалась причина использования автоматизированных тестов и конкретные цели, которых необходимо достичь при создании *полностью автоматизированных тестов* (Fully Automated Tests). Перед тем как переходить к главе 5, “Принципы автоматизации тестирования”, обратите внимание на главу 4, “Философия автоматизации тестов”, чтобы понять разницу между подходами, принятыми в среде специалистов по автоматизации тестирования.

Глава 4

Философия автоматизации тестов

О чём идет речь в этой главе

В главе 3, “Цели автоматизации”, рассматривались многие цели и преимущества эффективной автоматизации тестов. В этой главе рассматриваются некоторые различия в восприятии проекта, разработки и тестирования, которые влияют на способы применения данных шаблонов. Вопросы “общего вида” включают в себя выбор между написанием тестов до или после написания кода, восприятие тестов в качестве примеров, последовательность создания программного продукта (извне вовнутрь или наоборот), проверку состояния или поведения, а также генерацию тестовой конфигурации для всех тестов сразу или для каждого теста в отдельности.

Почему важна философия

Какое отношение философия имеет к автоматизации? Самое непосредственное! Отношение к жизни (и тестированию) активно влияет на отношение к автоматизации тестов. Во время обсуждения раннего черновика этой книги с Мартином Фаулером (редактором серии) были обнаружены философские отличия в подходах к автоматизации тестов на основе семейства xUnit. Эти отличия являются основной причиной слишком частого или, наоборот, достаточно редкого применения *подставных объектов* (Mock Object, с. 558).

После этого обсуждения автору захотелось найти и другие философские различия между разработчиками автоматизированных тестов. Альтернативные точки зрения обычно становятся результатом высказываний вида “Я никогда не пользовался данным шаблоном (так как для этого не было причин)” и “Я никогда не сталкивался с этим запахом”. Уточнив конкретные причины таких высказываний, можно многое узнать о философии тестирования конкретного разработчика. Такие обсуждения позволили выделить следующие философские различия.

- “Тесты после” или “тесты до” написания кода
- Тест за тестом или все тесты сразу

- “Извне вовнутрь” или “изнутри наружу” (применяется независимо от проектирования и написания кода)
- Проверка поведения или проверка состояния
- “Тестовая конфигурация для каждого теста” или “одна тестовая конфигурация для всех тестов сразу”

Некоторые философские отличия

Тестиовать до или после написания кода

В ходе традиционных процессов разработки программного обеспечения подготовка и запуск тестов выполняется после завершения проектирования и реализации программного продукта. Такая последовательность касается как приемочных, так и модульных тестов. С другой стороны, в гибких процессах разработки написание тестов до написания кода является стандартным подходом. Почему так важен порядок тестирования и разработки? Любой, кто пытался внедрить *полностью автоматизированные тесты* (Fully Automated Tests, с. 81) в унаследованную систему, скажет, что намного сложнее писать автоматизированные тесты для уже готового кода. Сложность заключается не в самих тестах, а в самоконтроле, который необходим для создания тестов, проверяющих “уже готовый” код. Даже при *проектировании с учетом тестирования* (design for testability) вероятность простого и естественного написания тестов без модификации кода продукта очень низка. Но если тесты писать до проверяемого кода, система изначально будет легко поддаваться тестированию.

Написание тестов до кода имеет и другие преимущества. В таком случае приходится писать только тот код, который заставит тест завершиться успешно, и код продукта будет более минималистичным. Необязательная функциональность просто оказывается ненаписанной; не приходится писать неработающий код обработки ошибок. Тесты более жизнеспособны, так как только необходимые методы предоставляются каждым объектом (в соответствии с потребностями теста).

Доступ к состоянию объектов для настройки тестовой конфигурации и проверки результата оказывается более естественным, если тесты написаны до самого кода. Например, в таком случае удается полностью избежать запаха *чувствительное сравнение* (Sensitive Equality; см. “Хрупкий” тест, Fragile Test, с. 277), так как в утверждениях используются правильные атрибуты объектов, а не выполняется сравнение строковых представлений этих объектов. Может оказаться, что строковое представление вообще не потребуется. Возможность подменять зависимости *тестовыми двойниками* (Test Double, с. 538) для проверки результата также значительно усиlena, так как *заменяемая зависимость* (substitutable dependency) интегрирована в программный продукт с самого начала.

Тесты как примеры

Каждый раз при упоминании концепции написания автоматизированных тестов до написания собственного программного обеспечения, у некоторых слушателей на лице появляется странное выражение. Сразу следует вопрос: “Как можно написать тест для еще не существующего программного обеспечения?” В таком случае в соответствии с идеей Брайана Маррика обсуждение переводится в сторону “примеров” и **разработки на**

основе примеров (*example-driven development*). Кажется, что многим значительно проще представить написание примеров, а не “тестов” перед кодом. Тот факт, что примеры являются выполняемыми и предъявляют требования, можно оставить для последующего обсуждения с разработчиками, обладающими более живым воображением.

К тому моменту, когда данная книга окажется в руках читателей, должны будут появиться инфраструктуры разработки на основе примеров. Переход от разработки на основе тестов к разработке на основе примеров начался с пакета RSpec для языка Ruby и пакета JBehave для языка Java. Основной дизайн таких “инфраструктур модульного тестирования” совпадает с дизайном xUnit, но терминология изменилась для отражения подхода *выполняемой спецификации* (*Executable Specification*; см. *Цели автоматизации тестов*, с. 77).

Еще одной популярной альтернативой описания компонентов бизнес-логики является применение тестов Fit. Такие тесты намного проще для чтения нетехническим персоналом в отличие от написанных на каком-либо языке программирования. И это не зависит от дружественности синтаксиса выбранного языка программирования!

Тест за тестом или все тесты сразу

Процесс разработки на основе тестов стимулирует к “написанию теста”, а после этого — к “написанию небольшого фрагмента кода” для обеспечения успешного завершения теста. Это не значит, что все тесты должны быть написаны до начала создания любого кода. Тесты и код пишутся поочередно для каждого небольшого фрагмента функциональности. “Немного тестов, немного кода, еще немного тестов” — самый характерный пример *инкрементной разработки* (*incremental development*). Является ли этот подход к разработке единственно возможным? Конечно, нет! Некоторые разработчики предпочитают определять все тесты, необходимые для текущей функции, перед созданием кода. Такая стратегия позволяет им “рассуждать, как клиент” или “думать, как тестер”, позволяя избежать слишком втягивания в “режим решения”.

Приверженцы чистоты разработки на основе тестов утверждают, что дизайн станет более инкрементным, если разрабатывать программный продукт по одному тесту за раз. “Намного проще сфокусироваться, если только один тест завершается неудачно”. Многие разработчики сообщают, что при таком подходе отпадает потребность в отладчике, так как тщательное тестирование и инкрементная разработка не оставляют сомнений в причинах неудачного завершения тестов. В такой ситуации тесты обеспечивают *локализацию дефектов* (*Defect Localization*; см. *Цели автоматизации тестов*, с. 77), когда последняя внесенная модификация (приводящая к появлению проблемы) еще не вышла из поля зрения разработчика.

Эти доводы оказываются особенно весомыми при обсуждении модульных тестов, так как остается возможность перечислять подробные требования (тесты) для каждого объекта или метода. В качестве разумного компромисса можно идентифицировать все тесты в начале выполнения *задачи* (*task*) — возможно, определяя пустые *тестовые методы* (*Test Method*, с. 378), но реализуя по одному *тестовому методу* (*Test Method*) за один раз. Кроме того, можно написать все *тестовые методы* (*Test Method*), но включить только один тест, чтобы сконцентрироваться на написании кода продукта для этого теста.

С другой стороны, не стоит передавать разработчику приемочные тесты в пределах одной пользовательской истории. Имеет смысл подготовить все тесты для одной истории еще до начала разработки. В некоторых группах разработчиков предпочтение от-

дается перечислению всех приемочных тестов (но не обязательно их полной реализации) до оценки трудозатрат на реализацию истории, так как тесты позволяют оценить масштабы истории.

Извне вовнутрь или изнутри наружу

Проектирование программного обеспечения извне вовнутрь подразумевает, что сначала рассматриваются приемочные тесты для “черного ящика” (также известные как тесты историй) для всей системы, а затем обдумываются модульные тесты для каждого элемента проектируемого продукта. В это же время можно создать тесты и для больших компонентов продукта.

Каждый из этих тестов заставляет “думать, как клиент” задолго до перехода к режиму мышления разработчика. Основное внимание уделяется интерфейсу, который предоставляется пользователю. Причем в качестве пользователя может выступать как живой человек, так и другой компонент программного обеспечения. Тесты описывают шаблоны использования и позволяют обнаружить различные сценарии, поддержку которых придется реализовать. Только после идентификации всех тестов спецификацию можно считать завершенной. Некоторые разработчики предпочитают проектировать извне вовнутрь, но писать код в направлении изнутри наружу. Это позволяет избежать “проблемы зависимости”. Такая тактика требует предугадывания запросов внешних уровней программного продукта во время написания тестов для внутренних компонентов. Кроме того, такой процесс обеспечивает тестирование внешних уровней в присутствии внутренних. Данная концепция проиллюстрирована на рис. 4.1. Движение сверху вниз на диаграмме подразумевает порядок написания программного обеспечения. Тесты для нижних и средних классов могут применять уже готовые классы над ними; такая стратегия позволит избавиться от применения *тестовых заглушек* (Test Stub, с. 544) и *подставных объектов* (Mock Object) в большинстве тестов. Возможно, *тестовые заглушки* (Test Stub) придется использовать, когда внутренние компоненты могут возвращать конкретные значения или генерировать исключения, но не могут делать это по запросу. В таком случае очень полезным может оказаться *диверсант* (Saboteur; см. *Тестовая заглушка*, Test Stub).

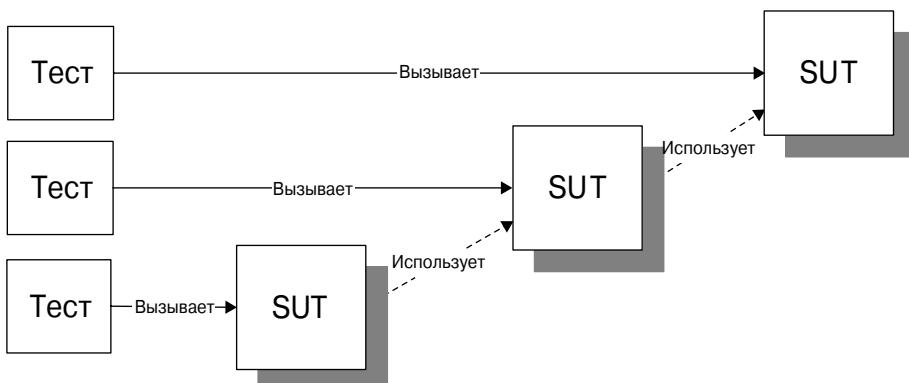


Рис. 4.1. Разработка функциональности изнутри наружу. Разработка начинается с самого внутреннего компонента и продвигается в направлении пользовательского интерфейса, основываясь на уже созданных компонентах

Другие разработчики тестов могут предпочесть проектирование и написание кода в направлении извне вовнутрь. Такой подход заставляет решать “проблему зависимости”. Вместо еще ненаписанного программного обеспечения можно применять *тестовые заглушки* (Test Stub). Это позволит запускать и тестируировать программное обеспечение внешних уровней. Кроме того, *тестовые заглушки* (Test Stub) можно будет применять для внедрения “невозможных” опосредованных входных данных (возвращаемых значений, исходящих параметров или исключений) в тестируемую систему для проверки ее корректной работы в подобных случаях.

На рис. 4.2 показан обратный порядок построения классов. Так как подчиненные классы еще не существуют, вместо них использовались *тестовые двойники* (Test Double).

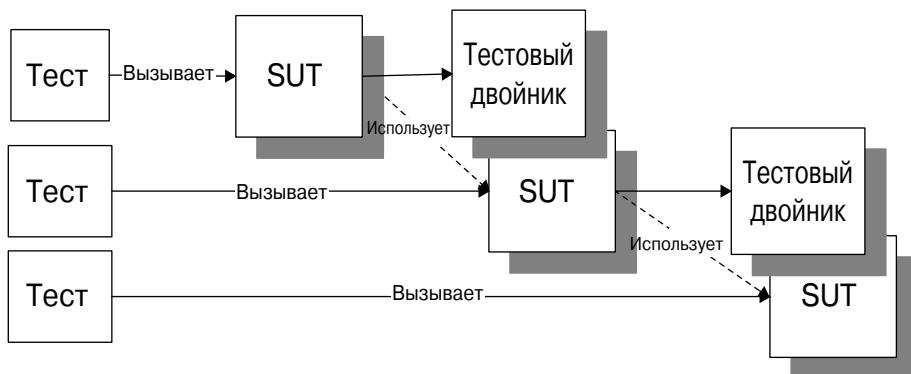


Рис. 4.2. Разработка функциональности в направлении извне вовнутрь, основанная на использовании тестовых двойников (Test Double). Разработка начинается с внешних уровней, где вместо вызываемых компонентов применяются тестовые двойники (Test Double), пока требования к вызываемым компонентам не будут formalизованы

После создания подчиненных классов двойники можно будет убрать из многих тестов. Их можно и оставить для обеспечения локализации дефектов (Defect Localization), но за счет большей стоимости обслуживания тестов.

Проверка поведения или проверка состояния

Написание кода начиная с внешних уровней является некоторым доводом в пользу проверки поведения, а не состояния. Подход сторонников “состояния” указывает, что достаточно перевести тестируемую систему в определенное состояние, отправить вызов и убедиться, что система перешла в ожидаемое состояние в момент завершения теста. Подход сторонников “поведения” требует проверки не только начального и конечного состояний, но и списка вызовов в соответствии с зависимостями тестируемой системы. Иначе говоря, необходимо подробно описать вызовы через исходящие интерфейсы. Такой **опосредованный вывод** (indirect output) тестируемой системы очень похож на возвращаемые значения функций, но для его перехвата потребуются специальные механизмы, так как обычным способом эту информацию ни тест, ни клиент получить не могут.

Школа бихевиористов иногда называется **разработкой на основе поведения** (behavior-driven development). Признаком такого подхода является широкое использование *подставных объектов* (Mock Object) и *тестовых агентов* (Test Spy, с. 552). Проверка поведе-

ния позволяет более тщательно выполнить изолированное тестирование каждого модуля программного продукта, за что приходится платить повышенной сложностью рефакторинга. Мартин Фаулер привел подробное описание обоих подходов в статье [MAS].

Тестовая конфигурация для каждого теста или одна тестовая конфигурация для всех тестов

Традиционно популярным подходом является определение “тестового стенда”, состоящего из приложения и базы данных, уже содержащих различные тестовые данные. Содержимое базы данных тщательно подбирается для выполнения различных сценариев тестирования.

Если также подходить к созданию тестовой конфигурации xUnit, автор тестов может определить *стандартную тестовую конфигурацию* (Standard Fixture, с. 338), которая будет использоваться всеми *тестовыми методами* (Test Method) одного или нескольких *классов теста* (Testcase Class, с. 401). Тестовую конфигурацию можно настраивать как *новую* (Fresh Fixture, с. 344) в каждом *тестовом методе* (Test Method) через *делегированную настройку* (Delegated Setup, с. 437) или в методе *setUp* с помощью *неявной настройки* (Implicit Setup, с. 449). С другой стороны, можно воспользоваться *общей тестовой конфигурацией* (Shared Fixture, с. 350), которая будет повторно использоваться несколькими тестами. В любом случае читатель теста может столкнуться с затруднениями при выделении из тестовой конфигурации реальных предварительных условий конкретного *тестового метода* (Test Method).

Более гибким подходом является проектирование *минимальной тестовой конфигурации* (Minimal Fixture, с. 336) для каждого *тестового метода* (Test Method). В таком случае не требуется сразу проектировать большую тестовую конфигурацию и применение *новой тестовой конфигурации* (Fresh Fixture) кажется наиболее оптимальным.

Почему возникают различия в философии

Конечно, не всегда удается убедить коллег в преимуществах своей философии. Но и в таком случае понимание приверженности другой философии позволяет понять, почему коллеги решают возникающие задачи по-другому. Это не значит, что не совпадают цели (например, программный продукт высокого качества, соответствие требованиям, своевременность, соблюдение бюджета). Просто решения для достижения цели принимаются на основе другой философии. Осознание таких различий является первым шагом к пониманию.

Философия автора

- Писать тесты до написания кода!
- Тесты являются примерами!
- Обычно тесты пишутся по одному за раз, но иногда в качестве “скелета” перечисляются все тесты, какие только можно придумать.
- Разработка в направлении извне вовнутрь позволяет определить тесты, которые потребуются для следующего уровня.

- Обычно выполняется *проверка состояния* (State Verification, с. 484), но при необходимости добиться хорошего покрытия кода можно обратиться к *проверке поведения* (Behavior Verification, с. 489).
 - Тестовая конфигурация создается для каждого теста в отдельности.
- Вот и все! Теперь должно быть ясно, из чего исходит автор в своих рассуждениях.

Что дальше

В этой главе рассматривались философии, влияющие на проектирование программного обеспечения, тестов и их автоматизации. В главе 5, “Принципы автоматизации тестирования”, рассматриваются главные принципы, позволяющие достичь целей, перечисленных в главе 3, “Цели автоматизации”. После этого можно будет переходить к общей стратегии автоматизации тестирования и к отдельным шаблонам.

Глава 5

Принципы автоматизации тестирования

О чём идет речь в этой главе

В главе 3, “Цели автоматизации”, рассматривались цели, которых необходимо достичь, чтобы считать автоматизацию модульных и приемочных тестов успешной. В главе 4, “Философия автоматизации тестов”, рассматривались отличия в подходах к проектированию и тестированию программного обеспечения. Все это может служить основой для принципов, которым стараются следовать опытные разработчики при автоматизации тестов. Они называются “принципами” по двум причинам: они находятся на слишком высоком уровне, чтобы считаться шаблонами, и они описывают систему ценностей, которая разделяется не всеми разработчиками. Различные системы ценностей могут спровоцировать выбор шаблонов, не описанных в данной книге. Явное описание системы ценностей позволит ускорить процесс определения существующих разногласий и их причин.

Принципы

Когда автору вместе с Шоном Смитом пришлось составлять список качеств тестов в оригинальном Манифесте автоматизации тестов (TAM), рассматривались причины именно такого написания тестов. Манифест представлял собой список свойств, которыми должен обладать тест, а не набор шаблонов, которые можно применить непосредственно. Это позволило определить несколько более конкретных принципов, и некоторые из них рассматриваются в данной главе. От целей их отличает большее количество разговаривающих вокруг них споров.

Принципы являются более “предписывающими”, чем шаблоны, и находятся на более высоком уровне абстракции. В отличие от шаблонов принципы не имеют альтернатив. Они просто представляются в формате “делайте так, потому что”. Для облегчения идентификации принципы формулируются в повелительном наклонении.

В большинстве случаев принципы одинаково хорошо применяются к модульным тестам и тестам историй. Возможным исключением является принцип *проверяйте одно условие за тест* (Verify One Condition per Test), который может оказаться непрактичным для приемочных тестов, проверяющих большие подмножества общей функциональности.

Но все равно стоит следовать этим принципам, отклоняясь от них только при полном понимании последствий.

Принцип: сначала пишите тесты (Write the Tests First)

Также известен как:

Разработка на основе тестов (Test-Driven Development), Разработка с тестами в начале (Test-First Development)

Разработка на основе тестов является приобретаемой привычкой. Как только она “врастает в пальцы”, написание кода другим способом кажется странным, как разработка на основе тестов для тех, кто никогда этим не занимался. В пользу такого подхода можно высказать два основных аргумента.

1. Модульные тесты значительно сокращают затраты на отладку (часто эти затраты выше затрат на автоматизацию тестов).
2. Написание тестов перед написанием кода вынуждает к проектированию кода с учетом тестов. При этом тесты не приходится рассматривать как отдельное требование к дизайну. Нужный результат просто получается, так как тесты уже написаны.

Принцип: проектируйте с учетом тестирования (Design for Testability)

На фоне предыдущего принципа этот принцип может показаться избыточным. Для разработчиков, проигнорировавших принцип *сначала пишите тесты* (Write the Tests First), принцип *проектируйте с учетом тестирования* (Design for Testability) оказывается еще более важным, так как, если тесты не учитывались на этапе проектирования, написать автоматизированные тесты после не получится. О возникающих трудностях может рассказать любой разработчик, которому пришлось добавлять автоматизированные модульные тесты в унаследованный программный продукт. Специальные методики добавления тестов в такой ситуации рассматривал Майк Фезерс [WEwLC].

Принцип: сначала используйте “главный” вход (Use the Front Door First)

Также известен как:

Сначала через главный вход (Front Door First)

Объекты имеют несколько типов интерфейсов. Существует “открытый” интерфейс, которым пользуются клиенты. Может существовать “частный” интерфейс, которым могут воспользоваться только близкие друзья.

Многие объекты имеют “исходящий интерфейс”, который состоит из фрагментов интерфейсов вызываемых объектов.

Типы интерфейсов оказывают влияние на устойчивость тестов к изменениям. Использование *манипуляций через “черный ход”* (Back Door Manipulation, с. 359) для настройки тестовой конфигурации или проверки ожидаемого результата может привести к появлению *слишком связанного теста* (Overcoupled Test; см. “Хрупкий” тест, Fragile Test, с. 277), что потребует более частого обслуживания. Злоупотребление *проверкой поведения* (Behavior Verification, с. 489) и *подставными объектами* (Mock Object, с. 558) приводит к появлению *зарегулированной программы* (Overspecified Software; см. “Хрупкий” тест, Fragile Test) и тестов, стимулирующих разработчиков не проводить необходимый рефакторинг.

Если все варианты одинаково эффективны, стоит использовать тест, работающий через открытый интерфейс. В таком случае объект тестируется через открытый интерфейс с **проверкой состояния** (State Verification, с. 484), чтобы определить правильность его поведения. Если не удается корректно описать ожидаемое поведение, можно создать *тест с пересечением уровней* (Layer-Crossing Test) и использовать *проверку поведения* (Behavior Verification) для контроля за обращениями тестируемой системы к **вызываемым компонентам** (depended-on components). Если приходится заменять медленный или недоступный вызываемый компонент более быстрым *тестовым двойником* (Test Double, с. 538), предпочтительнее использовать *поддельный объект* (Fake Object, с. 565), так как он вносит меньше предположений в тест (единственным предположением является необходимость компонента, заменяемого *поддельным объектом*, Fake Object).

Принцип: доносите намерение (Communicate Intent)

Полностью автоматизированные тесты (Fully Automated Tests), а особенно *тесты на основе сценария* (Scripted Test, с. 319), являются программами. Они должны иметь правильный синтаксис для успешной компиляции и правильную семантику для успешной работы. В teste должна быть реализована логика для перевода тестируемой системы в подходящее начальное состояние и для сравнения ожидаемого результата с фактическим. Хотя все эти характеристики и являются необходимыми, их недостаточно, так как игнорируется самый важный интерпретатор тестов: **ответственный за тесты** (test maintainer).

Также известен как:
Язык высокого уровня (Higher Level Language), Понимание с одного взгляда (Single-Glance Readable)

Большой объем кода (больше десяти строк) или *условная логика теста* (Conditional Test Logic, с. 243) обычно служат причиной появления *непонятного теста* (Obscure Test, с. 230). Его намного сложнее понять, так как приходится извлекать “общую картину” из множества деталей. Это значит, что потребуется больше времени при обслуживании теста или при попытке использования *тестов как документации* (Tests as Documentation). Кроме того, повышается стоимость владения тестом и снижается отдача от затраченных усилий.

При соблюдении данного принципа тесты становятся проще в понимании и обслуживании. В этом помогает вызов *вспомогательных методов теста* (Test Utility Method, с. 610) с описательными названиями [SBPP] для настройки тестовой конфигурации и проверки достижения ожидаемого результата. Из *тестового метода* (Test Method, с. 378) должно быть очевидно, как тестовая конфигурация влияет на ожидаемый результат каждого теста, т.е. какие именно входные данные позволяют получить конкретные выходные данные. Богатая библиотека *вспомогательных методов теста* (Test Utility Method) также упрощает написание тестов, поскольку не приходится кодировать все элементы в каждом teste.

Принцип: не модифицируйте тестируемую систему (Don't Modify the SUT)

Эффективное тестирование часто требует замены фрагмента приложения *тестовым двойником* (Test Double) или переопределить часть поведения с помощью *связанного с тестом подкласса* (Test-Specific Subclass, с. 591). Это может быть связано с необходимостью перехватить контроль над опосредованными входными данными или *проверить поведение* (Behavior Verification), перехватывая опосредованный вывод. Также причиной могут стать недопустимые побочные эффекты элементов приложения или зависимости, которые невозможно удовлетворить в тестовой среде или среде разработки.

Модификация тестируемой системы может оказаться опасным предприятием как при добавлении ловушек для тестов (Test Hook, с. 713), так и при переопределении поведения в *связанном с тестом подклассе* (Test-Specific Subclass) или замене вызываемого компонента *тестовым двойником* (Test Double). В любой из этих ситуаций может оказаться, что тестируется не тот код, который будет присутствовать в конечном продукте.

Необходимо обеспечить тестирование программного обеспечения в конфигурации, в которой оно будет использоваться во время реальной работы. Если действительно приходится заменять вызываемый компонент для получения большего контроля над контекстом тестируемой системы, необходимо делать это правильно. В противном случае может оказаться замененным компонент тестируемой системы, который тестируется. Предположим, что разрабатываются тесты для объектов X, Y и Z, где объект X зависит от объекта Y, который, в свою очередь, зависит от объекта Z. При написании тестов для объекта X имеет смысл заменить Y и Z *тестовым двойником* (Test Double). При тестировании Y объект Z можно заменить на *тестовый двойник* (Test Double). Но при тестировании объекта Z заменить его *тестовым двойником* (Test Double) нельзя, так как именно для этого объекта создаются тесты! Данная мысль является одной из самых важных при рефакторинге кода для упрощения тестирования.

При использовании *связанного с тестом подкласса* (Test-Specific Subclass) для переопределения части поведения объекта в целях тестирования необходимо внимательно следить, чтобы переопределялись только те методы, которые необходимо отключить или использовать для вставки опосредованных входных данных. Если принято решение повторно использовать *связанный с тестом подкласс* (Test-Specific Subclass), предназначенный для другого теста, убедитесь, что он не переопределяет поведение, проверяемое текущим тестом.

Этот принцип можно представить еще одним способом: термин *тестируемая система* определяется относительно создаваемого теста. В примере “X использует Y использует Z” для некоторых тестов компонентов в качестве тестируемой системы может выступать состоящий из X, Y и Z агрегат. С точки зрения модульных тестов в качестве тестируемой системы может выступать отдельно X, Y или Z. Единственным случаем, когда в качестве тестируемой системы выступает все приложение, является проведение приемочных тестов через пользовательский интерфейс с проверкой всех компонентов вплоть до базы данных. Даже в таком случае может тестироваться только один модуль всего приложения (например, “модуль управления клиентами”). Таким образом, “тестируемая система” очень редко является “приложением”.

Принцип: сохраняйте независимость тестов (Keep Tests Independent)

Также известен как:

Независимый тест
(Independent Test)

При тестировании вручную распространена практика создания длинных процедур тестирования, проверяющих множество аспектов поведения тестируемой системы за один тест. Такая агрегация задач необходима в связи

с объемом работы, которую необходимо выполнить для приведения системы в начальное состояние. Операции по настройке начального состояния одного теста могут просто повторять операции, используемые при проверке других элементов поведения. При тестировании вручную такое повторение может оказаться неэффективным. Кроме того, тестеры-люди могут распознать момент отказа теста, который не позволит продолжить тестирование, необходимость пропуска некоторых тестов или то, что результаты последующих тестов (даже завершившихся неудачно) не важны.

Если тесты зависят один от другого и, что еще хуже, от порядка выполнения, разработчик лишается полезной обратной связи, характерной для отказов отдельных тестов. *Взаимодействующие тесты* (Interacting Tests; см. *Нестабильный тест*, Erratic Test, с. 267) отказывают группами. Если отказал тест, который приводит систему в состояние, необходимое для следующего теста, следующий тест тоже завершится неудачно. Если оба теста отказали, нельзя быть уверенным, что это сигнализирует о проблеме в обоих тестируемых фрагментах кода, а не в фрагменте кода, который проверяет первый тест. Отказ обоих тестов просто не несет такой информации. Представьте, что речь идет не о двух тестах, а о десятках и даже сотнях *взаимодействующих тестов* (Interacting Tests).

Независимые тесты (Independent Tests) могут запускаться по отдельности. Каждый тест имеет собственную *новую тестовую конфигурацию* (Fresh Fixture, с. 344), которая переводит систему в необходимое состояние. Использующие *новую тестовую конфигурацию* (Fresh Fixture) тесты будут работать независимо с большей вероятностью, чем тесты на основе *общей тестовой конфигурации* (Shared Fixture, с. 350). Использование общей конфигурации может привести к появлению различных типов *нестабильных тестов* (Erratic Test), включая *одинокий тест* (Lonely Test), *взаимодействующие тесты* (Interacting Tests) и “войны” запуска тестов (Test Run War). При использовании независимых тестов отказ модульного теста выступает в роли средства *локализации дефектов* (Defect Localization) и позволяет обнаружить конкретную причину отказа.

Принцип: изолируйте тестируемую систему (Isolate the SUT)

Некоторые элементы программного обеспечения зависят только от (предположительно правильной) среды выполнения или операционной системы. Множество компонентов зависит от других компонентов собственной или сторонней разработки. Если одно программное обеспечение зависит от другого, меняющегося со временем, программного обеспечения, тесты могут неожиданно завершаться неудачно, если поведение других компонентов изменилось. Эта проблема известна как *чувствительность к контексту* (Context Sensitivity; см. “Хрупкий” тест, Fragile Test) и является вариантом “хрупкого” теста (Fragile Test).

Если одни программные компоненты зависят от других программных компонентов, поведение которых нельзя проконтролировать, очень сложно проверить корректность работы кода во всех ситуациях. Скорее всего, это приведет к появлению *не тестируемого кода* (Untested Code; см. *Ошибки в продукте*, Production Bugs, с. 303) или *не тестируемых требований* (Untested Requirement; см. *Ошибки в продукте*, Production Bugs). Обойти эту проблему можно, только вставив все возможные реакции вызываемого объекта в программный компонент под полным контролем со стороны тестов.

При тестировании приложения, компоненты, классы или методы необходимо максимально изолировать от всех остальных элементов программного продукта. Такая изоляция элементов позволяет *тестировать аспекты отдельно* (Test Concerns Separately) и *сохранять независимость тестов* (Keep Tests Independent). Кроме того, снижается *чувствительность к контексту* (Context Sensitivity), вызываемая слишком тесным связыванием тестируемой системы и окружающего ее программного кода.

Для соответствия данному принципу программное обеспечение имеет смысл проектировать с учетом замены каждого вызываемого компонента *тестовым двойником* (Test Double) через *вставку зависимости* (Dependency Injection, с. 684) или *поиск зависимости* (Dependency Lookup, с. 692). Также можно переопределить вызываемый компонент с по-

мошью *связанного с тестом подкласса* (Test-Specific Subclass), который позволяет контролировать опосредованные входные данные тестируемой системы. Подобный стиль проектирования позволяет упростить повторение и модификацию тестов.

Принцип: минимизируйте пересечения тестов (Minimize Test Overlap)

В большинстве приложений требуется проверять большое количество функций. Практически невозможно доказать правильную работу функций во всех возможных комбинациях и сценариях взаимодействия. Таким образом, создание набора тестов является задачей управления рисками.

Тесты должны быть структурированы таким образом, чтобы от конкретного элемента приложения зависело как можно меньше. Сначала такой подход может показаться интуитивно непонятным, поскольку для повышения покрытия тестами можно попытаться тестировать функции как можно чаще. К сожалению, несколько тестов для одной и той же функциональности отказывают одновременно. Кроме того, они требуют одинакового обслуживания при модификации функциональности тестируемой системы. Если несколько тестов проверяют одну и ту же функцию, повышается только стоимость обслуживания, а качество остается тем же.

Все действительно интересные условия должны проверяться тестами. Каждому условию должен соответствовать ровно один тест, ни больше, ни меньше. Если оказывается, что код имеет смысл проверять несколькими способами, то можно считать, что обнаружено несколько условий для тестов.

Принцип: минимизируйте нетестируемый код (Minimize Untestable Code)

Некоторые типы кода очень сложно тестировать с помощью *полностью автоматизированных тестов* (Fully Automated Tests). В качестве примеров такого кода можно назвать код компонентов графического интерфейса, многопотоковый код и *тестовые методы* (Test Method). Все они имеют один недостаток: они встраиваются в сложно создаваемый контекст или имеют сложности во взаимодействии с автоматизированными тестами.

Для нетестируемого кода просто не существует *полностью автоматизированных тестов* (Fully Automated Tests), которые должны защищать от мелких ошибок, “самостоятельно” проникающих в код. В результате осложняется безопасный рефакторинг и модификация становится более опасной.

Желательно уменьшить объем нетестируемого кода, если продукт придется сопровождать в дальнейшем. Такой код можно подвергнуть рефакторингу для выноса интересующей логики из класса, имеющего сложные отношения с тестами. Активные объекты и многопотоковый код могут быть преобразованы в *минимальный выполняемый файл* (Humble Executable; см. *Минимальный объект*, Humble Object, с. 700). Объекты пользовательского интерфейса можно преобразовать в *минимальный диалог* (Humble Dialog; см. *Минимальный объект*, Humble Object). Даже из *тестового метода* (Test Method) большую часть нетестируемого кода можно вынести во вспомогательный метод теста (Test Utility Method), который можно тестировать отдельно.

Если минимизировать нетестируемый код (Minimize Untestable Code), увеличивается общее покрытие тестами. Таким образом, усиливается уверенность в коде и повышается его способность к рефакторингу. Еще одним преимуществом является повышение качества кода.

Принцип: не вносите логику тестов в код продукта (Keep Test Logic Out of Production Code)

Если код продукта не проектировался с учетом тестов, может возникнуть соблазн добавить “ловушки” в продукт для облегчения тестирования. Такие ловушки обычно принимают вид `if testing then...` и могут как вызывать альтернативную логику, так и запрещать запуск определенной логики продукта.

Тестирование подразумевает проверку поведения системы. Если во время теста система имеет совершенно другое поведение, как убедиться, что код продукта действительно работает? Что еще хуже, “ловушки” для тестов могут привести к отказу продукта в реальной эксплуатации!

Код продукта не должен содержать условных операторов вида `if testing then`. Также он не должен содержать логики тестов. В хорошо спроектированной (с точки зрения тестов) системе поддерживается изоляция функциональности. Объектно-ориентированные системы хорошо поддаются тестированию, так как они состоят из отдельных объектов. К сожалению, даже объектно-ориентированные системы можно построить таким образом, что тестирование будет затруднено и в продукте будут встречаться фрагменты с логикой тестов.

Принцип: проверяйте одно условие за тест (Verify One Condition per Test)

Многим тестам требуется начальное состояние, отличное от принятого по умолчанию начального состояния тестируемой системы, а многие операции переводят тестируемую систему из исходного состояния в другое.

Возникает большой соблазн воспользоваться конечным состоянием после запуска одного теста в качестве начального состояния для другого теста, объединив проверку двух условий в один *тестовый метод* (Test Method). Но поддаваться такому соблазну не стоит, так как если одно утверждение окажется неудачным, остальные тесты просто не будут выполняться. В результате будет сложно добиться *локализации дефектов* (Defect Localization).

Проверка нескольких условий в одном teste имеет смысл при тестировании вручную, когда накладные расходы на формирование начального состояния велики и человек может адаптироваться к неудачному завершению тестов. При тестировании вручную слишком много времени уходит на формирование отдельной тестовой конфигурации для каждого теста, поэтому люди-тестеры стараются писать длинные проверки для нескольких условий¹. Кроме того, человек обладает достаточным интеллектом для обхода любых возникающих проблем. В результате не все оказывается потерянным, если один тест завершился неудачно. С другой стороны, при автоматизированном тестировании одно неудачно завершившееся утверждение приводит к остановке теста (остальные проверки не вернут информацию о работающих и неработающих функциях).

Каждый *тест на основе сценария* (Scripted Test) должен проверять единственное условие. Это возможно, так как тестовая конфигурация формируется программно, а не вручную. Программы создают тестовую конфигурацию очень быстро и не против повторить

Также известен как:

Не допускать тестовую логику в код продукта (No Test Logic in Production Code)

Также известен как:

Тест одного условия (Single-Condition Test)

¹ Умные тестеры обычно пользуются автоматизированным сценарием для перевода тестируемой системы в правильное начальное состояние, избегая длительных ручных операций.

одну и ту же последовательность операций сотни раз! Если нескольким тестам нужна одна и та же тестовая конфигурация, *тестовые методы* (Test Method) можно перенести в один класс теста для каждой тестовой конфигурации (Testcase Class per Fixture, с. 639), чтобы можно было воспользоваться *неявной настройкой* (Implicit Setup, с. 449). Кроме того, можно вызвать *вспомогательный метод теста* (Test Utility Method) для создания тестовой конфигурации через *делегированную настройку* (Delegated Setup, с. 437).

Каждый тест проектируется с учетом деления на четыре отдельные фазы (см. *Четырехфазный тест*; Four-Phase Test, с. 387), которые выполняются последовательно: *настройка тестовой конфигурации* (fixture setup), *вызов testируемой системы* (exercise SUT), *проверка результата* (result verification) и *очистка тестовой конфигурации* (fixture teardown).

- Во время первой фазы настраивается тестовая конфигурация (ситуация “до теста”), которая необходима testируемой системе для проявления ожидаемого поведения, а также содержит все необходимое для наблюдения за фактическим результатом (например, с использованием *тестового двойника*, Test Double).
- Во время второй фазы происходит взаимодействие с testируемой системой для запуска поведения, которое необходимо проверить. Это должен быть один ограниченный аспект поведения. Если происходит попытка вызова нескольких элементов testируемой системы, это будет не *тест одного условия* (Single Condition Test).
- На третьей фазе проверяется получение ожидаемого результата. Если результат не получен, тест завершается неудачно.
- Во время четвертой фазы удаляется тестовая конфигурация и среда выполнения возвращается в исходное состояние.

Обратите внимание на единственную фазу вызова testируемой системы и единственную фазу проверки результата. Рекомендуется избегать последовательности таких вызовов (вызов, проверка, вызов, проверка), поскольку это сделает необходимой проверку нескольких условий, что желательно поручить нескольким отдельным *тестовым методам* (Test Method).

Одним потенциально спорным аспектом принципа *проверяйте одно условие за тест* (Verify One Condition per Test) является значение фразы “одно условие”. Некоторые разработчики тестов подразумевают одно **утверждение** (assertion) на тест. Эта настойчивость может быть следствием использования организации *тестовых методов* (Test Method) вида *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture) с именованием каждого теста на основе проверочного утверждения, например `AwaitingApprovalFlight.validApproverRequestShouldBeApproved`. Единственное утверждение на тест делает такую схему именования очень удобной, но в то же время требует генерации большого количества тестовых методов на случай проверки утверждений относительно нескольких полей результата. Конечно, для следования такой интерпретации можно выделить *специальное утверждение* (Custom Assertion, с. 495) или *метод проверки* (Verification Method; см. *Специальное утверждение*, Custom Assertion), позволяющий сократить несколько вызовов методов до одного. Иногда такой подход делает тест более удобным для чтения. В противном случае не стоит слепо следовать рекомендациям по сохранению единственного утверждения на тест.

Принцип: тестируйте аспекты по-отдельности (Test Concerns Separately)

Поведение сложного приложения состоит из поведения большого количества меньших компонентов. Иногда компонент может вести себя по-разному. Каждый тип поведения является отдельным аспектом и может требовать значительного количества сценариев проверки.

Проблема при тестировании нескольких аспектов в одном *тестовом методе* (Test Method) заключается в нарушении работы теста при модификации любого из аспектов. Что еще хуже, причина отказа может оказаться неочевидной. Идентификация реальной причины обычно требует *отладки вручную* (Manual Debugging; см. *Частая отладка*, Frequent Debugging, с. 285) из-за недостаточной *локализации дефектов* (Defect Localization). В общем, можно отметить, что будет неудачно завершаться больше тестов и каждый из них потребует больше времени на диагностику и исправление. Рефакторинг также затрудняется при проверке нескольких аспектов в пределах одного теста. Поскольку каждый тест требует значительного перепроектирования, затрудняется разбиение на части слишком больших классов.

Тестирование аспектов по-отдельности позволяет использовать неудачное завершение теста в качестве указателя на причину проблемы в конкретной части системы, а не просто просигнализировать о проблеме “где-то”. Такой подход к тестированию делает поведение проще в понимании изначально и обеспечивает простоту рефакторинга в дальнейшем. Таким образом, появляется возможность перенести подмножество тестов в другой *класс теста* (Testcase Class, с. 401), проверяющий только что созданные классы. При этом потребуется только изменить имя класса тестируемой системы.

Принцип: обеспечьте адекватные усилия и ответственность (Ensure Commensurate Effort and Responsibility)

Объем трудозатрат на написание или модификацию теста не должен превышать объем затрат на реализацию соответствующей функциональности. Точно так утилиты для написания и обслуживания тестов не должны требовать больших знаний, чем утилиты для реализации функциональности. Например, если можно настроить поведение тестируемой системы с помощью метаданных и необходимо написать тесты, проверяющие правильность настройки метаданных, дополнительный код не должен требоваться. В такой ситуации более подходящим решением должен быть *управляемый данными тест* (Data-Driven Test, с. 322).

Что дальше

В предыдущих главах рассматривались распространенные “подводные камни” (в виде запахов тестов) и цели автоматизации тестирования. В этой главе явно описана система ценностей, на основе которой выбираются подходящие шаблоны. В главе 6, “Стратегия автоматизации тестирования”, рассматриваются решения, которые должны быть правильными в начале проекта, так как позднее их очень сложно изменить.

Глава 6

Стратегия автоматизации тестирования

О чём идет речь в этой главе

В предыдущих главах рассматривались некоторые проблемы автоматизации тестирования. В главе 5, “Принципы автоматизации тестирования”, были описаны принципы, позволяющие обойти эти проблемы. В настоящей главе рассматриваются более конкретные подходы, но основное внимание по-прежнему сконцентрировано на “высоте птичьего полета”. Логически стратегия тестирования рассматривается перед настройкой тестовой конфигурации, но является более сложной темой. Если у вас недостаточно опыта в автоматизации тестирования с помощью семейства пакетов xUnit, пропустите эту главу и сразу перейдите к главе 7, “Основы xUnit”, главе 8, “Управление временной тестовой конфигурацией” и последующим главам.

Что значит “стратегический”

Как было сказано в прологе, очень легко пойти по неправильному пути, особенно когда недостаточно опыта автоматизации тестирования и стратегия тестирования внедряется “вверх дном”. Если обнаружить проблемы достаточно рано, стоимость рефакторинга тестов будет вполне допустимой. Если же проблемы “проживут” достаточно долго или для их решения будет выбран неправильный подход, слишком много усилий окажутся потраченными впустую. Это не значит, что предлагается проектировать все тесты сразу. Практически всегда такой подход оказывается неверным. Вместо этого желательно иметь представление о необходимых стратегических решениях и принимать эти решения “вовремя”, а не “слишком поздно”. В этой главе рассматриваются некоторые стратегические проблемы, о которых имеет смысл помнить, чтобы не столкнуться с ними на более поздних этапах.

Что делает решение “стратегическим”? Стратегическое решение “трудно изменить”. Иначе говоря, уже принятые решения влияют на большое количество тестов (вплоть до того, что большинство или все тесты придется конвертировать в соответствии с другим подходом), т.е. любое решение, изменение которого стоит достаточно дорого, является стратегическим.

В качестве примеров стратегических решений можно привести следующие альтернативы.

- Какие тесты подвергать автоматизации?
- Каким инструментарием воспользоваться для автоматизации?
- Как управлять тестовыми конфигурациями?
- Как обеспечить простоту тестирования и взаимодействия с тестируемой системой?

Каждое из этих решений может иметь далеко идущие последствия, поэтому важно принимать их осознанно, в нужный момент и на основе максимально достоверной информации.

Описанные в данной книге стратегии и более подробные шаблоны одинаково применимы вне зависимости от выбранной *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332). Автор имеет большой опыт работы с семейством xUnit, поэтому именно xUnit уделяется основное внимание. Но не “выплескивайте ребенка вместе с водой”: даже при использовании другой *инфраструктуры автоматизации тестов* (Test Automation Framework) большинство описанных здесь навыков все равно будет применимо.

Какие тесты подвергать автоматизации

Грубо говоря, тесты можно разделить на две категории:

- тесты функциональности, которые проверяют поведение тестируемой системы в ответ на определенные стимулы;
- кроссфункциональные тесты, которые проверяют различные аспекты поведения системы, пересекающие границы отдельных функций.

На рис. 6.1 основные типы тестов показаны в виде двух столбцов. Каждый из них делится на более специализированные типы тестов.

Тесты функциональности

Тесты функциональности проверяют непосредственно наблюдаемое поведение фрагмента программного обеспечения. Функциональность может быть связана с бизнес-уровнем (например, основные сценарии использования системы) или с операционными требованиями (например, обслуживание системы и конкретные сценарии устойчивости к сбоям). Многие из этих требований можно выразить в виде **примеров использования** (use cases), **функций** (feature), **пользовательских историй** (user stories) или сценариев тестирования.

Тесты функциональности можно охарактеризовать уровнем проверяемой функциональности (бизнес-уровень или пользовательский уровень) и размером тестируемой системы, с которой они работают.

Приемочные тесты

Приемочные тесты проверяют поведение всей системы или всего приложения. Обычно они соответствуют сценариям одного или нескольких примеров использования, функций или пользовательских историй и называются тестами функциональности или

пользовательскими тестами. Хотя они могут быть автоматизированы разработчиками, ключевой характеристикой является возможность со стороны конечного пользователя распознать поведение, описываемое тестом.

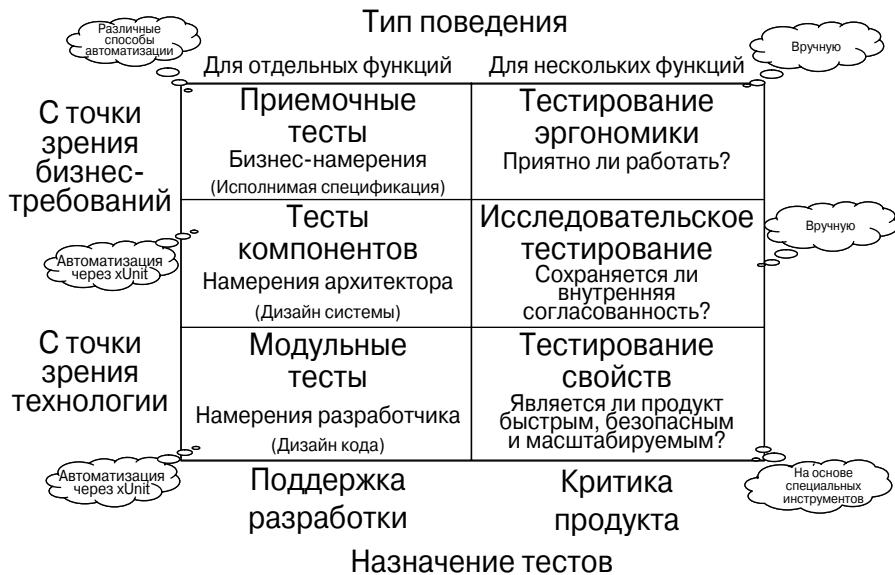


Рис. 6.1. Общая информация о типах тестов и причинах написания тестов таких типов. В левом столбце приведены тесты, описывающие функциональность продукта с различными уровнями детализации; эти тесты используются для поддержки процесса разработки. В правом столбце приведены тесты, пересекающие отдельные виды функциональности; эти тесты используются для критики продукта. В нижней части каждой ячейки указывается, что именно проверяется или какая мысль пытается донести тест

Модульные тесты

Модульные тесты проверяют поведение одного класса или метода, который является результатом решений, принятых на этапе проектирования. Обычно это поведение не связано с требованиями к системе, кроме случаев, когда ключевой элемент бизнес-логики реализован в конкретном классе или методе. Такие тесты пишутся разработчиками для себя. Они позволяют определить, что значит “готово”, описав поведение модуля в виде теста.

Тесты компонентов

Тесты компонентов проверяют компоненты, состоящие из групп классов и обеспечивающих определенную службу. С точки зрения размера тестируемой системы их место — где-то между модульными и приемочными тестами. Хотя некоторые разработчики называют их “интеграционными тестами” или “тестами подсистем”, эти термины могут иметь совершенно другое значение, отличающееся от значения “тесты конкретных крупных компонентов общей системы”.

Тесты с внедрением ошибок

Такие тесты обычно появляются на всех трех уровнях внутри тестов функциональности. На каждом уровне внедряются ошибки различных типов. С точки зрения автоматизации тестов внедрение ошибки является просто еще одним набором тестов на уровнях модулей и компонентов. Но на уровне всего приложения все становится еще интереснее. Ошибки на этом уровне сложнее автоматизировать, так как при их внедрении возникают сложности, решаемые только заменой фрагментов приложения.

Кроссфункциональные тесты

Тесты свойств

Тесты быстродействия проверяют различные “нефункциональные” (также известные как “экстрафункциональные” или “кроссфункциональные”) требования к системе. Отличием этих требований является пересечение границ функций. Обычно они соответствуют архитектурным “особенностям”. Среди них можно назвать следующие тесты:

- тесты времени реакции;
- тесты вместимости;
- стресс-тесты.

С точки зрения автоматизации многие тесты должны быть автоматизированы (как минимум частично), так как “живые тестеры” не смогут создать достаточную нагрузку для проверки поведения системы под нагрузкой. Хотя одни и те же тесты можно запускать несколько раз подряд с помощью xUnit, данная инфраструктура не очень хорошо подходит для автоматизации тестов быстродействия.

Одним из преимуществ гибких методов разработки является возможность запуска таких тестов на ранних этапах разработки — как только ключевые компоненты архитектуры выделены и основную функциональность можно запускать. Те же самые тесты можно запускать на протяжении всего проекта во время добавления новых функций к “скелету” системы.

Тесты эргономики

Тесты эргономики позволяют убедиться, что реальные пользователи в состоянии достигать заявленных целей с помощью приложения. Эти тесты очень сложно автоматизировать, так как необходима субъективная оценка простоты использования тестируемой системы. По этой причине тесты эргономики редко автоматизируются и в данной книге рассматриваться не будут.

Исследовательские тесты

Исследовательские тесты позволяют определить внутреннее единство продукта. Они используют продукт, наблюдают за его поведением, формируют гипотезы, проектируют тесты для проверки гипотез и вызывают продукт с использованием тестов. Из-за своей природы исследовательские тесты не могут быть автоматизированы, хотя автоматизированные тесты могут использоваться для подготовки тестируемой системы.

Инструментарий для автоматизации

Выбор подходящего инструментария важен настолько же, насколько важны навыки работы с выбранным инструментарием. На рынке существует множество инструментов, и очень легко соблазниться простотой использования некоторых из них. Выбор инструмента является стратегическим решением: после затрат времени на его изучение и автоматизацию множества тестов становится очень сложно перейти на другой инструмент.

В автоматизации тестов существует два фундаментально отличающихся подхода (рис. 6.2). Подход на основе *записанных тестов* (Recorded Test, с. 312) подразумевает использование утилит, отслеживающих взаимодействие пользователя и тестируемой системы при тестировании вручную. Полученная информация сохраняется в файл или базу данных и превращается в сценарий для воспроизведения относительно другой (или той же) версии тестируемой системы. Основной проблемой *записанных тестов* (Recorded Test) является обеспечиваемый уровень детальности. Большинство утилит записывают действия на уровне пользовательского интерфейса, что приводит к появлению “хрупких” тестов (Fragile Test, с. 277).

Второй подход к автоматизации на основе *созданных вручную сценариев тестов* (Hand-Scripted Test; см. *Тест на основе сценария*, Scripted Test, с. 319) подразумевает создание вызывающих систему тестовых программ (“сценариев”) вручную. Хотя в качестве *инфраструктуры автоматизации тестов* (Test Automation Framework) чаще всего используется xUnit, тесты могут создаваться и в других форматах, включая “пакетные” файлы, макроязыки и коммерческие или открытые утилиты тестирования. В качестве широко известных утилит для подготовки *тестов на основе сценария* (Scripted Test) можно назвать Watir (сценарии на языке Ruby, которые запускаются внутри Internet Explorer), Canoo WebTest (сценарии на языке XML, выполняемые утилитой WebTest) и Fit (с его основанным на формате Wiki родственным FitNesse). Некоторые из этих утилит обеспечивают функции захвата тестов, размыкая границу между тестами на основе сценария (Scripted Test) и записанными тестами (Recorded Test).

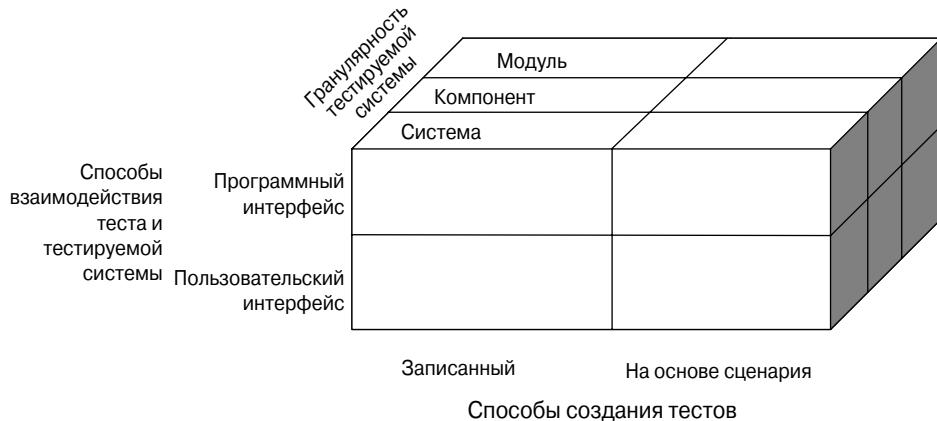


Рис. 6.2. Три измерения при выборе средств автоматизации тестов. Слева показаны два способа взаимодействия с тестируемой системой. Внизу перечислены способы создания сценариев тестов. Измерение “глубины” показывает различные размеры тестируемой системы

Выбор инструмента автоматизации тестов является значительной частью общей стратегии тестирования. Полный обзор различных средств автоматизации тестов здесь не приводится. Детальнее эта тема рассматривается в статье [ARTRP]. В следующих разделах приводится общий обзор слабых и сильных сторон каждого подхода.

Способы автоматизации тестирования и подходы к ней

На рис. 6.3 возможные варианты автоматизации тестирования показаны в виде матрицы. В теории матрица содержит $2 \times 2 \times 3$ возможных комбинаций, но даже передняя грань позволяет оценить основные отличия подходов; некоторые из них предназначены для автоматизации приемочных тестов.

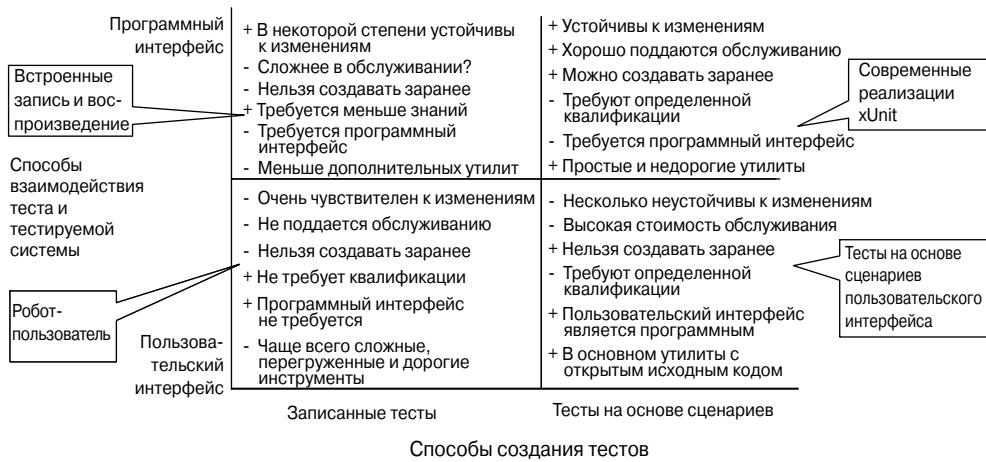


Рис. 6.3. Варианты на передней грани. Это более подробное представление передней грани, показанной на рис. 6.2. Здесь также показаны преимущества (+) и недостатки (-) каждого подхода

Верхний правый квадрант: современное семейство xUnit

В верхнем правом квадранте передней грани доминирует семейство xUnit. Инфраструктура xUnit предполагает использование написанных вручную тестов, вызывающих систему на всех уровнях абстракции (система, компонент, модуль) через внутренние интерфейсы. Хорошим примером могут служить модульные тесты, автоматизированные с помощью JUnit или NUnit.

Нижний правый квадрант: тесты пользовательского интерфейса на основе сценариев

В этом квадранте описывается вариант подхода на основе “современного эквивалента xUnit”. Наиболее распространенным примером такого подхода являются пакеты HttpUnit, JFCUnit, Watir и другие тесты пользовательского интерфейса на основе сценариев. Сценарии можно создавать и с помощью коммерческих утилит для работы с *записанными тестами* (Recorded Test), например QTP. Все эти подходы находятся в нижнем правом квадранте вне зависимости от уровня абстракции. Они могут использоваться для тестирования компонентов графического интерфейса (или даже некоторых модулей графиче-

ского интерфейса), хотя это потребует создания заглушек вместо настоящей системы, взаимодействующей с графическим интерфейсом.

Нижний левый квадрант: “робот”-пользователь

В этом квадранте основное внимание уделяется записи тестов, взаимодействующих с системой через пользовательский интерфейс. Большинство коммерческих утилит автоматизации тестов следуют такому подходу. В основном, он применяется на уровне абстракции “вся система”, но, как и тесты пользовательского интерфейса на основе сценариев, могут применяться к компонентам или модулям графического интерфейса, если остальную систему можно заменить заглушками.

Верхний левый квадрант: внутренняя запись

Для полноты картины в верхнем левом квадранте показано создание *записанных тестов* (Recorded Test) с помощью программного интерфейса за графическим интерфейсом, когда записываются все входные данные и ответы тестируемой системы. Это может потребовать вставки точек наблюдения между тестируемой системой (на интересующем уровне абстракции) и вызываемым компонентом. В процессе воспроизведения тестов программный интерфейс вставляет записанные входные данные и сравнивает записанный результат с ожидаемым.

Этот квадрант не переполнен коммерческими утилитами¹. Однако такой подход может быть оправдан при встраивании механизма *записи тестов* (Recorded Test) в само приложение.

Введение в xUnit

Семейство xUnit предназначено для автоматизации тестов, написанных разработчиками. Оно проектировалось для достижения следующих целей.

- Упростить написание тестов за счет отказа от нового языка программирования. Пакеты xUnit доступны для большинства современных языков.
- Упростить тестирование отдельных классов и объектов при недоступности остальных элементов приложения. Пакет xUnit позволяет тестировать программное обеспечение изнутри. Для этого достаточно следовать определенным принципам во время проектирования.
- Упростить запуск одного или нескольких тестов с помощью единственной команды. В пакете xUnit существует концепция набора тестов и *набора наборов* (Suite of Suites; см. *Объект набора тестов*, Test Suite Object, с. 414) для поддержки такого способа запуска тестов.
- Минимизировать стоимость запуска тестов таким образом, чтобы разработчики не стремились избежать их запуска. По этой причине каждый тест должен быть *само-*

¹ Большинство утилит в этом квадранте основное внимание уделяют регрессионным тестам, вставляя точки наблюдения в приложение на основе компонентов и записывая (удаленные) вызовы методов и ответы различных компонентов. Такой подход становится все более популярным с развитием **архитектуры на основе служб** (service-oriented architecture — SOA).

проверяющимся тестом (Self-Checking Test, с. 81) и содержать реализацию “принципа Голливуда”².

Семейство xUnit исключительно успешно достигает данных целей. Скорее всего, Эрик Гамма и Кент Бек даже не представляли, какое влияние окажет пакет JUnit на разработку программного обеспечения³. Однако те характеристики, которые делают семейство xUnit подходящим для автоматизации тестов разработчиков, делают это семейство менее подходящим для написания тестов других типов. В частности, поведение утверждений “останавливаться при первом отказе” часто критикуется (или переопределяется) разработчиками, которые пытаются автоматизировать с помощью xUnit многошаговые приемочные тесты и получить общую картину (что работает, а что нет) вместо обнаружения первого отклонения от ожидаемых результатов. В итоге можно выделить несколько соображений.

- “Остановка на первом отказе” является философским инструментом, а не характеристикой модульных тестов. Просто большинство разработчиков предпочитают, чтобы модульные тесты останавливались. При этом все осознают, что приемочные тесты должны быть больше модульных.
- Существует возможность модифицировать фундаментальное поведение xUnit в соответствии с собственными потребностями. Эта гибкость является одним из преимуществ инструментария с открытым исходным кодом.
- Необходимость модификации фундаментального поведения xUnit подсказывает, что имеет смысл рассмотреть другие варианты инструментария.

Например, инфраструктура Fit специально проектировалась для запуска приемочных тестов. В этом случае ограничения семейства xUnit, приводящие к “остановке на первом отказе”, преодолеваются за счет передачи состояния теста цветовым кодированием. Еще одной альтернативой для разработчиков на языке Java является пакет TestNG, обеспечивающий явное формирование последовательности *цепочки тестов* (Chained Tests, с. 477).

При этом стоит отметить, что выбор другого инструментария не избавляет от необходимости принимать стратегические решения (если выбранный инструмент не ограничивает множество решений). Например, даже при использовании тестов на основе инфраструктуры Fit придется формировать тестовую конфигурацию. Некоторые шаблоны, например *цепочки тестов* (Chained Tests), когда один тест создает тестовую конфигурацию для следующих, сложно поддаются автоматизации и могут оказаться меньше востребованными в инфраструктуре Fit, чем в xUnit. Именно гибкость семейства xUnit дает разработчику “веревку достаточной длины, чтобы прострелить себе ногу”, позволяя создавать *непонятные тесты* (Obscure Test, с. 230) приводящие к возникновению *высокой стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300).

Сильные стороны xUnit

Семейство xUnit лучше всего справляется с наборами тестов меньшего размера, каждый из которых требует небольшой и простой в создании тестовой конфигурации. Это

² Режиссеры из Голливуда при массовых просмотрах сообщают актерам: “Не звоните нам, мы позвоним вам сами (если вы нас заинтересуете)”.

³ Технически первым был реализован пакет SUnit, но стремительное развитие началось только после появления пакета JUnit и статьи “Test Infected” [TI].

позволяет создать отдельный тест для каждого сценария тестирования каждого объекта. Для управления тестовой конфигурацией должна использоваться стратегия на основе *новой тестовой конфигурации* (Fresh Fixture, с. 344) через создание *минимальной тестовой конфигурации* (Minimal Fixture, с. 336) для каждого теста.

Для максимальной эффективности тесты должны создаваться относительно программного интерфейса, после чего изолированно должны проверяться одиночные классы или небольшие группы классов. При таком подходе можно использовать небольшие тестовые конфигурации, не требующие много времени на генерацию.

При создании приемочных тестов семейство xUnit лучше всего работает при определении языка высокого уровня (Higher-Level Language, с. 95), с помощью которого и описываются тесты. В таком случае уровень абстракции повышается от технических подробностей до понятных для заказчика бизнес-концепций. После этого достаточно просто превратить тесты в *управляемые данными тесты* (Data-Driven Test, с. 322), реализованные на основе xUnit или Fit.

Обратите внимание, что многие описанные здесь высокоуровневые шаблоны и принципы одинаково хорошо подходят как для тестов на основе пакета Fit, так и для тестов на основе семейства xUnit. Кроме того, данные принципы можно использовать при работе с графическими инструментами тестирования, в которых обычно применяется метафора “записи и воспроизведения”. Шаблоны управления тестовыми конфигурациями особенно выделяются на этом фоне, как и повторно используемые “компоненты тестов”, которые можно комбинировать для создания различных тестовых сценариев. Такой подход полностью аналогичен подходу с применением однозадачных *тестовых методов* (Test Method, с. 378), вызывающих *вспомогательные методы теста* (Test Utility Method, с. 610) для снижения связности с программным интерфейсом тестируемой системы.

Управление тестовыми конфигурациями

Стратегия управления тестовыми конфигурациями оказывает значительное влияние на время работы и жизнеспособность тестов. Результат неверного выбора стратегии не будет заметен, пока не наберется несколько сотен тестов и не станет очевидным запах *медленный тест* (Slow Tests, с. 289); после этого через несколько месяцев разработки проявит себя и запах *высокая стоимость обслуживания тестов* (High Test Maintenance Cost). После обнаружения этих запахов станет очевидной необходимость смены стратегии автоматизации тестов (естественно, стоимость смены стратегии будет высокой из-за большого количества уже написанных тестов).

Что такое тестовая конфигурация

Каждый тест состоит из четырех частей, как показано в описании *четырехфазного теста* (Four-Phase Test, с. 387). На первой фазе создается тестируемая система и все, от чего она зависит. Созданные компоненты переводятся в состояние, в котором должна вызываться тестируемая система. В семействе xUnit все необходимые для вызова тестируемой системы компоненты называются **тестовой конфигурацией** (test fixture). Часть логики теста, которая создает тестовую конфигурацию, называется фазой **настройки тестовой конфигурации** (fixture setup).

На этом этапе стоит обратить внимание, что термин “тестовая конфигурация” (fixture) для разных людей имеет разные значения.

- В одних вариантах xUnit концепция тестовой конфигурации (fixture) отделена от *класса теста* (Testcase Class, с. 401), создающего конфигурацию. К этой категории можно отнести пакет JUnit и его непосредственных наследников в других языках.
- В других вариантах xUnit экземпляр *класса теста* (Testcase Class) “является” тестовой конфигурацией (fixture). Хорошим примером такого пакета считается NUnit.
- В третьей категории для тестовой конфигурации используется совершенно иное название. Например, в пакете RSpec предварительные условия теста описываются в классе контекста теста, содержащем *тестовые методы* (Test Method) — как и в NUnit, но с использованием другой терминологии.
- Термин *тестовая конфигурация* (fixture) имеет совершенно иное значение в других пакетах автоматизации тестирования. Например, в пакете Fit он используется для описания самостоятельно созданных фрагментов интерпретатора (Interpreter) для *управляемых данными тестов* (Data-Driven Test), который применяется при определении языка высокого уровня (Higher-Level Language).

Подход “класс является тестовой конфигурацией” предполагает организацию тестов в виде *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639). При выборе альтернативных способов организации тестов, например *класс теста для каждого класса* (Testcase Class per Class, с. 627) или *класс теста для каждой функции* (Testcase Class per Feature, с. 633), слияние концепций тестовой конфигурации и класса теста может вводить в заблуждение. На протяжении всей этой книги термин “тестовая конфигурация” используется для описания “предварительных условий теста”, а термин *класс теста* (Testcase Class) означает “класс, содержащий *тестовые методы* (Test Method) и весь остальной код, необходимый для создания тестовой конфигурации”.

Наиболее распространенным способом создания тестовой конфигурации является использование основного программного интерфейса тестируемой системы через вызов методов, создающих необходимые объекты. Когда состояние тестируемой системы хранится в других объектах или компонентах, можно воспользоваться *настройкой через “черный ход”* (Back Door Setup; см. *Манипуляция через “черный ход”*, Back Door Manipulation, с. 359) путем вставки необходимых записей непосредственно в другие компоненты, от которых зависит поведение тестируемой системы. Чаще всего *настройка через “черный ход”* (Back Door Setup) применяется по отношению к базам данных или при использовании *подставных объектов* (Mock Object, с. 558) или *тестовых двойников* (Test Double, с. 538). Более подробно эти концепции рассматриваются в главе 13, “Тестирование с использованием баз данных”, а также в главе 11, “Использование тестовых двойников”.

Основные стратегии работы с тестовыми конфигурациями

Вероятно, существует больше одного способа классификации любого множества. В данном случае стратегии работы с тестовыми конфигурациями будут классифицироваться в зависимости от задач, возникающих при разработке тестов.

Первая и самая простая стратегия управления тестовыми конфигурациями требует только соблюдения организации кода при создании тестовой конфигурации для каждого

теста, т.е. будет ли код размещен в *тестовом методе* (Test Method), выделен во *вспомогательный метод теста* (Test Utility Method), который вызывается из тестового метода, или добавлен в метод `setUp`, принадлежащий *классу теста* (Testcase Class). Такая стратегия предполагает использование *временной новой тестовой конфигурации* (Transient Fresh Fixture; см. *Новая тестовая конфигурация*, Fresh Fixture). Такие конфигурации существуют только в оперативной памяти и очень удобно исчезают сразу после использования.

Вторая стратегия предполагает использование *новой тестовой конфигурации* (Fresh Fixture), которая по тем или иным причинам существует дольше, чем работает *тестовый метод* (Test Method). Для защиты конфигурации от превращения в *общую тестовую конфигурацию* (Shared Fixture, с. 350) такие *постоянные новые тестовые конфигурации* (Persistent Fresh Fixture) нуждаются в коде, предназначенному для явного удаления конфигурации после завершения тестов. Данное требование является причиной появления различных шаблонов очистки тестовой конфигурации.

Третья стратегия подразумевает намеренное повторное использование постоянных тестовых конфигураций. Такая стратегия на основе *общей тестовой конфигурации* (Shared Fixture) часто применяется для ускорения работы тестов, основанных на *постоянной новой тестовой конфигурации* (Persistent Fresh Fixture), но вместе с этим приходится мириться с определенными накладными расходами. Такие тесты требуют использования одного из шаблонов создания и очистки тестовой конфигурации. Кроме того, тесты начинают взаимодействовать друг с другом (намеренно или случайно), что приводит к появлению *нестабильных тестов* (Erratic Test, с. 267) и *высокой стоимости обслуживания тестов* (High Test Maintenance Cost).

В табл. 6.1 показаны накладные расходы на управление тестовыми конфигурациями в соответствии с применяемой стратегией.

Таблица 6.1. Сводная таблица требований к настройке и очистке тестовой конфигурации для различных стратегий управления тестовыми конфигурациями

Код настройки	Код очистки	Триггер настройки/очистки
Временная новая тестовая конфигурация (Transient Fresh Fixture)	Да	
Постоянная новая тестовая конфигурация (Persistent Fresh Fixture)	Да	Да
Общая тестовая конфигурация (Shared Fixture)	Да	Да

Примечание: в строке “Общая тестовая конфигурация (Shared Fixture)” предполагается, что для каждого теста создается новая тестовая конфигурация, а не используется предварительно созданная тестовая конфигурация (Prebuilt Fixture, с. 454)

На рис. 6.4 показано взаимодействие между целями, повторным использованием тестовой конфигурации и временем жизни тестовой конфигурации.

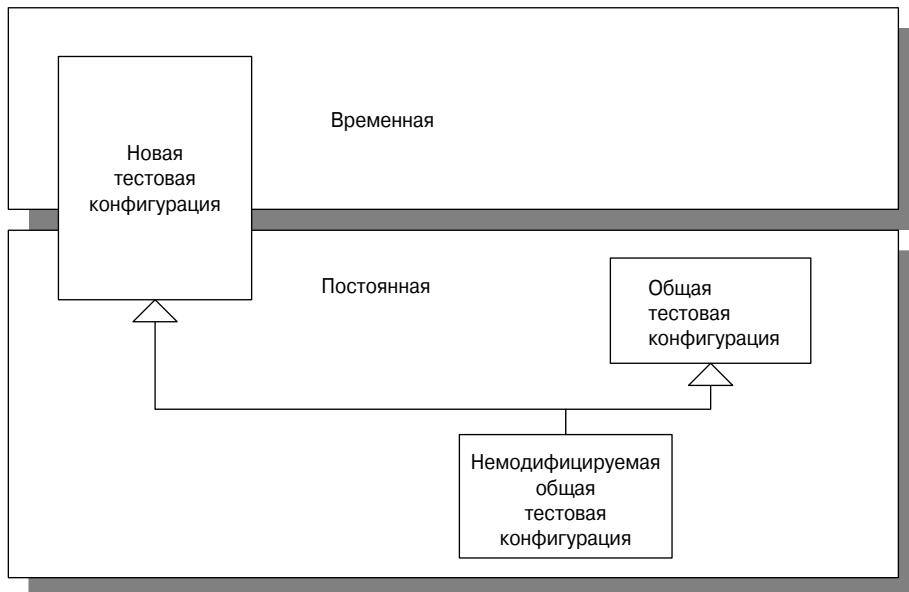


Рис. 6.4. Сводная информация о стратегиях управления тестовыми конфигурациями. Новая тестовая конфигурация (Fresh Fixture) может быть как временной, так и постоянной. Общая тестовая конфигурация (Shared Fixture) должна быть постоянной. Немодифицируемая общая тестовая конфигурация (Immutable Shared Fixture; см. Общая тестовая конфигурация, Shared Fixture) не должна модифицироваться ни одним тестом. В результате большинство тестов дополняют общую тестовую конфигурацию (Shared Fixture) модифицируемой новой тестовой конфигурацией (Fresh Fixture)

Для обеих комбинаций отношение между постоянностью и новизной кажется очевидным. Постоянная новая тестовая конфигурация (Fresh Fixture) более подробно рассматривается далее в этой главе. Временная общая тестовая конфигурация (Shared Fixture) является временной по своей природе — способ хранения ссылок на конфигурации делает их постоянными. Кроме того, временная и постоянная общие тестовые конфигурации должны рассматриваться как одинаковые.

Временная новая тестовая конфигурация

При таком подходе каждый тест создает временную *новую тестовую конфигурацию* (Fresh Fixture) в процессе работы. Любой необходимый объект или запись создается самим тестом (не обязательно внутри *тестового метода*, Test Method). Так как видимость тестовой конфигурации ограничена конкретным тестом, обеспечивается полная независимость отдельных тестов (ни один тест не будет зависеть от результата работы других тестов, использующих такую же тестовую конфигурацию).

Такой подход называется *новая тестовая конфигурация* (Fresh Fixture), поскольку каждый тест начинает работу с чистого листа. Никакая часть других тестовых конфигураций или *предварительно созданных тестовых конфигураций* (Prebuilt Fixture) не “наследуется” и не “используется повторно”. Каждый объект или запись в тестируемой системе является “свежим”, “полностью новым” и “не использованным ранее”.

Основным недостатком подхода на основе *новой тестовой конфигурации* (Fresh Fixture) является дополнительная вычислительная мощность, необходимая для создания всех объектов для каждого теста. Как следствие, тесты могут работать медленнее, чем при использовании *общей тестовой конфигурации* (Shared Fixture), особенно если применяется *постоянная новая тестовая конфигурация* (Persistent Fresh Fixture).

Постоянная новая тестовая конфигурация

Данный термин кажется оксюмороном, не так ли? В данном случае требуется, чтобы тестовая конфигурация была постоянно новой и при этом существовала дольше, чем работает отдельный тест. Что это за стратегия? Кому-то такая стратегия может показаться “глупой”, но у нее есть свои применения.

Разработчики вынуждены прибегать к данной стратегии при тестировании компонентов, тесно связанных с базой данных или другим механизмом постоянного хранения. Очевидным решением является отказ от тесной связности, но еще проще превратить базу данных в *заменяемую зависимость* (substitutable dependency) тестируемого компонента. Этот шаг может показаться не слишком практичным при тестировании *унаследованного программного обеспечения* (legacy software), но при этом могут потребоваться преимущества *новой тестовой конфигурации* (Fresh Fixture). Именно в таких обстоятельствах оправдана стратегия на основе *постоянной новой тестовой конфигурации* (Persistent Fresh Fixture). Ключевым отличием от стратегии на основе *временной новой тестовой конфигурации* (Transient Fresh Fixture) является требование к существованию кода, который очищает тестовую конфигурацию после каждого теста. Если в качестве механизма хранения используется база данных, файловая система или другой компонент с большими задержками, *постоянная новая тестовая конфигурация* (Persistent Fresh Fixture) может стать причиной возникновения *медленных тестов* (Slow Test).

Проблему *медленных тестов* (Slow Test) можно хотя бы частично решить с помощью одного или нескольких следующих шаблонов.

1. Создать *минимальную тестовую конфигурацию* (Minimal Fixture), насколько это возможно.
2. Ускорить создание через замену поставщика данных *тестовым двойником* (Test Double).
3. Если тесты все еще работают недостаточно быстро, уменьшить объем тестовой конфигурации, которая должна создаваться и очищаться при каждом запуске. Для этого не модифицируемые объекты должны быть вынесены в *немодифицируемую общую тестовую конфигурацию* (Immutable Shared Fixture).

Группы разработчиков, с которыми приходилось общаться автору, обнаружили, что тесты работают в 50 раз быстрее (время работы действительно составило 2% от исходного), если с помощью *вставки зависимости* (Dependency Injection, с. 684) или *поиска зависимости* (Dependency Lookup, с. 692) вся база данных заменялась *поддельной базой данных* (Fake Database; см. *Поддельный объект*, Fake Object, с. 565), основанной на хэш-таблицах, а не на простых таблицах. Каждый тест может потребовать множества операций с базами данных для настройки и очистки тестовой конфигурации для единственного запроса к тестируемой системе.

Многое можно сказать о минимизации размера и сложности тестовой конфигурации. Намного проще понять *минимальную тестовую конфигурацию* (Minimal Fixture) и оценить отношение “причина — следствие” между тестовой конфигурацией и ожидаемым результатом. Это основной фактор, позволяющий использовать *тесты как документацию* (Tests as Documentation, с. 79). В некоторых случаях тестовую конфигурацию можно сделать намного меньшей за счет *обрезания цепочки сущностей* (Entity Chain Snipping; см. *Тестовая заглушка*, Test Stub, с. 544) для отказа от создания объектов, от которых тест зависит опосредованно. Эта тактика позволяет значительно ускорить создание экземпляра тестовой конфигурации.

Стратегии на основе общей тестовой конфигурации

Иногда невозможно (или нерационально) использовать *новую тестовую конфигурацию* (Fresh Fixture). При таком подходе несколько тестов используют один и тот же экземпляр тестовой конфигурации.

Основным преимуществом *общей тестовой конфигурации* (Shared Fixture) является экономия времени на настройке и очистке конфигурации. Основной недостаток описывается одним из псевдонимов, *устаревшая тестовая конфигурация* (Stale Fixture), и запахом теста, который описывает наиболее распространенный побочный эффект: *взаимодействующие тесты* (Interacting Tests; см. *Нестабильный тест*, Erratic Test). Хотя общая тестовая конфигурация имеет и другие положительные стороны, большинство из них можно получить через применение других шаблонов к новой тестовой конфигурации (Fresh Fixture). *Стандартная тестовая конфигурация* (Standard Fixture, с. 338) позволяет избежать проектирования и написания кода для каждого теста, при этом не требуя совместного использования тестовой конфигурации.

Если *общая тестовая конфигурация* (Shared Fixture) такая плохая, почему она здесь рассматривается? Потому что каждый разработчик как минимум один раз пошел по этому пути (эта информация предназначена для разработчиков, оказавшихся в такой ситуации). Обратите внимание, что наше обсуждение ни в коем случае не должно рассматриваться, как рекомендация к использованию данного подхода. Этот путь усыпан битым стеклом, кишит ядовитыми змеями и... Ну, в общем, мысль должна быть понятна.

Какие возможны варианты, учитывая, что было принято решение использовать *общую тестовую конфигурацию* (Shared Fixture), ведь были рассмотрены все возможные альтернативы, правда? Необходимо принять еще несколько решений (рис. 6.5).

- Насколько широко применяется общая тестовая конфигурация (например, *класс теста* (Testcase Class), все тесты в наборе, все тесты одного пользователя)?
- Как часто создается новый экземпляр тестовой конфигурации?

Чем больше тестов пользуются тестовой конфигурацией, тем выше вероятность, что один из них что-то серьезно повредит и нарушит нормальные условия запуска последующих тестов. Чем реже создается тестовая конфигурация, тем дольше будут сохраняться побочные эффекты при ее повреждении. Например, *предварительно созданная тестовая конфигурация* (Prebuilt Fixture) может быть создана за пределами теста, что позволит вынести стоимость генерации тестовой конфигурации из стоимости запуска теста. К сожалению, это приведет к появлению *неповторяемого теста* (Unrepeatable Test), который не очищает результаты своей работы. Чаще всего такая стратегия применяется совместно с “*песочницей*” с базой данных (Database Sandbox, с. 658), инициализация которой проис-

ходит с помощью сценария в базе данных. Если *общая тестовая конфигурация* (Shared Fixture) доступна более чем одной *программе запуска тестов* (Test Runner, с. 405), может начаться “война” запуска тестов (Test Run War; см. *Нестабильный тест*, Erratic Test), когда тесты случайным образом завершаются неудачно при одновременном использовании одной и той же тестовой конфигурации несколькими тестами.

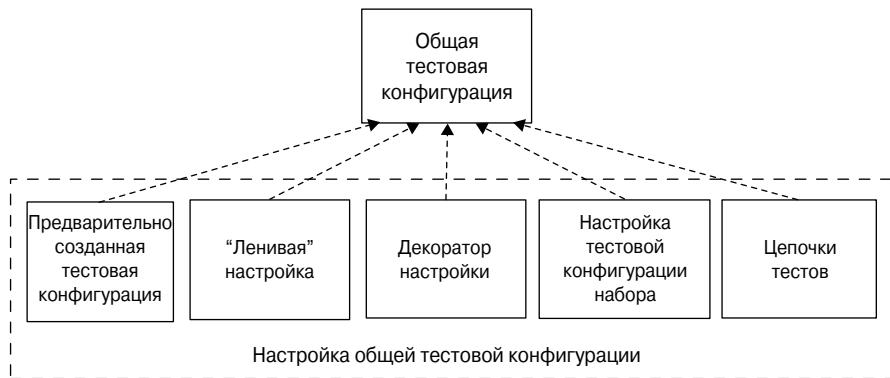


Рис. 6.5. Способы управления общей тестовой конфигурацией. Стратегии упорядочены по времени жизни тестовой конфигурации. Тестовая конфигурация с максимальным временем жизни показана слева

Во избежание *неповторяемых тестов* (Unrepeatable Test) и “войн” запуска тестов (Test Run War) можно создавать тестовую конфигурацию при каждом запуске набора тестов. Инфраструктура xUnit обеспечивает несколько выходов из такой ситуации, включая “ленивую” настройку (Lazy Setup, с. 460), настройку тестовой конфигурации набора (Suite Fixture Setup, с. 465) и декоратор настройки (Setup Decorator, с. 471). Концепция “ленивой инициализации” должна быть знакома большинству разработчиков объектно-ориентированных систем. В данном случае к тестовой конфигурации просто применяется идея конструктора. Два последних варианта обеспечивают возможность очистки тестовой конфигурации после завершения работы теста. Это делается за счет вызова метода `setUp` и соответствующего ему метода `tearDown`. “Ленивая” настройка (Lazy Setup) такой возможности не обеспечивает.

Цепочки тестов (Chained Tests) являются еще одним способом создания *общей тестовой конфигурации* (Shared Fixture) за счет запуска тестов в строго определенной последовательности и применения результата выполнения предыдущего теста в качестве тестовой конфигурации для следующего. К сожалению, если один тест завершится неудачно, все последующие тесты могут показать некорректный результат, так как их предварительные условия не были выполнены. Проблему можно обойти с помощью *сторожевых утверждений* (Guard Assertion, с. 510), проверяющих предварительные условия запуска тестов⁴.

Как было показано выше, *немодифицируемая общая тестовая конфигурация* (Immutable Shared Fixture) является стратегией для ускорения работы тестов, использующих *новую*

⁴ К сожалению, это может привести к замедлению работы тестов, если тестовая конфигурация находится в базе данных. Но такое решение будет намного более быстрым, чем вставка необходимых записей в базу данных для каждого теста в отдельности.

тестовую конфигурацию (Fresh Fixture). Кроме того, немодифицируемая общая тестовая конфигурация позволяет снизить вероятность ошибок в тестах на основе *общей тестовой конфигурации* (Shared Fixture) за счет локализации изменений в небольшой части тестовой конфигурации.

Обеспечение простоты тестирования и взаимодействия с тестируемой системой

Последним в этой главе стратегическим вопросом является обеспечение простоты тестирования. Обсуждение здесь не является исчерпывающим — данная тема слишком масштабна для одной главы о стратегии тестирования. Несмотря на это не стоит откладывать вопрос в долгий ящик, так как он очень влияет на автоматизацию тестирования в целом. Но сначала необходимо обратить внимание на сам процесс разработки.

Тестируйте после, но не говорите, что вас не предупреждали

Каждый, кому доводилось интегрировать модульные тесты в существующее приложение, сталкивался с множеством проблем! Это самая сложная ситуация при автоматизации тестов, имеющая к тому же самую низкую продуктивность. Большая часть преимуществ автоматизированных тестов проявляется на этапе отладки программного обеспечения, когда тесты значительно сокращают время работы с отладочными инструментами. Нельзя порекомендовать интеграцию модульных тестов в уже существующее программное обеспечение как способ знакомства с этой технологией, поскольку после такого опыта даже самые целеустремленные разработчики и руководители проектов отказываются от идеи применения тестов вообще.

Готовый проект с учетом тестов — прыжок выше собственной головы

Сделать готовый проект с учетом тестов очень сложно, так как заранее неизвестно, какие тесты потребуются в контрольных точках и точках наблюдения тестируемой системы. Очень легко написать сложный в тестировании программный продукт. Точно так можно потратить много времени на проектирование механизмов тестирования, которые окажутся недостаточными или невостребованными. В любом случае много усилий будет потрачено без видимого результата.

Возможность тестирования, обеспеченная тестами

Хорошей особенностью программного обеспечения, написанного на основе тестов, является отсутствие необходимости специально проектировать с учетом тестов; тесты просто пишутся и вынуждают писать сам продукт с учетом уже написанных тестов. Факт создания теста определяет контрольные точки и точки наблюдения, которые должна обеспечить тестируемая система. После успешного завершения всех тестов можно считать, что продукт проектировался с учетом тестирования.

На этом реклама *разработки на основе тестов* (Test-Driven Development) как процесса проектирования с учетом тестов завершена и можно переходить к обсуждению методик обеспечения тестируемости программного продукта.

Контрольные точки и точки наблюдения

Тест взаимодействует с программным продуктом⁵ через один или несколько интерфейсов или **точки взаимодействия** (interaction points). С точки зрения теста точки взаимодействия могут играть роль как контрольных точек, так и точек наблюдения (рис. 6.6).

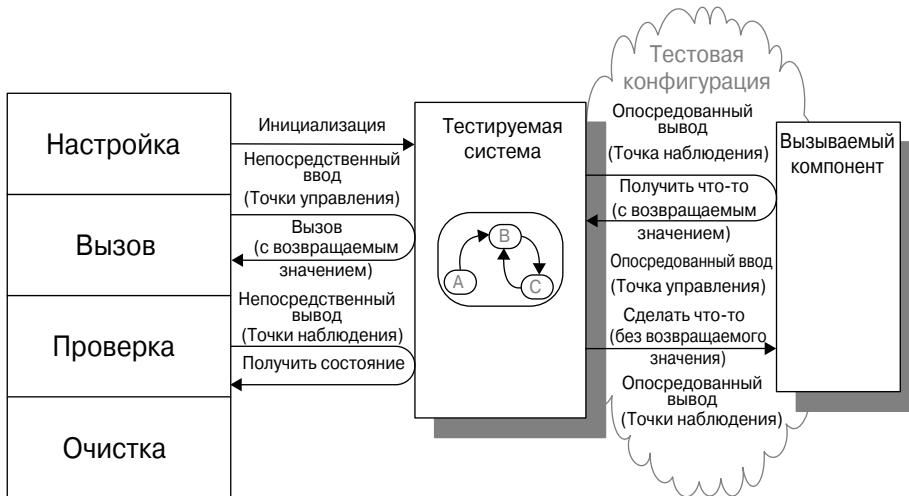


Рис. 6.6. Контрольные точки и точки наблюдения. Тест взаимодействует с тестируемой системой через точки взаимодействия. Точки непосредственного взаимодействия представляют собой синхронные вызовы методов со стороны теста; опосредованные точки взаимодействия требуют того или иного типа манипуляции через “черный ход” (Back Door Manipulation). Контрольные точки обозначаются стрелками, указывающими в сторону тестируемой системы. Точки наблюдения обозначаются стрелками, указывающими в сторону от тестируемой системы

Контрольная точка позволяет тесту запросить действие у программного продукта. Это может потребоваться для перевода продукта в конкретное состояние в процессе настройки или очистки тестовой конфигурации, а также для вызова тестируемой системы. Некоторые контрольные точки существуют только для тестов. Они не должны использоваться кодом продукта, так как пропускают проверку ввода или нарушают нормальный жизненный цикл тестируемой системы или объектов, от которых она зависит.

Точка наблюдения позволяет тесту получать информацию о поведении тестируемой системы на этапе проверки результатов. Точки наблюдения могут использоваться для получения состояния тестируемой системы или вызываемого компонента после запуска теста. Кроме того, они могут использоваться для взаимодействия между тестируемой системой и компонентами, с которыми она должна взаимодействовать в результате вызова. Проверка опосредованного вывода является примером *проверки через “черный ход”* (Back Door Verification; см. *Манипуляция через “черный ход”*, Back Door Manipulation).

⁵ Здесь намеренно не упоминается “тестируемая система”, так как взаимодействие происходит не только с тестируемой системой.

И контрольные точки, и точка наблюдения могут предоставляться тестируемой системой в виде синхронных вызовов методов. Это называется “использование главного входа”. Некоторые точки взаимодействия могут использовать “черный ход” к тестируемой системе. Это так называемая *манипуляция через “черный ход”* (Back Door Manipulation). На диаграммах контрольные точки обозначаются стрелками, направленными к тестируемой системе (как от тестов, так и от вызываемых компонентов). Точки наблюдения представлены стрелками, указывающими на сам тест. Обычно источником таких стрелок является тестируемая система или вызываемый компонент⁶. Кроме того, они могут начинаться с теста, взаимодействовать с тестируемой системой или вызываемым компонентом и возвращаться к тесту⁷.

Стили взаимодействия и шаблоны тестирования

При тестировании определенного фрагмента программного обеспечения тесты могут принимать одну из двух базовых форм.

Обычный тест взаимодействует с тестируемой системой только через открытый интерфейс, т.е. через “главный вход” (рис. 6.7). И контрольные точки, и точки наблюдения для такого теста представляют собой простые вызовы методов. Положительной стороной этого подхода является сохранение принципа инкапсуляции. Тесту достаточно знать только открытый интерфейс программного продукта и не требуется информация о внутреннем устройстве.

Основной альтернативой является *тест с пересечением уровней* (Layer-Crossing Test) (рис. 6.8), в котором тестируемая система вызывается через программный интерфейс, а за происходящим на “черном ход” наблюдает *тестовый двойник* (Test Double) одного из типов, например *тестовый агент* (Test Spy, с. 552) или *подставной объект* (Mock Object, с. 558). Это очень мощная техника тестирования для проверки некоторых типов архитектурных требований. К сожалению, при злоупотреблении такой подход может привести к появлению *зарегулированной программы* (Overspecified Software; см. “Хрупкий” тест, Fragile Test), так как изменения в реализации требований могут привести к неудачному завершению теста.

На рис. 6.8 тест справа использует замещающий вызываемый компонент *подставной объект* (Mock Object) в качестве точки наблюдения. Тест слева использует как контрольную точку *тестовую заглушку* (Test Stub), вставленную вместо вызываемого объекта. Тесты такого формата предполагают использование многоуровневой архитектуры, которая, в свою очередь, позволяет применять *тест уровня* (Layer Test, с. 368) для тестирования каждого уровня архитектуры в отдельности (рис. 6.9). Еще более общей концепцией является использование *теста компонента* (Component Test; см. Тест уровня, Layer Test) для изолированного тестирования каждого компонента в пределах уровня.

⁶ Асинхронная точка наблюдения.

⁷ Синхронная точка наблюдения.

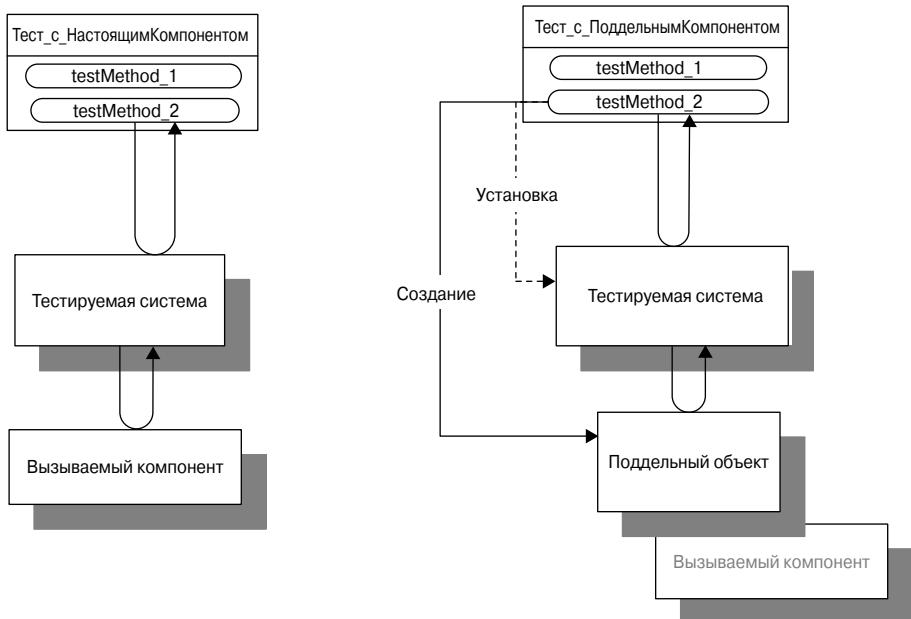


Рис. 6.7. Обычный тест взаимодействует с тестируемой системой только через “главный вход”. Тест справа замещает вызываемый компонент поддельным объектом (*Fake Object*) для обеспечения повторяемости и повышения быстродействия

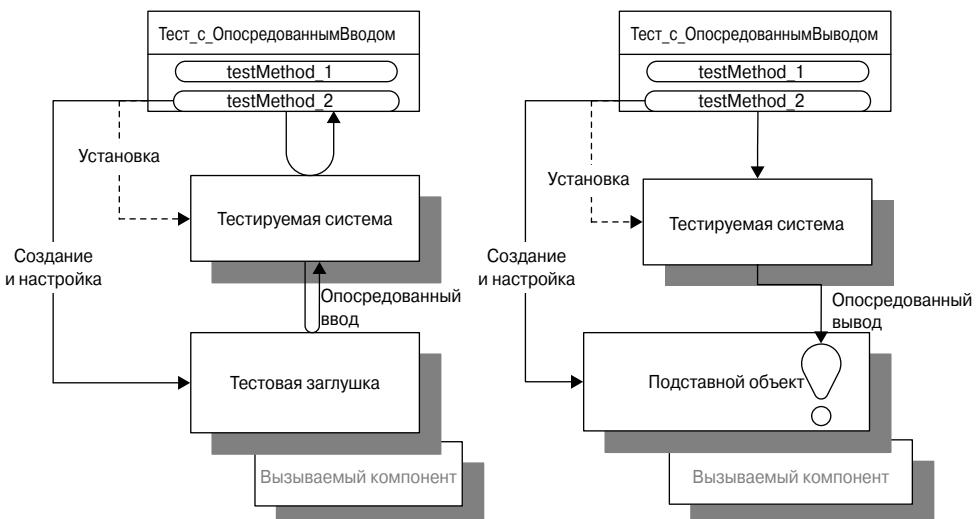


Рис. 6.8. Тест с пересечением уровней (*Layer-Crossing Test*) может взаимодействовать с тестируемой системой через “черный ход”. Тест слева управляет опосредованным вводом тестируемой системы с помощью тестовой заглушки (*Test Stub*). Тест справа проверяет опосредованный вывод с помощью подставного объекта (*Mock Object*)

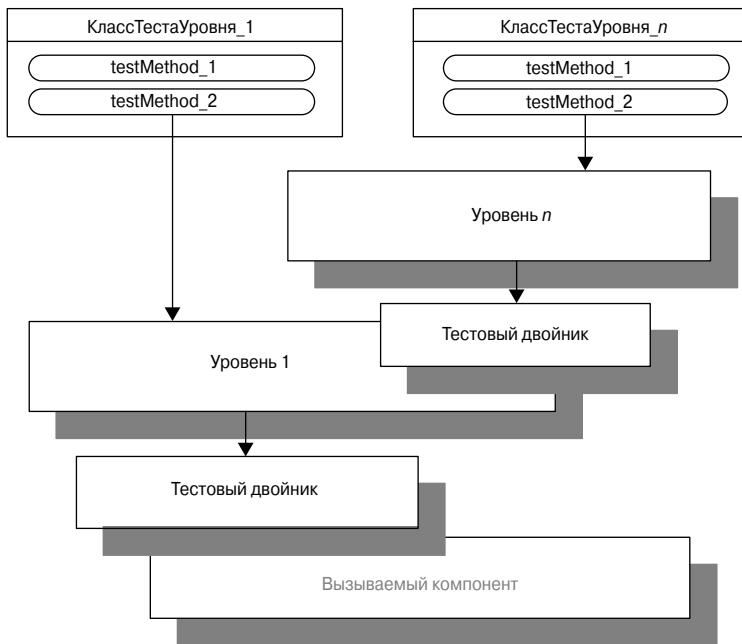


Рис. 6.9. Пара тестов уровня (Layer Test) проверяет разные уровни системы. Каждый уровень многоуровневой архитектуры можно тестировать независимо с помощью отдельного набора тестов. В результате обеспечиваются независимое тестирование различных аспектов и усиление разделения на уровне средствами тестов

При каждом использовании тестов с пересечением уровней необходимо обеспечить механизм заменяемой зависимости для каждого компонента, который вызывается тестируемой системой, но должен тестироваться независимо. Основными претендентами в данном случае являются любые варианты *вставки зависимости* (Dependency Injection; рис. 6.10) или некоторые формы *поиска зависимости* (Dependency Lookup), например *фабрика объектов* (Object Factory) и *локатор служб* (Service Locator). Такие механизмы подстановки зависимостей можно разработать самостоятельно, а можно воспользоваться инфраструктурой *инвертирования управления* (inversion control — IOC), если таковая доступна в используемой среде разработки. Запасным вариантом можно считать применение *связанного с тестом подкласса* (Test-Specific Subclass, с. 591) тестируемой системы или вызываемого компонента. Такой подкласс может применяться для переопределения доступа к зависимости или механизма создания внутри тестируемой системы, а также для замены поведения вызываемого компонента поведением, нужным тесту.

Решением “на крайний случай” является *ловушка для теста* (Test Hook, с. 713)⁸. Такие конструкции имеют право на жизнь как временные меры, позволяющие автоматизировать тесты для получения *страховочной сети* (Safety Net, с. 79) в качестве рефакторинга при внедрении тестов в существующий продукт. Но ни в коем случае не вырабатывайте привычку к использованию ловушек, так как длительное использование *ловушек для теста* (Test Hook) приведет к появлению *логики теста в продукте* (Test Logic in Production, с. 257).

⁸ Обычно ловушка имеет вид `if (testing) then ... else ... endif.`

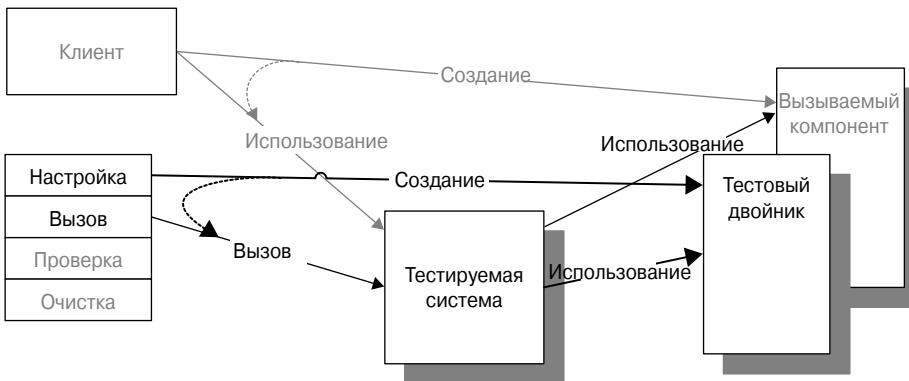


Рис. 6.10. Тест вставляет тестовый двойник (Test Double). Для замены вызываемого компонента подходящим тестовым двойником (Test Double) может использоваться вставка зависимости (Dependency Injection). Вызываемый компонент передается тестируемой системе во время или после создания

Третьим заслуживающим упоминания тестом является **асинхронный тест** (asynchronous test), который взаимодействует с тестируемой системой через настоящие сообщения. Поскольку ответы на запросы также поступают асинхронно, эти тесты должны содержать механизм межпроцессной синхронизации, например вызовы функции `wait`. К сожалению, необходимость ожидания ответа, который может никогда и не прийти, приводит к значительному увеличению времени работы тестов. Такого стиля тестирования необходимо избегать любой ценой при написании модульных тестов и тестов компонентов.

К счастью, шаблон **минимальный выполняемый файл** (Humble Executable; см. *Минимальный объект*, Humble Object, с. 700) позволяет избежать запуска модульных тестов подобным образом (рис. 6.11). При этом логика обработки входящих сообщений выносится в отдельный класс или компонент, который уже можно тестировать синхронно.

Смежной темой является тестирование бизнес-логики через пользовательский интерфейс. Обычно такое *опосредованное тестирование* (Indirect Testing; см. *Непонятный тест*, Obscure Test) является нежелательным, так как модификация пользовательского интерфейса нарушает работу тестов, пытающихся через него проверять работу бизнес-логики. Поскольку пользовательский интерфейс меняется достаточно часто (особенно в проектах с гибкой методологией разработки), такая стратегия значительно увеличивает стоимость обслуживания тестов. Врожденная асинхронность пользовательского интерфейса является еще одной причиной для того, чтобы не использовать подобный подход к тестированию. Тесты, взаимодействующие с системой через пользовательский интерфейс, должны быть асинхронными, а значит, получают все проблемы, связанные с тестами данного типа.

Разделяй и тестируй

Практически любой *сложный в тестировании код* (Hard-to-Test Code, с. 251) можно упростить для тестирования через рефакторинг, если имеющихся тестов достаточно для защиты от ошибок, вносимых в процессе такого рефакторинга.

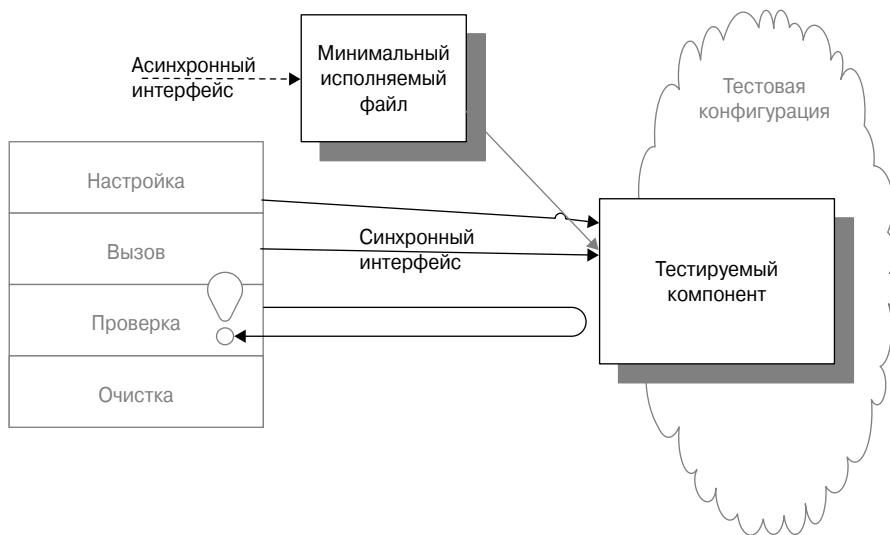


Рис. 6.11. Используя минимальный выполнимый файл (*Humble Executable*), можно упростить тесты. Данный шаблон улучшает повторяемость и скорость проверки логики, которую в противном случае пришлось бы проверять асинхронными тестами

Чтобы избежать обращения к пользовательскому интерфейсу в приемочных тестах, можно создать “подкожный” тест (Subcutaneous Test; см. *Тест уровня*, Layer Test). При этом, минуя уровень пользовательского интерфейса, тесты обращаются к бизнес-логике через *фасад служб* (Service Facade) [CJ2EPP], который предоставляет доступ к необходимым синхронным точкам взаимодействия. Пользовательский интерфейс зависит от того же фасада, позволяя проверить правильность бизнес-логики еще до подключения логики пользовательского интерфейса. Кроме того, многоуровневая архитектура позволяет тестировать пользовательский интерфейс до завершения работы над бизнес-логикой. В такой ситуации *фасад служб* (Service Facade) заменяется *тестовым двойником* (Test Double), обеспечивающий полностью детерминированное поведение, от которого зависят тесты⁹.

При модульном тестировании нетривиальных пользовательских интерфейсов¹⁰ можно воспользоваться *минимальным диалогом* (Humble Dialog; см. *Минимальный объект*, Humble Object) для выноса логики принятия решений в пользовательском интерфейсе из визуального уровня (который сложно тестировать синхронно) в уровень объектов поддержки, которые хорошо поддаются стандартным техникам модульного тестирования (рис. 6.12). Такой подход обеспечивает тщательное (на сопоставимом с бизнес-логикой уровне) тестирование поведения презентационной логики.

⁹ Тестовый двойник (Test Double) может быть жестко закодирован, а может управляться содержимым соответствующих файлов. В любом случае он не должен зависеть от настоящей реализации, чтобы тесты пользовательского интерфейса знали только данные для запуска конкретного поведения фасада служб, а не логику, реализующую данное поведение.

¹⁰ Любой пользовательский интерфейс, хранящий информацию о состоянии, поддерживающий условный вывод или разрешающий включение/отключение элементов, должен рассматриваться как нетривиальный.

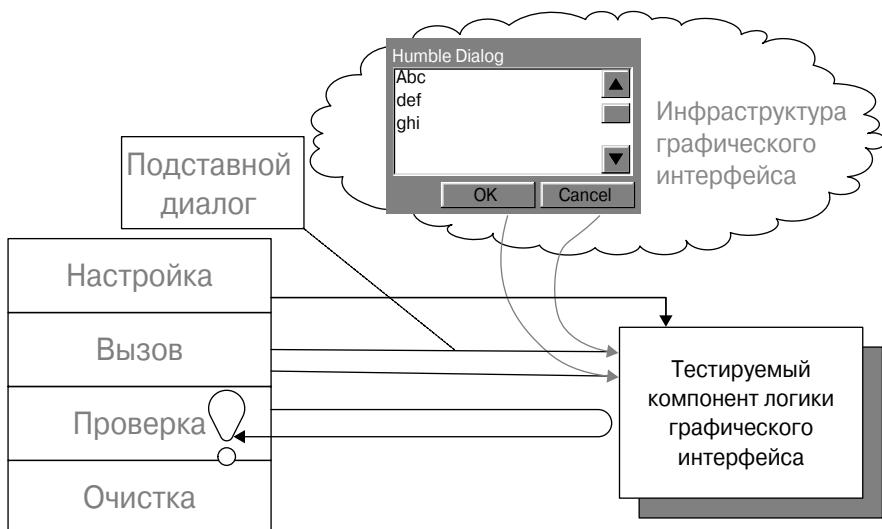


Рис. 6.12. Минимальный диалог (Humble Dialog) снижает зависимость теста от инфраструктуры пользовательского интерфейса. Логика, которая управляет состоянием компонентов пользовательского интерфейса, плохо поддается тестированию. После ее выделения в отдельный тестируемый компонент формируется минимальный диалог (Humble Dialog), практически не требующий тестирования

С точки зрения стратегии автоматизации тестирования ключевыми являются решения о стилях взаимодействия с тестируемой системой, которые следует использовать и которых следует избегать. Кроме того, важно обеспечить проектирование программного продукта в соответствии с принятыми решениями.

Что дальше

На этом обсуждение сложных в изменении решений по стратегии автоматизации тестирования завершается. Если вы дочитали до этого места, можно предположить, что инфраструктура xUnit выбрана вами как подходящий инструмент для автоматизации тестирования. В следующих главах подробно рассматриваются шаблоны реализации выбранной стратегии работы с тестовыми конфигурациями как на основе *новой тестовой конфигурации* (Fresh Fixture), так и на основе *общей тестовой конфигурации* (Shared Fixture). Сначала в главе 8, “Управление временной тестовой конфигурацией”, будет рассмотрен самый простой вариант — *временная новая тестовая конфигурация* (Transient Fresh Fixture). После этого в главе 9, “Управление постоянными тестовыми конфигурациями”, будет показано использование постоянных тестовых конфигураций. Но сначала в главе 7, “Основы xUnit”, будут описаны базовая терминология и нотация xUnit, которые используются на протяжении всей книги.

Глава 7

Основы xUnit

О чём идет речь в этой главе

В главе 6, “Стратегия автоматизации тестирования”, рассматривались “сложные в изменении” решения, которые приходится принимать на ранних этапах работы над проектом. В данной главе преследуются две цели. Во-первых, описываются терминология xUnit и формат диаграмм, которые применяются на протяжении всей книги. Во-вторых, рассматриваются внутренняя работа инфраструктуры xUnit и причины именно такой реализации поведения. Эта информация может пригодиться при создании новой *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332), являющейся адаптацией инфраструктуры xUnit. Также эта информация поможет авторам тестов понять, как использовать некоторые функции xUnit.

Введение в xUnit

Под термином “xUnit” подразумевается любой член семейства *инфраструктур автоматизации тестов* (Test Automation Framework), применяемых для автоматизации *созданных вручную сценариев тестов* (Hand-Scripted Test; см. *Тест на основе сценария*, Scripted Test, с. 319) и реализующих описанные здесь функции. Для большинства современных распространенных языков программирования существует как минимум одна реализация xUnit. Обычно для автоматизации тестов применяется тот же язык, который использовался для написания тестируемой системы. Хотя это не всегда так, использовать подобную стратегию обычно проще, поскольку тесты легко получают доступ к программному интерфейсу тестируемой системы. Используя знакомый язык программирования, можно с меньшими трудозатратами изучить подходы к автоматизации *полностью автоматизированных тестов* (Fully Automated Tests, с. 81)¹.

¹ См. врезку “Тестирование хранимых процедур с помощью JUnit” на с. 664, в которой приведен пример использования инфраструктуры на одном языке для работы с тестируемой системой, написанной на другом языке.

Общие функции

Учитывая, что большинство членов xUnit реализованы с использованием объектно-ориентированной парадигмы, именно они рассматриваются в первую очередь. После этого рассматриваются отличия, характерные для реализаций на основе других парадигм.

Во всех реализациях xUnit предоставляется базовый набор функций, которые позволяют решать следующие задачи:

- описывать тест как *тестовый метод* (Test Method, с. 378);
- описывать ожидаемые результаты внутри тестового метода в форме вызовов *методов с утверждением* (Assertion Method, с. 390);
- агрегировать тесты в наборы, которые могут запускаться с помощью одной команды;
- запускать один или несколько тестов для получения отчета о результатах запуска.

Поскольку во многих реализациях xUnit поддерживается *обнаружение тестовых методов* (Test Method Discovery; см. *Обнаружение тестов*, Test Discovery, с. 420), в них не используется *перечисление тестов* (Test Enumeration, с. 425) для добавления вручную каждого интересующего *тестового метода* (Test Method) в набор. В некоторых реализациях поддерживается тот или иной механизм *выбора тестов* (Test Selection, с. 429), позволяющий запускать подмножество тестов, основанное на каком-либо критерии.

Базовый минимум

Для понимания принципов работы инфраструктуры xUnit (рис. 7.1) как минимум необходимо знать следующее.

- Как определять тесты с помощью *тестовых методов* (Test Method) и *классов теста* (Testcase Class, с. 401)?
- Как создавать произвольные *наборы наборов* (Suite of Suites; см. *Объект набора тестов*, Test Suite Object, с. 414)²?
- Как запускать тесты?
- Как интерпретировать результат работы тестов?

Определение тестов

Каждому тесту соответствует *тестовый метод* (Test Method), который реализует *четырехфазный тест* (Four-Phase Test, с. 387) в соответствии с приведенной ниже последовательностью.

- Настройка тестовой конфигурации с помощью *встроенной настройки* (In-line Setup, с. 433), *делегированной настройки* (Delegated Setup, с. 437) или *неявной настройки* (Implicit Setup, с. 449).
- Вызов тестируемой системы через методы открытого или закрытого интерфейсов.

² Даже в вариантах xUnit, не имеющих явного класса или метода Suite, *объект набора тестов* (Test Suite Object) создается “за кулисами”.

- Проверка получения ожидаемого результата через *методы с утверждением* (Assertion Method).
- Удаление тестовой конфигурации с помощью *очистки со сборкой мусора* (Garbage-Collected Teardown, с. 518), *встроенной очистки* (In-line Teardown, с. 527), *неявной очистки* (Implicit Teardown, с. 533) или *автоматической очистки* (Automated Teardown, с. 521).

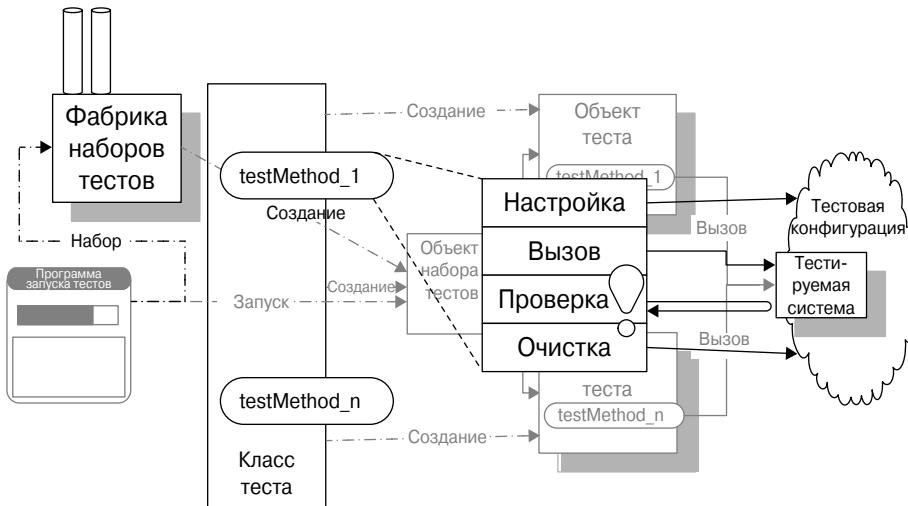


Рис. 7.1. Статическая структура теста с точки зрения разработчика. Разработчик теста видит только статическую структуру в процессе написания или чтения теста. Разработчик пишет один тестовый метод (Test Method) с разделением на четыре фазы для каждого теста в пределах класса теста (Testcase Class). Фабрика наборов тестов (Test Suite Factory; см. Перечисление тестов, Test Enumeration) используется только для перечисления тестов. Структуру во время выполнения (выделена серым цветом) разработчик может увидеть только в своем воображении

Самыми распространенными типами тестов являются *простой тест успешности* (Simple Success Test; см. *Тестовый метод*, Test Method), который проверяет правильность поведения тестируемой системы при правильных входных данных, и *тест на ожидаемое исключение* (Expected Exception Test; см. *Тестовый метод*, Test Method), который проверяет генерацию исключения при некорректном использовании тестируемой системы. Тест специального типа *тест конструктора* (Constructor Test; см. *Тестовый метод*, Test Method) проверяет правильность работы конструктора объектов. При использовании этого типа тестов может потребоваться проверка как “простого успеха”, так и “ожидаемого исключения”. *Тестовый метод* (Test Method), содержащий необходимую логику теста, должен где-то существовать, поэтому он определяется как метод *класса теста* (Testcase Class)³. После этого имя *класса теста* (Testcase Class) — или модуля/сборки, в котором он находится

³ В некоторых вариантах xUnit эта схема называется тестовой конфигурацией. Возможно, это следствие предположения, что будет использоваться единственный *класс теста* для каждой *тестовой конфигурации* (Testcase Class per Fixture, с. 639).

ся, — передается *программе запуска тестов* (Test Runner, с. 405). Это может происходить явно (например, при вызове программы запуска тестов из командного интерпретатора) или неявно, средствами интегрированной среды разработки.

Что такое тестовая конфигурация

Тестовая конфигурация (test fixture) включает в себя все необходимое для вызова тестируемой системы. Обычно тестовая конфигурация содержит как минимум экземпляр класса, содержащего тестируемый метод. Кроме того, она может содержать другие объекты, от которых зависит тестируемая система. Обратите внимание, что в некоторых реализациях xUnit под тестовой конфигурацией подразумевается *класс теста* (Testcase Class) — это явно указывает на предположение, что все *тестовые методы* (Test Method) в пределах *класса теста* (Testcase Class) должны использовать одну и ту же тестовую конфигурацию. К сожалению, такой конфликт названий делает обсуждение тестовых конфигураций особенно сложным. В данной книге для *класса теста* (Testcase Class) и создаваемой им тестовой конфигурации используются разные имена. Читателю остается перенести приведенную здесь терминологию на терминологию используемой конкретной реализации xUnit.

Определение наборов тестов

Большинство *программ запуска тестов* (Test Runner) “автомагически” создает набор тестов, состоящий из всех *тестовых методов* (Test Method) в пределах *класса теста* (Testcase Class). Часто этого достаточно. Но иногда необходимо запустить все тесты в пределах приложения. В других ситуациях может потребоваться запуск только подмножества тестов, относящихся к конкретной функциональности. В некоторых реализациях xUnit и утилитах сторонних производителей реализована функция *обнаружения классов теста* (Testcase Class Discovery), с помощью которой *программа запуска тестов* (Test Runner) выявляет наборы тестов в процессе поиска по файловой системе или выполняемым файлам.

Если такая функциональность недоступна, приходится использовать *перечисление наборов тестов* (Test Suite Enumeration; см. *Перечисление тестов*, Test Enumeration), при котором полный набор тестов для всей системы или приложения определяется как агрегат нескольких наборов меньшего объема. Для этого необходимо определить специальный класс *фабрики наборов тестов* (Test Suite Factory), метод suite которого возвращает *объект набора тестов* (Test Suite Object), содержащий коллекцию *тестовых методов* (Test Method) и другие *объекты набора тестов* (Test Suite Object).

Такая коллекция тестов вставляется в *набор наборов* (Suite of Suites), который часто используется для интеграции модульных тестов класса в набор тестов пакета или модуля, который, в свою очередь, включается в набор тестов для всей системы. Такая иерархическая организация обеспечивает запуск наборов различной полноты и позволяет запускать только то подмножество тестов, которое затрагивает разрабатываемый в данный момент фрагмент продукта. Кроме того, разработчик может запустить все существующие тесты одной командой перед включением изменений в хранилище исходного кода [SCMP].

Запуск тестов

Тесты запускаются с помощью *программ запуска тестов* (Test Runner). В большинстве реализаций xUnit существует несколько типов таких программ.

Программа запуска тестов с графическим интерфейсом (Graphical Test Runner; см. *Программа запуска тестов*, Test Runner) обеспечивает визуальный интерфейс для определения набора тестов, его запуска и контроля за его результатами. Одни такие программы позволяют пользователям выбирать тесты через ввод имени в *фабрику наборов тестов* (Test Suite Factory), другие предоставляют *графический обозреватель набора тестов* (Test Tree Explorer; см. *Программа запуска тестов*, Test Runner), который позволяет выбрать конкретные *тестовые методы* (Test Method) из всего дерева наборов тестов. Многие графические программы запуска тестов интегрированы в среды разработки, что сводит запуск тестов к выбору пункта Run as Test в контекстном меню.

Программа запуска тестов для командной строки (Command-Line Test Runner; см. *Программа запуска тестов*, Test Runner) может использоваться для запуска тестов из командного интерпретатора, как показано на рис. 7.2. При этом имя *фабрики наборов тестов* (Test Suite Factory), которая создаст соответствующий набор, передается программе в качестве параметра. Чаще всего такие программы применяются при вызове *программы запуска тестов* (Test Runner) из сценария интеграции [SCMP] или из внутренних процедур интегрированной среды разработки.

```
>ruby testrunner.rb c:/examples/tests/SmellHandlerTest.rb
Loaded suite SmellHandlerTest
Started
.....
Finished in 0.016 seconds.
5 tests, 6 assertions, 0 failures, 0 errors
>Exit code: 0
```

Рис. 7.2. Использование программы запуска тестов для командной строки (Command-Line Test Runner)

Результаты выполнения теста

Основной причиной запуска автоматизированных тестов является получение результатов. Для того чтобы результаты имели смысл, необходим стандартный способ их описания. Обычно в реализациях xUnit соблюдается голливудский принцип (“Не звоните нам, мы позвоним вам сами”). Другими словами, “отсутствие новостей — тоже хорошая новость”. Тесты сами сообщают разработчику о возникшей проблеме. Таким образом, можно основное внимание уделить неудачному завершению работы тестов, а не просматривать список успешных результатов.

Результаты выполнения тестов можно разделить на три категории, каждая из которых обрабатывается по-своему. Если тест завершает работу без ошибок и отказов, он считается успешным. Обычно в xUnit успешные тесты не требуют дополнительной обработки — нет никакой необходимости просматривать вывод при успешном завершении *самопроверяющегося теста* (Self-Checking Test, с. 81).

Тест завершается неудачно, если не выполняется *утверждение* (assertion). Таким образом, тест делает определенное утверждение через *метод с утверждением* (Assertion Method), но утверждение оказывается ложным. В таком случае *метод с утверждением* (Assertion

Method) генерирует исключение (или другую поддерживаемую языком конструкцию). *Инфраструктура автоматизации тестов* (Test Automation Framework) увеличивает значение счетчика неудачных завершений и добавляет подробную информацию об отказе в соответствующий список. Позднее данный список можно будет рассмотреть более подробно. При этом неудачное завершение одного теста не мешает выполнению оставшихся тестов. Такой подход обеспечивает следование принципу *сохранения независимости тестов* (Keep Tests Independent, с. 96).

Считается, что тест содержит ошибку, если тестируемая конфигурация или сам тест неудачно завершает работу непредсказуемым образом. В зависимости от используемого языка проблема может заключаться в неперехваченном исключении, ошибке или в чем-то другом. Как и в случае неудачных завершений через утверждение, *инфраструктура автоматизации тестов* (Test Automation Framework) увеличивает значение счетчика ошибочных тестов и добавляет подробную информацию об ошибке в соответствующий список, который можно просмотреть после завершения работы всех тестов.

Для каждой ошибки или для каждого неудачного завершения теста xUnit сохраняет информацию, которую можно будет проверить для поиска причин. Как минимум указываются имя *тестового метода* (Test Method) и *класса теста* (Testcase Class), а также природа проблемы (ложное утверждение или ошибка в работе программы). В большинстве интегрированных программ запуска *тестов с графическим интерфейсом* (Graphical Test Runner) достаточно щелкнуть на соответствующей строке с информацией для перехода к исходному коду, который стал причиной неудачного завершения или ошибки.

Поскольку ошибка теста кажется более серьезной проблемой по сравнению с неудачным завершением теста, некоторые разработчики стараются перехватить все ошибки тестируемой системы и превратить их в неудачные завершения теста. На самом деле в этом нет необходимости. Что интересно, чаще всего причину ошибки найти намного проще, чем причину неудачного завершения: содержимое стека после ошибки практически сразу показывает на проблемный код внутри тестируемой системы. Содержимое стека при неудачном завершении теста просто указывает на тест, в котором находилось ложное утверждение. При этом имеет смысл использовать *сторожевое утверждение* (Guard Assertion, с. 510) для защиты от запуска кода внутри *тестового метода* (Test Method), который приведет к появлению ошибки в самом *тестовом методе* (Test Method)⁴. Это обычная процедура проверки ожидаемого результата после вызова тестируемой системы, которая не приводит к удалению полезной диагностической информации из содержимого стека.

⁴ Например, перед выполнением утверждения относительно содержимого поля объекта, полученного от тестируемой системы, имеет смысл вызвать метод `assertNotNull` относительно ссылки на объект, чтобы убедиться в отсутствии ошибки “нулевой ссылки”.

Что происходит “под капотом” xUnit

Выше основное внимание уделялось *тестовым методам* (Test Method) и *классам тестов* (Testcase Class) с кратким упоминанием наборов тестов. Такого упрощенного представления с точки зрения “этапа компиляции” достаточно для большинства разработчиков, чтобы приступить к созданию автоматизированных тестов с использованием xUnit. Инфраструктуру xUnit можно использовать и без глубокого понимания принципов работы *инфраструктуры автоматизации тестов* (Test Automation Framework). Но без глубоких знаний разработчик может оказаться в сложном положении при создании и повторном использовании тестовых конфигураций. Таким образом, имеет смысл все-таки изучить, как инфраструктура xUnit запускает *тестовые методы* (Test Method). В большинстве реализаций xUnit каждый *тестовый метод* (Test Method) на этапе выполнения представлен *объектом теста* (Testcase Object, с. 410), так как намного проще управлять тестами, если они являются “полноправными” объектами (рис. 7.3)⁵. *Объекты тестов* (Testcase Object) агрегируются в *объект набора тестов* (Test Suite Object), который может использоваться для запуска нескольких тестов одним действием пользователя.

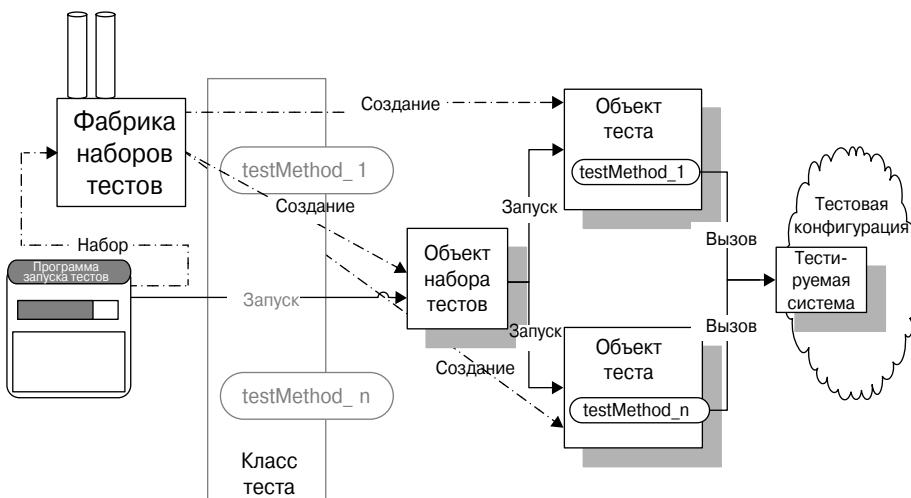


Рис. 7.3. Структура тестов на этапе выполнения с точки зрения инфраструктуры автоматизации тестов (Test Automation Framework). На этапе выполнения программа запуска тестов (Test Runner) запрашивает у класса теста (Testcase Class) или фабрики наборов тестов (Test Suite Factory) создание по одному объекту теста (Testcase Object) для каждого тестового метода (Test Method). При этом все объекты доступны в пределах одного объекта набора тестов (Test Suite Object). Программа запуска тестов (Test Runner) запрашивает у такого объекта (шаблон Composite [GOF]) запуск всех тестов и сбор результатов. Каждый объект теста (Testcase Object) запускает ровно один тестовый метод (Test Method)

⁵ Широко известным исключением является пакет NUnit. Дополнительная информация доступна во врезке “Всегда есть исключения” на с. 411.

Команды тестов

Программа запуска тестов (Test Runner) не может знать, как вызывать каждый *тестовый метод* (Test Method) в отдельности. Для обхода этого недостатка в большинстве реализаций xUnit каждый *тестовый метод* (Test Method) преобразовывается в специальный объект — шаблон *Command* [GOF], имеющий метод `run`. Чтобы создать такой *объект теста* (Testcase Object), *программа запуска тестов* (Test Runner) вызывает метод `suite` класса *теста* (Testcase Class) для получения объекта *набора тестов* (Test Suite Object). Метод `run` конкретного *объекта теста* (Testcase Object) вызывает соответствующий *тестовый метод* (Test Method), для которого он был создан, и возвращает информацию о его успешном или неудачном завершении. Метод `run` *объекта набора тестов* (Test Suite Object) перебирает члены коллекции и отслеживает количество запущенных и неудачно завершившихся тестов.

Объекты наборов тестов

Объект набора тестов (Test Suite Object) реализует шаблон Composite и имеет такой же стандартный интерфейс теста, как и все *объекты теста* (Testcase Object). Этот интерфейс (неявно реализованный в языках, не имеющих конструкции для определения интерфейсов) требует наличия метода `run`. Ожидается, что при вызове метода `run` все тесты, содержащиеся в получателе, будут запущены. *Объект теста* (Testcase Object) сам является “тестом” и запускает соответствующий *тестовый метод* (Test Method). *Объект набора тестов* (Test Suite Object) вызывает методы `run` всех содержащихся в нем *объектов тестов* (Testcase Object). Ценность использования шаблона Composite Command заключается в отсутствии различий между запуском одного теста и набора тестов.

Выше предполагалось, что *объект набора тестов* (Test Suite Object) уже создан и инициализирован. Но откуда он берется? В соответствии с соглашением каждый *класс теста* (Testcase Class) служит в качестве *фабрики наборов тестов* (Test Suite Factory). *Фабрика наборов тестов* (Test Suite Factory) предоставляет *метод класса* (class method), который возвращает *объект набора тестов* (Test Suite Object), содержащий по одному *объекту теста* (Testcase Object) для каждого *тестового метода* (Test Method). В языках с поддержкой механизмов интроспекции реализация xUnit может использовать *обнаружение тестовых методов* (Test Method Discovery) для поиска методов и автоматического создания *объекта набора тестов* (Test Suite Object). В других реализациях от разработчика требуется создание метода `suite` вручную. Подобная методика *перечисления тестов* (Test Enumeration) требует больше труда затрат и с большей вероятностью может привести к *потерянным тестам* (Lost Test; см. *Ошибки в продукте*, Production Bugs, с. 303).

Реализации xUnit в процедурной парадигме

Инфраструктуры автоматизации тестов (Test Automation Framework) и разработка на основе тестов стали популярны только после широкого распространения объектно-ориентированного программирования. Большинство членов семейства xUnit реализованы на объектно-ориентированных языках программирования, поддерживающих концепцию *объекта теста* (Testcase Object). Хотя отсутствие объектов не должно препятствовать тестированию процедурного кода, написание *самопроверяющихся тестов* (Self-

Checking Test) требует больше труда, а создание универсальных и готовых к повторному использованию *программ запуска тестов* (Test Runner) становится более сложным.

При отсутствии объектов или классов *тестовые методы* (Test Method) необходимо рассматривать как глобальные (открытые статичные) процедуры. Такие методы обычно хранятся в файлах или модулях (или с использованием другого механизма обеспечения модульности). Если язык поддерживает концепцию *процедурных переменных* (procedure variable), или *указателей на функцию* (function pointer), можно определить универсальную *процедуру набора тестов* (Test Suite Procedure; см. *Объект набора тестов*, Test Suite Object), которая принимает массив *тестовых методов* (Test Method), или “тестовых процедур”, в качестве параметра. Обычно *тестовые методы* (Test Method) агрегируются в массивы через *перечисление тестов* (Test Enumeration), так как очень мало необъектно-ориентированных языков поддерживают механизм интроспекции.

Если язык не поддерживает обработку *тестовых методов* (Test Method) как данных, наборы тестов придется определить через создание *процедур наборов тестов* (Test Suite Procedure), явно вызывающих *тестовые методы* (Test Method) и/или другие процедуры наборов тестов. При этом тесты могут запускаться с помощью определения метода main в пределах модуля.

Как последний вариант можно рассмотреть кодирование всех тестов в виде данных в файле и использовать один интерпретатор *управляемых данными тестов* (Data-Driven Test, с. 322). Основным недостатком такого подхода является ограничение множества тестов возможностями интерпретатора *управляемых данными тестов* (Data-Driven Test). При этом для каждой тестируемой системы такой интерпретатор должен быть реализован заново. Преимуществом такого подхода является возможность вынести написание кода тестов на уровень конечных пользователей и тестеров (что хорошо подходит для приемочных тестов).

Что дальше

В данной главе была приведена базовая терминология xUnit, которая будет использована при описании структуры тестов. Теперь можно уделить внимание новой задаче — созданию первой тестовой конфигурации в главе 8, “Управление временной тестовой конфигурацией”.

Глава 8

Управление временной тестовой конфигурацией

О чём идет речь в этой главе

В главе 6, “Стратегия автоматизации тестирования”, рассматривались стратегические решения, которые приходится принимать руководителям проектов. Также рассматривались термин “тестовая конфигурация” и вопросы, связанные с выбором стратегии управления тестовыми конфигурациями. В главе 7, “Основы xUnit”, были представлены базовая терминология xUnit и нотация, использованная в диаграммах. В настоящей главе описывается механизм реализации выбранной стратегии управления тестовыми конфигурациями.

Существует несколько способов создания *новой тестовой конфигурации* (Fresh Fixture, с. 344), и от выбранного решения будет зависеть объем трудозатрат на написание каждого теста и его обслуживание, а также эффективность *тестов как документации* (Tests as Documentation, с. 79). *Постоянная новая тестовая конфигурация* (Persistent Fresh Fixture) создается так же, как и *временная новая тестовая конфигурация* (Transient Fresh Fixture), но существуют некоторые отличия в процессе очистки тестовой конфигурации (рис. 8.1). С *общей тестовой конфигурацией* (Shared Fixture, с. 350) связан другой ряд особенностей. *Постоянная новая тестовая конфигурация* (Persistent Fresh Fixture) и *общая тестовая конфигурация* (Shared Fixture) более подробно рассматриваются в главе 9.

Тестовые конфигурации

Прежде всего необходимо определить, что же такое тестовая конфигурация.

Что такое тестовая конфигурация

Каждый тест состоит из четырех элементов, как следует из определения *четырехфазного теста* (Four-Phase Test, с. 387). Первый элемент отвечает за создание тестируемой системы и всего, от чего она зависит. После создания тестируемая система переводится в состояние, необходимое для вызова тестом. В xUnit все предварительные условия для вызова тестируемой системы называются **тестовой конфигурацией** (test fixture).

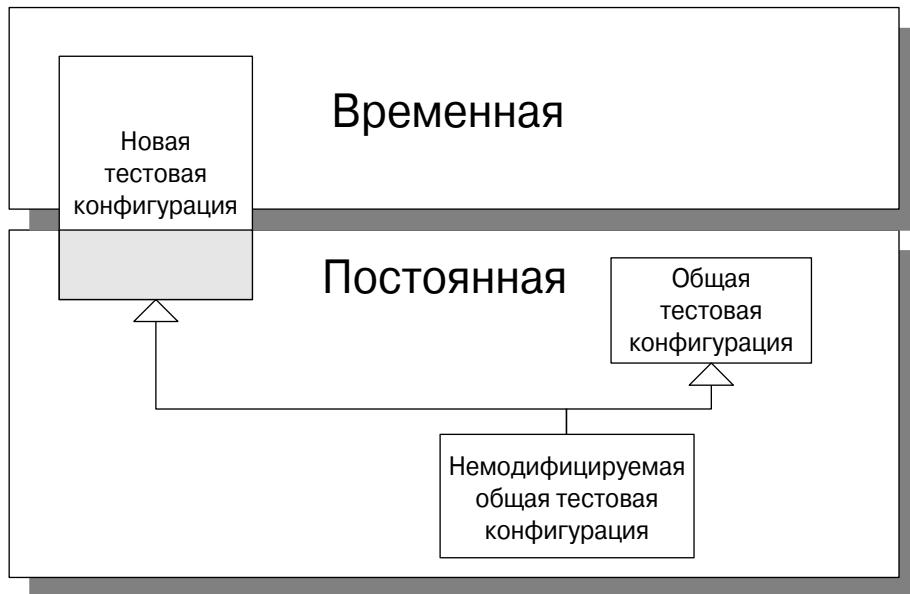


Рис. 8.1. Временная новая тестовая конфигурация (Transient Fresh Fixture). Новая тестовая конфигурация (Fresh Fixture) может быть двух видов — временной (transient) и постоянной (persistent). Оба типа требуют создания тестовой конфигурации. Последний тип также требует очистки тестовой конфигурации

Самым распространенным способом настройки тестовой конфигурации является настройка через “главный вход”, т.е. для перевода тестируемой системы в необходимое состояние вызываются ее методы в соответствующем порядке. Это может потребовать создания других объектов и их передачи тестируемой системе в качестве аргументов вызываемых методов. Если состояние тестируемой системы хранится в других объектах или компонентах, можно воспользоваться *настройкой через “черный ход”* (Back Door Setup; см. *Манипуляция через “черный ход”*, Back Door Manipulation, с. 359), когда необходимые записи вставляются непосредственно в другой компонент, от которого зависит поведение тестируемой системы. Чаще всего *настройка через “черный ход”* (Back Door Setup) применяется совместно с базами данных или при использовании *подставных объектов* (Mock Object, с. 558) и *тестовых двойников* (Test Double, с. 538). Такие ситуации рассматриваются в главе 13, “Тестирование с использованием баз данных”, и в главе 11, “Использование тестовых двойников”.

Стоит отметить, что термин “тестовая конфигурация” описывает разные сущности в зависимости от типа автоматизации тестирования. В реализациях xUnit для языков Microsoft под тестовой конфигурацией подразумевается *класс теста* (Testcase Class, с. 401). В большинстве других реализаций xUnit понятия *класса теста* (Testcase Class) и тестовой конфигурации (или контекста теста), которую он создает, разделяются. В инфраструктуре Fit [FitB] термин “тестовая конфигурация” описывает дополнительно созданные фрагменты интерпретатора управляемых данными тестов (Data-Driven Test, с. 322), который применяется для определения языка высокого уровня (Higher-Level Language, с. 95). Если в данной книге упоминается “тестовая конфигурация” без дополнительной квали-

фикации термина, имеется ввиду все, что настраивается перед вызовом тестируемой системы. Для описания класса, содержащего *тестовые методы* (Test Method, с. 378), вне зависимости от языка программирования используется термин *класс теста* (Testcase Class).

Что такое новая тестовая конфигурация

В стратегии на основе *новой тестовой конфигурации* (Fresh Fixture) новая тестовая конфигурация создается для каждого запускаемого теста (рис. 8.2), т.е. каждый *объект теста* (Testcase Object, с. 410) создает собственную тестовую конфигурацию перед вызовом тестируемой системы и делает это даже при повторных запусках. Именно это делает тестовую конфигурацию “новой”. В результате удается полностью избежать проблем, связанных с *взаимодействующими тестами* (Interacting Tests; см. *Нестабильный тест*, Erratic Test, с. 267).

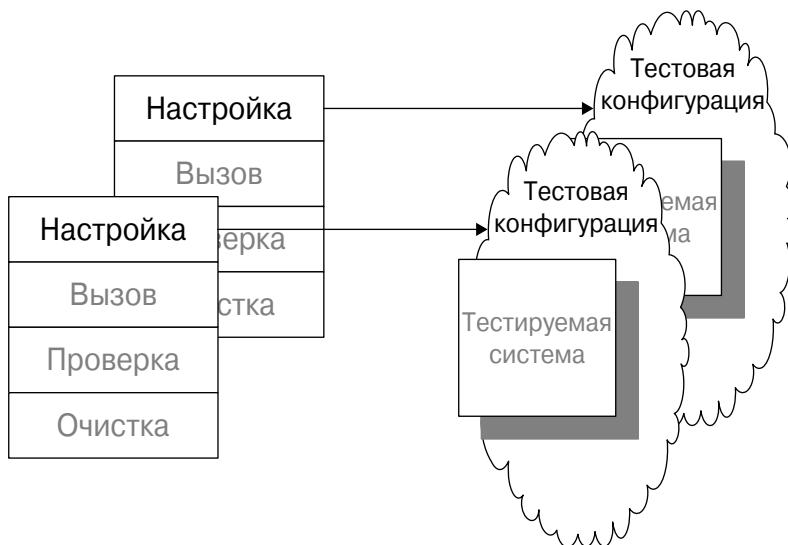


Рис. 8.2. Пара новых тестовых конфигураций (Fresh Fixture) и создающие их объекты. Новая тестовая конфигурация (Fresh Fixture) специально создается для одного теста, используется один раз и удаляется

Что такое временная новая тестовая конфигурация

Если тестовая конфигурация хранится только в памяти и на нее ссылаются только локальные переменные или переменные экземпляров¹, тестовая конфигурация просто исчезает после завершения работы тестов в результате *очистки со сборкой мусора* (Garbage-Collected Teardown, с. 518). С постоянными тестовыми конфигурациями этого не происходит. Следовательно, необходимо принять ряд решений о реализации стратегии с использованием *новой тестовой конфигурации* (Fresh Fixture). В частности, существует два способа обеспечения “свежести” тестовой конфигурации. Очевидным решением являет-

¹ См. врезку “Всегда есть исключения” на с. 411.

ся очистка после каждого теста. Менее очевидным является создание новой тестовой конфигурации, не затрагивая старой.

Большинство создаваемых *новых тестовых конфигураций* (Fresh Fixture) являются временными, поэтому данная тема будет рассматриваться в первую очередь. Управление *постоянными новыми тестовыми конфигурациями* (Persistent Fresh Fixture) будет рассматриваться в главе 9.

Создание новых тестовых конфигураций

Способы создания *временной новой тестовой конфигурации* (Transient Fresh Fixture) или *постоянной новой тестовой конфигурации* (Persistent Fresh Fixture) практически не отличаются. Логика генерации тестовой конфигурации включает в себя код создания экземпляра тестируемой системы², код перевода тестируемой системы в необходимое состояние и код создания и инициализации объектов, от которых зависит тестируемая система или которые будут передаваться в качестве аргумента. Самым очевидным способом создания *новой тестовой конфигурации* (Fresh Fixture) является *встроенная настройка* (In-line Setup, с. 434), при которой вся логика создания конфигурации находится в пределах *тестового метода* (Test Method). Также данная тестовая конфигурация может создаваться с помощью *делегированной настройки* (Delegated Setup, с. 437), подразумевающей вызов *вспомогательных методов теста* (Test Utility Method, с. 610). Наконец, можно воспользоваться *неявной настройкой* (Implicit Setup, с. 449), при которой *инфраструктура автоматизации тестов* (Test Automation Framework, с. 332) вызывает специальный метод `setUp` в составе *класса теста* (Testcase Class). Кроме того, можно использовать комбинацию этих подходов. Рассмотрим каждый вариант в отдельности.

Встроенная настройка тестовой конфигурации

При *встроенной настройке* (In-line Setup) создание тестовой конфигурации реализуется внутри *тестового метода* (Test Method). При этом создаются объекты, вызываются их методы, создается тестируемая система и вызываются ее методы для получения необходимого состояния. Все эти операции выполняются внутри *тестовых методов* (Test Method). Встроенную настройку (In-line Setup) можно рассматривать как создание тестовой конфигурации по методу “сделай сам”.

```
public void testStatus_initial() {
    // Встроенная настройка (In-line Setup)
    Airport departureAirport = new Airport("Calgary", "YYC");
    Airport destinationAirport = new Airport("Toronto", "YYZ");
    Flight flight = new Flight( flightNumber,
                               departureAirport,
                               destinationAirport );
    // Вызвать тестируемую систему и проверить результат
    assertEquals(FlightState.PROPOSED, flight.getStatus());
    // Очистка
    //     сборка мусора
}
```

² Предполагается, что в качестве тестируемой системы выступает объект, а не статический метод класса.

Основным недостатком *встроенной настройки* (In-line Setup) является вероятность *дублирования тестового кода* (Test Code Duplication, с. 254), поскольку каждому *тестовому методу* (Test Method) приходится создавать тестируемую систему и множество методов выполняют одни и те же задачи по настройке тестовой конфигурации. Результатом *дублирования тестового кода* (Test Code Duplication) является *высокая стоимость обслуживания тестов* (High Test Maintenance Cost, с. 300) как следствие “хрупких” тестов (Fragile Test, с. 277). Если процедура создания тестовой конфигурации достаточно сложная, это может привести к появлению *непонятных тестов* (Obscure Test, с. 230). С этим также связана проблема *фиксированных данных теста* (Hard-Coded Test Data; см. *Непонятный тест*, Obscure Test) в пределах каждого *тестового метода* (Test Method), так как создание локальной переменной с описывающим намерение именем может показаться слишком дорогостоящим.

Избежать появления всех этих запахов можно посредством выноса кода настройки тестовой конфигурации из *тестовых методов* (Test Method). Пункт назначения такого кода зависит от выбора альтернативной стратегии создания тестовой конфигурации.

Делегированная настройка тестовой конфигурации

Сократить *дублирование тестового кода* (Test Code Duplication) и уменьшить вероятность появления *непонятных тестов* (Obscure Test) можно с помощью рефакторинга *тестовых методов* (Test Method) с целью применения *делегированной настройки* (Delegated Setup). Для переноса последовательности операторов из нескольких *тестовых методов* (Test Method) в один *вспомогательный метод теста* (Test Utility Method) можно воспользоваться рефакторингом *выделение метода* (Extract Method) [Ref]. После этого *вспомогательный метод теста* (Test Utility Method) будет вызываться из модифицированных *тестовых методов* (Test Method). Это очень простой и безопасный рефакторинг, особенно если поручить среде разработки выполнение основных операций. Если выделенный метод содержит логику создания объекта, от которого зависит тест, он называется *методом создания* (Creation Method, с. 441)³. Методы создания с описывающими намерение именами делает предварительные условия теста очевидными для читателя, не приводя к ненужному *дублированию тестового кода* (Test Code Duplication). При этом и читатель, и разработчик тестов могут сосредоточиться на том, что создается, не отвлекаясь на то, как это делается. *Методы создания* (Creation Method) выступают в роли готовых к повторному использованию блоков создания тестовых конфигураций.

```
public void testGetStatus_inital() {
    // Настройка
    Flight flight = createAnonymousFlight();
    // Вызов тестируемой системы и проверка результата
    assertEquals(FlightState.PROPOSED, flight.getStatus());
    // Очистка
    //     сборка мусора
}
```

Одной из целей *методов создания* (Creation Method) является скрытие подробностей создания необходимых объектов от каждого теста. В результате снижается вероятность

³ С точки зрения *вспомогательного класса теста* (Test Helper, с. 651) они называются шаблоном *инкубатор объектов* (Object Mother).

появления “хрупких” тестов (Fragile Test), зависящих от изменения сигнатуры или семантики конструкторов. Если тесту не требуется идентификация объекта, можно воспользоваться *анонимным методом создания* (Anonymous Creation Method; см. *Метод создания*, Creation Method). При этом метод автоматически генерирует все необходимые объекту уникальные ключи. Использование *отдельного сгенерированного значения* (Distinct Generated Value; см. *Сгенерированное значение*, Generated Value, с. 726) позволяет гарантировать, что больше ни один экземпляр теста не будет обращаться к этому же объекту. Такой “предохранитель” позволяет избежать многих форм запаха поведения *нестабильный тест* (Erratic Test), включая *неповторяемый тест* (Unrepeatable Test), *взаимодействующие тесты* (Interacting Tests) и “войны” запуска тестов (Test Run War), даже если используется постоянное хранилище объектов, поддерживающее *общие тестовые конфигурации* (Shared Fixture).

Если тест зависит от атрибутов создаваемого объекта, используется *параметризованный анонимный метод создания* (Parameterized Anonymous Creation Method; см. *Метод создания*, Creation Method). Такому методу в качестве параметров передаются значения значимых атрибутов, а все остальные атрибуты устанавливаются методом автоматически. Обычно в таком случае рекомендуется следующий подход.

Если внутри тестового метода что-то отсутствует, важно, чтобы внутри тестового метода это не появлялось!

Делегированная настройка (Delegated Setup) часто используется при написании тестов проверки входных данных методов тестируемой системы (атрибутов объектов, переданных в качестве аргументов). В таком случае приходится писать отдельные тесты для каждого атрибута, значение которого должно проверяться. Создание всех некорректных объектов через *встроенную настройку* (In-line Setup) отняло бы слишком много времени. Для сокращения объема работы и во избежание вероятности *дублирования тестового кода* (Test Code Duplication) можно воспользоваться шаблоном *единственный дефектный атрибут* (One Bad Attribute; см. *Вычисляемое значение*, Derived Value, с. 722); сначала вызывается *метод создания* (Creation Method) для создания правильного объекта, а затем один из атрибутов получает некорректное значение, предположительно отвергаемое тестируемой системой. Точно так правильный объект можно создать с помощью *достигающего указанного состояния метода* (Named State Reaching Method; см. *Метод создания*, Creation Method).

Некоторые разработчики предпочитают *повторно использовать тесты для настройки тестовой конфигурации* (Reuse tests for fixture setup; см. *Метод создания*, Creation Method) в качестве альтернативы использованию *цепочки тестов* (Chained Tests, с. 477). В таком случае другие тесты вызываются непосредственно на этапе создания тестовой конфигурации. В подобном подходе нет ничего плохого, пока читатель теста в состоянии быстро определить, что другие тесты настраивают для данного теста. К сожалению, достаточно мало тестов имеют названия, описывающие их назначение. Именно по этой причине для получения эффекта от *тестов как документации* (Tests as Documentation) интересующие тесты нужно скрыть внутри *метода создания* (Creation Method), который имеет описывающее намерение имя. В таком случае читатель достаточно быстро поймет, как выглядит тестовая конфигурация.

Метод создания (Creation Method) может оставаться закрытым методом *класса теста* (Testcase Class), быть вынесенным в *суперкласс теста* (Testcase Superclass, с. 646) или перенесенным во *вспомогательный класс теста* (Test Helper, с. 651). “Матерью” всех методов

создания” является *инкубатор объектов* (Object Mother; см. *Вспомогательный класс теста*, Test Helper). Этот стратегический шаблон описывает семейство подходов по использованию *методов создания* (Creation Method) над одним или несколькими вспомогательными классами теста (Test Helper). Кроме того, в шаблон может входить *автоматическая очистка* (Automated Teardown, с. 522).

Неявная настройка тестовой конфигурации

В большинстве реализаций xUnit обеспечивается удобный способ вызова кода, который должен отработать перед каждым *тестовым методом* (Test Method). В одних случаях код вызывается с использованием специального имени (например, `setUp`). В других вызывается метод со специальной аннотацией (например, `@before` в JUnit) или атрибутом метода (например, `[Setup]` в NUnit). Чтобы избежать упоминания всех альтернативных методов вызова таких методов, в данной книге будет упоминаться просто метод `setUp` вне зависимости от механизма, реализованного в *инфраструктуре автоматизации тестов* (Test Automation Framework). Метод `setUp` является необязательным или инфраструктура обеспечивает принятую по умолчанию реализацию, чтобы не создавать этот метод самостоятельно для каждого *класса теста* (Testcase Class).

При *неявной настройке* (Implicit Setup) эта “ловушка” в инфраструктуре применяется через размещение всей логики создания тестовой конфигурации в метод `setUp`. Поскольку каждый *тестовый метод* (Test Method) в пределах *класса теста* (Testcase Class) использует одну и ту же логику создания тестовой конфигурации, все *тестовые методы* (Test Method) должны уметь использовать полученную конфигурацию. Такая тактика решает проблему *дублирования тестового кода* (Test Code Duplication), но имеет ряд других нежелательных последствий. Например, что проверяет приведенный ниже тест?

```
Airport departureAirport;
Airport destinationAirport;
Flight flight;
public void testGetStatus_initial() {
    // Неявная настройка (Implicit Setup)
    // Вызвать тестируемую систему и проверить результат
    assertEquals(FlightState.PROPOSED, flight.getStatus());
}
```

Первым следствием являются затруднения в понимании смысла теста, так как связь предварительных условий (тестовой конфигурации) и ожидаемого результата не видна в пределах *тестового метода* (Test Method). Для получения этой информации придется рассматривать метод `setUp`.

```
public void setUp() throws Exception{
    super.setUp();
    departureAirport = new Airport("Calgary", "YYC");
    destinationAirport = new Airport("Toronto", "YYZ");
    BigDecimal flightNumber = new BigDecimal("999");
    flight = new Flight(flightNumber, departureAirport,
                        destinationAirport);
}
```

Для решения этой проблемы можно выбрать имя *класса теста* (Testcase Class) в зависимости от создаваемой в методе `setUp` тестовой конфигурации. Конечно, это имеет смысл, только если всем *тестовым методам* (Test Method) необходима одна и та же тестовая конфигурация, — что фактически реализует стратегию класса теста для каждой *тестовой конфигурации* (Testcase Class per Fixture, с. 639). Как было показано ранее, в нескольких реализациях xUnit (например, VbUnit и NUnit) термин “тестовая конфигурация” ссылается на сущность, которая в данной книге называется *классом теста* (Testcase Class). Скорее всего, такая терминология основана на предположении, что будет использоваться стратегия *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture).

Еще одним следствием применения *неявной настройки* (Implicit Setup) является невозможность использования локальных переменных для хранения ссылок на объекты в составе тестовой конфигурации. Вместо этого для хранения объектов, создаваемых в методе `setUp` и необходимых для вызова тестируемой системы и проверки ожидаемого результата, используются переменные экземпляра. Эти же переменные используются для очистки тестовой конфигурации и выступают в роли глобальных переменных между фрагментами теста. Пока используются переменные экземпляров, а не переменные класса, тестовая конфигурация будет заново создаваться для каждого теста в пределах *класса теста* (Testcase Class). В большинстве реализаций xUnit обеспечивается изоляция между тестовыми конфигурациями, созданными для каждого *тестового метода* (Test Method), но, например, в NUnit такой изоляции нет. Дополнительная информация по данному вопросу приводится во врезке “Всегда есть исключения” на с. 411. В любом случае переменные всегда должны иметь описательные имена, чтобы не приходилось постоянно возвращаться к методу `setUp` и выяснять назначение переменных.

Злоупотребление методом `setUp`

Когда в руках новый молоток, хочется забить все гвозди!

Как и любой функцией любой системы, методом `setUp` можно злоупотреблять. Не пытайтесь применять метод `setUp` только потому, что его можно применить. Это один из нескольких механизмов повторного использования кода в приложении. Когда объектно-ориентированные языки программирования только появились, разработчики с энтузиазмом применяли наследование во всех возможных вариантах повторного использования. Со временем стало ясно, где имеет смысл применять наследование, а где лучше прибегнуть к другим механизмам повторного использования, например к делегированию. Метод `setUp` является аналогом наследования в xUnit.

Самым распространенным вариантом злоупотребления является использование метода `setUp` для создания *тестовой конфигурации общего характера* (General Fixture; см. *Непонятный тест*, Obscure Test), состоящей из нескольких независимых частей, предназначенных для разных *тестовых методов* (Test Method). Если создается *постоянная новая тестовая конфигурация* (Persistent Fresh Fixture), может появиться *медленный тест* (Slow Tests, с. 289). Кроме того, скрывается причинно-следственная связь между тестовой конфигурацией и ожидаемым результатом вызова тестируемой системы, что приводит к появлению *непонятных тестов* (Obscure Test).

Если не переходить к практике группирования *тестовых методов* (Test Method) в *классы теста* (Testcase Class) в соответствии с используемыми тестовыми конфигурациями, а продолжать применять метод `setUp`, в этом методе необходимо создать только

общую часть тестовой конфигурации. Иначе говоря, в методе `setUp` должна присутствовать только та логика, которая без проблем работает со всеми тестами. Даже если код создания тестовой конфигурации без проблем работает со всеми *тестовыми методами* (Test Method), он все равно может привести к другим проблемам, если метод `setUp` создает *тестовую конфигурацию общего характера* (General Fixture) вместо *минимальной тестовой конфигурации* (Minimal Fixture, с. 336). *Тестовая конфигурация общего характера* (General Fixture) часто является причиной возникновения *медленных тестов* (Slow Test), поскольку каждый тест тратит намного больше времени на создание тестовой конфигурации. Кроме того, частым результатом являются *непонятные тесты* (Obscure Test), так как читателю тестов сложно определить зависимость *тестового метода* (Test Method) от конкретного элемента тестовой конфигурации. Часто *тестовая конфигурация общего характера* (General Fixture) превращается в *“хрупкую” тестовую конфигурацию* (Fragile Fixture; см. *“Хрупкий” тест*, Fragile Test), так как отношение различных элементов к тестам со временем забывается. Внесение изменений в тестовую конфигурацию для обеспечения работы новых тестов может привести к нарушениям в работе уже существующих тестов.

Обратите внимание, что, если для хранения объекта будет использоваться переменная класса, тестовая конфигурация превратится в постоянную. Использование *“ленивой” настройки* (Lazy Setup, с. 460) для присвоения такой переменной значения делает тестовую конфигурацию общей, так как последующие тесты будут повторно использовать объекты, созданные предыдущими тестами, а значит, станут зависимыми от изменений, вносимых другими тестами.

Гибридная настройка тестовой конфигурации

В этой главе три стиля создания тестовой конфигурации рассматриваются как строгие альтернативы друг другу. На практике может оказаться полезным комбинирование различных методов. При автоматизации тестов *методы создания* (Creation Method) часто вызываются из *тестовых методов* (Test Method), но после этого дополнительная настройка выполняется внутри тестового метода. Метод `setUp` может оказаться проще для чтения, если из него вызывать *методы создания* (Creation Method) для генерации тестовой конфигурации. Кроме того, *методы создания* (Creation Method) имеют еще одно положительное свойство — они намного лучше подходят для модульного тестирования по сравнению со встроенной настройкой или методом `setUp`. Такие методы могут являться членами класса, не входящего в иерархию *классов теста* (Testcase Class), как, например, *вспомогательный класс теста* (Test Helper).

Очистка временной новой тестовой конфигурации

Одним из положительных качеств *временной новой тестовой конфигурации* (Transient Fresh Fixture) является простота очистки. Большинство членов семейства xUnit реализованы на языках, поддерживающих *сборку мусора* (garbage collection). Пока ссылки на тестовую конфигурацию хранятся в переменных с конечной областью видимости, можно рассчитывать на *очистку со сборкой мусора* (Garbage-Collected Teardown) как механизм удаления использованных тестовых конфигураций. Во врезке “Всегда есть исключения” на с. 411 показано, почему такой механизм не работает в пакете NUnit.

Если выбранная реализация xUnit не поддерживает сборку мусора, все *новые тестовые конфигурации* (Fresh Fixture) можно воспринимать как постоянные.

Что дальше

В этой главе рассматривались способы настройки и очистки хранящейся в памяти *новой тестовой конфигурации* (Fresh Fixture). При достаточном планировании и везении это все, что требуется в большинстве тестов. Управлять *новыми тестовыми конфигурациями* (Fresh Fixture) становится сложнее, если тестовая конфигурация сохраняется тестируемой системой или самим тестом. В главе 9, “Управление постоянными тестовыми конфигурациями”, рассматриваются дополнительные способы управления тестовыми конфигурациями, включая *постоянную новую тестовую конфигурацию* (Persistent Fresh Fixture) и *общую тестовую конфигурацию* (Shared Fixture).

Глава 9

Управление постоянными тестовыми конфигурациями

О чём идет речь в этой главе

В главе 8, “Управление временной тестовой конфигурацией”, рассматривалось обслуживание хранящихся в памяти *новых тестовых конфигураций* (Fresh Fixture, с. 344). Было показано, что управление тестовыми конфигурациями усложняется, если она сохраняется тестируемой системой или самим тестом. В этой главе рассматриваются дополнительные шаблоны, необходимые для управления постоянными тестовыми конфигурациями, включая *постоянную новую тестовую конфигурацию* (Persistent Fresh Fixture; см. *Новая тестовая конфигурация*, Fresh Fixture) и *общую тестовую конфигурацию* (Shared Fixture, с. 350).

Управление постоянными новыми тестовыми конфигурациями

Термин *постоянная новая тестовая конфигурация* (Persistent Fresh Fixture) может показаться оксюмороном, но на самом деле здесь нет противоречия. Стратегия с использованием *новой тестовой конфигурации* (Fresh Fixture) предполагает, что каждый запуск каждого *тестового метода* (Test Method, с. 378) использует заново созданную тестовую конфигурацию. Название отражает сущность: тестовая конфигурация не используется повторно! Это не значит, что тестовая конфигурация сразу исчезает — просто она не используется повторно (рис. 9.1). *Постоянная новая тестовая конфигурация* (Persistent Fresh Fixture) имеет ряд особенностей, усложняющих работу по сравнению с *временной новой тестовой конфигурацией* (Transient Fresh Fixture). В данной главе основное внимание уделяется проблемам, связанным с *неповторяемыми тестами* (Unrepeatable Test; см. *Нестабильный тест*, Erratic Test, с. 267), причиной появления которых являются остатки постоянной тестовой конфигурации, и с *медленными тестами* (Slow Tests, с. 289), причиной возникновения которых являются *общие тестовые конфигурации* (Shared Fixture, с. 350).

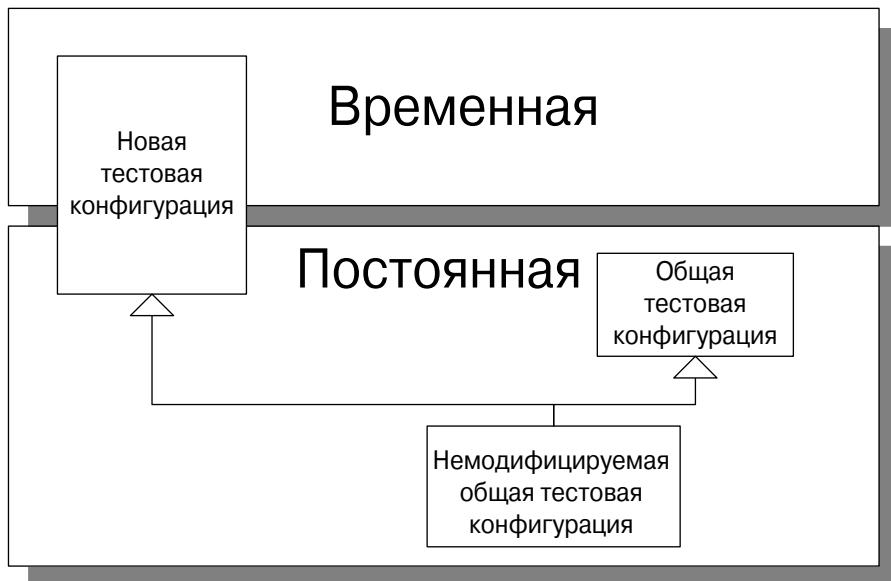


Рис. 9.1. Новая тестовая конфигурация (Fresh Fixture) может быть временной или постоянной. Стратегию на основе новой тестовой конфигурации (Fresh Fixture) можно применять даже для постоянных тестовых конфигураций, но приходится обеспечивать их удаление после каждого теста

Что делает тестовую конфигурацию постоянной

Тестовая конфигурация (новая или существующая) может стать постоянной по двум причинам. Первой причиной является сохранение состояния в тестируемой системе (объект “помнит”, как он использовался в прошлом). Такой сценарий обычно возникает, когда в состав тестируемой системы входит база данных или для хранения данных используются переменные классов. Второй причиной является хранение ссылки на тестовую конфигурацию в пределах *класса теста* (Testcase Class, с. 401). При этом ссылка на во всех отношениях *временную новую тестовую конфигурацию* (Transient Fresh Fixture) хранится таким образом, что тестовая конфигурация переживает повторные вызовы *тестового метода* (Test Method).

В некоторых реализациях xUnit обеспечивается механизм повторной загрузки всех классов перед каждым запуском теста. Такое поведение может включаться по выбору разработчика (через флагок “Повторно загрузить классы”) или автоматически. Такая функция защищает тестовые конфигурации от сохранения при хранении ссылок в переменных классов (кроме случаев, когда конфигурация сохраняется тестируемой системой или тестом на файловой системе или в базе данных).

Проблемы постоянных новых тестовых конфигураций

Если тестовые конфигурации сохраняются, при последующих запусках одних и тех же *тестовых методов* (Test Method) может оказаться, что тест создает уже существующую тестовую конфигурацию. Такое поведение может вызвать конфликт между новыми и су-

шествующими ресурсами. Хотя нарушение уникальности ключей в базе данных является самой распространенной проблемой, к конфликту может привести даже создание файла с именем уже существующего файла. Избежать появления *неповторяемых тестов* (Unrepeatable Test) можно, очистив тестовую конфигурацию после завершения работы теста. Еще один подход предполагает использование *отдельного сгенерированного значения* (Distinct Generated Value; см. *Сгенерированное значение*, Generated Value, с. 726) для всех идентификаторов, которые могут вступать в конфликт друг с другом.

Очистка постоянных новых тестовых конфигураций

В отличие от кода создания тестовой конфигурации, который также должен описывать предварительные условия теста, код очистки предназначен только для удаления остатков конфигурации. Он никак не помогает понять поведение тестируемой системы, но может скрыть назначение теста или сделать его сложным в понимании. Таким образом, самым лучшим является несуществующий код очистки. Написания кода очистки стоит избегать везде, где это возможно, предпочитая *очистку со сборкой мусора* (Garbage-Collected Teardown, с. 518). К сожалению, с постоянными новыми тестовыми конфигурациями сборка мусора не работает.

Код очистки, написанный вручную

Одним из способов удаления тестовой конфигурации является включение соответствующего кода в *тестовый метод* (Test Method). Такой механизм на самом деле сложнее, чем кажется на первый взгляд. Рассмотрим следующий пример.

```
public void testGetFlightsByOriginAirport_NoFlights()
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("1OF");
    // Вызов тестируемой системы
    List flightsAtDestination1 =
        facade.getFlightsByOriginAirport(outboundAirport);
    // Проверка результата
    assertEquals(0, flightsAtDestination1.size());
    facade.removeAirport(outboundAirport);
}
```

Простейшая встроенная очистка (Naive In-line Teardown; см. *Встроенная очистка*, In-line Teardown, с. 527) удалит тестовую конфигурацию в случае удачного завершения теста, но в случае ошибки или неудачного завершения очистка не сработает. Это связано с тем, что вызов *метода с утверждением* (Assertion Method, с. 390) генерирует исключение. Таким образом, работа теста завершается раньше еще до выполнения кода очистки. Для обеспечения гарантированного выполнения кода *встроенной очистки* (In-line Teardown) все генерирующие исключения конструкции внутри *тестового метода* (Test Method) должны заключаться в структуру управления *try/catch*. Ниже приведен тот же тест с описанными модификациями.

```
public void testGetFlightsByOriginAirport_NoFlights_td()
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("1OF");
```

```

try {
    // Вызов тестируемой системы
    List flightsAtDestination1 =
        facade.getFlightsByOriginAirport(outboundAirport);
    // Проверка результата
    assertEquals(0, flightsAtDestination1.size());
} finally {
    facade.removeAirport(outboundAirport);
}
}

```

К сожалению, механизм запуска кода очистки заметно усложняет *тестовый метод* (Test Method). Все становится еще сложнее, если необходимо удалить несколько ресурсов: даже если очистка одного из ресурсов завершается неудачно, необходимо обеспечить очистку остальных ресурсов. Часть проблемы можно решить за счет рефакторинга *выделение метода* (Extract Method) [Ref], переместив код очистки во *вспомогательный метод теста* (Test Utility Method, с. 610), который будет вызываться из конструкции обработки ошибок. Хотя в таком случае *делегированная очистка* (Delegated Teardown; см. *Встроенная очистка*, In-line Teardown) скрывает сложность работы с ошибками очистки, необходимость запуска метода очистки при ошибке или неудачном завершении теста остается.

В большинстве реализаций xUnit данная проблема решена за счет *неявной очистки* (Implicit Teardown, с. 533). *Инфраструктура автоматизации тестов* (Test Automation Framework, с. 332) вызывает специальный метод `tearDown` после завершения работы каждого *тестового метода* (Test Method) вне зависимости от результатов выполнения теста. Такой подход позволяет избежать добавления кода обработки ошибок в *тестовый метод* (Test Method), но налагает на тесты два дополнительных требования. Во-первых, тестовая конфигурация должна быть доступна из метода `tearDown`, а значит, необходимо использовать переменные экземпляров (предпочтительный вариант), переменные классов или глобальные переменные для хранения тестовой конфигурации. Во-вторых, метод `tearDown` должен правильно работать с каждым *тестовым методом* (Test Method) вне зависимости от создаваемой тестовой конфигурации. (Ситуация упрощается при использовании стратегии *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639), так как всегда будет использоваться одна и та же тестовая конфигурация. Если используется другая организация *классов теста* (Testcase Class), может потребоваться включение *пункта охраны очистки* (Teardown Guard Clause; см. *Встроенная очистка*, In-line Teardown) в метод `tearDown`, чтобы обеспечить отсутствие ошибок во время работы.)

Организация кода с соответствием настройки и очистки

Существует три способа организации кода настройки:

- *встроенная настройка* (In-line Setup, с. 433);
- *делегированная настройка* (Delegated Setup, с. 437);
- *неявная настройка* (Implicit Setup, с. 449).

Существует также три типа организации кода очистки:

- *встроенная очистка* (In-line Teardown);
- *делегированная очистка* (Delegated Teardown);
- *неявная очистка* (Implicit Teardown).

Всего доступно девять их комбинаций. Выбор подходящей комбинации не требует сложных решений, так как код очистки не обязательно должен быть доступен читателю тестов. Достаточно выбрать наиболее подходящий код настройки и эквивалентный или более скрытый вариант очистки (табл. 9.1). Например, *неявная очистка* (Implicit Teardown) может использоваться даже вместе со *встроенной настройкой* (In-line Setup) или *дeлегированной настройкой* (Delegated Setup). С другой стороны, *встроенную очистку* (In-line Teardown) имеет смысл использовать только вместе со *встроенной настройкой* (In-line Setup), но и такого варианта следует избегать.

Таблица 9.1. Совместимость стратегий настройки и очистки постоянных тестовых конфигураций

Механизм настройки	Механизм очистки		
	Встроенная очистка (In-line Teardown)	Делегированная очистка (Delegated Teardown)	Неявная очистка (Implicit Teardown)
Встроенная настройка (In-line Setup)	Не рекомендуется	Допустима	Рекомендуется
Делегированная настройка (Delegated Setup)	Не рекомендуется	Допустима	Рекомендуется
Неявная настройка (Implicit Setup)	Не рекомендуется	Не рекомендуется	Рекомендуется

Автоматизированная очистка

С написанным вручную кодом очистки связаны две проблемы: дополнительная работа при написании тестов и сложность в написании и тестировании. Если очистка работает некорректно, могут появляться *нестабильные тесты* (Erratic Test), причиной которых становится *утечка ресурсов* (Resource Leakage), так как неудачно завершившийся тест часто отличается от теста, некорректно очистившего тестовую конфигурацию.

В языках с поддержкой сборки мусора удаление *временной новой тестовой конфигурации* (Transient Fresh Fixture) выполняется автоматически. Пока на тестовую конфигурацию ссылаются только переменные экземпляров, которые выходят из области видимости после уничтожения *объекта теста* (Testcase Object, с. 410), сборка мусора обеспечит удаление. Но если применяются переменные классов или тестовые конфигурации включают в себя файлы или базы данных, сборка мусора не работает. В таком случае приходится выполнять очистку самостоятельно.

Не удивительно, что такая ситуация заставит ленивого, но изобретательного разработчика найти способ автоматизации логики очистки. Важно отметить, что код очистки не помогает в понимании смысла теста, поэтому его имеет смысл скрывать (в отличие от кода настройки, который часто очень важен для понимания назначения теста). Механизм *автоматической очистки* (Automated Teardown, с. 521) позволяет отказаться от написания кода очистки вручную для каждого *тестового метода* (Test Method) или *класса теста* (Testcase Class). Такой механизм содержит три компонента.

1. Хорошо протестированный механизм итерации по списку удаляемых объектов с перехватом и занесением в журнал любых ошибок в процессе удаления.

2. Механизм диспетчеризации для вызова подходящего кода, в зависимости от типа удаляемого объекта. Часто такой механизм реализован в виде объекта Command [GOF], содержащего в себе каждый удаляемый объект, но иногда достаточно просто вызвать метод `delete` самого объекта или воспользоваться конструкцией `switch`, зависящей от класса объекта.
3. Механизм регистрации для добавления созданных объектов (с соответствующей упаковкой) в очередь на удаление.

После создания механизма *автоматической очистки* (Automated Teardown) можно просто вызвать метод регистрации из *метода создания* (Creation Method, с. 441) и метод удаления из метода `tearDown`. Последняя операция может быть описана в *суперклассе теста* (Testcase Superclass, с. 646), наследуемого каждым *классом теста* (Testcase Class). Этот механизм можно даже расширить для удаления объектов, созданных тестируемой системой в результате вызова. Для этого можно воспользоваться наблюдаемой *фабрикой объектов* (Object Factory; см. *Поиск зависимости*, Dependency Lookup, с. 692) внутри тестируемой системы и зарегистрировать *суперкласс теста* (Testcase Superclass) как наблюдатель (Observer) [GOF] относительно создания объектов.

Очистка базы данных

Если постоянная новая *тестовая конфигурация* (Fresh Fixture) создается исключительно внутри реляционной базы данных, для удаления тестовой конфигурации можно воспользоваться возможностями базы данных. *Очистка усечением таблиц* (Table Truncation Teardown, с. 668) является методом “грубой силы”, так как одной командой удаляет содержимое целой таблицы. Конечно, такой подход оправдан, только если каждая *программа запуска тестов* (Test Runner, с. 405) имеет собственную “песочницу” с базой данных (Database Sandbox, с. 658). Менее радикальный подход заключается в использовании *очистки откатом транзакции* (Transaction Rollback Teardown, с. 675), когда отменяются все изменения, внесенные во время работы теста. Такой механизм основан на проектировании тестируемой системы с использованием *минимального контроллера транзакций* (Humble Transaction Controller; см. *Минимальный объект*, Humble Object, с. 700), что позволяет вызывать бизнес-логику из теста без автоматического сохранения результатов транзакции. Оба шаблона очистки с использованием базы данных чаще всего реализуются как *неявная очистка* (Implicit Teardown), чтобы не вносить логику очистки в *тестовые методы* (Test Method).

Как избежать очистки

Выше рассматривалась очистка тестовой конфигурации. Теперь выясним, как можно избежать очистки.

Обход коллизий тестовых конфигураций

Очистка тестовой конфигурации нужна по трем причинам.

1. Накопление остатков объектов конфигурации может замедлить работу тестов.
2. Остатки объектов конфигурации могут привести к изменению поведения тестируемой системы или утверждения будут возвращать некорректные результаты.

3. Остатки объектов конфигурации могут помешать созданию *новых тестовых конфигураций* (Fresh Fixture) для других тестов.

Проще всего устранить первую причину: можно регулярно возвращать механизм сохранения объектов к известному минимальному состоянию. К сожалению, такая тактика полезна, только если тесты можно заставить работать корректно у условиях накопления обломков конфигураций.

Вторую причину можно устраниТЬ, используя *дельта-утверждения* (Delta Assertion, с. 505) вместо “абсолютных” утверждений. *Дельта-утверждение* (Delta Assertion) сравнивает моментальный снимок тестовой конфигурации до запуска теста с состоянием тестовой конфигурации после вызова тестируемой системы.

Третья причина устраняется путем генерации каждым тестом нового набора объектов конфигурации. Таким образом, необходимые тесту объекты получают полностью уникальные идентификаторы, т.е. уникальные имена файлов, уникальные ключи и т.д. Для этого можно создать простой генератор уникальных идентификаторов, получать новый идентификатор в начале каждого теста и использовать его при создании каждого нового объекта тестовой конфигурации. Если тестовая конфигурация совместно используется несколькими *программами запуска тестов* (Test Runner), в уникальном идентификаторе придется учитывать текущего пользователя. Использование *отдельных сгенерированных значений* (Distinct Generated Value) в качестве ключей предоставляет и другие преимущества: появляется возможность реализовать *схему разбиения базы данных* (Database Partitioning Scheme; см. “*Песочница*” с базой данных, Database Sandbox), в которой абсолютные утверждения применимы несмотря на остатки тестовых конфигураций.

Защита тестовой конфигурации от сохранения

Очевидно, что приходится прилагать слишком много усилий, чтобы компенсировать недостатки постоянных *новых тестовых конфигураций* (Fresh Fixture). Нельзя ли избежать всей этой работы? Хорошая новость: можно! Плохая новость: *новая тестовая конфигурация* (Fresh Fixture) должна быть специально защищена от сохранения. Если за сохранение отвечает тестируемая система, механизм сохранения можно заменить *тестовым двойником* (Test Double, 538), удаляемым тестом по необходимости. Хорошим примером такого подхода является применение *поддельной базы данных* (Fake Database; см. *Поддельный объект*, Fake Object, с. 565). Если за сохранение конфигурации отвечает тест, решение выглядит еще проще: следует сократить использование сохраняющихся ссылок на конфигурацию.

Решение проблемы медленных тестов

Основным недостатком использования *постоянной новой тестовой конфигурации* (Persistent Fresh Fixture) является скорость, точнее, ее отсутствие. Файловые системы и базы данных работают с данными намного медленнее процессора и оперативной памяти. В результате при взаимодействии с базой данных тест работает намного медленнее, чем при работе только в оперативной памяти. Разница в скорости связана с тем, что часть тестовой конфигурации находится на жестком диске, но это не основная причина замедления. Настройка *новой тестовой конфигурации* (Fresh Fixture) в начале каждого теста и очистка после завершения обычно требуют больше обращений к диску, чем генерирует тестируемая система при доступе к тестовой конфигурации. В результате тесты с доступ-

пом к базе данных работают в 50 или в 100 (на два порядка!) раз дольше, чем тесты, которые полностью умещаются в оперативной памяти.

Обычной реакцией на замедление работы, связанное с использованием *постоянной новой тестовой конфигурации* (Persistent Fresh Fixture), является исключение накладных расходов на создание и очистку тестовой конфигурации через повторное использование тестовой конфигурации несколькими тестами. Предположив, что требуется пять операций доступа к диску на создание и удаление тестовой конфигурации и одна операция со стороны тестируемой системы, можно вычислить, что максимальная экономия от перехода к *общей тестовой конфигурации* (Shared Fixture) составит около одного порядка скорости (тесты будут работать в десять раз медленнее по сравнению с тестами в оперативной памяти). Конечно, в большинстве ситуаций это все еще слишком медленно и за такой результат придется заплатить высокую цену: тесты потеряют независимость. Это значит, что сразу появятся *взаимодействующие тесты* (Interacting Tests; см. *Нестабильный тест*, Erratic Test), *одинокий тест* (Lonely Test; см. *Нестабильный тест*, Erratic Test) и *неповторяемый тест* (Unrepeatable Test). И все это в дополнение к *медленным тестам* (Slow Test)!

Более правильным решением является отказ от использования хранящейся на диске базы данных. С небольшими усилиями можно заменить базу данных на диске *базой данных в памяти* (In-Memory Database; см. *Поддельный объект*, Fake Object) или *поддельной базой данных* (Fake Database). Лучше принять это решение на ранних этапах разработки, пока затраты на реализацию еще не высоки. Да, может возникнуть ряд проблем, например обработка хранимых процедур, но и их можно решить за обозримое время.

Конечно, подобная тактика является не единственным решением в борьбе с *медленными тестами* (Slow Test). Во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351 рассматриваются и другие возможные стратегии.

Управление общими тестовыми конфигурациями

Управление общими тестовыми конфигурациями во многом напоминает управление *постоянными новыми тестовыми конфигурациями* (Persistent Fresh Fixture), за исключением сознательно принятого решения не очищать тестовую конфигурацию после каждого теста и использовать ее повторно в последующих тестах (рис. 9.2). Это означает, что появились две особенности. Во-первых, тестовая конфигурация должна быть доступна из других тестов. Во-вторых, необходим механизм запуска генерации и очистки тестовой конфигурации.

Доступ к общей тестовой конфигурации

Вне зависимости от метода создания *общей тестовой конфигурации* (Shared Fixture) тесты должны иметь возможность находить используемую тестовую конфигурацию. Возможные варианты зависят от природы тестовой конфигурации. Если тестовая конфигурация хранится в базе данных (наиболее распространенный вариант общей конфигурации), тесты могут получать прямые ссылки на объекты конфигурации, если есть информация о базе данных. Может возникнуть соблазн воспользоваться *фиксированными значениями* (Hard-Coded Value; см. *Точное значение*, Literal Value, с. 718) при поиске в базе данных и доступе к объекту тестовой конфигурации. Практически всегда это очень пло-

хая идея, так как она приводит к тесному связыванию между тестами и реализацией тестовой конфигурации, а также к снижению документированности (см. *Непонятный тест*, Obscure Test, с. 230). Чтобы избежать таких потенциальных проблем, для доступа к тестовой конфигурации можно воспользоваться *методами поиска* (Finder Method; см. *Вспомогательный метод теста*, Test Utility Method, с. 610) с описательными именами. Имена *методов поиска* (Finder Method) могут напоминать имена *методов создания* (Creation Method), но в результате вызова возвращаются ссылки на существующие объекты тестовой конфигурации, а не создаются новые объекты.

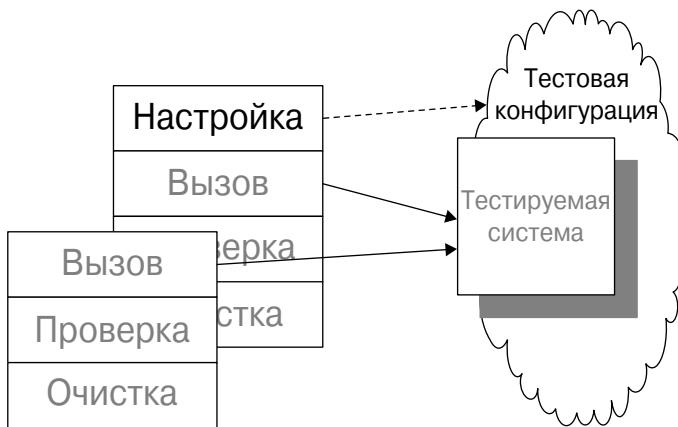


Рис. 9.2. Общая тестовая конфигурация (Shared Fixture) и два использующих ее тестовых метода (Test Method). Общая тестовая конфигурация (Shared Fixture) создается один раз и используется несколькими тестами, которые в результате намеренно или ненамеренно взаимодействуют друг с другом. Обратите внимание на отсутствие фазы настройки тестовой конфигурации во втором teste

Если тестовая конфигурация хранится в оперативной памяти, появляется целый набор решений. Если все совместно использующие конфигурацию тесты находятся в пределах одного *класса теста* (Testcase Class), для хранения ссылки на тестовую конфигурацию можно использовать переменную класса. Если переменная имеет описательное имя, читатель теста сможет быстро определить предварительные условия теста. Еще одной альтернативой является использование *метода поиска* (Finder Method).

Если тестовая конфигурация должна совместно использоваться несколькими *классами теста* (Testcase Class), потребуется более сложная техника. Конечно, можно позволить одному классу объявить переменную класса со ссылкой на конфигурацию, а другим тестам позволить получать доступ к этой переменной. К сожалению, такой подход создает нежелательную связность между тестами. Еще одна альтернатива предполагает перенос объявления в широкоизвестный объект, а именно — в *реестр тестовых конфигураций* (Test Fixture Registry; см. *Вспомогательный класс теста*, Test Helper, с. 651). Таким реестром может быть база данных с тестами или простой класс. Различные фрагменты тестовой конфигурации могут предоставляться через отдельные переменные класса или через *методы поиска* (Finder Method). Если *реестр тестовых конфигураций* (Test Fixture Registry) имеет один *метод поиска* (Finder Method), знающий, как получить доступ к объекту, но не хранящий ссылки, он называется *вспомогательным классом теста* (Test Helper).

Создание общей тестовой конфигурации

Для совместного использования общая тестовая конфигурация должна быть создана еще до вызова *тестовых методов* (Test Method). Генерация должна происходить сразу перед выполнением логики тестового метода, сразу перед запуском всего набора тестов или в любой момент до этих событий (рис. 9.3). В результате мы подошли к базовым шаблонам создания *общей тестовой конфигурации* (Shared Fixture).

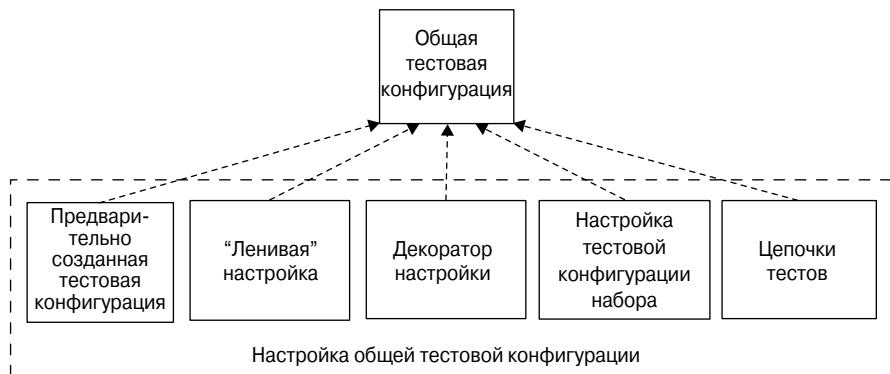


Рис. 9.3. Множество способов управления общей тестовой конфигурацией (Shared Fixture). Общая тестовая конфигурация (Shared Fixture) может создаваться в различные моменты; выбор конкретного варианта зависит от количества тестов, повторно использующих конфигурацию, и от количества обращений к конфигурации

Если вас устраивает идея создания тестовой конфигурации при первом обращении теста, можете воспользоваться “ленивой” настройкой (Lazy Setup, с. 460) в методе `setUp` соответствующего класса теста (Testcase Class), что приведет к настройке тестовой конфигурации в момент запуска первого теста. Последующие тесты обнаружат существующую конфигурацию и будут использовать ее повторно. Так как не существует индикатора запуска последнего теста в наборе (или в наборе наборов (Suite of Suites); см. *Объект набора тестов*, Test Suite Object, с. 414), неизвестно, в какой момент необходимо очищать тестовую конфигурацию. Это может привести к появлению *неповторяемых тестов* (Unrepeatable Test), так как прежняя тестовая конфигурация будет существовать в момент следующего запуска тестов (это зависит от способа доступа со стороны различных тестов).

Если тестовая конфигурация должна использоваться более широко, в начало набора тестов можно включить тест с *настройкой тестовой конфигурации* (Fixture Setup Testcase). Это специальный случай *цепочки тестов* (Chained Tests), страдающий от той же проблемы, что и “ленивая” настройка (Lazy Setup): неизвестен момент, когда необходимо удалять тестовую конфигурацию. Из-за зависимости от порядка запуска тестов в наборе лучше всего этот подход совместить с *перечислением тестов* (Test Enumeration, с. 425).

Если тестовая конфигурация должна удаляться после завершения работы набора тестов, необходим механизм управления, который будет сообщать о завершении последнего теста. Несколько реализаций xUnit поддерживают концепцию метода `setUp`, который запускается один раз после создания набора тестов на основе *класса теста* (Testcase Class). Такому методу *настройки тестовой конфигурации набора* (Suite Fixture Setup, с. 465) соответствует метод `tearDown`, который вызывается после завершения работы

последнего *тестового метода* (Test Method) (рассматривайте эту конструкцию как встроенный декоратор для единственного *класса теста*, Testcase Class). После этого можно гарантировать, что новая тестовая конфигурация будет создаваться при каждом запуске набора тестов. Тестовая конфигурация удаляется после завершения всех тестов, что решает проблему *неповторяемых тестов* (Unrepeatable Test); но не решается проблема *взаимодействующих тестов* (Interacting Tests). Такая возможность может быть добавлена в качестве расширения в любую реализацию xUnit. Если подобное решение не поддерживается или тестовая конфигурация должна совместно использоваться несколькими *классами теста* (Testcase Class), можно вернуться к использованию *декоратора настройки* (Setup Decorator, с. 471), который окружает выполнение набора тестов вызовами методов `setUp` и `tearDown`. Самым большим недостатком *декоратора настройки* (Setup Decorator) является невозможность независимого запуска зависящих от него тестов. Все такие тесты являются *одинокими тестами* (Lonely Test).

Последним вариантом является создание тестовой конфигурации еще до запуска тестов, т.е. применение *предварительно созданной тестовой конфигурации* (Prebuilt Fixture, с. 454). Такой подход обеспечивает максимальное количество вариантов при настройке тестовой конфигурации, так как эта процедура может выполняться за пределами xUnit. Например, тестовую конфигурацию можно создать вручную, с помощью сценариев базы данных, скопировав “эталонную” базу данных или воспользовавшись программой генерации данных. Основным недостатком *предварительно созданной тестовой конфигурации* (Prebuilt Fixture) является *ручное вмешательство* (Manual Intervention, с. 287), если один из тестов окажется *неповторяемым тестом* (Unrepeatable Test). В результате *предварительно созданная тестовая конфигурация* (Prebuilt Fixture) часто используется в комбинации с *новой тестовой конфигурацией* (Fresh Fixture) для создания *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture).

Что дальше

Определившись со способами настройки и очистки тестовых конфигураций, можно обратить внимание на вызов testируемой системы и проверку ожидаемого результата с помощью *методов с утверждением* (Assertion Method). Более подробно этот процесс рассматривается в главе 10, “Проверка результатов”.

Глава 10

Проверка результатов

О чём идет речь в этой главе

В главе 8, “Управление временной тестовой конфигурацией”, и в главе 9, “Управление постоянными тестовыми конфигурациями”, рассматривались настройка и очистка тестовой конфигурации до и после вызова тестируемой системы соответственно. В этой главе рассматриваются различные варианты проверки правильности поведения тестируемой системы, включая вызов тестируемой системы и сравнение ожидаемого результата с полученным.

Создание самопроверяющихся тестов

Одной из ключевых характеристик автоматизированных средствами xUnit тестов является возможность создать *самопроверяющийся тест* (Self-Checking Test; см. *Цели автоматизации тестов*, с. 77). Эта характеристика делает их достаточно эффективными с точки зрения затрат, чтобы оправдать очень частый запуск. В большинстве реализаций xUnit предоставляется коллекция встроенных методов с утверждением (Assertion Method, с. 390) и документация по их использованию. На первый взгляд, все выглядит достаточно просто, но создание хорошего теста требует не просто вызова встроенных методов с утверждением (Assertion Method). Необходимо изучить ключевые техники создания понятных тестов и избежания дублирования тестового кода (Test Code Duplication, с. 254).

Основная сложность в кодировании утверждений заключается в получении доступа к информации, которая будет сравниваться с ожидаемым результатом. Именно здесь важны точки наблюдения; они обеспечивают окно для просмотра состояния или поведения тестируемой системы, позволяя передавать эту информацию в *метод с утверждением* (Assertion Method). Пользоваться точками наблюдения, доступными через синхронный вызов методов, достаточно просто. Точки наблюдения других типов могут вызвать сложности, что и делает автоматизированные тесты интересной темой для исследований.

Обычно (но не всегда) утверждения вызываются из *тестовых методов* (Test Method, с. 378) сразу после вызова тестируемой системы. Некоторые авторы тестов вставляют утверждения после фазы создания тестовой конфигурации, чтобы убедиться в правильности настройки конфигурации. Практически всегда такая практика приводит к появлению *непонятных тестов* (Obscure Test, с. 229), поэтому предпочтительнее создавать модуль-

ные тесты для *вспомогательных методов теста* (Test Utility Method, с. 611)¹. Некоторые стили тестирования требуют формализовать ожидания до вызова тестируемой системы. Более подробная информация об этом приведена в главе 11, “Использование тестовых двойников”. Примеры вызова *методов с утверждением* (Assertion Method) из *вспомогательных методов теста* (Test Utility Method) будут рассмотрены в этой главе.

Возможным (но редко используемым) местом вызова *методов с утверждением* (Assertion Method) является метод `tearDown`, который применяется при *неявной очистке* (Implicit Teardown, с. 534). Поскольку этот метод запускается для каждого теста независимо от результата (если метод `setUp` завершился успешно), нет ничего плохого в размещении утверждений именно здесь. С такой схемой связаны те же накладные расходы, что и с *неявной настройкой* (Implicit Setup, с. 450) тестовой конфигурации. Такой вызов менее заметен, но выполняется автоматически. Дополнительная информация о добавлении утверждений в метод `tearDown` во время *неявной очистки* (Implicit Teardown) суперкласса для обнаружения оставленных тестом объектов в базе данных приводится во врезке “Использование дельта-утверждений для обнаружения утечки данных” на с. 507.

Что проверять: состояние или поведение?

В целом, автоматизированные тесты предназначены для проверки поведения тестируемой системы. Некоторые аспекты поведения можно проверить непосредственно. В этом случае в качестве хорошего примера выступают возвращаемые значения функций. Другие аспекты поведения проще проверяются опосредованно, через наблюдение за состоянием некоторых объектов. Фактическое поведение тестируемой системы можно проверить двумя способами.

1. Можно проверить состояние объектов, затрагиваемых тестируемой системой, извлекая каждое состояние через точку наблюдения и используя утверждения для сравнения полученной информации с ожидаемым состоянием.
2. Поведение тестируемой системы можно проверить непосредственно через точки наблюдения, вставленные между тестируемой системой и вызываемым компонентом. При этом наблюдаемое взаимодействие (список вызываемых методов) можно сравнить с ожидаемым списком вызовов.

Проверка состояния (State Verification, с. 485) осуществляется через утверждения и является более простой задачей. *Проверка поведения* (Behavior Verification, с. 490) сложнее в реализации и основана на утверждениях, проверяющих состояние.

¹ Единственным исключением является использование *общей тестовой конфигурации* (Shared Fixture, с. 350); в такой ситуации может потребоваться применение *сторожевого утверждения* (Guard Assertion, с. 510) для документирования требований теста к конфигурации и неудачного завершения теста при повреждении тестовой конфигурации. Также это можно сделать внутри *метода поиска* (Finder Method; см. *Вспомогательный метод теста*, Test Utility Method), который используется для получения объектов внутри *общей тестовой конфигурации* (Shared Fixture, с. 350).

Проверка состояния

“Нормальный” способ проверки получения ожидаемого результата называется *проверкой состояния* (State Verification, с. 484). Сначала вызывается тестируемая система, после чего с помощью утверждений рассматривается ее состояние. Кроме того, можно проверить все, что возвращает тестируемая система в виде результатов вызовов методов. Наиболее заметным является то, что не делается: в тестируемую систему не встраиваются дополнительные средства мониторинга для отслеживания взаимодействий с другими компонентами системы. Иначе говоря, проверяются только непосредственные выходные данные и в качестве точки наблюдения применяются только непосредственные вызовы методов.

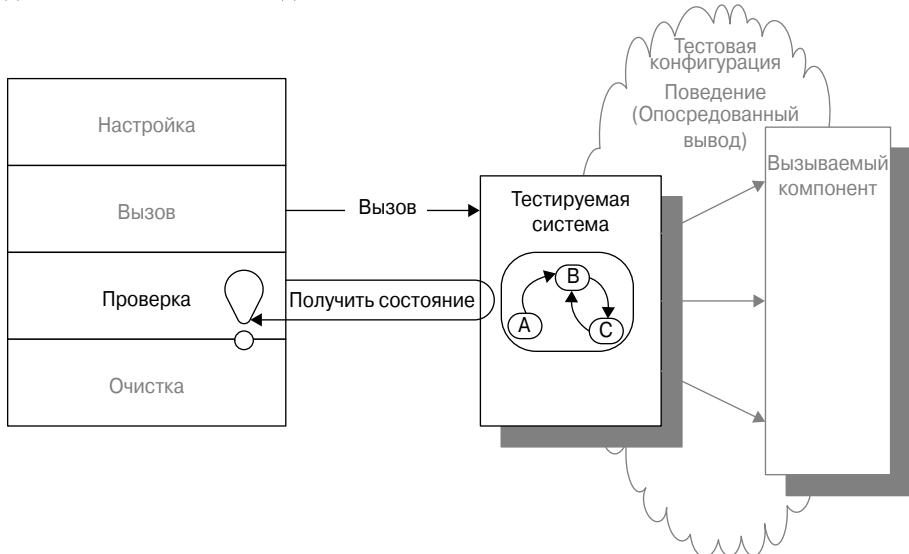


Рис. 10.1. Проверка состояния (State Verification). При проверке состояния утверждается, что после вызова тестируемая система и возвращаемые объекты находятся в ожидаемом состоянии. На “происходящее за кулисами” тест внимания не обращает

Проверка состояния (State Verification) может быть реализована двумя немного отличающимися способами. *Процедурная проверка состояния* (Procedural State Verification; см. *Проверка состояния*, State Verification) подразумевает создание последовательности утверждений, анализирующих конечное состояние тестируемой системы и сравнивающих его с ожидаемым. *Ожидаемый объект* (Expected Object; см. *Проверка состояния*, State Verification) описывает ожидаемое состояние таким образом, чтобы обеспечить сравнение одним вызовом *метода с утверждением* (Assertion Method). Такой подход минимизирует *дублирование тестового кода* (Test Code Duplication) и делает тесты более понятными (дополнительная информация приводится ниже в этой главе). Обе стратегии могут использовать как “встроенные”, так и *специальные утверждения* (Custom Assertion, с. 495).

Использование встроенных утверждений

Предоставленные инфраструктурой тестирования утверждения используются для описания “как должно быть” и обнаружения ситуаций, когда это не так! Но простое использование встроенных утверждений является лишь небольшой частью процесса.

Утверждения с описанием условий истины являются самой простой формой проверки результата. В большинстве реализаций xUnit предоставляется набор различных *методов с утверждением* (Assertion Method), включая следующие утверждения.

- *Утверждение с заявленным результатом* (Stated Outcome Assertion; см. *Метод с утверждением*, Assertion Method), например `assertTrue(aBooleanExpression)`.
- *Простое утверждение равенства* (Equality Assertion), например `assertEquals(expected, actual)`.
- *Нечеткое утверждение равенства* (Fuzzy Equality Assertion), например `assertEquals(expected, actual, tolerance)`, которое используется для сравнения чисел с плавающей точкой.

Конечно, язык, на котором пишутся тесты, оказывает некоторое влияние на утверждения. В пакетах JUnit, SUnit, CppUnit, NUnit и CsUnit большинство *утверждений равенства* (Equality Assertion) принимают в качестве параметров два объекта типа `Object`. Некоторые языки поддерживают “дополнение” типов параметров метода для получения различных реализаций утверждений в зависимости от типа передаваемых объектов. В некоторых языках, например в С, объекты не поддерживаются, поэтому сравниваются только значения.

Стоит обратить внимание на несколько особенностей, связанных с использованием *методов с утверждением* (Assertion Method). Естественно, максимальный приоритет имеет проверка истинности условий. Чем лучше утверждения, тем тоньше *страховочная сеть* (Safety Net, с. 79) и выше уверенность в коде. Следующей по значению является ценность утверждений как документации. Каждый тест должен очень ясно описывать, “когда система находится в состоянии S1 и выполняется операция X, должен возвращаться результат R и система должна переходить в состояние S2”. Состояние S1 достигается при настройке тестовой конфигурации. “Выполняется операция X” соответствует вызову тестируемой системы. “Результат R” и “система должна переходить в состояние S2” реализуются в виде утверждений. Это значит, что утверждения должно доступно описывать “R” и “S2”.

Также стоит обратить внимание, на сообщение, которое выводится при неудачном завершении теста. Сообщение должно содержать достаточный объем информации, чтобы позволить идентифицировать проблему². Таким образом, практически всегда необходимо включать *сообщение для утверждения* (Assertion Message, с. 398) в качестве значения необязательного параметра `message` (если такой параметр поддерживается в используемой реализации xUnit). Эта тактика позволяет избежать игры в *рулетку утверждений* (Assertion Roulette, с. 264), когда нельзя определить сработавшее утверждение без интерактивного запуска теста. При использовании сообщений ошибки во время интеграции

² В книге [TDD-APG] Дэйв Астелс заявляет, что никогда не использовал Eclipse Debugger, создавая примеры кода, так как утверждения всегда указывали на причину ошибки. К такому результату нужно стремиться всегда!

намного проще воспроизвести и исправить. Кроме того, упрощается диагностика некорректно работающих тестов, так как доступно описание того, что должно было произойти, а фактический результат показывает, что произошло.

При использовании *утверждения с заявленным результатом* (Stated Outcome Assertion), например `assertTrue` из пакета JUnit, сообщения об ошибке содержат недостаточно информации (например, “Assertion failed”). Вывод утверждения можно сделать более конкретным, используя *сообщение с описанием аргументов* (Argument-Describing Message; см. *Сообщение для утверждения*, Assertion Message), создаваемое с включением полезных фрагментов данных в сообщение. Для начала имеет смысл включать в аргумент *сообщения для утверждения* (Assertion Message) каждое значение.

Дельта-утверждения

При использовании *общей тестовой конфигурации* (Shared Fixture, с. 350) могут появляться *взаимодействующие тесты* (Interacting Tests; см. *Нестабильный тест*, Erratic Test, с. 267), поскольку каждый тест добавляет объекты/записи в базу данных и нельзя с уверенностью утверждать, что должно находиться в базе данных после вызова тестируемой системы. Избежать этой неопределенности можно, используя *дельта-утверждение* (Delta Assertion, с. 505) для проверки только новых объектов или записей. При таком подходе в начале теста создается “моментальный снимок” соответствующих таблиц/классов. После этого они исключаются из коллекции таблиц/классов, выводимых в конце теста перед сравнением с *ожидаемым объектом* (Expected Object). Хотя подобная тактика может привести к увеличению сложности тестов, дополнительную сложность можно переработать в *специальное утверждение* (Custom Assertion) и/или *метод проверки* (Verification Method; см. *Специальное утверждение*, Custom Assertion). Моментальный снимок “до” может создаваться внутри *тестового метода* (Test Method) или внутри метода `setUp`, если настройка тестовой конфигурации происходит до вызова *тестовых методов* (Test Method), например *неявная настройка* (Implicit Setup), *общая тестовая конфигурация* (Shared Fixture) или *предварительно созданная тестовая конфигурация* (Prebuilt Fixture, с. 454).

Внешняя проверка результата

Выше рассматривалась проверка ожидаемых результатов “в памяти”. На самом деле, возможен и другой подход, предполагающий сохранение ожидаемых и фактических результатов в файлах с поиском отличий средствами внешней программы. В результате получается *специальное утверждение* (Custom Assertion), выполняющее глубокую проверку двух ссылок на файлы. Часто программе сравнения требуется информация о фрагментах файлов, которые должны игнорироваться (как вариант эти фрагменты должны быть удалены из файлов предварительно), что в результате дает *нечеткое утверждение равенства* (Fuzzy Equality Assertion).

Внешняя проверка результата хорошо подходит для автоматизации приемочных тестов и регрессионного тестирования приложений, в которые не вносились значительные изменения. Основным недостатком такого подхода является появление *тайного гостя* (Mystery Guest; см. *Непонятный тест*, Obscure Test), с точки зрения читателя теста, так как ожидаемые результаты не видимы внутри теста. Чтобы обойти эту проблему, можно записать содержимое ожидаемого файла самим тестом, сделав содержимое види-

мым читателю тестов. Этот шаг имеет смысл, если объем данных достаточно невелик (еще один довод в пользу применения *минимальной тестовой конфигурации*, Minimal Fixture, с. 336).

Проверка поведения

Проверять поведение сложнее, чем состояние, из-за динамической природы поведения. Приходится “ловить” тестируемую систему “в движении”, в процессе генерации опосредованного вывода в направлении вызываемых объектов (рис. 10.2). Имеет смысл рассматривать два варианта проверки поведения: *процедурная проверка поведения* (Procedural Behavior Verification) и *ожидаемое поведение* (Expected Behavior). Оба требуют механизма доступа к исходящим вызовам методов со стороны тестируемой системы (опосредованный вывод). Эти и другие случаи применения *тестовых двойников* (Test Double, с. 538) более подробно рассматриваются в главе 11, “Использование тестовых двойников”.

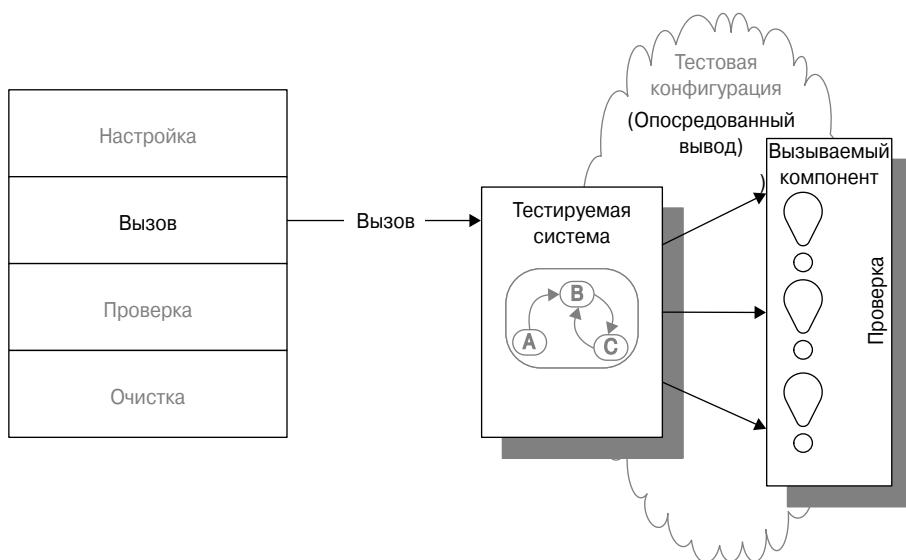


Рис. 10.2. Проверка поведения (Behavior Verification). При проверке поведения основное внимание уделяется опосредованному выводу (исходящим интерфейсам) тестируемой системы. Обычно это означает замену вызываемого компонента тем, что обеспечивает наблюдение и проверку исходящих вызовов

Процедурная проверка поведения

При процедурной проверке записывается поведение тестируемой системы в процессе выполнения и сохранения данных для последующего извлечения. После этого тест сравнивает каждый вывод тестируемой системы (один за другим) с соответствующим ожидаемым выводом. Таким образом, при процедурной проверке поведения тест выполняет процедуру (набор операций) для проверки поведения.

```

public void testRemoveFlightLogging_recordingTestStub()
    throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnUnregFlight();
    FlightManagementFacade facade =
        new FlightManagementFacadeImpl();
    // Настройка тестового двойника (Test Double)
    AuditLogSpy logSpy = new AuditLogSpy();
    facade.setAuditLog(logSpy);
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertEquals("number of calls", 1,
        logSpy.getNumberofCalls());
    assertEquals("action code",
        Helper.REMOVE_FLIGHT_ACTION_CODE,
        logSpy.getActionCode());
    assertEquals("date", helper.getTodaysDateWithoutTime(),
        logSpy.getDate());
    assertEquals("user", Helper.TEST_USER_NAME,
        logSpy.getUser());
    assertEquals("detail",
        expectedFlightDto.getFlightNumber(),
        logSpy.getDetail());
}

```

Ключевой задачей процедурной проверки поведения является захват и сохранение информации для последующего использования тестом. Эта задача решается за счет настройки тестируемой системы на использование *тестового агента* (Test Spy, с. 552) или *тестового шунта* (Self Shunt; см. *Фиксированный тестовый двойник*, Hard-Coded Test Double, с. 581)³ вместо вызываемого класса. После вызова тестируемой системы тест получает записи поведения и проверяет их с помощью утверждений.

Спецификация ожидаемого поведения

Если есть возможность создать *ожидаемый объект* (Expected Object) и сравнить его с фактическим объектом, который вернула тестируемая система, для проверки состояния, можно ли сделать что-то подобное для проверки поведения? Да, можно. *Ожидаемое поведение* (Expected Behavior) часто используется вместе с *тестом с пересечением уровней* (Layer-Crossing Test) для проверки опосредованного вывода объекта или компонента. Перед вызовом тестируемой системы создается *подставной объект* (Mock Object, с. 558), который настраивается на ожидаемые вызовы методов со стороны тестируемой системы.

```

public void testRemoveFlight_JMock() throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnonRegFlight();
    FlightManagementFacade facade =
        new FlightManagementFacadeImpl();
    // Настройка подставного объекта
    Mock mockLog = mock(AuditLog.class);
    mockLog.expects(once()).method("logMessage")
        .with(eq(helper.getTodaysDateWithoutTime()));
}

```

³ *Тестовый агент* (Test Spy), встроенный в класс *теста* (Testcase Class, с. 401).

```

        eq(Helper.TEST_USER_NAME),
        eq(Helper.REMOVE_FLIGHT_ACTION_CODE),
        eq(expectedFlightDto.getFlightNumber()) );
// Установка подставного объекта
facade.setAuditLog((AuditLog) mockLog.proxy());
// Вызов
facade.removeFlight(expectedFlightDto.getFlightNumber());
// Проверка
// JMock вызывает метод verify() автоматически
}

```

Сокращение дублирования кода

Одним из наиболее распространенных запахов является *дублирование тестового кода* (Test Code Duplication). С каждым написанным тестом повышается вероятность дублирования, но особенно опасным является создание новых тестов методом копирования и вставки. Может показаться, что дублирование тестового кода не так опасно, как дублирование кода продукта. *Дублирование тестового кода* (Test Code Duplication) опасно, если приводит к появлению других запахов, например “хрупкого” теста (Fragile Test, с. 277), “хрупкой” тестовой конфигурации (Fragile Fixture, с. 284) или *высокой стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300), из-за слишком большого количества тестов, тесно связанных со *стандартной тестовой конфигурацией* (Standard Fixture, с. 338) или программным интерфейсом тестируемой системы. Кроме того, *дублирование тестового кода* (Test Code Duplication) иногда может быть симптомом другой проблемы — сокрытия намерений теста за слишком большим объемом кода (например, *непонятный тест*, Obscure Test).

В логике проверки результата *дублирование тестового кода* (Test Code Duplication) обычно проявляется в виде повторений утверждений. Существует несколько методик, позволяющих сократить количество утверждений в таких случаях:

- *ожидаемый объект* (Expected Object);
- *специальное утверждение* (Custom Assertion);
- *метод проверки* (Verification Method).

Ожидаемый объект

Часто возникает необходимость проверять утверждения относительно разных полей одного объекта. Если приходится повторять такую группу утверждений (несколько раз в одном тесте или в нескольких тестах), следует искать способы сокращения *дублирования тестового кода* (Test Code Duplication). Ниже приведен *тестовый метод* (Test Method), сравнивающий несколько атрибутов одного объекта. Скорее всего, практически такая же последовательность утверждений используется в нескольких других тестовых методах.

```

public void testInvoice_addLineItem7() {
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();

```

```

LineItem actual = (LineItem)lineItems.get(0);
assertEquals(expItem.getInv(), actual.getInv());
assertEquals(expItem.getProd(), actual.getProd());
assertEquals(expItem.getQuantity(), actual.getQuantity());
}

```

Самой очевидной альтернативой является использование единственного *утверждения равенства* (Equality Assertion) для сравнения двух объектов вместо нескольких утверждений для сравнения этих же объектов поле за полем. Если значения хранятся в отдельных переменных, возможно, потребуется создать новый объект соответствующего класса и инициализация его свойств значениями из этих переменных. Такой способ работает при возможности в любой момент создавать *ожидаемый объект* (Expected Object) и при наличии метода `equals`, который сравнивает только интересующие свойства.

```

public void testInvoice_addLineItem8() {
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    LineItem actual = (LineItem)lineItems.get(0);
    assertEquals("Item", expItem, actual);
}

```

Но что если не нужно сравнивать все свойства объекта или метод `equals` проверяет равенство идентификаторов, а не содержимого? Что если необходимо *специфическое для теста равенство* (test-specific equality)? Что если невозможно создать *ожидаемый объект* (Expected Object) из-за отсутствия конструктора? В таком случае можно реализовать *специальное утверждение* (Custom Assertion), в котором определено интересующее равенство, или реализовать специфическое для теста равенство в методе `equals` класса *ожидаемого объекта* (Expected Object), который передается в *метод с утверждением* (Assertion Method). Это необязательно должен быть класс интересующего объекта. Достаточно, чтобы он содержал реализацию метода `equals`, сравнивающего экземпляр с интересующим объектом. Таким образом, это может быть простой *объект передачи данных* (data transfer object) [CJ2EEP] или *связанный с тестом подкласс* (Test-Specific Subclass, с. 591) настоящего класса с переопределенным методом `equals`.

Некоторые разработчики стараются не полагаться на метод `equals`, реализованный в тестируемой системе при создании утверждений, так как эта реализация может измениться, приводя к неудачному завершению зависящих от нее тестов (или к игнорированию важного отличия объектов друг от друга). Рекомендуется подходить к этому решению с практической точки зрения. Если использование метода `equals` из тестируемой системы кажется осмысленным, используйте именно его. Если необходима другая реализация, определите *специальное утверждение* (Custom Assertion) или класс для *ожидаемого объекта* (Expected Object). Кроме того, задайте себе вопрос: “Насколько сложно будет изменить стратегию, если метод `equals` в будущем будет модифицирован?” Например, в статически типизированных языках, поддерживающих доопределение функций для других типов параметров (как в Java), можно создать *специальное утверждение* (Custom Assertion), которое будет использоваться вместо принятой по умолчанию реализации для параметров других типов. Обычно добавление такого кода в готовый проект после изменения метода `equals` проблем не вызывает.

Специальные утверждения

Специальное утверждение (Custom Assertion) создается разработчиком с учетом особенностей предметной области. Специальные утверждения скрывают процедуру проверки результатов за описательным именем, делая логику проверки более доступной для восприятия. Кроме того, специальные утверждения помогают бороться с *непонятными тестами* (Obscure Test), скрывая большой объем отвлекающего внимание кода. Еще одним преимуществом переноса кода в *специальное утверждение* (Custom Assertion) является возможность создания модульных тестов для логики проверки. После этого утверждения перестают быть *нетестируемым кодом теста* (Untestable Test Code; см. *Сложный в тестировании код*, Hard-to-Test Code, с. 251)!

```
static void assertLineItemsEqual(
    String msg, LineItem exp, LineItem act) {
    assertEquals (msg+" Inv", exp.getInv(), act.getInv());
    assertEquals (msg+" Prod", exp.getProd(), act.getProd());
    assertEquals (msg+" Quan", exp.getQuantity(), act.getQuantity());
}
```

Существует два способа создания *специального утверждения* (Custom Assertion).

1. Через рефакторинг существующего кода тестов для сокращения *дублирования тестового кода* (Test Code Duplication).
2. Через запись вызовов несуществующих *методов с утверждением* (Assertion Method) в процессе написание тестов. Тело каждого метода разрабатывается после определения полного набора необходимых *специальных утверждений* (Custom Assertion).

Второй вариант позволяет напоминать себе об ожидаемом результате вызова тестируемой системы, даже если код проверки еще не написан. В любом случае определение набора *специальных утверждений* (Custom Assertion) является первым этапом создания языка *высокого уровня* (Higher-Level Language, с. 95) для описания тестов.

В первом варианте достаточно использовать рефакторинг *выделения метода* (Extract Method) [Ref] над повторяющимися утверждениями. Используемые существующей логикой проверки объекты передаются в новые методы в качестве аргументов. Кроме того, в качестве параметра передается *сообщение для утверждения* (Assertion Message), что позволяет идентифицировать разные вызовы одного и того же *метода с утверждением* (Assertion Method).

Метод проверки с описанием результата

Жестокий рефакторинг тестового кода породил еще один способ — “описывающий результат” *метод проверки* (Verification Method). Предположим, обнаружена группа тестов, в которых разделы вызова тестируемой системы и проверки результата совпадают, а отличается только часть настройки тестовой конфигурации. Если применить рефакторинг *выделение метода* (Extract Method) к общей части и присвоить ей осмысленное имя, станет меньше кода, тест станет более понятным и будет получена пригодная для тестирования логика проверки!

```

void assertInvoiceContainsOnlyThisLineItem(
    Invoice inv,
    LineItem expItem) {
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    assertLineItemsEqual("",expItem, actual);
}

```

Основным отличием *метода проверки* (Verification Method) от специального утверждения (Custom Assertion) является взаимодействие первого с тестируемой системой (обычно через вызовы методов). Еще одним отличием является наличие стандартной сигнатуры *утверждения равенства* (Equality Assertion) в *специальном утверждении* (Custom Assertion): `assertSomething(message, expected, actual)`. С другой стороны, *метод проверки* (Verification Method) может иметь произвольный набор параметров, так как дополнительные параметры могут передаваться тестируемой системе. Фактически метод проверки — это что-то среднее между *специальным утверждением* (Custom Assertion) и *параметризованным тестом* (Parameterized Test, с. 618).

Параметризованный и управляемый данными тест

В выделении общих фрагментов тестов можно пойти еще дальше. Если логика создания тестовой конфигурации совпадает, но использует разные данные, можно выделить общие фазы настройки тестовой конфигурации, вызова тестируемой системы и проверки результата в новый *параметризованный тест* (Parameterized Test). Он не вызывается *инфраструктурой автоматизации тестов* (Test Automation Framework, с. 332) автоматически, так как требует передачи аргументов. Вместо этого для каждого теста определяются простые *тестовые методы* (Test Method), которые, в свою очередь, вызывают *параметризованный тест* (Parameterized Test) и передают ему данные, обеспечивающие уникальность теста. Эти данные могут потребоваться при настройке тестовой конфигурации, вызове тестируемой системы и проверке ожидаемого результата. В следующих тестах метод `generateAndVerifyHtml` является *параметризованным тестом* (Parameterized Test).

```

def test_extref
    sourceXml = "<extref id='abc' />"
    expectedHtml = "<a href='abc.html'>abc</a>"
    generateAndVerifyHtml(sourceXml,expectedHtml,"<extref>")
end
def test_testterm_normal
    sourceXml = "<testterm id='abc' />"
    expectedHtml = "<a href='abc.html'>abc</a>"
    generateAndVerifyHtml(sourceXml,expectedHtml,"<testterm>")
end
def test_testterm_plural
    sourceXml = "<testterms id='abc' />"
    expectedHtml = "<a href='abc.html'>abcs</a>"
    generateAndVerifyHtml(sourceXml,expectedHtml,"<plural>")
end

```

В *управляемом данными тесте* (Data-Driven Test, с. 322) код теста универсален и может вызываться инфраструктурой тестирования. Аргументы извлекаются из файла данных в процессе работы теста. *Управляемый данными тест* (Data-Driven Test) можно рассматри-

вать как вывернутый наизнанку *параметризованный тест* (Parameterized Test). *Тестовый метод* (Test Method) передает данные в *параметризованный тест* (Parameterized Test). *Управляемый данными тест* (Data-Driven Test) является *тестовым методом* (Test Method) и читает данные из файла самостоятельно. Содержимое файла представляет собой язык *высокого уровня* (Higher-Level Language) для тестирования. *Управляемый данными тест* (Data-Driven Test) является интерпретатором этого языка. Такая схема является эквивалентом инфраструктуры Fit, реализованным средствами xUnit. Простой пример *управляемого данными теста* (Data-Driven Test) приведен в следующем коде на языке Ruby.

```
def test_crossref
  executeDataDrivenTest "CrossrefHandlerTest.txt"
end
def executeDataDrivenTest filename
  dataFile = File.open(filename)
  dataFile.each_line do | line |
    desc, action, part2 = line.split(",")
    sourceXml, expectedHtml, leftOver = part2.split(",")
    if "crossref"==action.strip
      generateAndVerifyHtml sourceXml, expectedHtml, desc
    else # new "verbs" go before here as elsif's
      report_error("unknown action" + action.strip)
    end
  end
end
```

Ниже представлен файл с разделенными запятой полями данных, которые будут за-прашиваться *управляемым данными тестом* (Data-Driven Test).

```
ID, Action, SourceXml, ExpectedHtml
Extref,crossref,<extref id='abc' />,<a href='abc.html'>abc</a>
TTerm,crossref,<testterm id='abc' />,<a href='abc.html'>abc</a>
TTerms,crossref,<testterms id='abc' />,<a href='abc.html'>abcs</a>
```

Как избежать условной логики в тестах

Еще одним свойством тестов, которого стоит избегать, является условная логика. *Условная логика теста* (Conditional Test Logic, с. 243) заставляет тест вести себя по-разному в разных ситуациях. При этом снижается доверие к тестам, так как код самого теста тестируемому не поддается. Почему доверие важно? Потому что единственным способом проверки *тестового метода* (Test Method) является модификация тестируемой системы для генерации обнаруживаемой тестом ошибки. Если *тестовый метод* (Test Method) имеет несколько ветвей выполнения, придется обеспечивать правильность каждой из них. Намного проще иметь одну последовательность выполнения внутри теста. Вот причины, по которым в тесты может попадать условная логика:

- необходимо избежать вызова некоторых утверждений, так как они не имеют смысла с учетом информации, полученной на текущий момент (обычно определено условие неудачного завершения);
- необходимо учсть различные варианты полученного результата при сравнении с ожидаемым результатом;

- делается попытка повторного использования *тестового метода* (Test Method) в различных ситуациях (фактически несколько тестов собираются в один *тестовый метод*, Test Method).

Проблема *условной логики теста* (Conditional Test Logic) в первых двух вариантах заключается в сложности кода теста для чтения и в скрытии случаев повторного использования тестовых методов через *гибкий тест* (Flexible Test; см. *Условная логика теста*, Conditional Test Logic). Последняя “причина” является просто плохой идеей, в пользу которой нет ни одного довода. Существуют более подходящие способы повторного использования тестовой логики, не подразумевающие повторное использование *тестовых методов* (Test Method). Одни из них уже рассматривались ранее (см. раздел “Сокращение дублирования кода”). Другие способы рассматриваются в остальных главах книги. В этом случае просто скажите “Нет”!

Хорошая новость заключается в том, что все допустимые варианты *условной логики теста* (Conditional Test Logic) можно достаточно легко удалить из существующих тестов.

Удаление операторов `if`

Что делать, если утверждение нельзя вызывать, так как оно приведет к ошибке в работе теста, но нужно получить более осмысленное сообщение о неудачном завершении? Обычно такое утверждение помещается внутрь конструкции `if`, как показано в следующем примере. К сожалению, такой подход приводит к появлению *условной логики теста* (Conditional Test Logic), чего стоит избегать, так как при каждом запуске теста должен работать один и тот же код.

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem expected =
        new LineItem(invoice, product, 5,
                     new BigDecimal("30"),
                     new BigDecimal("69.96"));
    LineItem actItem = (LineItem) lineItems.get(0);
    assertEquals("invoice", expected, actItem);
} else {
    fail("Счет должен иметь только один пункт");
}
```

Более предпочтительным решением является использование *сторожевого утверждения* (Guard Assertion, с. 510), показанного ниже в модифицированной версии кода.

```
List lineItems = invoice.getLineItems();
assertEquals("number of items", lineItems.size(), 1);
LineItem expected =
    new LineItem(invoice, product, 5,
                 new BigDecimal("30"),
                 new BigDecimal("69.96"));
LineItem actItem = (LineItem) lineItems.get(0);
assertEquals("invoice", expected, actItem);
```

Положительным эффектом *сторожевого утверждения* (Guard Assertion) является возможность предотвратить вызов утверждения, которое приведет к ошибке теста, без введения *условной логики в тест* (Conditional Test Logic). Через некоторое время такие

утверждения становятся очевидными и интуитивно понятными. Может даже возникнуть соблазн вставлять утверждения относительно предварительных условий метода в код продукта!

Исключение циклов

Условная логика теста (Conditional Test Logic) может проявляться в виде циклов, проверяющих коллекции, которые возвращает тестируемая система. Добавление цикла непосредственно в *тестовый метод* (Test Method) вызывает три проблемы.

- Создается *нетестируемый код теста* (Untestable Test Code), так как цикл является частью теста и не может контролироваться другими тестами.
- Тест превращается в *непонятный тест* (Obscure Test), так как циклы скрывают намерения разработчика.
- Цикл внутри теста может привести к появлению запаха проекта *разработчики не пишут тесты* (Developers Not Writing Tests, с. 298), так как сложность генерации циклов может помешать разработчикам создавать *самопроверяющиеся тесты* (Self-Checking Test).

Более правильным решением будет делегирование такой логики во *вспомогательный метод теста* (Test Utility Method) с описательным именем. Такой метод можно тестиировать и использовать повторно.

Другие способы

В этом разделе рассматриваются другие способы написания простых для понимания тестов.

Разработка в порядке “извне вовнутрь”

Хорошим способом написания понятного кода является разработка “извне вовнутрь”. Это реализация идеи Стивена Коуви: “Начинайте, думая о завершении”. Для этого сначала следует написать последнюю строку функции или теста. Единственным смыслом существования функции является возвращаемое значение. Процедура создается ради побочных эффектов (модификации значений). Тест предназначен для проверки совпадения ожидаемого и полученного результатов (через утверждения).

Разработка в обратном порядке предполагает, что сначала записываются утверждения. В качестве параметров в утверждения передаются локальные переменные с описательными именами. После этого остается только написать код теста, присваивающий необходимые значения этим переменным. Поскольку хотя бы один аргумент утверждения должен быть получен от тестируемой системы, она должна вызываться из теста. Объявление и инициализация переменной после ее использования заставляет лучше понимать назначение переменной. Кроме того, эта схема приводит к появлению более осмысленных имен переменных вместо `invoice1` и `invoice2`.

Разработка “извне вовнутрь” (или “сверху вниз”, как ее иногда называют) означает сохранение уровня абстракции. *Тестовый метод* (Test Method) должен уделять основное внимание созданию условий для получения интересующего поведения тестируемой системы. Механизмы вызова этой логики теста должны быть реализованы на “нижних уровнях” программного обеспечения для тестирования. На практике такое поведение реализовано в виде вызовов *вспомогательных методов теста* (Test Utility Method), что при написании *тестовых методов* (Test Method) позволяет основное внимание уделять требованиям тестируемой системы. Появляется возможность не думать, как будет создаваться объект или проверяться результат. Достаточно просто описать, какими должны быть интересующие вас объект и результат. Использованные, но еще ненаписанные вспомогательные методы служат шаблонами для незавершенной логики автоматизации тестов. (Такой метод всегда должен иметь описательное имя и его заглушка должна вызывать утверждение `fail`, чтобы напоминать разработчикам о необходимости реализовать тело метода.) При этом можно переходить к реализации других тестов, пока требования еще свежи в памяти. Позднее можно одеть “шапку мастера по инструментам” и реализовать необходимые *вспомогательные методы теста* (Test Utility Method).

Использование разработки на основе тестов для создания вспомогательных методов теста

После создания всех *тестовых методов* (Test Method) можно переходить к реализации *вспомогательных методов теста* (Test Utility Method). При этом можно воспользоваться преимуществом разработки на основе тестов, написав тесты *вспомогательных методов теста* (Test Utility Test; см. *Вспомогательный метод теста*, Test Utility Method). Много времени на их написание не потребуется, но наличие модульных тестов придаст уверенность в правильности вспомогательных методов.

Следует начать с самого простого случая (например, с утверждения о равенстве двух идентичных коллекций, содержащих один и тот же элемент), а затем перейти к самому сложному случаю, действительно необходимому для *тестового метода* (Test Method) (например, двух коллекций, содержащих два одинаковых элемента, но в разном порядке). Разработка на основе тестов позволяет найти минимальную реализацию *вспомогательного метода теста* (Test Utility Method), которая может оказаться намного проще универсального решения. Нет смысла создавать универсальную логику, обрабатывающую ненужные варианты, но желательно вставить *сторожевое утверждение* (Guard Assertion) внутри *специального утверждения* (Custom Assertion), чтобы в неподдерживаемых случаях тест завершался неудачно.

Расположение повторно используемой логики проверки

Предположим, необходимо с помощью рефакторинга *выделение метода* (Extract Method) создать повторно используемые *специальные утверждения* (Custom Assertion) или с помощью *методов проверки* (Verification Method) написать тесты. Куда в таком случае нужно вставить пригодную к повторному использованию логику тестов? Наиболее очевидным местом является *сам класс теста* (Testcase Class, с. 401). Логика может использоваться более широко, если воспользоваться рефакторингом *подъем метода* (Pull-Up Method) [Ref] для переноса методов в *суперкласс теста* (Testcase Superclass, с. 646). Кроме

того, с помощью рефакторинга *перемещение метода* (Move Method) [Ref] его можно переместить во *вспомогательный класс теста* (Test Helper, с. 651). Более подробно эта проблема рассматривается в главе 12, “Организация тестов”.

Что дальше

Обсуждением способов проверки ожидаемого результата завершается введение в базовые способы автоматизации тестов средствами xUnit. В главе 11, “Использование тестовых двойников”, рассматриваются расширенные способы, предполагающие использование *тестовых двойников* (Test Double).

Глава 11

Использование тестовых двойников

О чём идет речь в этой главе

В нескольких предыдущих главах рассматривались базовые механизмы запуска тестов с помощью *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332) xUnit. В большинстве случаев предполагалось, что тестируемая система проектируется для обеспечения простоты тестирования в изоляции от других компонентов программного продукта. Если некоторый класс не зависит от других классов, при его тестировании не возникает сложностей и описанные здесь способы не потребуются. Но если класс все же зависит от других классов, остается два варианта: тестировать его вместе с другими классами, от которых он зависит, или попытаться изолировать его для отдельного тестирования. В этой главе рассматриваются способы изоляции тестируемой системы от других программных компонентов, от которых она зависит.

Что такое опосредованные ввод и вывод

Проблема тестирования классов в группах или кластерах заключается в сложности выявления всех ветвей выполнения кода. Вызываемые компоненты могут возвращать значения или генерировать исключения, влияющие на поведение тестируемой системы, но некоторые из таких случаев практически невозможно спровоцировать. Опосредованный ввод, полученный от вызываемого компонента, может быть непредсказуемым (например, как в случае с системными часами или календарем). В других случаях вызываемый компонент может быть недоступен в тестовой среде или может вообще не существовать. Как в таких условиях тестировать зависящие от них классы?

Иногда необходимо проверить определенные побочные эффекты вызова тестируемой системы. Если наблюдение за опосредованным выводом тестируемой системы затруднено (или извлечение этих данных слишком дорого обходится), эффективность автоматизированных тестов будет снижаться.

Из названия главы несложно догадаться, что для решения проблемы следует использовать *тестовые двойники* (Test Double, с. 538). Начнем с применения *тестовых двойников* (Test Double) для проверки опосредованных вводов и выводов. После этого будут описаны другие полезные применения данного механизма.

Назначение информации об опосредованном вводе

Обращения к вызываемому объекту часто возвращают объекты или значения, модифицируют собственные аргументы или даже генерируют исключения. Многие ветви выполнения тестируемой системы предназначены для работы с возвращаемыми значениями и обработки возможных исключений. Если этот код не протестировать, появится запах *не протестированный код* (Untested Code; см. *Ошибки в продукте*, Production Bugs, с. 303). Эффективное тестирование этих ветвей кода может оказаться самой сложной задачей, но именно они с большой вероятностью приводят к катастрофическим отказам при первом же запуске в процессе эксплуатации.

Очень нежелательно, чтобы код обработки исключения выполнялся при первом же запуске продукта. Что если в нем содержатся ошибки? Очевидно, что такой код должен контролироваться автоматизированными тестами. Основной проблемой таких тестов является необходимость заставить вызываемый компонент сгенерировать исключение, чтобы можно было проверить код обработки ошибки. Ожидаемое от вызываемого компонента исключение является примером **опосредованного ввода** (indirect input test condition) (рис. 11.1). Ввод был вставлен в систему через **точку управления** (control point).

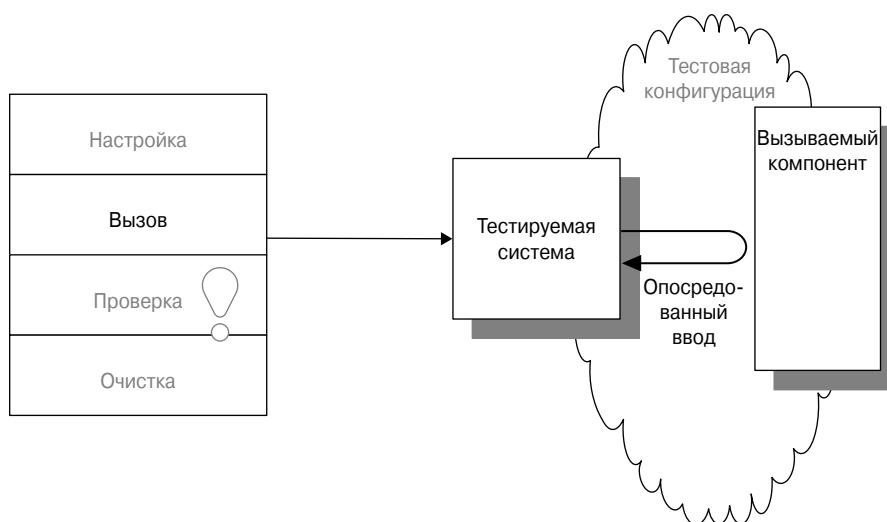


Рис. 11.1. Опосредованный ввод был получен тестируемой системой от вызываемого компонента. Не каждый ввод тестируемой системы предоставляетя тестом. Иногда опосредованный ввод предоставляется другими вызываемыми компонентами в виде возвращаемых значений, модифицированных параметров или исключений

Назначение информации об опосредованном выводе

Концепция инкапсуляции часто позволяет не думать о конкретной реализации. В конце концов, для этого инкапсуляция и предназначена — скрывать от клиентов интерфейса подробности реализации. При тестировании проверяется именно реализация, чтобы у клиентов не возникало желания интересоваться этим вопросом.

Рассмотрим компонент с программным интерфейсом, в который входит ничего не возвращающий метод. Отсутствие возвращаемого значения не позволяет определить, успешно ли завершилась работа метода. В такой ситуации остается только тестирование через “черный ход”. Хорошим примером такой ситуации является система ведения журнала сообщений. Вызовы в программном интерфейсе системы редко возвращают значения, указывающие на корректное завершение операции. Единственным способом проверки работоспособности является взаимодействие через другой интерфейс — через интерфейс, позволяющий получать внесенные в журнал сообщения.

Клиент может вызывать систему при выполнении некоторых условий. Эти вызовы не видны через интерфейс клиента, но обычно являются требованиями к клиенту, а значит, будут проверяться тестом. Условия, которые приводят к занесению сообщения в журнал, являются условием теста **опосредованного ввода** (indirect output test condition) (рис. 11.2). Для него придется создавать тесты, чтобы избежать появления **нетестированного требования** (Untested Requirement; см. *Ошибки в продукте*, Production Bugs). Опосредованный вывод проверяется через **точку наблюдения** (observation point).

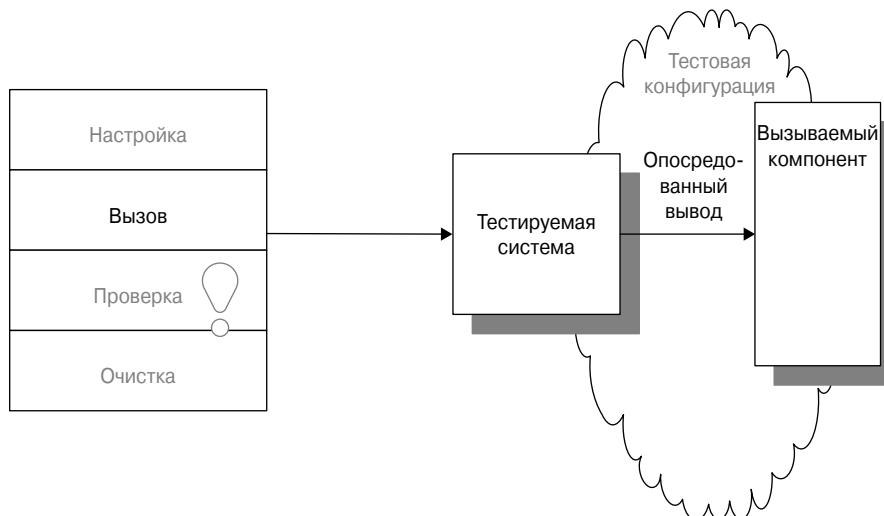


Рис. 11.2. Опосредованный вывод передается тестируемой системе. Не каждый вывод тестируемой системы непосредственно доступен тесту. Некоторые типы опосредованного вывода передаются другим компонентам в виде вызовов методов или сообщений

В других ситуациях тестируемая система имеет видимое поведение, которое можно тестировать через основной интерфейс, но при этом ожидается и ряд побочных эффектов. Оба результата должны проверяться тестами. Иногда тестирование заключается в добавлении утверждений относительно опосредованного вывода в существующие тесты. Это позволит проверить каждое *нетестированное требование* (Untested Requirement).

Управление опосредованным вводом

Тестирование с опосредованным вводом выполняется немного проще, чем тестирование с опосредованным выводом, так как способы для проверки вывода основаны на способах проверки ввода. Сначала рассмотрим тестирование с опосредованным вводом.

Для проверки тестируемой системы с опосредованным вводом необходим контроль над вызываемым компонентом, позволяющий получать от него любое возвращаемое значение. Это означает наличие подходящей точки управления.

Ниже приведены примеры опосредованного ввода через такие точки управления:

- возвращаемые значения функций/методов;
- значения модифицированных аргументов;
- генерируемые исключения.

Часто тест может взаимодействовать с вызываемым компонентом для настройки ответа на запросы. Например, если компонент обеспечивает доступ к данным в базе данных, можно воспользоваться *настройкой через “черный ход”* (Back Door Setup; см. *Манипуляция через “черный ход”*, Back Door Manipulation, с. 359) для вставки конкретных значений в базу данных, которые заставляют компонент реагировать необходимым образом (например, не найдено ни одного пункта, найден один пункт, найдено несколько пунктов). В данном конкретном случае в качестве точки управления можно использовать базу данных (рис. 11.3).

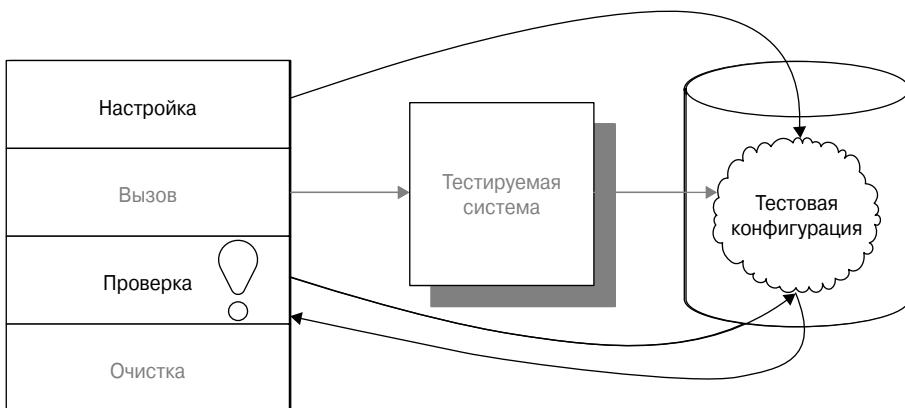


Рис. 11.3. Использование манипуляций через “черный ход” (Back Door Manipulation) для опосредованного управления и контроля над тестируемой системой. Если тестируемая система хранит свое состояние в другом компоненте, можно попытаться модифицировать состояние через непосредственное взаимодействие теста с другим компонентом через “черный ход”

Но в большинстве случаев такой подход будет непрактичным или даже невозможным. Использование реального компонента может оказаться невозможным по следующим причинам.

- Реальным компонентом нельзя управлять для получения необходимого опосредованного ввода. Только настоящие ошибки в программном обеспечении компонента могут привести к необходимому вводу в тестируемую систему.

- Реальный компонент можно заставить сгенерировать необходимый ввод, но это потребует слишком больших накладных расходов.
- Реальный компонент можно заставить сгенерировать необходимый ввод, но это будет сопровождаться недопустимыми побочными эффектами.

Если реальный компонент нельзя использовать в качестве точки управления, его придется заменить тем, что можно контролировать. Такая замена может выполняться одним из нескольких способов, которые рассматриваются в разделе “Установка тестового двойника”. Самым распространенным подходом является настройка *тестовой заглушки* (Test Stub, с. 544) с набором возвращаемых значений и установка заглушки в тестируемую систему. В процессе вызова тестируемой системы *тестовая заглушка* (Test Stub) получает запросы и возвращает настроенные ответы (рис. 11.4). В такой ситуации заглушка превращается в точку управления.

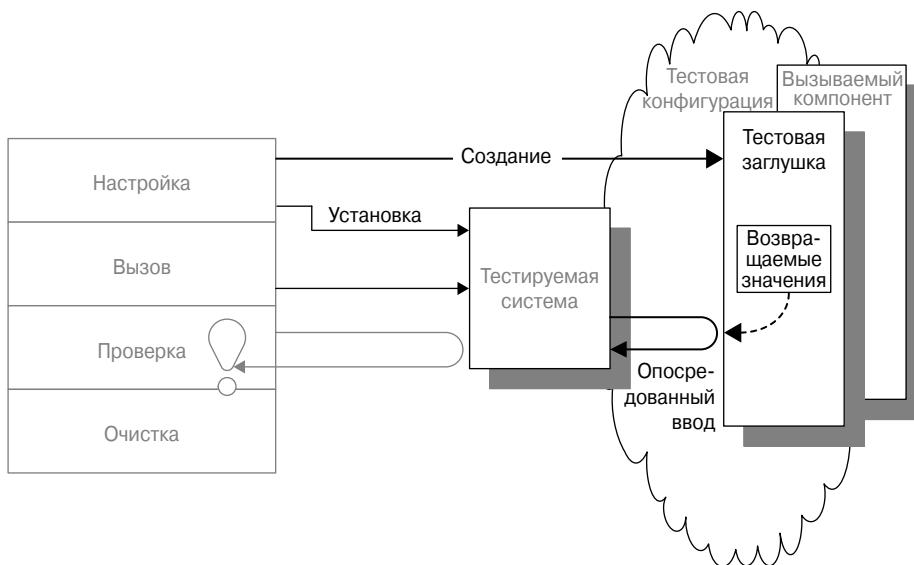


Рис. 11.4. Использование тестовой заглушки (Test Stub) в качестве точки управления опосредованным вводом. Одним из способов вставки опосредованного ввода в тестируемую систему является установка тестовой заглушки (Test Stub) вместо вызываемого компонента.

Перед вызовом тестируемой системы тестовая заглушка (Test Stub) получает набор возвращаемых значений. Такая стратегия позволяет заставить тестируемую систему пройти по всем ветвям выполнения кода

Проверка опосредованного вывода

В нормальной ситуации в процессе вызова тестируемая система естественно взаимодействует с вызываемым компонентом. Для проверки опосредованного вывода приходится наблюдать за вызовами программного интерфейса вызываемого компонента со стороны тестируемой системы (рис. 11.5). Более того, если необходимо тестировать происходящее после этого, потребуется способ управления возвращаемыми значениями (как было показано при обсуждении опосредованного ввода).

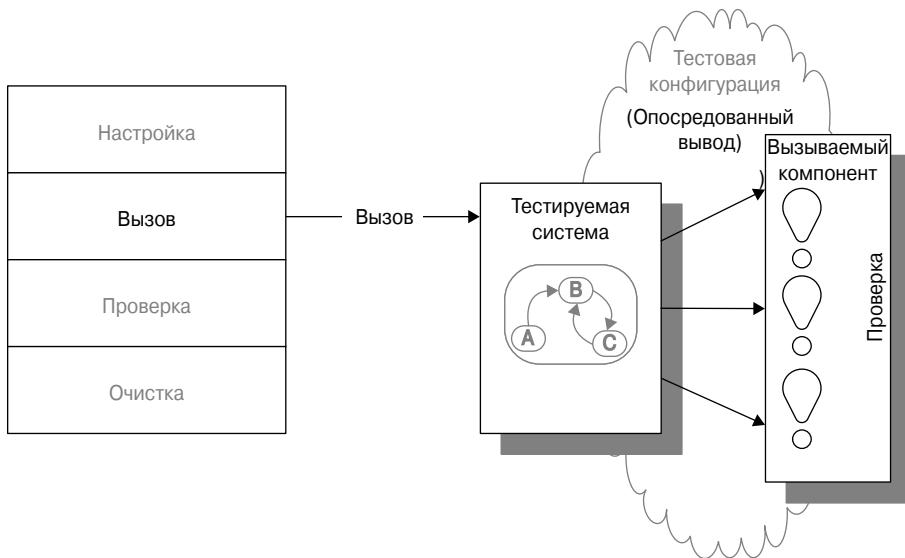


Рис. 11.5. Использование проверки поведения (Behavior Verification) для проверки опосредованного вывода тестируемой системы. Чтобы определить множество вызовов других компонентов, которые совершает тестируемая система, придется обратиться к проверке поведения (Behavior Verification), а не просто проверить состояние тестируемой системы после вызова

В большинстве случаев тест может использовать вызываемый компонент в качестве точки наблюдения для выяснения, как он используется.

- У файловой системы можно запросить содержимое файла, записанного тестируемой системой. Таким образом можно проверить, записан ли файл и содержит ли он ожидаемые данные.
- У базы данных можно запросить содержимое таблицы или конкретной записи, чтобы убедиться, что тестируемая система внесла в базу данных ожидаемые записи.
- У компонента отправки электронной почты можно запросить запись об обращении тестируемой системы для отправки конкретного сообщения.

Это примеры проверки через “черный ход” (Back Door Verification; см. *Манипуляция через “черный ход”, Back Door Manipulation*, с. 359). Некоторые вызываемые компоненты позволяют настраивать поведение таким образом, чтобы уведомлять тест об их использовании.

- Файловая система может уведомлять тест при каждом создании или при каждой модификации файла для своевременной проверки его содержимого.
- Триггер в базе данных может уведомлять тест о внесении или удалении записей.
- Компонент отправки электронной почты может пересыпать все исходящие сообщения тесту.

Иногда, как было показано при рассмотрении опосредованного ввода, непрактично использовать настоящий компонент в виде точки наблюдения. Если все остальные способы не дают результата, можно попытаться заменить настоящий компонент тестовой альтернативой. Например, такой вариант может быть оправдан в следующих ситуациях.

- Вызовы (или внутреннее состояние) вызываемого объекта невозможно определить.
- Запросы к реальному компоненту возможны, но связаны со слишком большими накладными расходами.
- Запросы к реальному компоненту возможны, но связаны с недопустимыми побочными эффектами.
- Реальный компонент еще не существует.

Имеется несколько способов замены реального компонента, как показано в разделе “Установка тестового двойника”.

Существует два основных стиля проверки опосредованного вывода. *Процедурная проверка поведения* (Procedural Behavior Verification; см. *Проверка поведения*, Behavior Verification) перехватывает обращения (или результаты обращений) к вызываемому компоненту в процессе вызова тестируемой системы, после чего сравнивает их с ожидаемыми вызовами. Такая проверка предполагает замещение **заменяемой зависимости** (substitutable dependency) *тестовым агентом* (Test Spy, с. 552). В процессе вызова тестируемой системы он получает и записывает полученные обращения. После вызова тестируемой системы *тестовый метод* (Test Method, с. 378) запрашивает у *тестового агента* (Test Spy) записанные данные и с помощью *методов с утверждением* (Assertion Method, с. 390) сравнивает их с ожидаемыми вызовами (рис. 11.6).

Ожидаемое поведение (Expected Behavior; см. *Проверка поведения*, Behavior Verification) включает создание “спецификации поведения” в процессе настройки тестовой конфигурации и последующее сравнение ожидаемого поведения с фактическим. Обычно для этого загружается *подставной объект* (Mock Object, с. 558) с описанием ожидаемых вызовов процедур. Подставной объект устанавливается в тестируемую систему (рис. 11.7). В процессе вызова тестируемой системы *подставной объект* (Mock Object) получает вызовы и сравнивает их с предварительно определенным набором (“спецификацией поведения”). Если в процессе работы *подставной объект* (Mock Object) получает неожиданный вызов, происходит немедленное неудачное завершение теста. Содержимое стека на момент неудачного завершения покажет точное место возникновения проблемы в тестируемой системе, так как *метод с утверждением* (Assertion Method) вызывается из *подставного объекта* (Mock Object), а он, в свою очередь, вызывается тестируемой системой. Кроме того, можно узнать, в каком месте *тестового метода* (Test Method) вызывалась тестируемая система.

Тестовый агент (Test Spy) или *подставной объект* (Mock Object) можно использовать в качестве точек управления опосредованным вводом.

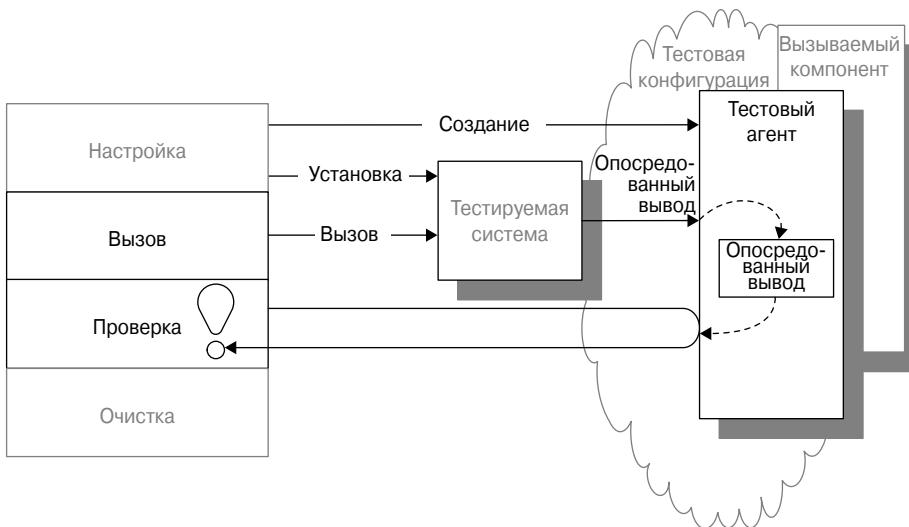


Рис. 11.6. Использование тестового агента (*Test Spy*) в качестве точки наблюдения для опосредованного вывода тестируемой системы. Одним из способов реализации проверки поведения (*Behavior Verification*) является установка тестового агента (*Test Spy*) в точке получения опосредованного вывода. После вызова тестируемой системы тест запрашивает у тестового агента (*Test Spy*) информацию о его использовании и сравнивает полученную информацию с ожидаемым поведением с помощью утверждений

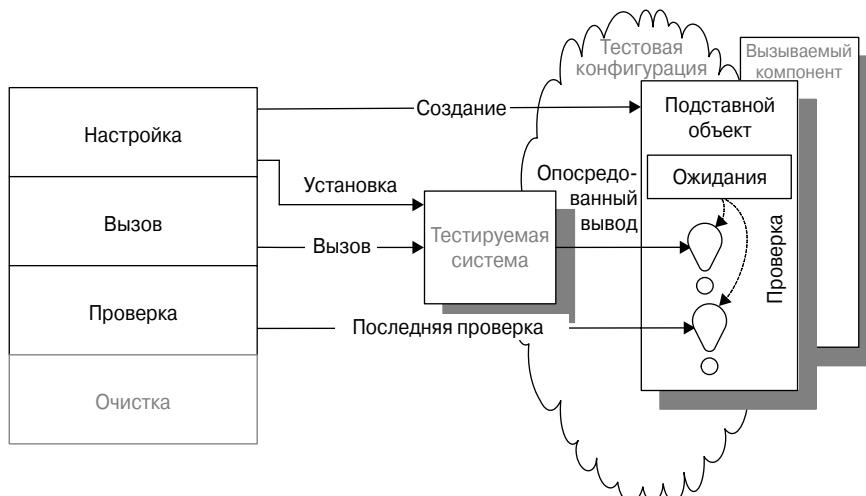


Рис. 11.7. Использование подставного объекта (*Mock Object*) в качестве точки наблюдения для опосредованного вывода тестируемой системы. Еще одним способом реализации проверки поведения (*Behavior Verification*) является установка подставного объекта (*Mock Object*) в точке получения опосредованного вывода. В то время как тестируемая система обращается к вызываемому компоненту, подставной объект (*Mock Object*) с помощью утверждений сравнивает фактические вызовы и аргументы с ожидаемыми

Тестирование с помощью двойников

На этом этапе часто возникает вопрос: “Как заменить негибкие и сложные во взаимодействии реальные компоненты тем, что поддерживает управление опосредованным вводом и проверку опосредованного вывода?”

Как было показано, для тестирования опосредованного ввода необходимо достаточно полно контролировать вызываемый компонент, чтобы заставить его возвращать любые значения (допустимые, недопустимые и исключения). Для проверки опосредованного вывода необходимо иметь возможность перехватывать обращения тестируемой системы к другим компонентам. С *тестовым двойником* (Test Double) намного проще взаимодействовать и создавать тесты необходимой структуры.

Типы тестовых двойников

Тестовый двойник (Test Double) — это любой объект или компонент, который устанавливается вместо реального компонента на время работы теста. В зависимости от причин применения *тестовый двойник* (Test Double) может проявлять один из четырех типов поведения (рис. 11.8).

- *Объект-заглушка* (Dummy Object, с. 730) просто передается тестируемой системе в качестве аргумента (или атрибута аргумента), но на самом деле никогда не используется.
- *Тестовая заглушка* (Test Stub) представляет собой объект, заменяющий реальный компонент, от которого зависит тестируемая система. Он позволяет тесту управлять опосредованным вводом тестируемой системы и провоцировать выполнение ветвей кода, которые в других условиях не выполняются. *Тестовый агент* (Test Spy), являясь более функциональной версией *тестовой заглушки* (Test Stub), позволяет проверять опосредованный вывод тестируемой системы, передавая его тесту для проверки.
- *Подставной объект* (Mock Object) заменяет реальный компонент, от которого зависит тестируемая система. Позволяет тесту проверять опосредованный вывод.
- *Поддельный объект* (Fake Object, с. 565) заменяет функциональность вызываемого компонента альтернативной реализацией.

Объект-заглушка

Объект-заглушка (Dummy Object) является вырожденным случаем *тестового двойника* (Test Double). Он предназначен для передачи из метода или в метод и по-другому никогда не используется, т.е. *объект-заглушка* (Dummy Object) просто существует. В большинстве случаев вместо него можно использовать значения “null” (или “nil”, или “nothing”). В других ситуациях приходится создавать настоящие объекты, так как код ожидает ненулевого значения. В динамически типизированных языках подойдет практически любой объект. В статически типизированных языках необходимо обеспечить совместимость типов между *объектом-заглушкой* (Dummy Object) и передаваемым параметром или переменной, которой он присваивается.

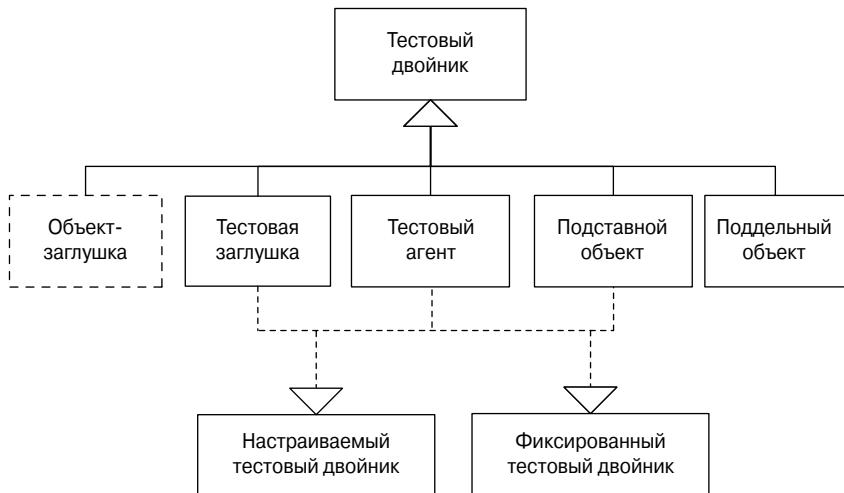


Рис. 11.8. Существует несколько типов тестовых двойников (Test Double). Объект-заглушка (Dummy Object) представляет собой альтернативу шаблонам значений. Тестовая заглушка (Test Stub) используется для проверки опосредованного ввода. Тестовый агент (Test Spy) и подставной объект (Mock Object) проверяют опосредованный вывод. Поддельный объект (Fake Object) эмулирует поведение настоящих вызываемых компонентов, но обладает дружественными к тесту характеристиками

В следующем примере объект `DummyCustomer` передается в конструктор `Invoice` как один из обязательных аргументов. Объект `DummyCustomer` тестируемым кодом не используется.

```

public void testInvoice_addLineItem_DO() {
    final int QUANTITY = 1;
    Product product = new Product("Dummy Product Name",
                                    getUniqueNumber());
    Invoice inv = new Invoice( new DummyCustomer() );
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    assertLineItemsEqual("", expItem, actual);
}
  
```

Обратите внимание, что *объект-заглушка* (Dummy Object) и **Null Object** [PLoPD3] — это не одно и то же. *Объект-заглушка* (Dummy Object) не используется тестируемой системой, а значит, его поведение никого не интересует. С другой стороны, **Null Object** используется тестируемой системой, но специально создан для того, чтобы ничего не делать. Небольшая, но очень важная разница!

Объект-заглушка (Dummy Object) отличается от остальных тестовых двойников (Test Double), так как они являются альтернативой шаблонам значений атрибутов, например *точное значение* (Literal Value, с. 718), *сгенерированное значение* (Generated Value, с. 726) и *вы-*

числяемое значение (Derived Value, с. 722). Таким образом, их “настройка” или “установка” не требуется. На самом деле никакие из рассмотренных ниже характеристик других тестовых двойников не относятся к *объекту-заглушке* (Dummy Object), поэтому в данной главе они больше не рассматриваются.

Тестовая заглушка

Тестовая заглушка (Test Stub) представляет собой объект, выступающий в роли точки управления. Используется для доставки опосредованного ввода в тестируемую систему при вызове его методов. Использование *тестовой заглушки* (Test Stub) позволяет выполнять ветви кода, которые в других случаях в процессе тестирования не затрагиваются. *Генератор ответов* (Responder; см. *Тестовая заглушка*, Test Stub) является базовым вариантом заглушки, используемым для вставки действительного и недействительного опосредованного ввода в тестируемую систему через возвращаемые значения методов. *Диверсант* (Saboteur; см. *Тестовая заглушка*, Test Stub) является специальным вариантом заглушки для генерации исключений или ошибок в качестве некорректного опосредованного ввода в тестируемую систему. Поскольку процедурные языки программирования не поддерживают объекты, в таких языках приходится использовать *процедурные тестовые заглушки* (Procedural Test Stub; см. *Тестовая заглушка*, Test Stub).

В следующем примере реализованный в виде анонимного внутреннего класса Java *диверсант* (Saboteur) генерирует исключение в момент вызова метода `getTime` тестируемой системой. Это позволяет проверить поведение тестируемой системы в такой ситуации.

```
public void testDisplayCurrentTime_exception()
    throws Exception {
    // Настройка тестовой конфигурации
    // Определение и создание тестовой заглушки
    TimeProvider testStub = new TimeProvider()
    { // Анонимная внутренняя тестовая заглушка
        public Calendar getTime() throws TimeProviderEx {
            throw new TimeProviderEx("Sample");
        }
    };
    // Создать тестируемую систему
    TimeDisplay sut = new TimeDisplay();
    sut.setTimeProvider(testStub);
    // Вызвать тестируемую систему
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверить непосредственный вывод
    String expectedTimeString =
        "<span class=\"error\">Invalid Time</span>";
    assertEquals("Exception", expectedTimeString, result);
}
```

В процедурных языках программирования *процедурная тестовая заглушка* (Procedural Test Stub) имеет форму:

- замены еще ненаписанной процедуры;
- альтернативной реализации процедуры, которая связывается с программой вместо основной реализации.

Обычно *процедурная тестовая заглушка* (Procedural Test Stub) позволяет выполнять отладку в то время, когда часть кода еще не готова. Очень редко заглушки подменяются на этапе выполнения — в большинстве процедурных языков это связано с определенными проблемами. Если нет возражений против *логики теста в продукте* (Test Logic in Production, с. 257), процедурные заглушки можно реализовать с помощью *ловушки для теста* (Test Hook, с. 713) в виде конструкции `if testing then... else` внутри тестируемой системы. Подобная реализация показана в следующем примере.

```
public Calendar getTime() throws TimeProviderEx {
    Calendar theTime = new GregorianCalendar();
    if (TESTING) {
        theTime.set(Calendar.HOUR_OF_DAY, 0);
        theTime.set(Calendar.MINUTE, 0);
    } else {
        // Просто вернуть календарь
    }
    return theTime;
};
```

Основным исключением являются языки с поддержкой **процедурных переменных** (procedural variables)¹. Такие переменные позволяют реализовать динамическое связывание, если код клиента получает доступ к замещаемой процедуре через переменную.

Тестовый агент

Тестовый агент (Test Spy) представляет собой объект, выступающий в роли точки наблюдения для опосредованного вывода тестируемой системы. К остальным возможностям *тестовой заглушки* (Test Stub) он добавляет возможность незаметной записи всех вызовов методов, которые выполняет тестируемая система. После этого тест выполняет *процедурную проверку поведения* (Procedural Behavior Verification), сравнивая фактические вызовы с ожидаемыми с помощью последовательности утверждений.

В следующем примере для проверки правильности информации, передаваемой в метод `logMessage` тестируемой системой, используется *интерфейс извлечения* (Retrieval Interface; см. *Тестовый агент*, Test Spy).

```
public void testRemoveFlightLogging_recordingTestStub()
    throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnUnregFlight();
    FlightManagementFacade facade =
        new FlightManagementFacadeImpl();
    // Настройка тестового двойника
    AuditLogSpy logSpy = new AuditLogSpy();
    facade.setAuditLog(logSpy);
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка состояния
    assertFalse("полет существует после удаления",
        facade.flightExists(expectedFlightDto.
            getFlightNumber()));
    // Проверить опосредованный вывод с помощью
```

¹ Или **указателей на функцию** (function pointers).

```
// интерфейса извлечения
assertEquals("number of calls", 1,
    logSpy.getNumberOfCalls());
assertEquals("action code",
    Helper.REMOVE_FLIGHT_ACTION_CODE,
    logSpy.getActionCode());
assertEquals("date", helper.getTodaysDateWithoutTime(),
    logSpy.getDate());
assertEquals("user", Helper.TEST_USER_NAME,
    logSpy.getUser());
assertEquals("detail",
    expectedFlightDto.getFlightNumber(),
    logSpy.getDetail());
}
```

Подставной объект

Подставной объект (Mock Object) может выступать в роли точки наблюдения для опосредованного вывода тестируемой системы. Как и от *тестовой заглушки* (Test Stub), от него может потребоваться информация в ответ на вызовы методов. Как и тестовый агент, *подставной объект* (Mock Object) следит за вызовами со стороны тестируемой системы. Но в отличие от агента он сравнивает фактические вызовы с определенными ранее с помощью утверждений и неудачно завершает тест от имени *тестового метода* (Test Method). В результате логика проверки опосредованного вывода тестируемой системы может повторно использоваться всеми тестами, применяемыми один и тот же *подставной объект* (Mock Object). Подставной объект встречается в двух вариантах.

- Строгий *подставной объект* (Mock Object) неудачно завершает тест, если ожидаемые вызовы приходят в порядке, отличающемся от порядка в ранее заданной спецификации.
- Ослабленный *подставной объект* (Mock Object) мирится с неправильным порядком вызовов. Некоторые из них даже принимают неожиданные или отсутствующие вызовы. В таком случае объект проверяет только те вызовы, которые соответствуют ожидаемым.

В следующем teste настраивается *подставной объект* (Mock Object) для перехвата ожидаемого вызова метода `logMessage`. Как только тестируемая система (метод `removeFlight`) вызывает `logMessage`, *подставной объект* (Mock Object) проверяет утверждение о соответствии каждого фактического аргумента ожидаемому. Если методу передаются некорректные аргументы, тест завершается неудачно.

```
public void testRemoveFlight_Mock() throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnonRegFlight();
    // Настройка подставного объекта
    ConfigurableMockAuditLog mockLog =
        new ConfigurableMockAuditLog();
    mockLog.setExpectedLogMessage(
        helper.getTodaysDateWithoutTime(),
        Helper.TEST_USER_NAME,
        Helper.REMOVE_FLIGHT_ACTION_CODE,
        expectedFlightDto.getFlightNumber());
    mockLog.setExpectedNumberCalls(1);
```

```

// Установка подставного объекта
FlightManagementFacade facade =
    new FlightManagementFacadeImpl();
facade.setAuditLog(mockLog);
// Вызов
facade.removeFlight(expectedFlightDto.getFlightNumber());
// Проверка
assertFalse("полет существует после удаления",
    facade.flightExists(expectedFlightDto.
        getFlightNumber()));
mockLog.verify();
}

```

Как и *тестовая заглушка* (Test Stub), *подставной объект* (Mock Object) часто поддерживает настройку с использованием опосредованного ввода, необходимого тестируемой системе для достижения состояния, когда генерируется проверяемый опосредованный вывод.

Поддельный объект

Поддельный объект (Fake Object) отличается от *тестовой заглушки* (Test Stub) или *подставного объекта* (Mock Object), так как он не контролируется и не наблюдается тестом непосредственно. *Поддельный объект* (Fake Object) используется для замещения функциональности настоящего вызываемого компонента в ситуациях, не предполагающих проверку опосредованного ввода или вывода. Обычно *поддельный объект* (Fake Object) реализует ту же функциональность или подмножество функциональности настоящего вызываемого компонента, хотя и более простым способом. Чаще всего причиной использования *поддельного объекта* (Fake Object) является недоступность настоящего вызываемого объекта, его недостаточное быстродействие или недоступность в тестовой среде.

Во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351 рассматривается инкапсуляция доступа к базе данных внутри уровня сохранения данных с последующей заменой этого уровня компонентом, использующим хранящиеся в памяти хэш-таблицы вместо настоящей базы данных. Результатом стало пятидесятитрехкратное ускорение работы тестов. Для этого использовалась *поддельная база данных* (Fake Database; см. *Поддельный объект*, Fake Object), которая выглядела, как показано ниже.

```

public class InMemoryDatabase implements FlightDao{
    private List airports = new Vector();
    public Airport createAirport(String airportCode,
        String name, String nearbyCity)
        throws DataException, InvalidArgumentException {
        assertParamtersAreValid(airportCode, name, nearbyCity);
        assertAirportDoesntExist(airportCode);
        Airport result = new Airport(getNextAirportId(),
            airportCode, name, createCity(nearbyCity));
        airports.add(result);
        return result;
    }
    public Airport getAirportByPrimaryKey(BigDecimal airportId)
        throws DataException, InvalidArgumentException {
        assertAirportNotNull(airportId);
        Airport result = null;
        Iterator i = airports.iterator();
        while (i.hasNext()) {
            Airport airport = (Airport) i.next();

```

```

        if (airport.getId().equals(airportId)) {
            return airport;
        }
    } else
        throw new DataException("Airport not found:" + airportId);
}

```

Предоставление тестового двойника

Существует два способа предоставления тестовых двойников: *созданный вручную тестовый двойник* (Hand-Built Test Double; см. *Настраиваемый тестовый двойник*, Configurable Test Double, с. 571) и *динамически генерируемый тестовый двойник* (Dynamically Generated Test Double), который генерируется на этапе выполнения средствами инфраструктуры или инструментария от стороннего разработчика². Все сгенерированные *тестовые двойники* (Test Double) по своей природе должны быть *настраиваемыми тестовыми двойниками* (Configurable Test Double). Более подробно эти компоненты рассматриваются в следующем разделе. С другой стороны, *созданные вручную тестовые двойники* (Hand-Built Test Double) чаще всего оказываются *фиксированными тестовыми двойниками* (Hard-Coded Test Double, с. 581), но при определенных усилиях также могут поддерживать настройку. В следующих примерах кода показан созданный вручную *внутренний тестовый двойник* (Inner Test Double), основанный на анонимном внутреннем классе Java.

```

public void testDisplayCurrentTime_AtMidnight_PS()
    throws Exception {
    // Настройка тестовой конфигурации
    // Определение и создание тестовой заглушки
    TimeProvider testStub = new PseudoTimeProvider()
    { // Анонимная внутренняя заглушка
        public Calendar getTime(String timeZone) {
            Calendar myTime = new GregorianCalendar();
            myTime.set(Calendar.MINUTE, 0);
            myTime.set(Calendar.HOUR_OF_DAY, 0);
            return myTime;
        }
    };
    // Создание экземпляра тестируемой системы
    TimeDisplay sut = new TimeDisplay();
    // Вставка тестовой заглушки в тестируемую систему
    sut.setTimeProvider(testStub);
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка вывода
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}

```

² Пакет JMock и его реализации для других языков программирования могут служить хорошим примером такого инструментария. С помощью других инструментов, например EasyMock, реализуются *статически генерируемые тестовые двойники* (Statically Generated Test Double). При этом сначала генерируется код, который позднее компилируется, как и в случае с созданными вручную тестовыми двойниками.

Разработку *созданных вручную тестовых двойников* (Hand-Built Test Double) в статически типизированных языках (Java и C#) можно значительно упростить, если предоставить набор базовых классов, которые называются *псевдообъектами* (Pseudo-Object; см. *Фиксированный тестовый двойник*, Hard-Coded Test Double) и служат основой для создания подклассов. *Псевдообъекты* (Pseudo-Object) могут сократить количество реализуемых методов в каждой *тестовой заглушки* (Test Stub), *тестовом агенте* (Test Spy) и *подставном объекте* (Mock Object) до тех методов, вызов которых ожидается. Они оказываются особенно полезными при использовании *внутренних тестовых двойников* (Inner Test Double) или *тестовых шунтов* (Self Shunt; см. *Фиксированный тестовый двойник*, Hard-Coded Test Double). Определение класса *псевдообъекта* (Pseudo-Object) для предыдущих примеров будет выглядеть следующим образом.

```
/**
 * Базовый класс для созданных
 * вручную тестовых заглушек
 * и подставных объектов
 */
public class PseudoTimeProvider implements ComplexTimeProvider {
    public Calendar getTime() throws TimeProviderEx {
        throw new PseudoClassException();
    }
    public Calendar getTimeDifference(Calendar baseTime,
                                     Calendar otherTime)
        throws TimeProviderEx {
        throw new PseudoClassException();
    }
    public Calendar getTime(String timeZone)
        throws TimeProviderEx {
        throw new PseudoClassException();
    }
}
```

Настройка тестового двойника

Некоторые *тестовые двойники* (Test Double) (особенно *тестовая заглушка*, Test Stub, *подставной объект*, Mock Object) должны получить возвращаемые и ожидаемые значения. *Фиксированный тестовый двойник* (Hard-Coded Test Double) получает такие инструкции во время проектирования от разработчика. *Настраиваемый тестовый двойник* (Configurable Test Double) получает эту информацию на этапе выполнения от самого теста (рис. 11.9). *Тестовая заглушка* (Test Stub) и *тестовый агент* (Test Spy) во время настройки получают только значения, которые будут возвращаться методами, предположительно вызываемыми тестируемой системой. Кроме того, *подставной объект* (Mock Object) должен получить имена всех методов и их аргументов, которые будут вызываться тестируемой системой. Во всех случаях именно разработчик определяет, на какие значения настраивать *тестовый двойник* (Test Double). Не удивительно, что основным условием во время принятия этого решения является сохранение простоты понимания теста и возможность повторного использования кода *тестового двойника* (Test Double).

Поддельные объекты (Fake Object) в “настройке” во время выполнения не нуждаются, так как они просто используются тестируемой системой. Вывод впоследствии зависит от более ранних вызовов со стороны тестируемой системы. Точно так *объект-заглушка*

(*Dummy Object*) не требует “настройки”, поскольку он никогда не вызывается³. *Процедурная тестовая заглушка* (*Procedural Test Stub*) обычно создается в виде *фиксированного тестового двойника* (*Hard-Coded Test Double*), т.е. он программируется на возврат конкретного значения в ответ на вызов функции; таким образом, он является простейшей формой *тестового двойника* (*Test Double*).

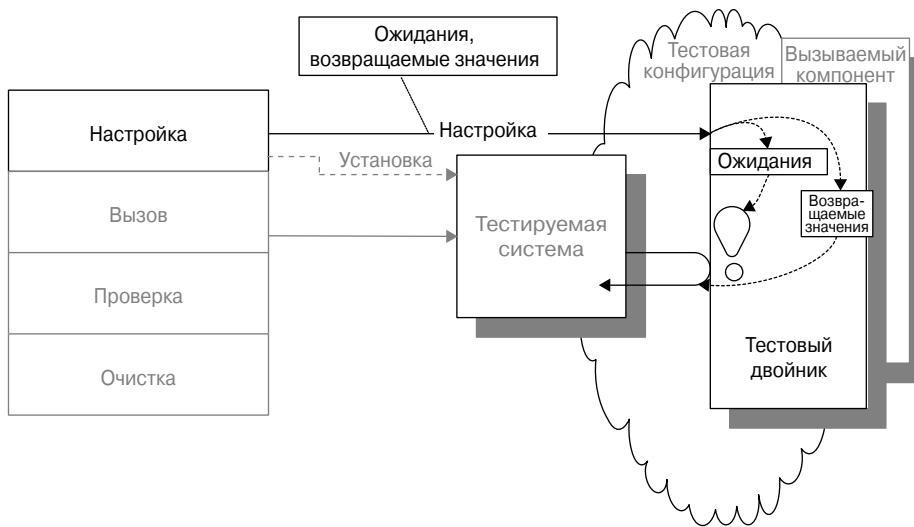


Рис. 11.9. Тестовый двойник (*Test Double*) настраивается тестом. Передавая возвращаемые или ожидаемые значения в настраиваемый тестовый двойник (*Configurable Test Double*), можно избежать размножения классов фиксированных тестовых двойников (*Hard-Coded Test Double*)

Настраиваемый тестовый двойник (*Configurable Test Double*) предоставляет *конфигурационный интерфейс* (*Configuration Interface*; см. *Настраиваемый тестовый двойник*, *Configurable Test Double*) или *режим настройки* (*Configuration Mode*), который может использоваться тестом для настройки двойника на возвращаемые и ожидаемые значения. Как следствие *настраиваемый тестовый двойник* (*Configurable Test Double*) может повторно использоваться многими тестами. Кроме того, в результате тесты становятся более понятными, так как внутренние значения *тестового двойника* (*Test Double*) видимы в тексте теста. Это снижает вероятность появления запаха *таинственный гость* (*Mystery Guest*; см. *Непонятный тест*, *Obscure Test*, с. 230).

На каком этапе должна происходить эта настройка? Установка *тестового двойника* (*Test Double*) должна рассматриваться, как любая другая часть настройки тестовой конфигурации. Возможны такие варианты, как *встроенная настройка* (*In-line Setup*, с. 434), *неявная настройка* (*Implicit Setup*, с. 449) и *делегированная настройка* (*Delegated Setup*, с. 437).

³ *Объект-заглушка* (*Dummy Object*) может использоваться в качестве точки наблюдения, чтобы можно было убедиться, что он никогда не используется. При этом *объект-заглушка* (*Dummy Object*) генерирует исключение, если вызывается любой из его методов.

Установка тестового двойника

Перед вызовом тестируемой системы необходимо “установить” все *тестовые двойники* (Test Double), от которых зависит тест. Слово “установить” описывает процесс принуждения тестируемой системы к использованию *тестового двойника* (Test Double). Нормальная последовательность предполагает создание экземпляра *тестового двойника* (Test Double), его настройку в случае *настраиваемого тестового двойника* (Configurable Test Double) и предоставление двойника тестируемой системе до или в процессе вызова. Существует несколько способов “установки” *тестового двойника* (Test Double). Выбор конкретного варианта больше зависит от стиля, чем от особенностей кода, если тестируемая система создавалась с учетом тестов. Если тесты внедряются в унаследованную систему, выбор способа установки может оказаться более ограниченным.

Основное множество вариантов установки состоит из *вставки зависимости* (Dependency Injection, с. 684), когда клиент сообщает тестируемой системе, какой вызываемый компонент использовать, *поиска зависимости* (Dependency Lookup, с. 692), когда тестируемая система делегирует создание и получение вызываемого компонента другому объекту, и *ловушки для теста* (Test Hook), когда модифицируются вызываемый компонент или обращения к нему внутри тестируемой системы.

Если в языке программирования доступен механизм *инвертирования управления* (inversion of control), тесты могут заменять зависимости без дополнительных усилий разработчика. В результате исчезает необходимость создания механизмов *вставки зависимости* (Dependency Injection) или *поиска зависимости* (Dependency Lookup).

Вставка зависимости

Вставка зависимости (Dependency Injection) является классом ослабления связности дизайна, при котором клиент предоставляет тестируемой системе вызываемый компонент на этапе выполнения (рис. 11.10). Разработка на основе тестов сделала *вставку зависимости* (Dependency Injection) более популярным решением, так как в результате получается проще тестируемый дизайн. Этот шаблон позволяет более широко повторно использовать тестируемую систему, так как знание о зависимостях из тестируемой системы извлекается и остается представление только об универсальном интерфейсе, который должен быть реализован вызываемым компонентом. Существует несколько вариантов *вставки зависимости* (Dependency Injection).

- *Вставка метода установки* (Setter Injection; см. *Вставка зависимости*, Dependency Injection). Тестируемая система получает доступ к вызываемому компоненту через открытый атрибут (переменную или свойство). После создания тестируемой системы тест явно присваивает значение атрибуту, устанавливая *тестовый двойник* (Test Double). Значение атрибута могло устанавливаться ранее конструктором тестируемой системы (в таком случае тестируемая система не будет пытаться устанавливать настоящий вызываемый компонент).
- *Вставка конструктора* (Constructor Injection; см. *Вставка зависимости*, Dependency Injection). Тестируемая система получает доступ к вызываемому компоненту через закрытый атрибут. Тест передает *тестовый двойник* (Test Double) в систему через конструктор, принимающий вызываемый компонент в виде явного аргумента, и сохраняет его в виде значения атрибута. Это может быть основной, применяемый в коде продукта или альтернативный конструктор. В последнем случае ос-

новной конструктор должен вызывать альтернативный, передавая принятый по умолчанию вызываемый компонент в качестве аргумента.

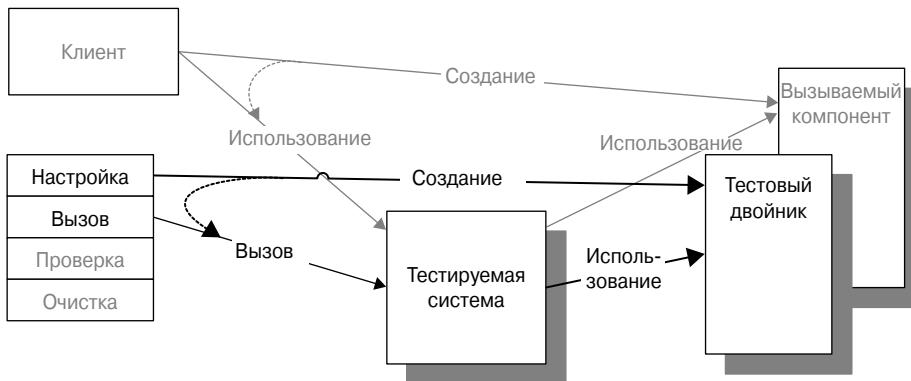


Рис. 11.10. Тестовый двойник (Test Double) вставляется в систему самим тестом. Для использования двойников должен существовать механизм подмены вызываемого компонента. Использование вставки зависимости (Dependency Injection) требует от вызывающей стороны предоставить вызываемый компонент тестируемой системе

- *Вставка параметра* (Parameter Injection; см. *Вставка зависимости*, Dependency Injection). Тестируемая система получает вызываемый компонент в виде параметра метода. В таком случае тест передает *тестовый двойник* (Test Double), а код продукта передает настоящий объект⁴. Такой подход оправдан, если программный интерфейс тестируемой системы принимает заменяемый объект в качестве параметра. Хотя апологеты *подставных объектов* (Mock Object) утверждают, что проектирование программного интерфейса подобным образом улучшает дизайн тестируемой системы, не всегда возможно или практически оправдано передавать все необходимые объекты в качестве параметров методов.

Поиск зависимости

Если программное обеспечение проектировалось без учета тестов или *вставка зависимости* (Dependency Injection) была недопустима, *поиск зависимости* (Dependency Lookup) может оказаться достаточно удобным. Этот шаблон удаляет из тестируемой системы знание о конкретном вызываемом компоненте, но вместо этого тестируемая система должна запрашивать другой компонент для поиска необходимого вызываемого компонента (рис. 11.11). В результате появляется возможность замены вызываемого компонента на этапе выполнения без модификации кода тестируемой системы. При этом приходится каким-то образом изменять поведение промежуточного компонента, вследствие чего и возникают различные варианты *поиска зависимости* (Dependency Lookup).

⁴ Этот подход был описан в первой статье о *подставных объектах* (Mock Object) [ET]. В ней *подставные объекты* (Mock Object), передаваемые в качестве параметров методов, называются *умными указателями* (Smart Handlers).

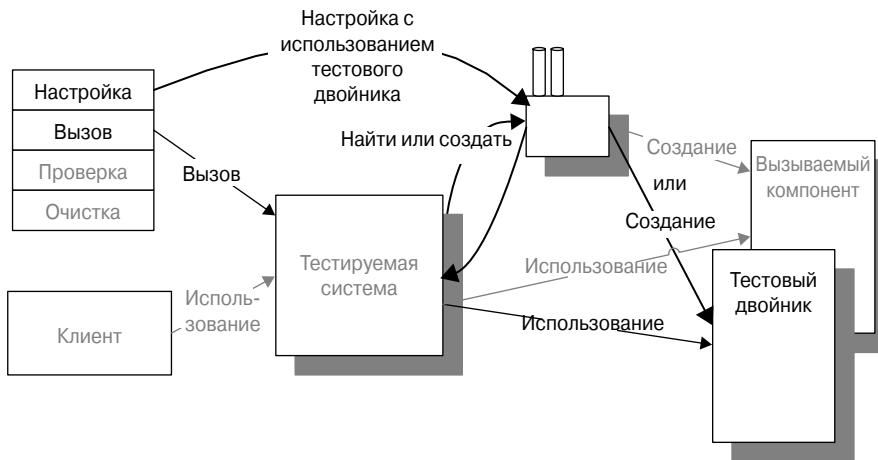


Рис. 11.11. Локатор служб (Service Locator) “настроен” тестом на предоставление тестового двойника (Test Double) тестируемой системе. Использование тестовых двойников (Test Double) предполагает наличие механизма подмены вызываемого компонента. При использовании поиска зависимости (Dependency Lookup) тестируемая система должна запрашивать у широкоизвестного объекта ссылку на вызываемый компонент. Тест может предоставить локатор служб (Service Locator), который будет возвращать ссылку на тестовый двойник (Test Double)

- **Фабрика объектов** (Object Factory; см. *Поиск зависимости*, Dependency Lookup). Тестируемая система создает вызываемый объект, вызывая **метод-фабрику** (Factory Method) [GOF] широкоизвестного объекта вместо конструктора. Тест заставляет **фабрику объектов** (Object Factory) создавать **тестовый двойник** (Test Double) вместо нормального вызываемого компонента.
- **Локатор служб** (Service Locator; см. *Поиск зависимости*, Dependency Lookup). Тестируемая система получает ранее созданный служебный объект, консультируясь с широкоизвестным объектом реестра (Registry) [PEAA]. Тест настраивает **локатор служб** (Service Locator) на предоставление **тестового двойника** (Test Double) вместо вызываемого компонента.

Граница между этими шаблонами оказывается достаточно размытой, если для создания объекта, возвращаемого **локатором служб** (Service Locator), применяется “ленивая” инициализация (Lazy Initialization). Может, стоит переименовать локатор в **фабрику объектов** (Object Factory)? Есть ли разница между названиями? Скорее всего, название менять не стоит — *поиск зависимости* (Dependency Lookup).

Интеграция тестов с помощью связанного с тестом подкласса

Даже если ни один из этих механизмов не встроен в тестируемую систему, можно попытаться интегрировать возможность тестирования через **связанный с тестом подкласс** (Test-Specific Subclass).

Достаточно часто шаблон Singleton [GOF] специально используется в качестве **фабрики объектов** (Object Factory) или **локатора служб** (Service Locator). Если Singleton имеет фиксированное поведение, придется превратить его в **заменяемый единственный экземпляр**.

ляр класса (Substitutable Singleton; см. *Связанный с тестом подкласс*, Test-Specific Subclass, с. 591), что позволит переопределить возвращаемый вызываемый компонент *тестовым двойником* (Test Double). Можно избежать использования *класса с единственным экземпляром* (Singleton), если воспользоваться инвертированием управления или созданным вручную механизмом *вставки зависимости* (Dependency Injection). Оба варианта являются более предпочтительными, по сравнению с использованием класса с единственным экземпляром, так как зависимость теста от *тестового двойника* (Test Double) становится более очевидной. Классы, реализующие шаблон *класса с единственным экземпляром* (Singleton), используемые по другим причинам, практически всегда приводят к проблемам во время написания тестов. Необходимо избегать их использования любой ценой.

Тест может создать экземпляр связанныго с *тестом подкласса* (Test-Specific Subclass) и добавить механизм *вставки зависимости* (Dependency Injection) или заменить другие методы тестируемой системы ориентированным на тест поведением (рис. 11.12). Можно переопределить логику, которая используется для доступа к вызываемому компоненту. В результате без внесения модификаций в код продукта появляется возможность возврата *тестового двойника* (Test Double) вместо стандартного компонента. Кроме того, можно заменить реализации любых методов, вызываемых тестируемым методом, поведением *тестовой заглушки* (Test Stub). Таким образом, тестируемая система превращается в собственный *тестовый двойник* в виде подкласса (Subclassed Test Double; см. *Связанный с тестом подкласс*, Test-Specific Subclass). Это один из способов вставки опосредованного ввода в тестируемую систему.

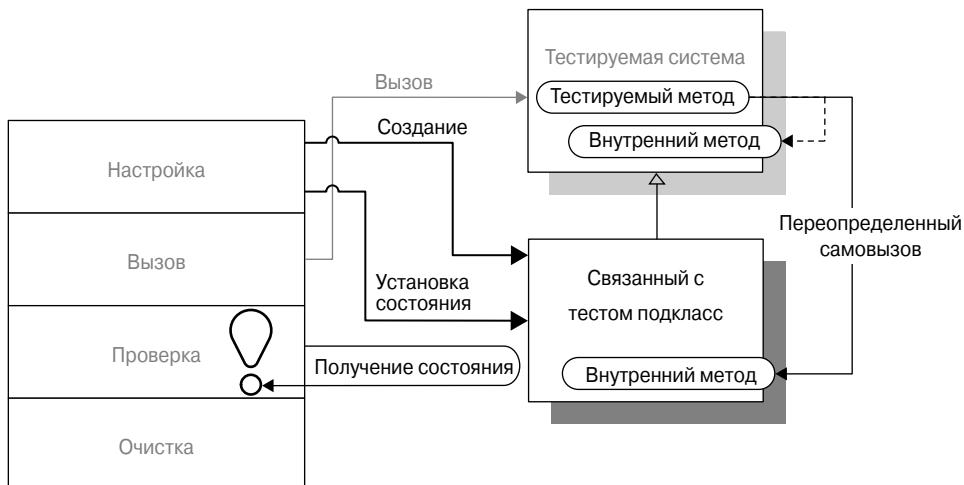


Рис. 11.12. Использование связанныго с тестом подкласса (Test-Specific Subclass). Если все остальные попытки завершатся неудачно, всегда можно попытаться создать подкласс тестируемой системы для изменения или предоставления необходимой для тестирования функциональности

Основным предварительным условием использования *связанного с тестом подкласса* (Test-Specific Subclass) тестируемой системы является необходимость самовызыва открытых методов тестируемой системы, в которых реализована переопределенная тестом функциональность. Лучше всего это удается при использовании небольших методов, ре-

шающих единственную задачу! Основным недостатком этого подхода является опасность случайного переопределения поведения, которое проверяется тестом.

Также можно создать подкласс вызываемого компонента и вставить в него связанное с тестом поведение, превращая его в *тестовый двойник в виде подкласса* (Subclassed Test Double) (рис. 11.13). Такая стратегия безопаснее, чем создание подкласса тестируемой системы, так как отсутствует вероятность случайного переопределения тестируемого поведения. Главная задача заключается в том, чтобы заставить тестируемую систему использовать связанный с тестом подкласс (Test-Specific Subclass) вместо вызываемого компонента. На практике это означает, что придется воспользоваться одним из механизмов вставки или поиска зависимости (кроме случаев, когда вызываемый компонент представляет собой Singleton). Если тестируемая система использует Singleton через вызов статического метода `soleInstance` класса с жестко закодированным методом, для возврата из метода `soleInstance` экземпляра *тестового двойника* (Test Double) можно создать подкласс класса Singleton и присвоить его переменной класса `soleInstance` экземпляр *тестового двойника* (Test Double). Если переменная класса имеет фиксированный тип Singleton, возвращаемый двойник должен быть *тестовым двойником в виде подкласса* (Subclassed Test Double). Хотя такая техника часто используется для получения другой службы от *локатора служб* (Service Locator), *тестовый двойник в виде подкласса* (Subclassed Test Double) можно использовать непосредственно без промежуточного локатора служб (Service Locator).

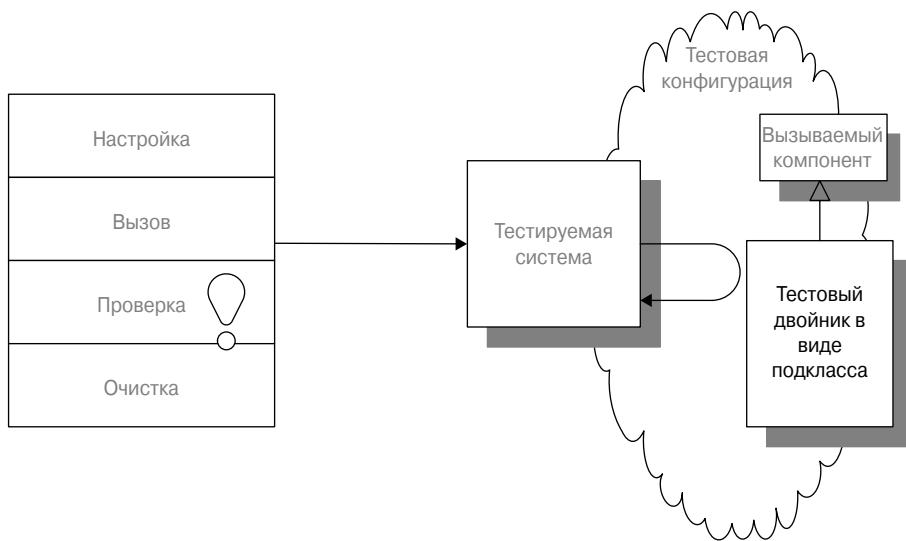


Рис. 11.13. Использование тестового двойника (Test Double), являющегося подклассом вызываемого компонента. Одним из способов создания двойника является написание подкласса настоящего класса и переопределение реализации методов, в которых необходимо контролировать опосредованный ввод или проверять опосредованный вывод

Другие способы интеграции тестов в существующий код

Если ни один из перечисленных выше способов не подходит для интеграции тестов, не все потеряно. В рукаве осталось еще немногих фокусов.

Ловушка для теста (Test Hook) — это “слон в посудной лавке”, о котором никто не хочет даже говорить, так как это приводит к появлению *логики теста в продукте* (Test Logic in Production). Но такие ловушки считаются допустимыми, когда приходится тестируировать существующий код и ни один из перечисленных способов не позволяет безболезненно интегрировать тесты. Лучше всего использовать ловушки в виде “переходной” стратегии, позволяющей автоматизировать *тесты на основе сценария* (Scripted Test, с. 319) или *записанные тесты* (Recorded Test, с. 312) для создания *страховочной сети* (Safety Net, с. 79) на время широкомасштабного рефакторинга для более полного внедрения тестов. В идеальном случае после переработки кода можно создать тесты на основе описанных ранее способов и полностью избавиться от *ловушек для теста* (Test Hook).

Майкл Фезэрс [WEwLC] описывал еще несколько способов замены зависимостей связанным с тестом кодом под общим названием “поиск *объектных швов*”. Например, вызываемую библиотеку можно заменить библиотекой, специально разработанной в целях тестирования. Таким образом, можно разорвать кажущуюся жесткой зависимость. Большинство этих способов плохо подходят в ситуации динамической подмены зависимости отдельными тестами, так как требуют внесения изменений в характеристики среды. Но “*объектные швы*” являются хорошим способом тестирования унаследованного кода в целях рефакторинга с последующим применением описанных ранее способов разрыва зависимости.

Для установки поведения *тестовый двойник* (Test Double) можно воспользоваться ориентированным на аспекты программированием. Для этого можно определить тестовую точку, соответствующую месту обращения тестируемой системы к вызываемому компоненту. Хотя для этого необходима поддерживающая данную парадигму среда разработки, сгенерированный ею код необязательно вставлять в продукт. В результате такой способ может применяться даже в средах, враждебных ориентированному на аспекты программированию.

Другие сферы применения тестовых двойников

Выше рассматривалось тестирование опосредованного ввода и вывода. Теперь рассмотрим другие сферы применения *тестовых двойников* (Test Double).

Эндоскопическое тестирование

Тим Маккиннон впервые ввел концепцию **эндоскопического тестирования** (endoscopic testing) [ET] в своей первой публикации о *подставных объектах* (Mock Object). При таком подходе основное внимание уделяется тестированию изнутри через передачу *подставного объекта* (Mock Object) в качестве аргумента тестируемого метода. Это позволяет проверять некоторые аспекты внутреннего поведения тестируемой системы, которые снаружи увидеть невозможно.

Классическим примером является использование подставного класса коллекции, предварительно наполненного всеми ожидаемыми элементами коллекции. Если тестируемая система пытается добавить неожиданный элемент, утверждение внутри подставной коллекции оказывается ложным. При этом в отчете о неудачном завершении теста выводится весь внутренний стек вызовов. Если среда разработки поддерживает остановку выполнения при генерации определенного исключения, в момент неудачного завершения теста можно проверить значения локальных переменных.

Разработка на основе потребностей

Дальнейшим развитием эндосякопического тестирования является “разработка на основе потребностей” [MRNO], при которой зависимости тестируемой системы определяются вместе с написанием тестов. Такой подход “извне внутрь” к написанию и тестированию программного обеспечения комбинирует в себе элегантность традиционного подхода “сверху вниз” к написанию кода на основе тестов при поддержке *подставных объектов* (Mock Object). Он позволяет создавать и тестировать программное обеспечение уровень за уровнем, и разработка внешнего уровня начинается до реализации более низких уровней.

Разработка на основе потребностей сочетает преимущества разработки на основе тестов (описание программного обеспечения языком тестов до начала разработки) и инкрементного подхода к проектированию, исключающего догадки о возможных применениях вызываемого класса.

Ускорение создания тестовой конфигурации

Ускорение генерации *новой тестовой конфигурации* (Fresh Fixture, с. 344) является еще одним назначением тестовых двойников (Test Double). Если тестируемой системе необходимо взаимодействовать с другими объектами (сложными в создании из-за множества зависимостей), вместо сложной сети объектов можно создать единственный *тестовый двойник* (Test Double). Если такой способ применяется к сетям объектов сущностей, она называется *обрезание цепочки сущностей* (Entity Chain Snipping; см. *Тестовая заглушка*, Test Stub).

Ускорение работы тестов

Тестовый двойник (Test Double) может служить заменой для медленных компонентов. Например, замещение реляционной базы данных хранящимся в памяти *поддельным объектом* (Fake Object) может сократить время работы теста на несколько порядков! Дополнительные трудозатраты на создание *поддельной базы данных* (Fake Database) более чем компенсируется сокращенным временем ожидания и повышением качества из-за ускорения обратной связи и повышения частоты запуска тестов. Дополнительная информация по данной теме приводится во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351.

Другие аргументы

Поскольку многие тесты требуют замены настоящего вызываемого компонента *тестовым двойником* (Test Double), как убедиться в правильности работы продукта с реальным вызываемым компонентом? Конечно, приемочные тесты должны проверить поведение с использованием настоящего вызываемого компонента (кроме случаев, когда настоящие вызываемые компоненты являются интерфейсами к другим системам, которые приходится менять на заглушки во время тестирования на одном компьютере). Необходимо написать специальный вариант *теста конструктора* (Constructor Test; см. *Тестовый метод*, Test Method), или “тест инициализации заменяемого компонента”, для проверки правильности установки настоящего вызываемого компонента. Моментом созда-

ния такого теста обычно является запуск первого теста, замещающего вызываемый компонент *тестовым двойником* (Test Double). Обычно в этот момент начинается использование механизма установки тестовых двойников.

Наконец, стоит соблюдать осторожность и не попадаться в “ловушку нового молотка”⁵. Злоупотребление *тестовыми двойниками* (Test Double), а особенно *подставными объектами* (Mock Object) и *тестовыми заглушками* (Test Stub), может привести к появлению *зарегулированных программ* (Overspecified Software; см. “Хрупкий” тест, Fragile Test, с. 277), так как в тестах кодируется зависящая от реализации информация о дизайне. Если *тестовый двойник* (Test Double) зависит от изменения дизайна и слишком много тестов зависят от этого двойника, изменение дизайна оказывается слишком сложной задачей.

Что дальше

В этой главе рассматривались способы тестирования программного обеспечения с опосредованными вводом и выводом. В частности, рассматривались концепция *тестовых двойников* (Test Double) и различные способы их установки. В главе 12, “Организация тестов”, основное внимание будет уделено стратегиям организации кода тестов в *тестовые методы* (Test Method) и *вспомогательные методы теста* (Test Utility Method, с. 610), реализованные в *классах теста* (Testcase Class, с. 401) и *вспомогательных классах теста* (Test Helper, с. 651).

⁵ Когда в руках новый молоток, хочется забить все гвозди.

Глава 12

Организация тестов

О чём идет речь в этой главе

В предыдущих главах рассматривались способы взаимодействия с тестируемой системой в целях проверки ее поведения. В данной главе основное внимание уделяется организации тестового кода с целью упрощения поиска и улучшения понимания.

Основным элементом организации тестового кода является *тестовый метод* (Test Method, с. 378). Выбор содержимого и места расположения тестового метода является центральным вопросом организации тестов. Если в системе присутствует несколько тестов, их организация не будет играть большой роли. Но с ростом количества тестов до сотен организация становится критическим фактором при оценке простоты поиска и понимания тестов.

В начале этой главы рассматривается содержимое тестовых методов. Затем описываются *классы теста* (Testcase Class, с. 401), внутри которых размещаются *тестовые методы* (Test Method). Именование тестов тесно связано с подходом к их организации, а значит, данная проблема будет рассматриваться следующей. После этого можно обратить внимание на организацию *классов теста* (Testcase Class) в наборы тестов и размещение кода тестов. В конце главы речь идет о повторном использовании кода тестов и приводятся рекомендации по его размещению.

Базовые механизмы инструментария xUnit

Инфраструктура xUnit обеспечивает множество средств для организации тестов. На вопрос “Где разместить код тестов?” можно ответить, вставив код в *тестовый метод* (Test Method) *класса теста* (Testcase Class). После этого для создания *объекта набора тестов* (Test Suite Object, с. 414), содержащего все тесты из *класса теста* (Testcase Class), можно использовать *обнаружение тестов* (Test Discovery, с. 420) или *перечисление тестов* (Test Enumeration, с. 425). Программа запуска тестов (Test Runner, с. 405) вызывает метод *объекта набора тестов* (Test Suite Object) для запуска всех *тестовых методов* (Test Method).

Размер тестовых методов

Тестовое условие (test condition) — это требующая доказательства функциональность тестируемой системы. Его можно описать в терминах начального состояния системы, способа обращения к системе, ожидаемого ответа и ожидаемого завершающего состояния. **Тестовый метод** (Test Method) представляет собой последовательность операторов на языке написания тестов, описывающих одно или несколько тестовых условий (рис. 12.1). Что должно входить в *тестовый метод* (Test Method)?

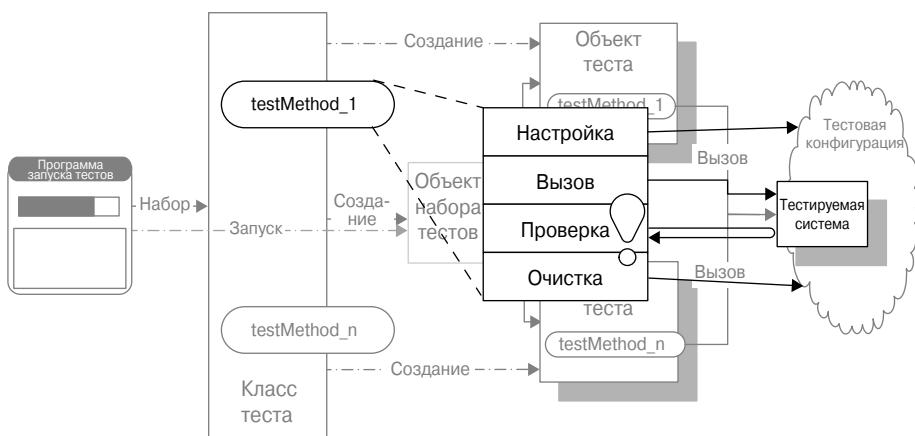


Рис. 12.1. Четыре фазы типичного теста. Каждый тестовый метод (Test Method) содержит реализацию четырехфазного теста (Four-Phase Test, с. 388), проверяющего единственное тестовое условие. Не все фазы четырехфазного теста должны находиться внутри тестового метода

Многие приверженцы чистого подхода предпочитают *проверять одно условие за тест* (Verify One Condition per Test, с. 99), так как при этом обеспечивается хорошая локализация дефектов (Defect Localization, с. 78). Иначе говоря, когда тест завершается неудачно, источник проблемы в тестируемой системе очевиден, так как каждый тест проверяет ровно одно тестовое условие. Такой подход очень отличается от тестирования вручную, когда разработчик создает длинные тесты с проверкой нескольких условий, чтобы сэкономить время на создании предварительных условий для каждого теста. При создании автоматизированных тестов на базе инфраструктуры xUnit существует несколько способов много-кратной генерации тестовой конфигурации (как показано в главе 8, “Управление временной тестовой конфигурацией”), поэтому в автоматизированных тестах будет проверяться по одному условию на тест. Тест, проверяющий слишком много тестовых условий, называется “энергичный” тест (Eager Test; см. *Рулетка утверждений*, Assertion Roulette, с. 264) и рассматривается как запах кода.

Проверяющий единственное условие тест вызывает одну ветвь выполнения тестируемой системы и только она вызывается при каждом запуске теста. В результате получается *повторяемый тест* (Repeatable Test, с. 81). Это значит, что количество тестовых методов не должно быть меньше ветвей выполнения кода. Можно ли по-другому обеспечить полное покрытие кода тестами? Управляемость такого шаблона является следствием изоля-

ции тестируемой системы (Isolate the SUT, с. 97) при создании модульных тестов для каждого класса. В результате основное внимание уделяется ветвям выполнения единственного объекта. Кроме того, поскольку каждый тест должен проверять только одну ветвь выполнения, каждый тестовый метод должен состоять из строго последовательных операторов, описывающих происходящее в этой ветви¹. Еще одной причиной проверки единственного условия на тест является *минимизация пересечения тестов* (Minimize Test Overlap, с. 98), чтобы снизить количество тестов, модифицируемых одновременно с поведением тестируемой системы.

Брайан Маррик разработал интересное компромиссное решение, которое называется “пока мы здесь” и использует тестовую конфигурацию для запуска дополнительных проверок и утверждений. Маррик явно сопровождает эти элементы комментариями, чтобы при модификации тестируемой системы устаревшие элементы можно было просто удалить. Подобная стратегия позволяет избежать обслуживания дополнительного тестового кода.

Тестовые методы и классы тестов

Тестовый метод (Test Method) существует в пределах *класса теста* (Testcase Class). Должен ли существовать один *класс теста* (Testcase Class) на все приложение или необходимо создавать один класс на каждый *тестовый метод* (Test Method)? Конечно, правильный ответ где-то между этими крайними точками зрения, и он будет меняться с развитием проекта.

Класс теста для каждого класса

Первые *тестовые методы* (Test Method) можно разместить в одном классе теста. С ростом количества *тестовых методов* (Test Method) возникает необходимость разделения *класса теста* (Testcase Class), чтобы на каждый тестируемый класс приходился один *класс теста* (Testcase Class). Это снизит количество *тестовых методов* (Test Method) в каждом классе (рис. 12.2). С ростом полученных *классов теста* (Testcase Class) происходит дальнейшее дробление. В процессе деления придется принимать решение о распределении тестовых методов по классам.

Класс теста для каждой функции

Один из подходов (рис. 12.3) предполагает размещение в одном *классе теста* (Testcase Class) всех *тестовых методов* (Test Method), проверяющих определенную функцию тестируемой системы (под функцией подразумевается один или несколько методов и атрибутов, реализующих некоторую возможность тестируемой системы). Это позволяет легко видеть все тестовые условия для данной функции. (Для достижения подобной ясности можно воспользоваться подходящим соглашением об именовании тестов.) Недостатком такого решения является вероятность дублирования кода создания тестовой конфигурации в нескольких *классах теста* (Testcase Class).

¹ Содержащий *условную логику теста* (Conditional Test Logic, с. 243) *тестовый метод* (Test Method) является признаком приспособления теста к различным обстоятельствам, когда тест не управляет всем опосредованным выводом тестируемой системы или пытается проверять сложные ожидаемые состояния внутри одного *тестового метода* (Test Method).

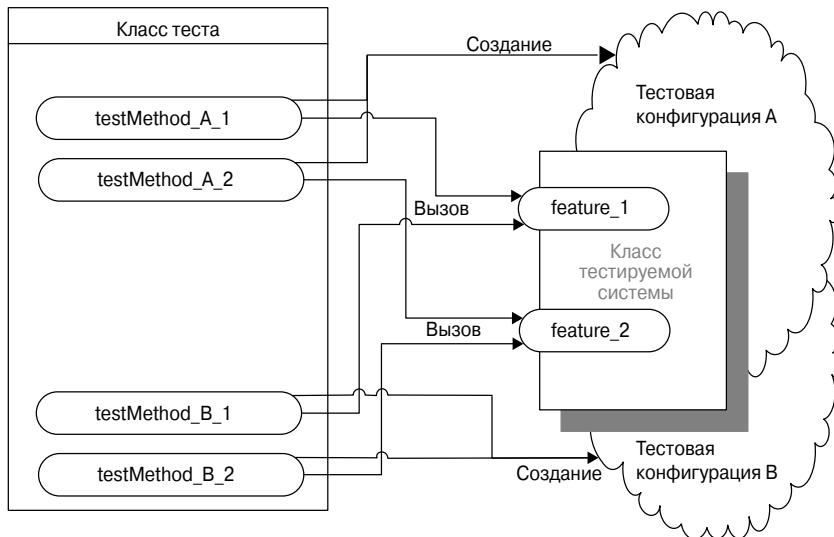


Рис. 12.2. Класс продукта с единственным классом теста (Testcase Class). При использовании данного шаблона все тестовые методы (Test Method) для поведения тестируемого класса расположены в одном классе теста (Testcase Class). Каждый тестовый метод создает отдельную тестовую конфигурацию, делая это самостоятельно или с помощью методов создания (Creation Method, с. 441)

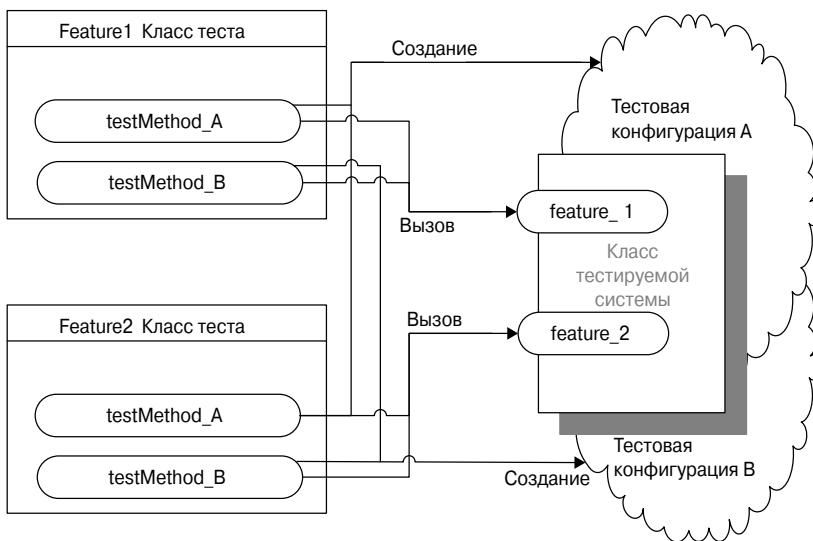


Рис. 12.3. Класс продукта и класс теста (Testcase Class) для каждой функции. При использовании этого шаблона отдельный класс теста (Testcase Class) создается для каждой существенной функции или возможности тестируемой системы. Тестовые методы (Test Method) класса теста создают необходимые тестовые конфигурации и проверяют различные аспекты функции

Класс теста для каждой тестовой конфигурации

Противоположный подход (рис. 12.4) подразумевает группирование *тестовых методов* (Test Method), требующих одинаковой тестовой конфигурации (одинаковых предусловий), в один *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639). В таком случае код создания тестовой конфигурации можно разместить в методе *setUp* (*неявная настройка*, Implicit Setup, с. 449), но тогда тестовые условия для каждой функции будут распределены между несколькими *классами теста* (Testcase Class).

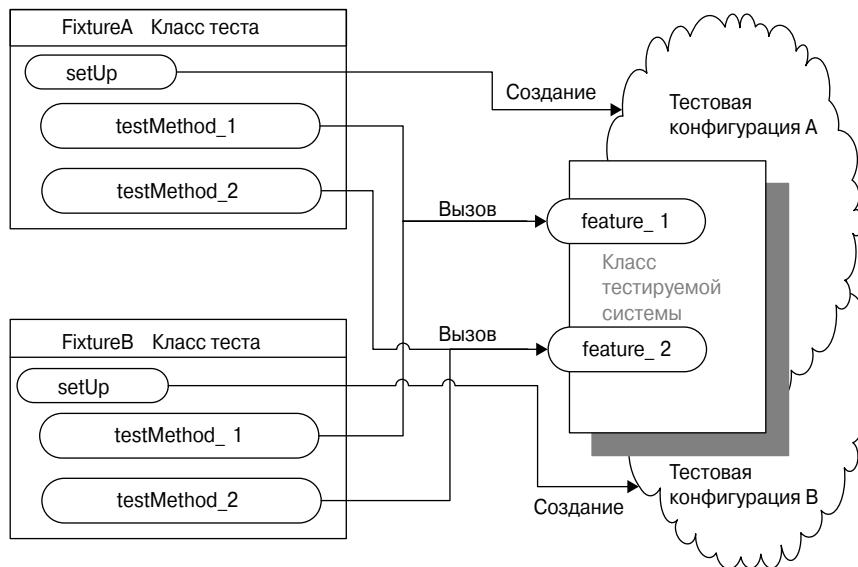


Рис. 12.4. Класс продукта и класс теста (Testcase Class) для каждой тестовой конфигурации. При использовании этого шаблона отдельный класс теста создается для каждой возможной тестовой конфигурации. Тестовые методы (Test Method) этого класса проверяют различные функции, используя общую отправную точку

Выбор стратегии организации тестовых методов

Очевидно, что не существует единственно верного подхода, которому всегда стоит следовать. Вариант может быть лучшим только в конкретной ситуации. *Класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture) обычно используется при создании модульных тестов для имеющих состояние объектов, когда каждый метод должен проверяться в каждом состоянии объекта. *Класс теста для каждой функции* (Testcase Class per Feature, с. 633) является более подходящим при создании приемочных тестов для *фасада служб* (Service Facade) [CJ2EEP]. Все тесты для заметной клиенту функции оказываются в одном месте. Данный шаблон часто используется, когда применяется *предварительно созданная тестовая конфигурация* (Prebuilt Fixture, с. 454), так как тесты могут не иметь логики создания тестовой конфигурации. Если каждому тесту необходима немножко отличающаяся тестовая конфигурация, наиболее подходящими считаются шаб-

лоны класса для каждой функции (Testcase Class per Feature) и делегированная настройка (Delegated Setup, с. 437) для создания тестовых конфигураций.

Соглашения об именовании тестов

Имена классов и методов тестов имеют решающее значение для обеспечения простоты поиска и понимания тестов. Покрытие кода тестами можно сделать более очевидным, если каждый *тестовый метод* (Test Method) систематически именовать в соответствии с проверяемым условием. Вне зависимости от схемы организации тестовых методов комбинация имен тестового пакета, *класса теста* (Testcase Class) и *тестового метода* (Test Method) должна доносить следующую информацию:

- имя класса в тестируемой системе;
- имя вызываемого метода или функции;
- важные характеристики любых входных значений, связанных с вызовом тестируемой системы;
- важную информацию о состоянии тестируемой системы или ее зависимостях.

Эти элементы являются “входной” частью тестового условия. Очевидно, это достаточно большой объем информации, чтобы донести его в двух именах, но если этого достаточно, результат превзойдет все ожидания. Один взгляд на имена классов и методов позволит узнать, для каких условий существуют тесты. Пример такой схемы именования показан на рис. 12.5.

На рис. 12.5 продемонстрированы преимущества включения в имя “ожиданий” из тестового условия:

- ожидаемый вывод (ответы) при вызове тестируемой системы;
- ожидаемое состояние тестируемой системы после вызова и ее зависимости.

Эта информация может быть включена в имя *тестового метода* (Test Method) с префиксом “should”. Если подобная схема делает имена слишком длинными, можно всегда узнать ожидаемый вывод, заглянув внутрь *тестового метода* (Test Method). (Во многих реализациях xUnit “поощряется” вставка слова “test” в начало имен тестов, чтобы методы автоматически обнаруживались и добавлялись в *объект набора тестов* (Test Suite Object). Это ограничивает свободу именования по сравнению с вариантами индикации тестовых методов с помощью **атрибутов методов** (method attribute) или **аннотаций** (annotation).)

Организация наборов тестов

Класс теста (Testcase Class) выступает в роли *фабрики наборов тестов* (Test Suite Factory; см. *Перечисления тестов*, Test Enumeration), возвращая *объект набора тестов* (Test Suite Object) с *коллекцией объектов теста* (Testcase Object, с. 410). Каждый объект представляет *тестовый метод* (Test Method) (рис. 12.6). Это принятый по умолчанию механизм организации, предоставляемый инфраструктурой xUnit. Большинство программ запуска тестов (Test Runner) позволяют любому классу выступать в роли *фабрики наборов тестов* (Test Suite Factory) через реализацию шаблона Factory Method [GOF], которая обычно называется *suite*.



Рис. 12.5. Класс продукта и класс теста для каждой тестовой конфигурации. В такой ситуации имя класса может описывать тестовую конфигурацию, а имя метода описывает входные параметры и ожидаемый вывод

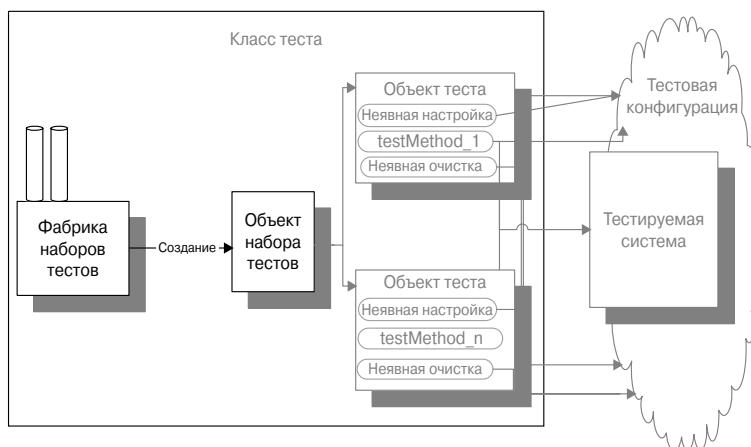


Рис. 12.6. Класс теста (Testcase Class) выступает в роли фабрики наборов тестов (Test Suite Factory). По умолчанию класс теста (Testcase Class) выступает в роли фабрики наборов тестов (Test Suite Factory) и создает объект набора тестов (Test Suite Object), необходимый программе запуска тестов (Test Runner). Кроме того, можно перечислить конкретное подмножество запускаемых тестов, создав фабрику наборов тестов (Test Suite Factory), возвращающую объект набора тестов (Test Suite Object) с интересующими разработчика тестами

Запуск групп тестов

Часто возникает необходимость запускать группы тестов (набор тестов), но это не должно влиять на способ организации тестов. Популярным решением считается создание специальной *фабрики наборов тестов* (Test Suite Factory) под названием AllTests для каждого пакета тестов. Но не стоит на этом останавливаться: можно создать *именованные наборы тестов* (Named Test Suite, с. 604) для любой коллекции тестов, которые должны запускаться вместе. Хорошим примером может служить *набор с подмножеством* (Subset Suite; см. *Именованный набор тестов*, Named Test Suite), позволяющий запускать только те тесты, которым нужно программное обеспечение, “выложенное” (или “не выложенное”!) на сервер. Обычно существует как минимум один *набор с подмножеством* (Subset Suite) для всех модульных тестов и еще один — для приемочных тестов (так как они работают очень долго). В некоторых реализациях xUnit поддерживается *выбор тестов* (Test Selection, с. 429), который можно использовать вместо создания *наборов с подмножеством* (Subset Suite).

Подобное группирование тестов на этапе выполнения часто является отражением среды, в которой они запускаются. Например, может существовать один *набор с подмножеством* (Subset Suite), включающий в себя все тесты, работающие без базы данных, и еще один набор с тестами, зависящими от базы данных. Точно так может существовать отдельный *набор с подмножеством* (Subset Suite) для тестов, зависящих или не зависящих от сервера. Если пакет содержит различные наборы тестов, можно определить *набор наборов* (Suite of Suites) под названием AllTests, который состоит из *наборов с подмножеством* (Subset Suite). В результате любой тест, который входит в любой *набор с подмножеством* (Subset Suite), автоматически станет частью набора AllTests.

Запуск единственного теста

Предположим, что один *тестовый метод* (Test Method) во всем классе теста завершился неудачно. Принято решение установить точку останова на определенном методе, но он вызывается каждым тестом. Первой реакцией будет щелчок на кнопке Go при каждом срабатывании точки останова, пока метод не будет вызван интересующим разработчика тестом. Одним из решений является комментирование остальных *тестовых методов* (Test Method), чтобы они не запускались. Другие тестовые методы также можно переименовать, чтобы механизм *обнаружения тестов* (Test Discovery) не распознал их как тесты. В реализациях xUnit, использующих атрибуты методов и аннотации, остальным тестовым методам можно назначить атрибут Ignore. С каждым из подходов связан потенциальный риск *потери теста* (Lost Test; см. *Ошибки в продукте*, Production Bugs, с. 303), хотя подход на основе атрибута Ignore создает напоминание об игнорируемых тестах. В реализациях xUnit, предоставляющих *обозреватель набора тестов* (Test Tree Explorer; см. *Программа запуска тестов*, Test Runner), можно просто выбрать один тест из иерархического представления набора (рис. 12.7).

Если ни один из вариантов не является доступным, для запуска одного теста можно воспользоваться *фабрикой набора тестов* (Test Suite Factory). Но возникает вопрос: “Не предназначены ли наборы тестов для запуска групп тестов из разных *классов теста* (Testcase Class)?” Вообще-то, предназначены, но это не значит, что их нельзя использовать для других целей. Можно определить *набор из одного теста* (Single Test Suite; см. *Именованный набор тестов*, Named Test Suite), обычно называемый MyTest, который бу-

дет запускать определенный тест. Для этого вызывается конструктор *класса теста* (Testcase Class), которому в качестве аргумента передается имя конкретного *тестового метода* (Test Method).

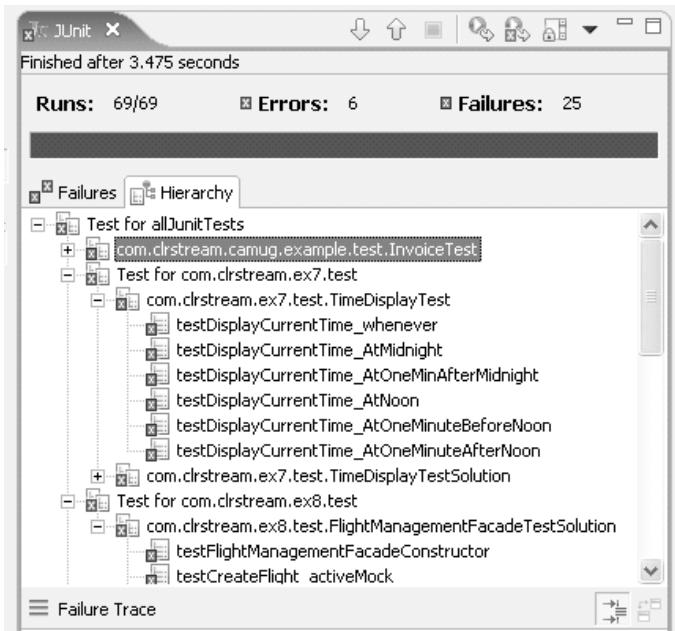


Рис. 12.7. Обозреватель набора тестов (Test Tree Explorer), в котором показана структура тестов в наборе. С его помощью можно просматривать структуру набора тестов и запускать отдельные тесты или подмножества набора

Повторное использование кода тестов

Дублирование тестового кода (Test Code Duplication, с. 254) может значительно увеличить стоимость написания и обслуживания тестов. К счастью, существует достаточно способов повторного использования тестовой логики. Очень важно, чтобы повторное использование не снижало ценность *тестов как документации* (Tests as Documentation, с. 79). Не рекомендуется повторно использовать *тестовые методы* (Test Method) в других условиях (например, с другими тестовыми конфигурациями), так как подобный подход обычно является признаком *гибких тестов* (Flexible Test; см. *Условная логика теста*, Conditional Test Logic, с. 243), проверяющих разные условия в разных обстоятельствах. В большинстве случаев повторное использование тестового кода достигается через *неявную настройку* (Implicit Setup) или *вспомогательные методы теста* (Test Utility Method, с. 610). Основным исключением можно считать повторное использование *тестовых двойников* (Test Double, с. 538) несколькими тестами. В таком случае классы тестовых двойников можно рассматривать как специальные *вспомогательные классы теста* (Test Helper, с. 651).

Во многих реализациях xUnit предоставляется специальный *суперкласс теста* (Testcase Superclass, с. 646), который обычно называется “*TestCase*”. От него должны (иногда обязательно) непосредственно или опосредованно наследовать все *классы теста* (Testcase Class) (рис. 12.8). Если в пределах *класса теста* (Testcase Class) существуют полезные вспомогательные методы, которые необходимо повторно использовать в других классах, имеет смысл создать один или несколько *суперклассов теста* (Testcase Superclass), от которых будут наследовать другие классы теста. В такой ситуации необходимо внимательно следить за методами, которые должны видеть типы или классы, расположенные в различных пакетах тестируемой системы, — корневой *суперкласс теста* (Testcase Superclass) не должен непосредственно зависеть от этих типов или классов, так как это приведет к циклическому графу зависимостей. Можно попытаться создать *суперкласс теста* (Testcase Superclass) в каждом тестовом пакете, чтобы сделать зависимости тестовых классов ациклическими. Альтернативным вариантом является создание *вспомогательного класса теста* (Test Helper) для каждого пакета предметной области и размещение различных *вспомогательных классов теста* (Test Helper) в соответствующих пакетах тестов. Таким образом, *класс теста* (Testcase Class) не связан с единственным *суперклассом теста* (Testcase Superclass) и может просто “использовать” подходящие вспомогательные *классы теста* (Test Helper).

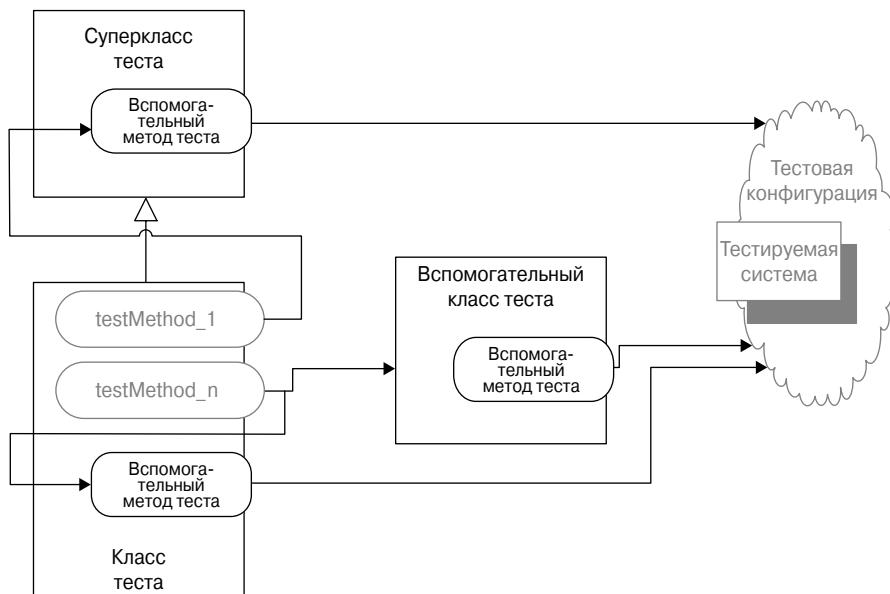


Рис. 12.8. Варианты размещения вспомогательных методов теста (Test Utility Method). Основным критерием при принятии решения является прогнозируемый масштаб повторного использования тестовых методов (Test Method)

Наследование и повторное использование класса теста

Самой распространенной причиной наследования методов из *суперкласса теста* (Testcase Superclass) является доступ к вспомогательным *методам теста* (Test Utility Method). Еще одной причиной является тестирование инфраструктур и их встраиваемых

модулей. В такой ситуации имеет смысл создать тест соответствия, описывающий общее поведение встраиваемого модуля с помощью шаблона Template Method [GOF]. Реализация шаблона будет вызывать методы подкласса, связанного с тестируемым встраиваемым модулем, для проверки конкретных условий модуля. Подобный сценарий встречается достаточно редко, поэтому здесь он рассматриваться не будет. Дополнительная информация приводится в статье [FaT].

Организация тестовых файлов

На данном этапе возникает следующий вопрос: “Где размещать *классы теста* (Testcase Class)?” Очевидно, эти классы должны находиться в хранилище исходного кода вместе с кодом продукта. Даже при выполнении этого условия остается богатый выбор ответов на поставленный вопрос. Выбранная стратегия упаковки тестов зависит от используемой среды — во многих интегрированных средах существуют ограничения, делающие некоторые стратегии нереализуемыми. Ключевым условием является *недопуск логики тестов в код продукта* (Keep Test Logic Out of Production Code, с. 99). При этом должна существовать возможность найти соответствующий тест для каждого фрагмента кода или каждой функции.

Встроенные автотесты

При использовании встроенных автотестов тесты включаются в код продукта и могут быть запущены в любой момент. Никаких усилий по их разделению не прилагается. Во многих организациях соблюдается условие *недопуска логики тестов в код продукта* (Keep Test Logic Out of Production Code, с. 99), поэтому встроенные автотесты не являются предпочтительным вариантом. Соблюдение этого условия особенно важно в средах с ограниченным объемом памяти, когда тесты не должны отнимать ценный ресурс.

В некоторых средах разработки хранение тестов и кода продукта в одном месте поощряется. Например, в пакете SAP ABAP Unit поддерживается ключевое слово `For Testing`, позволяющее системе удалять тесты при переносе кода в продукт.

Пакеты тестов

Если принято решение разместить *классы теста* (Testcase Class) в отдельных пакетах, их можно организовать несколькими способами. Тесты можно хранить отдельно, разместив их в одном или нескольких пакетах, но в пределах одного дерева исходного кода. Также тесты можно хранить в том же логическом пакете, где хранится код, но физически разместить в параллельном дереве исходного кода. Последний подход часто используется в языке Java, так как позволяет избежать проблемы, когда тесты не видят “зашитенных пакетом” методов тестируемой системы². Некоторые среды разработки могут отказать в реализации такого подхода, настаивая, чтобы пакет был целиком расположен в преде-

² В языке Java обеспечивается еще один способ обхода проблемы видимости: можно определить собственный тестовый `Security Manager`, чтобы позволить тестам получать доступ ко всем методам тестируемой системы, а не только к “зашитенным пакетом”. Это универсальное решение проблемы, но оно требует понимания принципов работы загрузчиков классов Java. В других языках эквивалентной функциональности (или такой проблемы!) может просто не существовать.

лах одной папки или проекта. При использовании тестовых пакетов в каждом пакете с кодом продукта на этапе компиляции может потребоваться применение **фильтра для тестов** (test stripper), который исключит тесты из откомпилированной версии продукта.

Зависимости тестов

Независимо от подходов к хранению исходного кода и управлению им необходимо обеспечить удаление любых *зависимостей продукта от теста* (Test Dependency in Production; см. *Логика теста в продукте*, Test Logic in Production, с. 257), так как даже фильтр для тестов не сможет их удалить, если код продукта зависит от их присутствия. Такое требование обуславливает повышенное внимание к зависимостям классов. Кроме того, нельзя допускать *присутствия логики теста в продукте* (Test Logic in Production), так как при этом тестируемый код не совпадает с тем кодом, который будет вводиться в эксплуатацию. Более подробно эта тема рассматривается в главе 6, “Стратегия автоматизации тестирования”.

Что дальше

Рассмотрев подходы к организации кода тестов, можно переходить к шаблонам тестирования. Первые шаблоны рассматриваются в главе 13, “Тестирование с использованием баз данных”.

Глава 13

Тестирование с использованием баз данных

О чём идет речь в этой главе

В главе 12, “Организация тестов”, были приведены способы организации тестового кода. В этой главе рассматриваются проблемы, возникающие, когда приложение включает в себя базу данных. Приложения с базами данных создают определенные сложности при написании автоматизированных тестов. Операции с базами данных выполняются значительно медленнее, чем операции с данными в оперативной памяти. В результате взаимодействующие с базой тесты работают на порядок медленнее, чем тесты, полностью умещающиеся в оперативной памяти.

Даже если игнорировать вероятность появления *медленных тестов* (Slow Tests, с. 289), базы данных являются источником множества запахов тестов. Одни из этих запахов являются следствием сохранения данных между запусками тестов. Другие представляют собой результат решения повторно использовать одну и ту же конфигурацию несколькими тестами. Некоторые из этих запахов рассматривались в главе 9, “Управление постоянными тестовыми конфигурациями”. В настоящей главе данная тема и особенности тестирования с использованием баз данных рассматривается более подробно.

Тестирование с использованием баз данных

Вот первый и самый важный совет по данной теме:

Если есть возможность не использовать базу данных при тестировании, не используйте ее!

Совет может выглядеть категоричным, но на это есть свои причины. С базами данных связан ряд сложностей в проектировании приложений и особенно в проектировании тестов. Тесты, нуждающиеся в базе данных, работают в среднем на два порядка медленнее, чем тесты без базы данных.

Причины тестирования с базами данных

Многие приложения используют базу данных для длительного хранения объектов или данных. База данных является необходимым строительным блоком приложения. Таким образом, использование “песочницы” с базой данных (Database Sandbox, с. 658) для изолирования разработчиков и тестеров от эксплуатируемого продукта (и друг от друга) является частью практически каждого проекта (рис. 13.1).

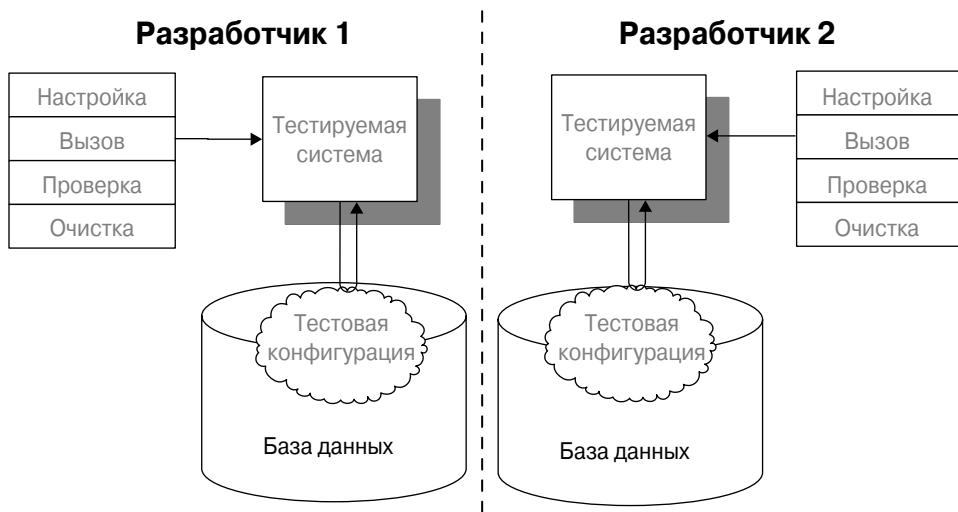


Рис. 13.1. “Песочница” с базой данных (Database Sandbox) для каждого разработчика.
Совместное использование “песочницы” несколькими разработчиками — не что иное, как “экономия на спичках” (это все равно что электрик и сантехник, работающие над одной стеной одновременно)

Проблемы, связанные с базами данных

Использование баз данных приводит к появлению ряда проблем, усложняющих автоматизированное тестирование. Многие из этих проблем связаны с постоянной природой тестовой конфигурации. Часть этих вопросов рассматривалась в главе 9, “Управление постоянными тестовыми конфигурациями”, и кратко перечисляется здесь.

Постоянные тестовые конфигурации

Приложения с базами данных имеют ряд особенностей с точки зрения создания автоматизированных тестов. Базы данных работают значительно медленнее, чем приложения, полностью расположенные в оперативной памяти. В результате взаимодействующие с базой тесты работают на порядок медленнее, чем тесты, полностью умещающиеся в оперативной памяти. Но даже если игнорировать проблему *медленных тестов* (Slow Test), базы данных являются основным источником запахов автоматизированных тестов. Чаще всего встречаются запахи *неустойчивый тест* (Erratic Test, с. 267) и *непонятный тест* (Obscure Test, с. 230). Так как данные в базе данных потенциально существуют дольше, чем работает тест, на них стоит обратить внимание, чтобы не создавать тесты, работаю-

щие только один раз или взаимодействующие друг с другом. *Неповторяющиеся тесты* (Unrepeatable Test; см. *Нестабильный тест*, Erratic Test) и *взаимодействующие тесты* (Interacting Tests; см. *Нестабильный тест*, Erratic Test) являются прямым следствием постоянных тестовых конфигураций и причиной увеличения стоимости обслуживания тестов с развитием приложения.

Общие тестовые конфигурации

Сохранение тестовой конфигурации и совместное использование тестовой конфигурации — это далеко не одно и то же. Намеренное совместное использование тестовой конфигурации может привести к появлению *одиноких тестов* (Lonely Test; см. *Нестабильный тест*, Erratic Test), когда одни тесты зависят от других в части создания конфигурации, т.е. получаются так называемые *цепочки тестов* (Chained Tests, с. 477). Если каждому разработчику не была предоставлена “песочница” с базой данных (Database Sandbox), между разработчиками может начаться “война” запуска тестов (Test Run War, с. 274; см. *Нестабильный тест*, Erratic Test). Эта проблема возникает, когда две или более *программ запуска тестов* (Test Runner, с. 405) взаимодействуют одна с другой через доступ к одной и той же тестовой конфигурации в общем экземпляре базы данных. Каждый из этих запахов поведения является прямым следствием решения совместно использовать тестовую конфигурацию. Длительность хранения и масштаб совместного использования тестовой конфигурации непосредственно влияют на наличие или отсутствие перечисленных запахов.

Тестовые конфигурации общего характера

Существует еще одна проблема, связанная с базами данных: со временем они разрастаются в большие тестовые конфигурации *общего характера* (General Fixture; см. *Непонятный тест*, Obscure Test), которые используются разными тестами в разных целях. Такой результат особенно вероятен при использовании предварительно созданных *тестовых конфигураций* (Prebuilt Fixture, с. 454), чтобы избежать создания конфигурации для каждого теста. Также это может стать следствием использования *стандартной тестовой конфигурации* (Standard Fixture, с. 338) при выборе стратегии с *новой тестовой конфигурацией* (Fresh Fixture, с. 344). Подобный подход значительно усложняет понимание спецификации теста. В целом, база данных выступает в роли *тайного гостя* (Mystery Guest; см. *Непонятный тест*, Obscure Test) для всех тестов.

Тестирование без баз данных

Современная многоуровневая архитектура программного обеспечения [DDD, PEAA, WWW] предоставляет возможность тестирования бизнес-логики вообще без использования базы данных. Уровень бизнес-логики можно тестировать отдельно от других уровней системы. Для этого можно применять *тест уровня* (Layer Test, с. 368) с заменой **уровня доступа к данным** (data access layer) *тестовым двойником* (Test Double, с. 538) (рис. 13.2).

Если архитектура недостаточно разделена на уровни, чтобы использовать *тест уровня* (Layer Test), все еще остается возможность тестирования без настоящей базы данных с помощью *поддельной базы данных* (Fake Database; см. *Поддельный объект*, Fake Object, с. 565) или *базы данных в памяти* (In-Memory Database; см. *Поддельный объект*, Fake Ob-

jest). *База данных в памяти* (In-Memory Database) отличается тем, что хранит данные только в оперативной памяти, а значит, работает значительно быстрее, чем при чтении данных с диска. Поддельная база данных (Fake Database) на самом деле не является базой данных; это уровень доступа к данным, который выдает себя за базу данных. Как правило, *поддельная база данных* (Fake Database) значительно упрощает создание независимых тестов, поскольку она создается вместе с остальной тестовой конфигурацией как часть реализации стратегии на основе *временной новой тестовой конфигурации* (Transient Fresh Fixture; см. *Новая тестовая конфигурация*, Fresh Fixture). Несмотря на это обе стратегии предоставляют возможность работать на скоростях обращения к памяти, позволяя избежать проблем *медленных тестов* (Slow Test). Если тесты работают через открытый интерфейс test-тируемой системы, дополнительные знания о ее структуре не будут влиять на структуру тестов.

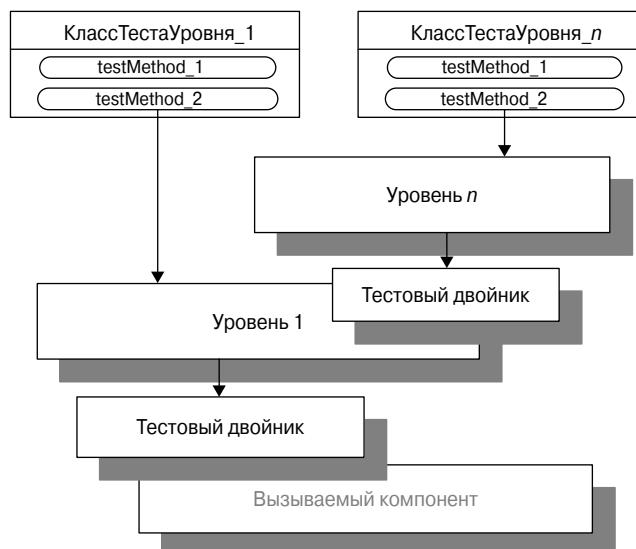


Рис. 13.2. Пара тестов уровня (Layer Test), каждый из которых проверяет свой уровень системы. Тест уровня (Layer Test) позволяет создавать каждый уровень независимо от других уровней. Подобные тесты особенно полезны, если уровень хранения данных можно заменить тестовым двойником (Test Double), снижающим чувствительность к контексту (Context Sensitivity; см. “Хрупкий” тест, Fragile Test, с. 277)

Замена базы данных *тестовым двойником* (Test Double) дает хорошие результаты, пока база данных применяется только как хранилище. Все становится интереснее, если используется функциональность, характерная для конкретного типа базы данных, например генерация номеров последовательностей или хранимых процедур. В таком случае замена базы данных становится более сложной, так как требует более тщательного проектирования с учетом тестов. Основной стратегией является инкапсуляция взаимодействия с базой данных внутри уровня доступа к данным. Если такой уровень обеспечивает функциональность доступа к данным, можно просто делегировать эти задачи “объекту базы данных”. От разработчика потребуется предоставить связанные с тестом реализации для всех элементов интерфейса уровня доступа к данным, предоставляющих

уникальную функциональность производителя; для этого отлично подходит *тестовая заглушка* (Test Stub, с. 544).

Если используются уникальные для производителя функции базы данных, придется предоставлять функциональность при выполнении тестов в памяти. Обычно замена такой функциональности *тестовым двойником* (Test Double) не требуется, так как она скрыта внутри базы данных. Подобную функциональность можно добавить в находящуюся в памяти версию приложения с помощью реализации шаблона Strategy [GOF], который по умолчанию инициализируется объектом **null** [PLoPD3]. Во время эксплуатации продукта объект **null** ничего не делает; при запуске в памяти во время теста объект Strategy предоставляет отсутствующую функциональность. Дополнительным положительным эффектом является простота смены производителя базы данных, так как ловушки для предоставления отсутствующей функциональности уже существуют (это еще один пример того, как проектирование с учетом тестирования улучшает общий дизайн приложения).

Замена базы данных (или уровня доступа к данным) в автоматизированных тестах предполагает, что тестируемую систему можно заставить использовать заменяемый объект. Обычно это делается одним из двух способов: через непосредственную *вставку зависимости* (Dependency Injection, с. 684) или через интеграцию в уровень бизнес-логики *поиск зависимости* (Dependency Lookup, с. 692) для обнаружения уровня доступа к данным.

Тестирование базы данных

Предположим, что найден способ тестирования большей части программного обеспечения без базы данных. Что будет после этого? Исчезнет ли необходимость тестирования базы данных? Конечно, нет! Нужно убедиться, что функция базы данных работает правильно, как и другие программные компоненты. Но при тестировании базы данных внимание можно сфокусировать таким образом, чтобы сократить количество тестов. Поскольку тесты с использованием базы данных работают значительно медленнее, чем тесты в памяти, количество таких тестов должно быть минимизировано.

Какие тесты базы данных потребуются? Ответ на этот вопрос зависит от способа работы с базой данных, применяемого в приложении. Если применяются хранимые процедуры, придется создавать модульные тесты для проверки их логики. Если уровень доступа к данным скрывает базу данных от бизнес-логики, необходимо создавать тесты для функциональности доступа к данным.

Тестирование хранимых процедур

Тесты для хранимых процедур можно создавать двумя способами. *Удаленный тест хранимой процедуры* (Remoted Stored Procedure Test; см. *Тест хранимой процедуры*, Stored Procedure Test, с. 662) создается на том же языке программирования, что и все остальные модульные тесты. Доступ к хранимой процедуре осуществляется через тот же механизм вызова, который применяется в логике приложения (через шаблоны Remote Proxy, Facade или объект Command [GOF]). С другой стороны, можно создавать *хранящийся в базе данных тест хранимой процедуры* (In-Database Stored Procedure Test) на том же языке, что и сами хранимые процедуры. Такие тесты работают внутри базы данных (рис. 13.3). Реализации xUnit доступны для нескольких популярных языков хранимых процедур. Одним из примеров является пакет utPLSQL.

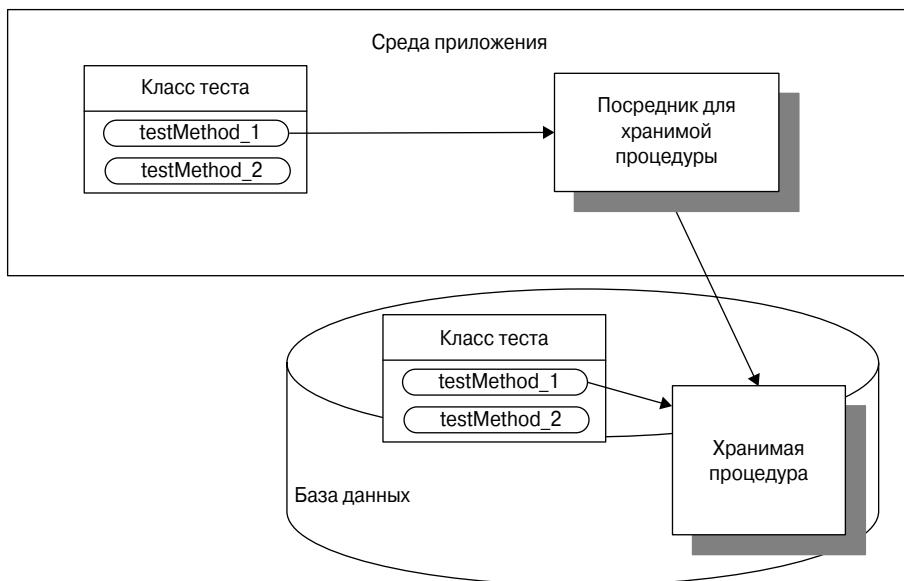


Рис. 13.3. Тестирование хранимой процедуры с помощью самопроверяющегося теста (Self-Checking Test, с. 81). Автоматизированные регрессионные тесты хранимых процедур являются очень ценным инструментом, если их можно запускать повторно и они работают достаточно быстро

Тестирование уровня доступа к данным

Для уровня доступа к данным также необходимы модульные тесты. Большая часть этих тестов должна работать через открытый интерфейс. Но некоторые тесты должны пересекать уровни, чтобы убедиться, что информация вставляется туда, куда нужно. Для этого можно воспользоваться расширениями xUnit для тестирования баз данных (например, DbUnit для Java). Расширения позволяют непосредственно вставлять данные в базу (для тестов “Read”) или проверять содержимое базы данных после тестов (для тестов “Create/Update/Delete”).

Для защиты тестовой конфигурации от сохранения во время тестирования уровня доступа к данным воспользуйтесь очисткой откатом транзакции (Transaction Rollback Teardown, с. 675). Для этого применяется шаблон DFT минимального контроллера транзакций (Humble Transaction Controller; см. *Минимальный объект*, Humble Object, с. 700). Таким образом, код чтения или записи в базу данных никогда не завершает транзакций и изменения со стороны тестируемой системы не будут сохраняться.

Еще одним способом очистки изменений, внесенных в базу данных на этапе создания тестовой конфигурации и вызова тестируемой системы, является очистка усечением таблиц (Table Truncation Teardown, с. 668). Такая техника “грубой силы” при удалении данных работает только при выделении каждому разработчику “песочницы” с базой данных (Database Sandbox) и при удалении всех данных в одной или нескольких таблицах.

Обеспечение независимости разработчиков

Тестирование базы данных требует наличия настоящей базы данных для запуска тестов. В ходе тестирования каждому разработчику нужна собственная “песочница” с базой данных (Database Sandbox). Попытка совместного использования такой “песочницы” несколькими разработчиками является “экономией на спичках”. Разработчики будут просто путаться друг у друга под ногами и тратить время¹. Существует множество различных оправданий не предоставления каждому разработчику собственной базы данных, но ни одно из них не выдерживает критики. Самые оправданные опасения связаны со стоимостью лицензии на базу данных для каждого разработчика, но даже это ограничение можно обойти через один из вариантов “виртуальной «песочницы»”. Если технология базы данных поддерживает такой подход, можно воспользоваться *схемой БД на программу запуска тестов* (DB-Schema per Test-Runner) или *схемой разбиения базы данных* (Database Partitioning Scheme).

Тестирование с базами данных (опять!)

Предположим, что удалось удачно разделить систему на уровни и обеспечить запуск большинства тестов без доступа к настоящей базе данных. Какие типы тестов должны запускаться с использованием настоящей базы данных? Ответ прост: “Как можно меньше, но не меньше!” На практике вместе с базой данных необходимо запускать минимальную презентативную выборку приемочных тестов, чтобы убедиться, что тестируемая система ведет себя одинаково как с базой данных, так и без нее. Такие тесты не обязательно должны обращаться к бизнес-логике через пользовательский интерфейс (если функциональность конкретного пользовательского интерфейса не зависит от базы данных). “Под кожей” тест (Subcutaneous Test; см. *Тест уровня*, Layer Test) подойдет в большинстве ситуаций.

Что дальше

В этой главе рассматривались специальные способы тестирования с базами данных. Это только поверхностное обсуждение взаимодействия гибких методов разработки и баз данных. (Более подробно данная тема рассматривается в [RDb].) В главе 14, “План эффективной автоматизации тестирования”, приводится обзор приведенного материала и даются рекомендации по ускорению автоматизации тестов разработчиками.

¹ Представьте, что бригада плотников пользуется одним молотком.

Глава 14

План эффективной автоматизации тестирования

О чём идет речь в этой главе

В главе 13, “Тестирование с использованием баз данных”, рассматривался набор шаблонов, которые характерны для тестирования приложений, использующих базу данных. Все эти шаблоны основаны на способах, приведенных в главе 6, “Стратегия автоматизации тестирования”, в главе 9, “Управление постоянными тестовыми конфигурациями”, и в главе 11, “Использование тестовых двойников”. Этот достаточно большой объем материала обязательно нужно освоить перед тем, как начинать эффективное тестирование с использованием и без использования баз данных!

Из этого следует важный вывод: нельзя стать экспертом в автоматизации тестирования за один вечер — необходимые навыки вырабатываются со временем. Кроме того, требуется время на изучение различных инструментов и шаблонов, доступных разработчику. В этой главе приводится план изучения шаблонов и формирования навыков и рассматривается концепция “зрелости автоматизации тестирования”, которая напоминает Capability Maturity Model (CMM).

Сложность автоматизации тестирования

Одни тесты написать сложнее, чем другие. Причиной этого может служить применение недостаточно распространенных способов или недостаток необходимых инструментов автоматизации. Ниже распространенные типы тестов перечислены в порядке возрастания сложности.

1. Простые **объекты сущностей** (entity objects; Domain Model [PEAA]).
 - Простые бизнес-классы без зависимостей.
 - Сложные бизнес-классы с зависимостями.
2. Обслуживающие объекты без состояния.
 - Отдельные компоненты с компонентными тестами.
 - Целый уровень бизнес-логики с *тестом уровня* (Layer Test, с. 368).

3. Обслуживающие объекты с состоянием.
 - Приемочные тесты с использованием *фасада служб* (Service Facade) [CJ2EEP] на основе “подкожного” теста (Subcutaneous Test; см. *Тест уровня*, Layer Test).
 - Компоненты с состоянием и компонентные тесты.
4. Сложный в тестировании код.
 - Логика пользовательского интерфейса, предоставленная через *минимальный диалог* (Humble Dialog; см. *Минимальный объект*, Humble Object, с. 700).
 - Логика базы данных.
 - Многопотоковое программное обеспечение.
5. Унаследованное объектно-ориентированное программное обеспечение (создававшееся без тестов).
6. Унаследованное не объектно-ориентированное программное обеспечение.

По мере приближения к концу списка программное обеспечение становится все сложнее в тестировании. Как это ни удивительно, многие разработчики пытаются познакомиться с тестами, интегрируя их в уже существующее приложение. Это значит, что они соответствуют последним двум пунктам списка, для которых на самом деле нужен достаточно большой опыт. К сожалению, во многих случаях тестирование унаследованного программного продукта завершается неудачно, а значит, возникает предубеждение относительно использования автоматизированных тестов вообще. Если оказалось, что вы пытаетесь изучить автоматизированные тесты на примере интеграции тестов в существующее приложение, примите два совета: во-первых, обратитесь к тому, кто уже имел опыт такой интеграции; во-вторых, прочитайте отличную книгу Майкла Фезерса [WEwLC], в которой рассматривается множество способов интеграции тестов в существующие приложения.

План создания простых в обслуживании автоматизированных тестов

Учитывая, что некоторые тесты писать значительно сложнее, чем все остальные, имеет смысл сначала научиться писать простые тесты. При обучении разработчиков автоматизации тестирования способы рассматриваются в приведенном ниже порядке. Данный план основан на пирамиде потребностей Маслоу [HoN]: потребности на более высоком уровне удовлетворяются только после потребностей на всех низких уровнях.

1. Выполните код на “счастливом маршруте”.
 - Настройте простое предтестовое состояние тестируемой системы.
 - Вызовите тестируемый метод.
2. Проверьте непосредственный вывод “счастливого маршрута”.
 - Вызовите *метод с утверждением* (Assertion Method, с. 390) для вывода тестируемой системы.
 - Вызовите *метод с утверждением* (Assertion Method) для проверки состояния после теста.
3. Проверьте альтернативные ветви кода.

- Передайте другие аргументы тестируемому методу.
 - Измените предтестовое состояние тестируемой системы.
 - Перехватите управление опосредованным вводом с помощью *тестовой заглушки* (Test Stub, с. 544).
- 4.** Проверьте поведение опосредованного вывода.
- Воспользуйтесь *подставным объектом* (Mock Object, с. 558) или *тестовым агентом* (Test Spy, с. 552) для перехвата исходящих вызовов методов.
- 5.** Оптимизируйте запуск и обслуживание тестов.
- Заставьте тесты работать быстрее.
 - Убедитесь, что тесты просты для понимания и обслуживания.
 - Проектируйте программное обеспечение с учетом теста.
 - Сократите риск пропуска ошибок.

Приведенный порядок не подразумевает порядок реализации конкретного теста¹. Скорее, данный порядок описывает последовательность изучения способов автоматизации тестирования.

Рассмотрим каждый пункт более подробно.

Выполните код на “счастливом маршруте”

Для этого необходимо автоматизировать *простой тест успешности* (Simple Success Test; см. *Тестовый метод*, Test Method, с. 378), как тест для открытого программного интерфейса тестируемой системы. Для успешного завершения теста достаточно жестко вписать часть логики в тестируемую систему, например, в местах, где для получения информации вызываются другие компоненты и принимается решение, которое “выталкивает” тест со “счастливого маршрута”. Перед вызовом тестируемой системы необходимо настроить тестовую конфигурацию, переведя тестируемую систему в необходимое состояние. Пока тестируемая система выполняется без ошибок, тест можно считать успешно завершившимся. На этом этапе изучения тестов сравнивать ожидаемый результат с фактическим необязательно.

Проверьте непосредственный вывод “счастливого маршрута”

После успешного вызова тестируемой системы можно добавить логику проверки результатов и превратить тест в *самопроверяющийся* (Self-Checking Test, с. 81). Это потребует добавления вызовов *методов с утверждением* (Assertion Method), которые будут сравнивать ожидаемые результаты с фактически полученными. Это очень легко сделать для любых объектов или значений, возвращаемых тесту тестируемой системой. Кроме того, можно вызывать другие методы тестируемой системы или использовать открытые поля для доступа к состоянию системы после теста; полученное состояние можно также проверить с помощью *методов с утверждениями* (Assertion Method).

¹ Хотя и можно воспользоваться таким порядком при создании тестов, желательно начинать с утверждений.

Проверьте альтернативные ветви кода

На этом этапе основная ветвь кода уже проверена. Но альтернативные ветви по своей сути все еще являются *нетестированным кодом* (Untested Code; см. *Ошибки в продукте*, Production Bugs, с. 303), а значит, следующим этапом будет создание тестов для этих ветвей кода (как при уже написанном коде продукта, так и при попытке автоматизации тестов, которые потребуют написания этого кода). В данном случае основным является вопрос: “Что приводит к выполнению альтернативных ветвей кода?” Наиболее распространенными ответами являются:

- различные значения, передаваемые клиентом в качестве аргументов;
- различные начальные состояния тестируемой системы;
- различные результаты вызова методов компонентов, от которых зависит тестируемая система.

Первый случай можно проверить, модифицировав логику тестов, вызывающих методы тестируемой системы, с передачей различных значений в качестве аргументов. Второй случай требует инициализации тестируемой системы другим состоянием. Ни один из этих случаев не требует “запредельных знаний”. Но в третьем случае все становится значительно интереснее.

Управление опосредованным вводом

Поскольку ответы от других компонентов заставляют тестируемую систему выполнять альтернативную ветвь кода, необходимо перехватить управление таким опосредованным вводом. Для этого можно воспользоваться *тестовой заглушки* (Test Stub), которая возвращает значение, переводящее тестируемую систему на интересующую ветвь кода. В процессе создания тестовой конфигурации необходимо заставить систему воспользоваться заглушкой, а не настоящим компонентом. Существует два варианта создания *тестовой заглушки* (Test Stub): *фиксированная тестовая заглушка* (Hard-Coded Test Stub), содержащая написанный вручную код, который возвращает конкретные значения, и *настраиваемая тестовая заглушка* (Configurable Test Stub), которая настраивается тестом на возврат интересующих значений. В обоих случаях тестируемая система должна использовать *тестовую заглушки* (Test Stub) вместо реального компонента.

Многие альтернативные ветви кода приводят к “успешному” выводу от тестируемой системы. Такие тесты считаются *простыми тестами успешности* (Simple Success Test) и используют *тестовую заглушки* (Test Stub), которая называется *генератором ответов* (Responder). Другие ветви приводят к генерации сообщения об ошибке или исключения, а значит, тесты называются *тестами на ожидаемое исключение* (Expected Exception Test) и используют тестовую заглушки типа *диверсанта* (Saboteur).

Создание повторяемых и простых в модификации тестов

Факт замены настоящего вызываемого компонента *тестовой заглушки* (Test Stub) имеет один очень полезный побочный эффект: тесты становятся повторяемыми. Используя *тестовую заглушки* (Test Stub), можно заменить компонент с потенциально неопределенным поведением полностью контролируемым тестовым аналогом. Это хороший пример принципа *Изолируйте тестируемую систему* (Isolate the SUT, с. 97).

Проверьте поведение опосредованного вывода

Выше основное внимание уделялось получению управления опосредованным вводом тестируемой системы и проверкой видимого непосредственного вывода через проверку состояния системы после теста. Такой контроль результатов называется *проверкой состояния* (State Verification, с. 484). Но иногда нельзя подтвердить правильность поведения тестируемой системы, рассматривая только ее состояние после теста. Иначе говоря могут существовать *нетестированные требования* (Untested Requirement; см. *Ошибки в продукте*, Production Bugs), которые можно проверить только с помощью *проверки поведения* (Behavior Verification, с. 489).

На основе уже имеющихся знаний и навыков можно создать близкого родственника *тестовой заглушки* (Test Stub), который будет перехватывать исходящие от тестируемой системы вызовы методов. *Тестовый агент* (Test Spy) “запоминает”, как его вызывали, а тест может позднее получить информацию и воспользоваться *методами с утверждением* (Assertion Method) для сравнения с ожидаемым вариантом использования. *Подставной объект* (Mock Object) может получить список ожидаемых выводов во время создания тестовой конфигурации. Он сравнивает фактические вызовы с ожидаемыми в процессе взаимодействия с тестируемой системой.

Оптимизируйте запуск и обслуживание тестов

На этом этапе автоматизированные тесты написаны для всех ветвей кода. Но такие тесты могут оказаться не самыми оптимальными.

- Это могут быть *медленные тесты* (Slow Tests, с. 289).
- Тесты могут содержать *дублирование тестового кода* (Test Code Duplication, с. 254), затрудняющее понимание тестов.
- Некоторые из них могут быть сложными для понимания и обслуживания *непонятными тестами* (Obscure Test, с. 230).
- Среди написанных тестов могут существовать *тесты с ошибками* (Buggy Test, с. 296), причиной которых являются ненадежные *вспомогательные методы теста* (Test Utility Method, с. 610) или *условная логика теста* (Conditional Test Logic, с. 243).

Заставьте тесты работать быстрее

Медленный тест (Slow Test) часто оказывается первым запахом в очереди на устранение. Для ускорения работы тестов можно повторно использовать тестовую конфигурацию в нескольких тестах, например, через применение *общей тестовой конфигурации* (Shared Fixture, с. 350). К сожалению, такая тактика обычно становится причиной целого ряда новых проблем. Замена вызываемого компонента функционально эквивалентным, но более быстрым в работе *поддельным объектом* (Fake Object, с. 565) практически всегда является более подходящим решением. Использование *поддельных объектов* (Fake Object) основано на способах проверки опосредованного ввода и вывода.

Сделайте тесты более понятными и простыми в обслуживании

Непонятный тест (Obscure Test) можно сделать более понятным. *Дублирование тестового кода* (Test Code Duplication) можно сократить через рефакторинг *тестовых методов* (Test Method) и вызов *вспомогательных методов теста* (Test Utility Method), содержащих часто используемую логику. *Метод создания* (Creation Method, с. 441), *специальное утверждение* (Custom Assertion, с. 495), *метод поиска* (Finder Method; см. *Вспомогательный метод теста*, Test Utility Method) и *параметризованный тест* (Parameterized Test, с. 618) являются примерами такого подхода.

Если *классы теста* (Testcase Class, с. 401) становятся слишком большими, чтобы быстро в них разобраться, можно реорганизовать классы относительно тестовых конфигураций или функций. Кроме того, намерение можно донести через систему именования *классов теста* (Testcase Class) и *тестовых методов* (Test Method), описывающую проверяемые тестовые условия.

Сократите вероятность появления пропущенных ошибок

Если возникла проблема, связанная с появлением *тестов с ошибками* (Buggy Test) или *ошибок в продукте* (Production Bugs), можно сократить вероятность ложных несрабатываний (тестов, завершающихся успешно, когда должно быть неудачное завершение) через инкапсуляцию тестовой логики. При этом для вспомогательных *методов теста* (Test Utility Method) необходимо использовать описательные имена. Поведение *вспомогательных методов теста* (Test Utility Method) необходимо проверять с помощью *тестов вспомогательных методов тестов* (Test Utility Test; см. *Вспомогательный метод теста*, Test Utility Method).

Что дальше

Этой главой завершается часть I, “Общая информация”. В главах 1–14 рассматривались цели, принципы, философия, шаблоны, запахи и способы создания эффективных автоматизированных тестов. В части II, “Запахи тестов”, и части III, “Шаблоны”, приводится подробное описание каждого запаха и шаблона, упомянутого в части I.

Часть II

Запахи тестов

Глава 15

Запахи кода

Запахи в этой главе:

Непонятный тест (Obscure Test).....	230
Условная логика теста (Conditional Test Logic)	243
Сложный в тестировании код (Hard-to-Test Code).....	251
Дублирование тестового кода (Test Code Duplication).....	254
Логика теста в продукте (Test Logic in Production).....	257

Непонятный тест (Obscure Test)

Также известен как:

*Длинный тест (Long Test),
Сложный тест (Complex Test),
Подробный тест (Verbose Test)*

Очень сложно разобраться в teste, просто взглянув на него.

Автоматизированные тесты выполняют две функции.

Во-первых, они должны выступать в роли документации по необходимому поведению тестируемой системы. Это соответствует принципу *тесты как документация* (Tests as Documentation, с. 79). Во-вторых, тесты должны быть само-проверяющейся выполняемой спецификацией. Часто эти функции противоречат одна другой, так как необходимый для выполнения код может сделать тест таким переполненным, что он станет сложным для понимания.

Симптомы

Сложно понять, какое поведение проверяет тест.

Влияние

Первой проблемой *непонятного теста* (Obscure Test) является сложность понимания, а значит, и обслуживания. Практически всегда такие тесты не могут служить в качестве документации (Tests as Documentation), что приводит к *высокой стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300).

Вторая проблема связана с ошибками, которые могут “проскакивать” мимо теста из-за ошибок в коде, скрытых в недрах *непонятного теста* (Obscure Test). В результате появляется *тест с ошибками* (Buggy Test, с. 296). Более того, отказ одного утверждения в “энергичном” teste (Eager Test) может скрыть значительно больше ошибок, до которых просто не доходит выполнение, что приводит к потере отладочной информации теста.

Причины

Это может показаться парадоксальным, но причиной возникновения *непонятного теста* (Obscure Test) является слишком большой или слишком малый объем информации в *тестовом методе* (Test Method, с. 378). Примером слишком малого объема информации служит *тайныственный гость* (Mystery Guest). “Энергичный” teste (Eager Test) и *ненужная информация* (Irrelevant Information) — примеры слишком большого объема информации.

Обычно основной причиной появления *непонятного теста* (Obscure Test) является недостаточное внимание к обеспечению чистоты и простоты кода теста. Код теста настолько же важен, насколько важен код продукта, и требует того же объема рефакторинга. В основе *непонятного теста* (Obscure Test) лежит подход “Сделать все здесь и сразу”. Размещение всего кода в пределах одного метода приводит к появлению огромных *тестовых методов* (Test Method), так как некоторые операции просто нуждаются в большом объеме кода.

Некоторые причины появления непонятных тестов связаны с размещением внутри теста не той информации.

- “*Энергичный*” *тест* (Eager Test). Тест проверяет слишком большое количество функций в одном *тестовом методе* (Test Method).
- *Таинственный гость* (Mystery Guest). Читатель теста просто не в состоянии увидеть причинно-следственную связь между тестовой конфигурацией и логикой проверки, так как часть проверки происходит за пределами *тестового метода* (Test Method).

Основная проблема *подробных тестов* (Verbose Test) (которые используют слишком много кода для донесения содержащейся в них сути) может быть разбита на несколько причин.

- *Тестовая конфигурация общего характера* (General Fixture). Тест создает или использует конфигурацию, которая больше, чем необходимо для проверки интересующей тест функциональности.
- *Неуместная информация* (Irrelevant Information). Тест содержит большой объем ненужной информации о тестовой конфигурации, что отвлекает читателя от реальных факторов, влияющих на поведение тестируемой системы.
- *Фиксированные данные теста* (Hard-Coded Test Data). Значения в тестовой конфигурации, утверждения или аргументы тестируемой системы жестко закодированы в *тестовом методе* (Test Method), скрывая причинно-следственную связь между входными параметрами и ожидаемым выводом.
- *Опосредованное тестирование* (Indirect Testing). *Тестовый метод* (Test Method) взаимодействует с системой опосредованно через другой объект, усложняя взаимодействие.

Причина: “энергичный” тест (Eager Test)

Тест проверяет слишком много функций с помощью единственного *тестового метода* (Test Method).

Симптомы

Тест работает очень медленно, проверяя все, что можно проверить. Очень сложно определить, какой фрагмент отвечает за создание тестовой конфигурации, а какой вызывает тестируемую систему.

```
public void testFlightMileage_asKm2() throws Exception {
    // Создать тестовую конфигурацию
    // Вызвать конструктор
    Flight newFlight = new Flight(validFlightNumber);
    // Проверить созданный объект
    assertEquals(validFlightNumber, newFlight.number);
    assertEquals("", newFlight.airlineCode);
    assertNull(newFlight.airline);
    // Настроить пробег
    newFlight.setMileage(1122);
    // Вызвать транслятор пробега
    int actualKilometres = newFlight.getMileageAsKm();
    // Проверить результаты
    int expectedKilometres = 1810;
    assertEquals(expectedKilometres, actualKilometres);
    // Попробовать сделать то же, но с отмененным полетом
```

```

newFlight.cancel();
try {
    newFlight.getMileageAsKm();
    fail("Expected exception");
} catch (InvalidRequestException e) {
    assertEquals(
        "Не удается получить пробег для отмененного полета",
        e.getMessage());
}
}

```

Основная причина

При тестировании вручную имеет смысл выстраивать несколько логически различных тестовых условий в один тест, чтобы сократить общие накладные расходы. Такой подход работает, поскольку тесты выполняет думающий человек. В любой момент он может принять решение о продолжении или остановке работы теста в зависимости от текущей информации.

Возможное решение

При автоматизации тестов лучше создавать набор независимых *тестов одного условия* (Single Condition Test, с. 99), так как они намного лучше справляются с *локализацией дефектов* (Defect Localization, с. 78).

Причина: таинственный гость (Mystery Guest)

Читатель теста не видит причинно-следственную связь между тестовой конфигурацией и логикой проверки, так как часть проверки выполняется за пределами *тестового метода* (Test Method).

Симптомы

Любой тест передает данные в тестируемую систему. Данные на этапах создания тестовой конфигурации и вызова системы *четырехфазного теста* (Four-Phase Test, с. 387) определяют предварительные условия для тестируемой системы и влияют на ее поведение. Постусловия (ожидаемый результат) представлены данными, которые передаются в качестве аргументов *методов с утверждением* (Assertion Method, с. 390) на этапе проверки результата.

Если фаза создания тестовой конфигурации или проверки результатов зависит от информации, не отраженной внутри теста, и читатель не может определить проверяемое поведение, пока не найдет и не прочитает внешнюю информацию, такая ситуация называется *таинственным гостем* (Mystery Guest). Вот пример, когда невозможно определить смысл тестовой конфигурации, а значит, сложно связать ожидаемый результат с предварительными условиями теста.

```

public void testGetFlightsByFromAirport_OneOutboundFlight_mg()
    throws Exception {
    loadAirportsAndFlightsFromFile("test-flights.csv");
    // Вызвать систему
    List flightsAtOrigin =

```

```

        facade.getFlightsByOriginAirportCode("YYC");
// Проверить результат
assertEquals(1, flightsAtOrigin.size());
FlightDto firstFlight = (FlightDto) flightsAtOrigin.get(0);
assertEquals("Calgary", firstFlight.getOriginCity());
}

```

Влияние

Таинственный гость (Mystery Guest) усложняет определение причинно-следственной связи между тестовой конфигурацией (предварительными условиями теста) и ожидаемым результатом теста. Как следствие тесты не в состоянии играть роль документации (*Tests as Documentation*). Что еще хуже, кто-то может удалить или модифицировать внешний ресурс, даже не подозревая о влиянии своих действий на запускаемые тесты. Такой запах поведения имеет свое название: *оптимизм по отношению к ресурсу* (Resource Optimism; см. *Нестабильный тест*, Erratic Test, с. 267)!

Если в качестве *таинственного гостя* (Mystery Guest) выступает *общая тестовая конфигурация* (Shared Fixture, с. 350), модификация конфигурации другим тестом приводит к появлению *нестабильных тестов* (Erratic Test).

Основная причина

Тест зависит от загадочных внешних ресурсов, что не позволяет понять проверяемое поведение. *Таинственный гость* (Mystery Guest) может принимать одну из следующих форм.

- Имя существующего внешнего файла передается методу тестируемой системы в качестве аргумента. Содержимое файла определяет поведение тестируемой системы.
- Содержимое записи в базе данных идентифицируется буквенным ключом и загружается в объект. Объект используется тестом или передается в тестируемую систему.
- Содержимое файла читается и используется в вызовах *методов с утверждением* (Assertion Method) для проверки ожидаемого результата.
- Для создания *общей тестовой конфигурации* (Shared Fixture) используется *декоратор настройки* (Setup Decorator, с. 471). Обращение к объектам в пределах тестовой конфигурации выполняется через переменные внутри логики проверки результата.
- *Тестовая конфигурация общего характера* (General Fixture) создается с помощью *неявной настройки* (Implicit Setup, с. 449), а *тестовый метод* (Test Method) получает к ним доступ через переменные экземпляров или переменные классов.

Во всех перечисленных сценариях результат один: очень сложно оценить причинно-следственную связь между тестовой конфигурацией и ожидаемым результатом теста, так как необходимые данные в коде теста не видны. Если в такой ситуации данные не описываются именами переменных и файлов, то появляется *таинственный гость* (Mystery Guest).

Возможное решение

Очевидным решением является использование *новой тестовой конфигурации* (Fresh Fixture, с. 344) с помощью *встроенной настройки* (In-line Setup, с. 433). На примере с файлом это означает создание содержимого файла в виде строки в самом тесте с последующей записью в файловую систему (рефакторинг *настройка внешнего ресурса*, Setup External Resource, с. 741) или размещение его в *тестовой заглушки* (Test Stub) вместо файловой системы в процессе создания тестовой конфигурации (см. рефакторинг *встраивание ресурса*, In-

line Resource, с. 738). Во избежание появления *неуместной информации* (Irrelevant Information), возможно, потребуется скрыть подробности создания файла в одном или нескольких *методах создания* (Creation Method, с. 441) с описательными именами.

Если приходится использовать *общую тестовую конфигурацию* (Shared Fixture) или *неявную настройку* (Implicit Setup), для доступа к объектам в пределах конфигурации воспользуйтесь *методами поиска* (Finder Method; см. *Вспомогательный метод теста*, Test Utility Method, с. 610) с описательными именами. Если приходится пользоваться внешними ресурсами, например файлами, разместите их в специальной папке или в каталоге, а в именах явно укажите, какие данные они содержат.

Причина: тестовая конфигурация общего характера (General Fixture)

Тест создает или использует конфигурацию, которая больше, чем необходимо для проверки интересующей тест функциональности.

Симптомы

Код создания тестовой конфигурации выглядит слишком большим, намного большим, чем необходимо в большинстве тестов. Сложно оценить причинно-следственную связь между тестовой конфигурацией, вызываемыми фрагментами тестируемой системы и ожидаемым результатом.

Рассмотрим следующий набор тестов.

```
public void testGetFlightsByFromAirport_OneOutboundFlight()
    throws Exception {
    setupStandardAirportsAndFlights();
    FlightDto outboundFlight = findOneOutboundFlight();
    // Вызвать систему
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlight.getOriginAirportId());
    // Проверить результат
    assertOnly1FlightInDtoList("Flights at origin",
        outboundFlight,
        flightsAtOrigin);
}
public void testGetFlightsByFromAirport_TwoOutboundFlights()
    throws Exception {
    setupStandardAirportsAndFlights();
    FlightDto[] outboundFlights =
        findTwoOutboundFlightsFromOneAirport();
    // Вызвать систему
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlights[0].getOriginAirportId());
    // Проверить результат
    assertExactly2FlightsInDtoList("Flights at origin",
        outboundFlights,
        flightsAtOrigin);
}
```

Чтение этапов вызова тестовой конфигурации и проверки результата подсказывает, что этим тестам нужны совершенно разные тестовые конфигурации. Хотя здесь используется стратегия на основе *новой тестовой конфигурации* (Fresh Fixture), логика при создании тестовой конфигурации используется одна и та же (вызывается метод `setUpStan-`

`dardAirportsAndFlights`). Имя метода подсказывает, что это классический и легко узнаваемый пример *тестовой конфигурации общего характера* (General Fixture). Более сложным был бы случай создания *стандартной тестовой конфигурации* (Standard Fixture, с. 338) в каждом teste или создание разных тестовых конфигураций, содержащих больше данных, чем необходимо тесту.

Кроме того, могут проявляться *медленные тесты* (Slow Tests, с. 289) или “*хрупкая тестовая конфигурация*” (Fragile Fixture; см. “*Хрупкий*” тест, Fragile Test, с. 277).

Основная причина

Самой распространенной причиной данной проблемы являются тесты, использующие предназначенную для различных тестов конфигурацию. В качестве примеров можно привести использование *неявной настройки* (Implicit Setup) или *общей тестовой конфигурации* (Shared Fixture) несколькими тестами с различными требованиями к тестовой конфигурации. В результате тестовая конфигурация становится большой и сложной для понимания. Кроме того, тестовая конфигурация может расти со временем. Главной причиной этого является использование *стандартной тестовой конфигурации* (Standard Fixture), которая должна удовлетворять требования всех использующих ее тестов. Чем разностороннее требования, тем больше вероятность создания *тестовой конфигурации общего характера* (General Fixture).

Влияние

При создании тестовой конфигурации, поддерживающей несколько различных тестов, очень сложно понять, как с ней взаимодействует каждый тест. Подобная сложность снижает вероятность использования *тестов как документации* (Tests as Documentation) и приводит к появлению “*хрупкой*” *тестовой конфигурации* (Fragile Fixture), поскольку каждый разработчик модифицирует конфигурацию для создаваемых тестов. Кроме того, становится заметной проблема *медленных тестов* (Slow Test), так как больший размер тестовой конфигурации требует больше времени на создание, особенно при использовании файловой системы или базы данных.

Возможное решение

Для решения проблемы необходимо перейти к использованию *минимальной тестовой конфигурации* (Minimal Fixture, с. 336). Для этого каждый тест должен иметь *новую тестовую конфигурацию* (Fresh Fixture, с. 344). Если приходится использовать *общую тестовую конфигурацию* (Shared Fixture), следует применить рефакторинг *создания уникального ресурса* (Make Resource Unique, с. 739) для создания виртуальной “*песочницы*” с базой данных (Database Sandbox, с. 658) для каждого теста. (Переход к использованию *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture) не решает проблему полностью, так как не позволяет определить фрагменты тестовой конфигурации, необходимые каждому тесту; удается идентифицировать только модифицируемые фрагменты!)

Причина: неуместная информация (Irrelevant Information)

Тест показывает слишком много информации о тестовой конфигурации, что отвлекает читателя от основных факторов, затрагивающих поведение тестируемой системы.

Симптомы

Читателю теста сложно определить, какие из передаваемых в объекты значений фактически влияют на ожидаемый результат.

```
public void testAddItemQuantity_severalQuantity_v10() {
    // Настройка тестовой конфигурации
    Address billingAddress =
        createAddress("1222 1st St SW", "Calgary", "Alberta",
                      "T2N 2V2", "Canada");
    Address shippingAddress =
        createAddress("1333 1st St SW", "Calgary", "Alberta",
                      "T2N 2V2", "Canada");
    Customer customer =
        createCustomer(99, "John", "Doe", new BigDecimal("30"),
                       billingAddress, shippingAddress);
    Product product =
        createProduct(88, "SomeWidget", new BigDecimal("19.99"));
    Invoice invoice = createInvoice(customer);
    // Вызов testируемой системы
    invoice.addItemQuantity(product, 5);
    // Проверка результата
    LineItem expected =
        new LineItem(invoice, product, 5, new BigDecimal("30"),
                     new BigDecimal("69.96"));
    assertContainsExactlyOneLineItem(invoice, expected);
}
```

Логика создания тестовой конфигурации может выглядеть большой и сложной, так как связывает вместе множество несвязанных объектов. В результате из-за непонятных предварительных условий сложно определить, что именно проверяет тест.

```
public void testGetFlightsByOriginAirport_TwoOutboundFlights()
    throws Exception {
    FlightDto expectedCalgaryToSanFran = new FlightDto();
    expectedCalgaryToSanFran.setOriginAirportId(calgaryAirportId);
    expectedCalgaryToSanFran.setOriginCity(CALGARY_CITY);
    expectedCalgaryToSanFran.setDestinationAirportId(sanFranAirportId);
    expectedCalgaryToSanFran.setDestinationCity(SAN_FRAN_CITY);
    expectedCalgaryToSanFran.setFlightNumber(
        facade.createFlight(calgaryAirportId, sanFranAirportId));
    FlightDto expectedCalgaryToVan = new FlightDto();
    expectedCalgaryToVan.setOriginAirportId(calgaryAirportId);
    expectedCalgaryToVan.setOriginCity(CALGARY_CITY);
    expectedCalgaryToVan.setDestinationAirportId(vancouverAirportId);
    expectedCalgaryToVan.setDestinationCity(VANCOUVER_CITY);
    expectedCalgaryToVan.setFlightNumber(facade.createFlight(
        calgaryAirportId, vancouverAirportId));
```

Проверяющий результат код тоже может оказаться слишком сложным.

```
List lineItems = inv.getLineItems();
assertEquals("number of items", lineItems.size(), 2);
// Проверить первый элемент
LineItem actual = (LineItem)lineItems.get(0);
assertEquals(expItem1.getInv(), actual.getInv());
```

```

assertEquals(expItem1.getProd(), actual.getProd());
assertEquals(expItem1.getQuantity(), actual.getQuantity());
// Проверить второй элемент
actual = (LineItem)lineItems.get(1);
assertEquals(expItem2.getInv(), actual.getInv());
assertEquals(expItem2.getProd(), actual.getProd());
assertEquals(expItem2.getQuantity(), actual.getQuantity());
}

```

Основная причина

Тест содержит большой объем данных, являющихся *точными значениями* (Literal Value, с. 718) или переменными. Часто *неуместная информация* (Irrelevant Information) встречается вместе с *фиксированными данными теста* (Hard-Coded Test Data) или *тестовой конфигурацией общего характера* (General Fixture); также она может возникать из-за видимости всех данных, необходимых для запуска, вместо данных, необходимых для понимания. При создании тестов путь наименьшего сопротивления заключается в использовании всех доступных методов (тестируемой системы и других объектов) и присвоении значений всем параметрам, вне зависимости от их отношения к тесту.

Еще одной причиной сложности является включение всего кода, необходимого для проверки результата, через *процедурную проверку состояния* (Procedural State Verification; см. *Проверка состояния*, State Verification, с. 484) вместо использования более компактного “декларативного” стиля при описании результата.

Влияние

Сложно создавать *тесты как документацию* (Tests as Documentation), если они содержат визуально случайные фрагменты *непонятных тестов* (Obscure Test), не формирующие очевидной связи между пред- и постусловиями. Точно так анализ этапов создания тестовой конфигурации или проверка результата может привести к *высокой стоимости обслуживания тестов* (High Test Maintenance Cost) и повышению вероятности возникновения *ошибок в продукте* (Production Bugs, с. 303) или появления *тестов с ошибками* (Buggy Test).

Возможное решение

Избавиться от *неуместной информации* (Irrelevant Information) в логике создания тестовой конфигурации лучше всего, заменив непосредственные вызовы конструкторов или *методов фабрик* (Factory Method) [GOF] вызовами *параметризованных методов создания* (Parameterized Creation Method; см. *Метод создания*, Creation Method), принимающих в качестве параметров только относящуюся к тесту информацию. Значения тестовой конфигурации, не относящиеся к тесту (т.е. не влияющие на ожидаемый результат), должны получать принятые по умолчанию значения внутри *метода создания* (Creation Method) или заменяться *объектом-заглушкой* (Dummy Object, с. 730). Таким образом, читателю теста сообщается, что “невидимые значения не влияют на ожидаемый результат”. Значения в тестовой конфигурации, которые встречаются в настройке конфигурации и в проверке результата, можно заменить константами с подходящими именами (пока для этого используется *новая тестовая конфигурация*, Fresh Fixture).

Для скрытия *неуместной информации* (Irrelevant Information) в логике проверки результата можно воспользоваться утверждениями относительно всего *ожидаемого объекта* (Expected Object; см. *Проверка состояния*, State Verification) вместо утверждений относи-

тельно отдельных полей. Также можно создать *специальное утверждение* (Custom Assertion, с. 495), скрывающее сложную процедурную логику проверки.

Причина: фиксированные данные теста (Hard-Coded Test Data)

Данные в тестовой конфигурации, утверждениях или аргументах тестируемой системы жестко зафиксированы в *тестовых методах* (Test Method), скрывая причинно-следственную связь между вводом и ожидаемым выводом.

Симптомы

Читателям тестов сложно определить связь жестко закодированных значений внутри теста друг с другом и с поведением тестируемой системы. Кроме того, могут возникать такие запахи поведения, как *нестабильный тест* (Erratic Test).

```
public void testAddItemQuantity_severalQuantity_v12() {
    // Настройка тестовой конфигурации
    Customer cust = createACustomer(new BigDecimal("30"));
    Product prod = createAProduct(new BigDecimal("19.99"));
    Invoice invoice = createInvoice(cust);
    // Вызов тестируемой системы
    invoice.addItemQuantity(prod, 5);
    // Проверка результата
    LineItem expected = new LineItem(invoice, prod, 5,
        new BigDecimal("30"), new BigDecimal("69.96"));
    assertContainsExactlyOneLineItem(invoice, expected);
}
```

Этот пример не так уж и плох, поскольку значений не очень много. Если выполнение математических операций в уме не является сильной стороной разработчика, он может и не заметить соотношение между ценой единицы (19.99), количеством единиц (5), скидкой (30) и общей ценой (69.96).

Основная причина

Фиксированные данные теста (Hard-Coded Test Data) появляются, когда тест содержит множество визуально не связанных *точных значений* (Literal Value). Тест всегда выполняет передачу данных в тестируемую систему. Используемые при создании тестовой конфигурации и вызове системы данные определяют предусловия для тестируемой системы и влияют на ее поведение. Постусловия (ожидаемый результат) отражаются данными, передаваемыми в виде аргументов *методов с утверждением* (Assertion Method). При создании тестов путь наименьшего сопротивления заключается в использовании всех доступных методов (тестируемой системы и других объектов) и присвоении значений всем параметрам вне зависимости от их отношения к тесту.

Если для повторного использования логики теста применяется метод “*копирования-вставки*”, точные значения копируются в производные тесты.

Влияние

Сложно создавать *тесты как документацию* (Tests as Documentation), если они содержат визуально случайные фрагменты *непонятных тестов* (Obscure Test), не формирующие очевидной связи между пред- и постусловиями. Несколько точных значений не выглядят проблемой. В конце концов, они не слишком усложняют тест. Но с ростом количества таких

значений понимание дается все сложнее и сложнее, особенно когда резко падает отношение “сигнал/шум” из-за несвязанности большинства значений с самим тестом.

Также огромное влияние оказывают коллизии тестов, использующих одни и те же значения. Такая ситуация возникает только при использовании *общей тестовой конфигурации* (Shared Fixture), поскольку стратегия на основе *новой тестовой конфигурации* (Fresh Fixture) не оставляет объекты, с которыми могут сталкиваться следующие тесты.

Возможное решение

Лучшим способом избавиться от запаха *непонятных тестов* (Obscure Test) является замена точных констант чем-либо другим. Значения в тестовой конфигурации, определяющие выполняемый сценарий (так называемые коды типов), являются единственными значениями, которые менять не стоит, но даже их можно превратить в именованные константы.

Не влияющие на тест (от которых не зависит ожидаемый результат) значения в конфигурации должны присваиваться в *методах создания* (Creation Method). Таким образом, читателю теста сообщается, что “невидимые значения не влияют на ожидаемый результат”. Значения в тестовой конфигурации, которые встречаются в настройке конфигурации и в проверке результата, можно заменить константами с подходящими именами (пока для этого используется *новая тестовая конфигурация*, Fresh Fixture).

Значения в логике проверки результата, основанные на значениях в тестовой конфигурации или использующиеся в качестве аргументов тестируемой системы, должны быть заменены *вычисляемыми значениями* (Derived Value, с. 722), делающими вычисления очевидными для читателя тестов.

Если применяется один из вариантов *общей тестовой конфигурации* (Shared Fixture), можно попытаться использовать *отдельное сгенерированное значение* (Distinct Generated Value; см. *Сгенерированное значение*, Generated Value, с. 726) для обеспечения уникальности значений при каждом запуске теста. Такой подход особенно полезен для полей, играющих роль уникального ключа в базах данных. Часто подобная логика скрывается внутри *анонимного метода создания* (Anonymous Creation Method; см. *Метод создания*, Creation Method).

Причина: опосредованное тестирование (Indirect Testing)

Тестовый метод (Test Method) опосредованно взаимодействует с тестируемой системой через другой объект, усложняя процесс взаимодействия.

Симптомы

Тест, в основном, взаимодействует не с тем объектом, поведение которого проверяется, а с другими объектами. Тесту приходится создавать и вызывать объекты, содержащие ссылки на тестируемую систему, а не с самой тестируемой системой. Тестирование бизнес-логики через презентационный уровень является распространенным примером *опосредованного тестирования* (Indirect Testing).

```
private final int LEGAL_CONN_MINS_SAME = 30;
public void testAnalyze_sameAirline_LessThanConnectionLimit () throws Exception {
    // Настройка
    FlightConnection illegalConn =
```

```

        createSameAirlineConn( LEGAL_CONN_MINS_SAME - 1);
// Вызов
FlightConnectionAnalyzerImpl sut =
    new FlightConnectionAnalyzerImpl();
String actualHtml =
    sut.getFlightConnectionAsHtmlFragment(
        illegalConn.getInboundFlightNumber(),
        illegalConn.getOutboundFlightNumber());
// Проверка
StringBuffer expected = new StringBuffer();
expected.append("<span class='boldRedText'>");
expected.append("Connection time between flight ");
expected.append(illegalConn.getInboundFlightNumber());
expected.append(" and flight ");
expected.append(illegalConn.getOutboundFlightNumber());
expected.append(" is ");
expected.append(illegalConn.getActualConnectionTime());
expected.append(" minutes.</span>");
assertEquals("html", expected.toString(), actualHtml);
}

```

Влияние

Иногда невозможно протестировать “все, что может сломаться” внутри тестируемой системы через промежуточный объект. Такие тесты редко оказываются понятными. И никогда они не будут служить документацией (Tests as Documentation).

Опосредованное тестирование (Indirect Testing) может привести к появлению “хрупких” тестов (Fragile Test), так как изменения в промежуточном объекте могут потребовать модификации тестов даже в случае неизменности тестируемой системы.

Основная причина

Тестируемая система может быть “закрытой” относительно класса, используемого для ее проверки. Непосредственное создание тестируемой системы может оказаться невозможным, поскольку конструкторы также являются закрытыми. Подобная проблема является одним из признаков непригодности дизайна для тестирования.

Может оказаться, что результат вызова тестируемой системы нельзя наблюдать непосредственно. В таком случае ожидаемый результат теста должен проверяться через промежуточный объект.

Возможное решение

Для устранения этого запаха может потребоваться улучшение дизайна в сторону простоты тестирования. С помощью рефакторинга *выделение тестируемого компонента* (Extract Testable Component) [WEwLC] можно обеспечить доступ теста непосредственно к тестируемой системе. Такой подход потенциально приводит к появлению нетестируемого *минимального объекта* (Humble Object, с. 700) и простого в тестировании объекта, который будет содержать большую часть логики.

```

public void testAnalyze_sameAirline_EqualsConnectionLimit()
throws Exception {
    // Настройка
    Mock flightMgmtStub = mock(FlightManagementFacade.class);
    Flight firstFlight = createFlight();
    Flight secondFlight = createConnectingFlight(

```

```

        firstFlight, LEGAL_CONN_MINS_SAME);
flightMgmtStub.expects(once()).method("getFlight")
    .with(eq(firstFlight.getFlightNumber()))
    .will(returnValue(firstFlight));
flightMgmtStub.expects(once()).method("getFlight")
    .with(eq(secondFlight.getFlightNumber()))
    .will(returnValue(secondFlight));
// Вызов
FlightConnAnalyzer theConnectionAnalyzer =
    new FlightConnAnalyzer();
theConnectionAnalyzer.facade =
    (FlightManagementFacade) flightMgmtStub.proxy();
FlightConnection actualConnection =
    theConnectionAnalyzer.getConn(
        firstFlight.getFlightNumber(),
        secondFlight.getFlightNumber());
// Проверка
assertNotNull("actual connection", actualConnection);
assertTrue("IsLegal", actualConnection.isLegal());
}

```

Иногда тест вынужден взаимодействовать с системой опосредованно, так как рефакторинг кода невозможен. В таком случае сложную логику, обеспечивающую *опосредованное тестирование* (Indirect Testing), необходимо скрыть внутри *вспомогательного метода теста* (Test Utility Method). Точно так настройка тестовой конфигурации может быть скрыта внутри *метода создания* (Creation Method), а проверка результата — в *методе проверки* (Verification Method; см. *Специальное утверждение*, Custom Assertion). Оба подхода являются примерами *инкапсуляции программного интерфейса тестируемой системы* (SUT API Encapsulation; см. *Вспомогательный метод теста*, Test Utility Method).

```

public void testAnalyze_sameAirline_LessThanConnLimit()
throws Exception {
    // Настройка
    FlightConnection illegalConn =
        createSameAirlineConn(LEGAL_CONN_MINS_SAME - 1);
    FlightConnectionAnalyzerImpl sut =
        new FlightConnectionAnalyzerImpl();
    // Вызов тестируемой системы
    String actualHtml =
        sut.getFlightConnectionAsHtmlFragment(
            illegalConn.getInboundFlightNumber(),
            illegalConn.getOutboundFlightNumber());
    // Проверка
    assertConnectionIsIllegal(illegalConn, actualHtml);
}

```

В следующем *специальном утверждении* (Custom Assertion) скрываются ужасы извлечения бизнес-результата из шума презентационного уровня. Данное утверждение создано с помощью рефакторинга *выделение метода* (Extract Method) [Ref]. Конечно, данный пример был бы более эффективным при поиске ключевых строк внутри кода HTML вместо создания большой ожидаемой строки и ее сравнения с возвращаемым значением. После этого в других *тестах презентационного уровня* (Presentation level test; см. *Тест уровня*, Layer Test, с. 368) можно проверить правильность форматирования HTML-строки.

```

private void assertConnectionIsIllegal(FlightConnection conn,
                                     String actualHtml) {
    // Создание ожидаемого значения
    StringBuffer expected = new StringBuffer();
    expected.append("<span class='boldRedText'>");
    expected.append("Connection time between flight ");
    expected.append(conn.getInboundFlightNumber());
    expected.append(" and flight ");
    expected.append(conn.getOutboundFlightNumber());
    expected.append(" is ");
    expected.append(conn.getActualConnectionTime());
    expected.append(" minutes.</span>");
    // Проверка
    assertEquals("html", expected.toString(), actualHtml);
}

```

Шаблоны решений

Хорошая стратегия тестирования помогает делать код тестов понятным. Но не забывайте, что как “любой план сражения не выдерживает первого контакта с противником”, так и никакая инфраструктура тестирования не может удовлетворить всех требований тестов. Инфраструктура будет развиваться вместе с программным продуктом и навыками автоматизации тестов.

Повторное использование логики тестов в различных сценариях обеспечивается вызовом *вспомогательных методов теста* (Test Utility Method) или запросом у общего *параметризованного теста* (Parameterized Test, с. 618) передачи уже созданной тестовой конфигурации или *ожидаемого объекта* (Expected Object).

Создание тестов в направлении “снаружи внутрь” позволяет снизить вероятность появления *непонятных тестов* (Obscure Test), требующих дальнейшего рефакторинга. При таком подходе *четырехфазный тест* (Four-Phase Test) описывается вызовами несуществующих *вспомогательных методов теста* (Test Utility Method). После завершения тестов можно написать вспомогательные методы. Написание тестов, в первую очередь, позволяет более ясно определить необходимые вспомогательные методы. Конечно, “зараженный” идеей тестирования разработчик сначала напишет *тесты вспомогательных методов теста* (Test Utility Test) и только после этого — *вспомогательные методы теста* (Test Utility Method).

Условная логика теста (Conditional Test Logic)

Тест содержит код, который может выполняться или не выполняться.

Также известен как:
Код теста со смещением
(*Indented Test Code*)

Полностью автоматизированные тесты (Fully Automated Tests, с. 81) представляют собой код, проверяющий поведение другого кода. Но, если этот код слишком сложен, как убедиться в его правильности? Можно написать тесты для тестов, но когда остановится такая рекурсия? Ответ прост: *тестовые методы* (Test Method, с. 378) должны быть настолько простыми, чтобы не требовать собственных тестов.

Условная логика теста (Conditional Test Logic) является одним из факторов, делающих тесты сложнее, чем необходимо.

Симптомы

Как запах кода *условная логика теста* (Conditional Test Logic) может не провоцировать симптомы в поведении, но ее присутствие очевидно для любого читателя тестов. Относитесь с подозрением к любым управляющим структурам внутри *тестового метода* (Test Method)! Кроме того, читатель теста может задаться вопросом о ветви кода, которая будет выполняться. Ниже показан пример *условной логики теста* (Conditional Test Logic), в которой присутствуют как циклы, так и операторы *if*.

```
// Убедиться, что Vancouver включен в список
actual = null;
i = flightsFromCalgary.iterator();
while (i.hasNext()) {
    FlightDto flightDto = (FlightDto) i.next();
    if (flightDto.getFlightNumber().equals(
        expectedCalgaryToVan.getFlightNumber()))
    {
        actual = flightDto;
        assertEquals("Flight from Calgary to Vancouver",
                    expectedCalgaryToVan,
                    flightDto);
        break;
    }
}
```

После чтения приведенного кода напрашивается вопрос: “Что делает этот тест и как определить, что он делает это правильно?” Одним из поведенческих симптомов может стать присутствие связанного запаха *высокая стоимость обслуживания тестов* (High Test Maintenance Cost, с. 300) на уровне проекта, который является результатом сложности, вносимой *условной логикой теста* (Conditional Test Logic).

Влияние

Условная логика теста (Conditional Test Logic) значительно усложняет понимание, что же тест делает на самом деле. Код с единственной ветвью выполнения всегда выполняется одинаково. Код с несколькими ветвями значительно сложнее для понимания и не позволяет всегда быть уверенным в результатах его работы.

Самопроверяющиеся тесты (Self-Checking Test, с. 81) создаются для повышения уверенности в правильности кода продукта. Как быть уверенными в правильности кода тестов, если он выполняется по-разному при каждом запуске? В такой ситуации очень сложно определить (или доказать), что тест проверяет именно то поведение, которое интересует разработчика. Тест с управляющими структурами, циклами или различными значениями при каждом запуске оказывается очень сложным в отладке, так как его поведение не полностью определено.

Усложнение процесса написания тестов является еще одним следствием *условной логики теста* (Conditional Test Logic). Поскольку тесты сложно поддаются тестированию, как определить, что тест действительно обнаруживает ошибки, для перехвата которых он создавался? (Это общая проблема для всех *непонятных тестов* (Obscure Test, с. 229), так как они являются распространенной причиной появления *тестов с ошибками* (Buggy Test, с. 296).)

Причины

Разработчик автоматизированных тестов может добавлять *условную логику теста* (Conditional Test Logic) по целому ряду причин.

- Операторы `if` могут использоваться для перехода к оператору `fail` или для обхода некоторых фрагментов кода теста, если тестируемая система не возвращает действительные данные.
- Циклы могут использоваться для проверки содержимого коллекции объектов (*условная логика проверки*, Conditional Verification Logic). Такое решение может привести к появлению *непонятных тестов* (Obscure Test).
- *Условная логика теста* (Conditional Test Logic) может использоваться для проверки сложных объектов или полиморфных структур данных (еще один вариант *условной логики проверки*, Conditional Verification Logic). Это просто реализация метода `equals` в соответствии с шаблоном Foreign Method [Ref].
- *Условная логика теста* (Conditional Test Logic) может использоваться для инициализации тестовой конфигурации или *ожидаемого объекта* (Expected Object; см. *Проверка состояния*, State Verification, с. 484), что позволит одному тесту проверять несколько различных условий (*гибкий тест*, Flexible Test).
- Операторы `if` могут использоваться для защиты от очистки несуществующей тестовой конфигурации (*сложная очистка*, Complex Teardown).

Имеет смысл некоторые из этих причин рассмотреть более подробно.

Причина: гибкий тест (Flexible Test)

Код теста проверяет разные функции в зависимости от времени или места запуска.

Симптомы

Тест содержит условную логику, выполняющую различные операции в зависимости от текущего окружения. Чаще всего такая функциональность принимает форму *условной логики теста* (Conditional Test Logic) для определения ожидаемого результата в зависимости от внешних факторов.

Рассмотрим следующий тест, использующий текущее время для определения ожидаемого вывода тестируемой системы.

```
public void testDisplayCurrentTime_whenever() {
    // Настройка тестовой конфигурации
    TimeDisplay sut = new TimeDisplay();
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    Calendar time = new DefaultTimeProvider().getTime();
    StringBuffer expectedTime = new StringBuffer();
    expectedTime.append("<span class=\"tinyBoldText\">");
    if ((time.get(Calendar.HOUR_OF_DAY) == 0)
        && (time.get(Calendar.MINUTE) <= 1)) {
        expectedTime.append(" Midnight");
    } else if ((time.get(Calendar.HOUR_OF_DAY) == 12)
        && (time.get(Calendar.MINUTE) == 0)) { // полдень
        expectedTime.append(" Noon");
    } else {
        SimpleDateFormat fr = new SimpleDateFormat("h:mm a");
        expectedTime.append(fr.format(time.getTime()));
    }
    expectedTime.append("</span>");
    assertEquals(expectedTime, result);
}
```

Основная причина

Причиной *гибкого теста* (Flexible Test) является недостаточный контроль над средой. Вероятно, разработчику теста не удалось отделить тестируемую систему от ее зависимостей, поэтому пришлось адаптировать логику теста с учетом состояния среды.

Влияние

Первым недостатком *гибких тестов* (Flexible Test) является сложность в понимании, а значит, и в обслуживании. Вторым недостатком является невозможность определить, какие сценарии тестирования выполняются, и все ли сценарии выполняются регулярно. Например, выполняется ли в приведенном примере полуночный сценарий? Как часто это происходит? Скорее всего, редко или никогда, так как для этого тест должен запускаться ровно в полночь, что маловероятно, даже если специально спланировать ночную компиляцию.

Возможное решение

От *гибких тестов* (Flexible Test) лучше всего избавляться через разделение тестируемой системы и ее зависимостей, вынудивших разработчика добавлять гибкость в тесты. Это касается и рефакторинга тестируемой системы для поддержки **заменяемой зависимости** (substitutable dependency). После этого зависимость можно заменить *тестовым двойником* (Test Double, с. 538), например *тестовой заглушкой* (Test Stub, с. 544) или *подставным объектом* (Mock Object, с. 558). Затем для каждого аспекта гибкого теста можно написать отдельные тесты.

Причина: условная логика проверки (Conditional Verification Logic)

Условная логика проверки (Conditional Verification Logic) может стать причиной проблемы при проверке ожидаемого результата. Обычно такая ситуация возникает, когда разработчик пытается предотвратить генерацию исключения, когда тестируемая система не возвращает подходящий объект или использует циклы для проверки содержимого возвращаемых системой коллекций.

```
// Убедиться, что Vancouver включен в список
actual = null;
i = flightsFromCalgary.iterator();
while (i.hasNext()) {
    FlightDto flightDto = (FlightDto) i.next();
    if (flightDto.getFlightNumber().equals(
        expectedCalgaryToVan.getFlightNumber()))
    {
        actual = flightDto;
        assertEquals("Flight from Calgary to Vancouver",
                    expectedCalgaryToVan,
                    flightDto);
        break;
    }
}
```

Возможное решение

Управляющие выполнением операторы `if` можно заменить вызовами функции `fail` со *сторожевым утверждением* (Guard Assertion, с. 510), которое заставляет тест завершаться неудачно еще перед достижением кода, выполнение которого нежелательно. Данный подход хорошо работает во всех случаях, кроме случая использования *теста на ожидаемое исключение* (Expected Exception Test; см. *Тестовый метод*, Test Method). В такой ситуации следует прибегнуть к стандартной идиоме *теста на ожидаемое исключение* (Expected Exception Test), принятой в данной реализации пакета xUnit.

Условную логику теста (Conditional Test Logic) для проверки сложных объектов можно заменить *утверждением равенства* (Equality Assertion; см. *Метод с утверждением*, Assertion Method, с. 390) по отношению к *ожидаемому объекту* (Expected Object). Если метод `equals` в коде продукта является слишком строгим, можно воспользоваться *специальным утверждением* (Custom Assertion, с. 495) для определения **равенства в пределах теста** (test-specific equality).

Любые циклы в логике проверки необходимо выносить в *специальное утверждение* (Custom Assertion). Поведение утверждения можно будет проверить с помощью *теста*

специального утверждения (Custom Assertion Test; см. *Специальное утверждение*, Custom Assertion).

Вызов *вспомогательных методов теста* (Test Utility Method, с. 610) позволяет повторно использовать логику тестов. Также можно воспользоваться *параметризованным тестом* (Parameterized Test, с. 618), который передает уже созданную тестовую конфигурацию и *ожидаемый объект* (Expected Object).

Причина: логика продукта внутри теста (Production Logic in Test)

Симптомы

Некоторые варианты *условной логики теста* (Conditional Test Logic) встречаются в разделе проверки результата. Рассмотрим внимательно внутренние циклы этого теста.

```
public void testCombinationsOfInputValues() {
    // Настроить тестовую конфигурацию
    Calculator sut = new Calculator();
    int expected; // Определяется внутри циклов
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            // Вызвать тестируемую систему
            int actual = sut.calculate( i, j );
            // Проверить результат
            if (i==3 & j==4) // Специальный случай
                expected = 8;
            else
                expected = i+j;
            assertEquals(message(i,j), expected, actual);
        }
    }
    private String message(int i, int j) {
        return "Cell( " + String.valueOf(i)+ ", "
            + String.valueOf(j) + ")";
    }
}
```

Вложенные циклы внутри данного теста на *основе цикла* (Loop-Driven Test; см. *Параметризованный тест*, Parameterized Test) вызывают систему с различными комбинациями значений *i* и *j*. Ниже основное внимание уделяется *условной логике теста* (Conditional Test Logic) внутри цикла.

Основная причина

Данный случай *логики продукта внутри теста* (Production Logic in Test) является непосредственным результатом проверки нескольких условий внутри единственного *тестового метода* (Test Method). Учитывая, что тестируемой системе передается несколько значений, на выходе стоит ожидать нескольких результатов. Сложно перечислить ожидаемые результаты для всех наборов входных параметров, если тестируемой системе передается несколько комбинаций разных аргументов во вложенных циклах. Распространенным решением данной проблемы является использование вычисляемых на основе входных параметров значений (Calculated Value; см. *Вычисляемое значение*, Derived Value, с. 722). Потенциальным недостатком (как можно видеть в данном примере) является повторение логики тестируемой системы по вычислению ожидаемого результата внутри теста.

Возможное решение

По возможности лучше перечислить наборы предварительно полученных значений, которые будут передаваться в тестируемую систему. В следующем примере та же логика проверяется с помощью (меньшего) набора предварительно перечисленных значений.

```
public void testMultipleValueSets() {
    // Настроить тестовую конфигурацию
    Calculator sut = new Calculator();
    TestValues[] testValues = {
        new TestValues(1,2,3),
        new TestValues(2,3,5),
        new TestValues(3,4,8), // Специальный случай!
        new TestValues(4,5,9)
    };
    for (int i = 0; i < testValues.length; i++) {
        TestValues values = testValues[i];
        // Вызвать тестируемую систему
        int actual = sut.calculate(values.a, values.b);
        // Проверить результат
        assertEquals(message(i), values.expectedSum, actual);
    }
}
private String message(int i) {
    return "Row " + String.valueOf(i);
}
```

Причина: сложная очистка (Complex Teardown)

Симптомы

Сложный код очистки тестовой конфигурации с большей вероятностью приводит к повреждению тестового окружения, если очистка выполняется некорректно. Код очистки сложно проверять, но его некорректная работа может вызвать “утечку данных”, позднее приводящую к неудачному завершению тестов без видимых на то причин. Рассмотрим следующий пример.

```
public void testGetFlightsByOrigin_NoInboundFlight_SMRTD()
    throws Exception {
    // Настроить тестовую конфигурацию
    BigDecimal outboundAirport = createTestAirport("1OF");
    BigDecimal inboundAirport = null;
    FlightDto expFlightDto = null;
    try {
        inboundAirport = createTestAirport("1IF");
        expFlightDto =
            createTestFlight(outboundAirport, inboundAirport);
        // Вызвать тестируемую систему
        List flightsAtDestination1 =
            facade.getFlightsByOriginAirport(inboundAirport);
        // Проверить результат
        assertEquals(0, flightsAtDestination1.size());
    } finally {
        try {
            facade.removeFlight(expFlightDto.getFlightNumber());
        } finally {
            try {
```

```
        facade.removeAirport(inboundAirport);
    } finally {
        facade.removeAirport(outboundAirport);
    }
}
```

Основная причина

Обычно очистка требуется при использовании сохраняемых ресурсов, оказывающих-
ся за пределами системы **сборки мусора** (garbage collection). *Сложная очистка* (Complex Teardown) появляется, когда множество таких ресурсов используется в одном и том же *тестовом методе* (Test Method).

Возможное решение

Чтобы избавиться от сложной логики очистки, следует прибегнуть к *неявной очистке* (Implicit Teardown, с. 533), позволяющей повторно использовать и тестировать код, или к *автоматической очистке* (Automated Teardown, с. 521), которую можно проверять с помощью автоматизированных модульных тестов.

Кроме того, от необходимости выполнять очистку объектов тестовой конфигурации можно избавиться за счет перехода к стратегии на основе *новой тестовой конфигурации* (Fresh Fixture, с. 344) и отказа использовать сохраняемые объекты в тестах с применением одного из типов *тестовых двойников* (Test Double).

Причина: несколько тестовых условий (Multiple Test Conditions)

Симптомы

Тест пытается применить одну и ту же логику к нескольким наборам входных значений, с каждым из которых связан собственный ожидаемый результат. В следующем примере происходит перебор коллекции тестируемых значений и к каждому члену коллекции применяется логика теста.

```
public void testMultipleValueSets() {
    // Настроить тестовую конфигурацию
    Calculator sut = new Calculator();
    TestValues[] testValues = {
        new TestValues(1, 2, 3),
        new TestValues(2, 3, 5),
        new TestValues(3, 4, 8), // особый случай!
        new TestValues(4, 5, 9)
    };
    for (int i = 0; i < testValues.length; i++) {
        TestValues values = testValues[i];
        // Вызвать тестируемую систему
        int actual = sut.calculate(values.a, values.b);
        // Проверить результат
        assertEquals(message(i), values.expectedSum, actual);
    }
}
private String message(int i) {
    return "Row " + String.valueOf(i);
}
```

Основная причина

Разработчик теста пытается проверять несколько условий с применением одной и той же логики *тестового метода* (Test Method). В приведенном выше примере проявляется простая *условная логика теста* (Conditional Test Logic). Но все может значительно усложниться, если код содержит несколько вложенных циклов и даже операторы *if* для расчета различных вариантов ожидаемого результата.

Возможное решение

Из всех источников *условной логики теста* (Conditional Test Logic) *несколько тестовых условий* (Multiple Test Conditions) являются наиболее безобидными. Основным недостатком теста является его неудачное завершение после первого отказа и как следствие отсутствие *локализации дефектов* (Defect Localization, с. 78) при обнаружении ошибки в коде. Проблему простоты чтения можно решить за счет использования рефакторинга *выделение метода* (Extract Method) [Ref] для создания вызова *параметризованного теста* (Parametrized Test) из тела цикла. Проблема отсутствия *локализации дефектов* (Defect Localization) решается вызовом *параметризованного теста* (Parameterized Test) из отдельного *тестового метода* (Test Method) для каждого тестового условия. Для больших наборов значений более подходящим решением может оказаться *управляемый данными тест* (Data-Driven Test, с. 322).

Сложный в тестировании код (Hard-to-Test Code)

Код тестировать сложно.

Автоматизированное тестирование — это мощный инструмент, позволяющий быстро разрабатывать программное обеспечение даже при больших объемах уже написанного кода. Конечно, все эти преимущества доступны, только когда большая часть кода защищена *полностью автоматизированными тестами* (Fully Automated Tests, с. 81). Трудозатраты на написание таких тестов добавляются к затратам на создание самого кода. Не удивительно, что возникает желание упростить написание автоматизированных тестов. (Кроме того, затраты на тесты желательно компенсировать снижением затрат на другую деятельность. Лучше всего это достигается сокращением *частой отладки* (Frequent Debugging, с. 285) за счет написания тестов до написания кода и использования *локализации дефектов* (Defect Localization, с. 78).)

Сложный в тестировании код (Hard-to-Test Code) является одним из факторов, усложняющих эффективное с точки зрения затрат создание полных и корректных автоматизированных тестов.

Симптомы

Некоторые типы кода по своей природе с трудом поддаются тестированию, например компоненты графического интерфейса, многопотоковый код и код тестов. Причиной сложности является невидимость такого кода для тестов. Еще одной причиной сложности тестирования является слишком тесная связь кода с другими классами. Сложно создавать экземпляр объекта, когда конструктор не существует, является закрытым или принимает слишком много других объектов в качестве параметров.

Влияние

При наличии *сложного в тестировании кода* (Hard-to-Test Code) сложно проверить его качество автоматизированными средствами. Хотя проверка качества вручную возможна, она плохо поддается масштабированию, так как оценка качества после каждой модификации требует таких усилий, что это просто не будет делаться. Кроме того, подобная стратегия нераздельно связана с высокой стоимостью документации тестов.

Шаблоны решений

Лучше сделать код более приспособленным к тестированию. Это достаточно обширная тема, чтобы посвятить ей целую главу. В данном разделе будут рассмотрены только самые характерные примеры.

Причины

Существует множество причин появления *сложного в тестировании кода* (Hard-to-Test Code). Здесь рассматриваются самые распространенные из них.

Причина: тесно связанный код (Highly Coupled Code)**Симптомы**

Не удается протестировать класс изолированно от нескольких других классов.

Также известен как:

**Фиксированная зависимость
(Hard-Coded Dependency)**

Влияние

Тесно связанный код плохо приспособлен для модульного тестирования, так как он не работает в изоляции.

Основная причина

Причиной появления *тесно связанного кода* (Highly Coupled Code) могут быть несколько факторов, включая плохой дизайн, недостаток опыта в объектно-ориентированном проектировании, а также недостаточную стимуляцию к созданию слабо связанных кодов.

Возможное решение

Ключом к тестированию тесно связанного кода является ослабление связи. При разработке на основе тестов это является естественным результатом процесса разработки.

Для ослабления связи в целях тестирования обычно применяется *тестовый двойник* (Test Double, с. 538), а точнее — *тестовая заглушка* (Test Stub, с. 544) или *подставной объект* (Mock Object, с. 558). Более подробно данная тема рассматривается в главе 11, “Использование тестовых двойников”.

Интеграция тестов в существующий код является более сложной задачей, особенно при работе с унаследованным кодом. Это настолько обширная тема, что Майкл Фезерс написал целую книгу о способах интеграции тестов [WEwLC].

Причина: асинхронный код (Asynchronous Code)**Симптомы**

Класс невозможно протестировать с помощью непосредственных вызовов методов. Тест должен запустить выполняемый фрагмент (например, поток, процесс или приложение) и ожидать завершения настройки перед началом взаимодействия.

Влияние

Код с асинхронным интерфейсом сложен в тестировании, так как тестам приходится координировать свою работу с тестируемой системой. Данное требование вносит дополнительную сложность в тесты и приводит к значительному увеличению времени их работы. Последнее обстоятельство является определяющим для модульных тестов, которые должны работать как можно быстрее, иначе разработчики откажутся запускать их достаточно часто.

Основная причина

Тестируемый код реализации алгоритма тесно связан с активным объектом, внутри которого он обычно выполняется.

Возможное решение

При тестировании асинхронного кода ключевым действием является отделение логики от асинхронного механизма доступа. Шаблон *минимальный объект* (Humble Object, с. 700), включая *минимальный диалог* (Humble Dialog) и *минимальный выполняемый файл* (Humble Executable), является хорошим примером реструктуризации асинхронного кода для обеспечения синхронного тестирования.

Причина: нетестируемый код теста (Untestable Test Code)

Симптомы

Тело *тестового метода* (Test Method, с. 378) оказывается настолько непонятным (см. *Непонятный тест*, Obscure Test, с. 230) или содержит столько *условной логики теста* (Conditional Test Logic, с. 243), что возникают сомнения в корректности теста.

Влияние

Любая *условная логика теста* (Conditional Test Logic) внутри *тестовых методов* (Test Method) с большой вероятностью приводит к появлению *тестов с ошибками* (Buggy Test, с. 296) и становится причиной *высокой стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300). Слишком большой объем кода внутри тестового метода делает тест сложным для понимания и модификации.

Основная причина

Код внутри *тестового метода* (Test Method) по определению сложен для тестирования с помощью *самопроверяющихся тестов* (Self-Checking Test, с. 81). Для этого придется заменить тестируемую систему *тестовым двойником* (Test Double), возвращающим интересующую тест ошибку и запускающим тестовый метод внутри другого метода *теста на ожидаемое исключение* (Expected Exception Test; см. *Тестовый метод*, Test Method) — слишком сложная конструкция, нежелательная в большинстве ситуаций.

Возможное решение

Необходимости тестировать код внутри *тестовых методов* (Test Method) можно избежать, использовав предельно простой код и вынеся *условную логику теста* (Conditional Test Logic) во *вспомогательные методы теста* (Test Utility Method, с. 610). С написанием *самопроверяющихся тестов* (Self-Checking Test) для вспомогательных методов проблем обычно не возникает.

Дублирование тестового кода (Test Code Duplication)

Один и тот же тестовый код присутствует в нескольких местах.

Многие тесты в наборе выполняют аналогичные операции. Например, тесты часто повторяют сценарии, являющиеся вариацией одного и того же сценария. Многим тестам требуется схожая тестовая конфигурация. В некоторых случаях даже на этапе вызова тестируемой системы многие тесты содержат одну и ту же повторяющуюся нетривиальную логику.

Необходимость решать одинаковые задачи часто приводит к *дублированию тестового кода* (Test Code Duplication).

Симптомы

Некоторые тесты могут содержать общее подмножество одинаковых операторов, как показано в следующем примере.

```
public void testInvoice_addOneLineItem_quantity1_b() {
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    // Проверить единственный элемент
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    LineItem actual = (LineItem)lineItems.get(0);
    assertEquals(expItem.getInv(), actual.getInv());
    assertEquals(expItem.getProd(), actual.getProd());
    assertEquals(expItem.getQuantity(), actual.getQuantity());
}
public void testRemoveLineItemsForProduct_oneOfTwo() {
    // Настройка
    Invoice inv = createAnonInvoice();
    inv.addItemQuantity(product, QUANTITY);
    inv.addItemQuantity(anotherProduct, QUANTITY);
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.removeLineItemForProduct(anotherProduct);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    assertEquals(expItem.getInv(), actual.getInv());
    assertEquals(expItem.getProd(), actual.getProd());
    assertEquals(expItem.getQuantity(), actual.getQuantity());
}
```

Даже один тест может содержать повторяющиеся группы одинаковых операторов.

```
public void testInvoice_addTwoLineItems_sameProduct() {
    Invoice inv = createAnonInvoice();
    LineItem expItem1 = new LineItem(inv, product, QUANTITY1);
    LineItem expItem2 = new LineItem(inv, product, QUANTITY2);
    // Вызов
```

```

inv.addItemQuantity(product, QUANTITY1);
inv.addItemQuantity(product, QUANTITY2);
// Проверка
List lineItems = inv.getLineItems();
assertEquals("number of items", lineItems.size(), 2);
// Проверка первого элемента
LineItem actual = (LineItem)lineItems.get(0);
assertEquals(expItem1.getInv(), actual.getInv());
assertEquals(expItem1.getProd(), actual.getProd());
assertEquals(expItem1.getQuantity(), actual.getQuantity());
// Проверка второго элемента
actual = (LineItem)lineItems.get(1);
assertEquals(expItem2.getInv(), actual.getInv());
assertEquals(expItem2.getProd(), actual.getProd());
assertEquals(expItem2.getQuantity(), actual.getQuantity());
}

```

В обоих примерах явно видно *дублирование тестового кода* (Test Code Duplication). Для сравнения: дублирование в разных тестовых методах сложнее определить, чем в разных *классах теста* (Testcase Class, с. 401).

Влияние

Метод “копирования и вставки” часто приводит к появлению нескольких копий одного и того же кода. Такой код требует обслуживания при каждой модификации семантики методов тестируемой системы (включая количество аргументов, атрибуты аргументов, атрибуты возвращаемых объектов, последовательность вызова). Это значительно повышает стоимость добавления новой функциональности (см. *Высокая стоимость обслуживания тестов*, High Test Maintenance Cost, с. 300), так как приходится модифицировать все тесты, содержащие связанный с изменением код.

Причины

Причина: повторное использование через копирование (Cut-and-Paste Code Reuse)

“Копирование и вставка” является мощным инструментом быстрого написания кода, но в результате получается множество копий одного и того же кода, каждую из которых приходится обслуживать по-отдельности.

Основная причина

Повторное использование через копирование (Cut-and-Paste Code Reuse) часто применяется как единственный метод повторного использования. Разработчики, уделяющие внимание тому, “как” что-либо сделать, часто повторяют один и тот же код, поскольку не могут (или не желают тратить время) оценить общую картину (намерение) теста.

Фактором, способствующим применению такого подхода, может оказаться недостаток опыта или навыков рефакторинга, мешающий разработчикам понять общую картину на основе только что написанного подробного кода. Конечно, ограниченные временные рамки могут стать еще одной причиной отказа от рефакторинга. В результате со временем сложность тестового кода растет, хотя должно быть совсем наоборот.

Возможные решения

После появления *дублирования тестового кода* (Test Code Duplication) лучшим решением является применение рефакторинга *выделение метода* (Extract Method) [Ref] для создания *вспомогательного метода теста* (Test Utility Method, с. 610) на основе одного из примеров с последующим обобщением этого метода на работу в каждом случае дублирования. Если *дублированию тестового кода* (Test Code Duplication) подвержена настройка тестовой конфигурации, в результате рефакторинга получаются *методы создания* (Creation Method, с. 441) или *методы поиска* (Finder Method; см. *Вспомогательный метод теста*, Test Utility Method). Если дублирование наблюдается в логике проверки результата, рефакторинг дает *специальные утверждения* (Custom Assertion, с. 495) или *методы проверки* (Verification Method; см. *Специальные утверждения*).

Рефакторинг *введение параметра* (Introduce Parameter) [JBrians] позволяет преобразовать любую константу внутри извлеченного метода в параметр, который может передаваться для модификации поведения метода каждым вызывающим тестом.

Более простой способ избежать *дублирования тестового кода* (Test Code Duplication) — создать *тестовые методы* (Test Method) по направлению “извне внутрь”, уделяя основное внимание заложенному в тест намерению. Каждый раз, когда необходимо сделать что-то, требующее нескольких строк кода, просто вызывается несуществующий *вспомогательный метод теста* (Test Utility Method). Сначала пишутся тесты и только после этого создается реализация *вспомогательных методов теста* (Test Utility Method), позволяющая тестам компилироваться и запускаться. (Современные интегрированные среды разработки помогают такому процессу, обеспечивая автоматическую генерацию “скелетов” методов по щелчку мышью.)

Причина: изобретение колеса (Reinventing the Wheel)

В то время как *повторное использование через копирование* (Cut-and-Paste Code Reuse) приводит к намеренному созданию копии существующего кода для сокращения трудозатрат при создании тестов, иногда одинаковые последовательности операторов в разных тестах получаются случайно.

Основная причина

Обычно такая проблема возникает из-за недостатка информации о доступных *вспомогательных методах теста* (Test Utility Method). Кроме того, причиной может быть склонность разработчика к созданию собственного кода вместо использования чужого.

Возможное решение

Технически решение практически полностью совпадает с решением проблемы *повторного использования через копирование* (Cut-and-Paste Code Reuse), но с точки зрения процесса разработки решение отличается. Перед повторным изобретением колеса (созданием собственного кода) автор теста должен ознакомиться со всеми доступными *вспомогательными методами теста* (Test Utility Method).

Источники дополнительной информации

Впервые *дублирование тестового кода* (Test Code Duplication) было описано в докладе на конференции XP2001, который назывался “Refactoring Test Code” [RTC].

Логика теста в продукте (Test Logic in Production)

Предоставленный заказчику код продукта содержит логику, которая должна работать только во время тестирования.

Тестируемая система может содержать логику, которую невозможно запустить в тестовой среде. Тестам может потребоваться поведение, обеспечивающее полное покрытие кода тестируемой системы.

Симптомы

Тестируемая система содержит логику, предназначенную только для взаимодействия с тестами. Это может быть “дополнительная функциональность”, необходимая тесту для получения доступа к внутреннему состоянию тестируемой системы во время настройки тестовой конфигурации или проверки результатов. Также эта логика может модифицировать тестируемую систему, когда обнаруживается запуск под управлением теста.

Влияние

Желательно не оставлять *логику теста в продукте* (Test Logic in Production), так как это усложняет тестируемую систему и открывает доступ новым типам ошибок, которых хотелось бы избежать. Система, поведение которой отличается в тестовой лаборатории и в промышленной эксплуатации, может считаться настоящим рецептом катастрофы!

Причины

Причина: ловушка для теста (Test Hook)

Условная логика внутри тестируемой системы определяет, когда работает “настоящий” код, а когда код, предназначенный для тестов.

Симптомы

При наличии такого запаха кода симптомы на уровне поведения могут не проявляться или проблемы могут возникнуть только во время промышленной эксплуатации. В тестируемой системе могут возникать следующие фрагменты кода.

```
if (testing) {
    return hardCodedCannedData;
} else { // Настоящая логика...
    return gatheredData;
}
```

ARIANE

Первый полет ракеты Ariane 5 стал полной катастрофой: ракета взорвалась через 37 секунд после старта. Причиной стал, на первый взгляд, безобидный фрагмент кода, который использовался, когда ракета находилась на Земле, но, к сожалению, продолжил работу в первые 40 секунд полета. При попытке присвоить 64-разрядное число, означающее боко-

вую скорость, доступному в программном обеспечении ракеты регистру разрядностью 16 бит навигационный компьютер решил, что ракета движется в неправильном направлении! Компьютер попытался исправить курс, но резкая смена направления просто разорвала ракетный ускоритель. Хотя это не очень характерный пример *логики теста в продукте* (Test Logic in Production, с. 257), он иллюстрирует риски, связанные с подобными ошибками.

Можно ли было предотвратить эту катастрофу с помощью автоматизированных тестов? Сложно сказать с полной уверенностью, так как можно утверждать лишь то, что соответствующая модификация процессов разработки позволила бы избежать этой проблемы еще до ее появления, но вполне возможно, что автоматизированные тесты смогли бы предотвратить катастрофу.

В частности, тесты смогли бы обнаружить переполнение, когда число больше, чем можно сохранить в данном случае. Такой тест защитил бы от появления исключения при первой же попытке промышленной эксплуатации.

Кроме того, тесты, оставшиеся от ракеты Ariane 4, документировали бы максимальную скорость в направлении сверху вниз. Вполне возможно, что эти тесты были бы обновлены во время разработки программного обеспечения для Ariane 5 и завершились бы неудачно из-за более высокой скорости новой ракеты.

Более подробное (и очень интересное) описание “небольшой ошибки, которая смогла...”, приводится по адресу <http://www.around.com/ariane.html>.

Влияние

Не предназначенный для промышленной эксплуатации код может быть случайно запущен именно в таких условиях и привести к серьезным проблемам.

Ракета Ariane 5 взорвалась через 37 секунд после старта, так как фрагмент кода, предназначенный для запуска только на Земле, продолжал работать на протяжении 40 секунд во время полета. Этот код попытался присвоить 64-разрядное число, означающее боковую скорость ракеты, 16-разрядному полю. Данная операция заставила навигационный компьютер думать, что ракета движется в неправильном направлении (см. приведенную выше врезку). Несмотря на всю уверенность в том, что *ловушка для теста* (Test Hook) не сработает в промышленной эксплуатации, стоит ли искушать судьбу?

Основная причина

В одних случаях *логика теста в продукте* (Test Logic in Production) вводится для получения более определенного поведения от тестируемой системы за счет возврата известных (фиксированных) значений. В других ситуациях *логика теста в продукте* (Test Logic in Production) вводится для защиты от выполнения кода, который не может работать в тестовой среде. К сожалению, такой подход может приводить к невозможности запуска этого кода в промышленной эксплуатации, если в конфигурацию продукта вкрадлась ошибка.

Иногда от тестируемой системы требуется выполнение дополнительного кода, который в противном случае выполнялся бы в вызываемом компоненте. Например, код триггера в базе данных не будет работать, если заменить ее *поддельной базой данных* (Fake Database; см. *Поддельный объект*, Fake Object, с. 565). Таким образом, тест пытается обеспечить выполнение эквивалентной логики где-то внутри тестируемой системы.

Возможное решение

Вместо добавления логики теста в код продукта логику можно перенести в заменяемую зависимость. Код, работающий только в промышленных условиях, можно вставить в объект Strategy [GOF], который устанавливается по умолчанию и заменяется на Null Object [PLoPD3] при запуске тестов. С другой стороны, код, который должен работать только во время тестирования, может быть вставлен в объект Strategy [GOF], который по умолчанию заменяется на Null Object. Таким образом, во время тестирования дополнительный код вводится через замену объекта Strategy. Для обеспечения правильной настройки данного механизма необходим *тест конструктора* (Constructor Test; см. *Тестовый метод*, Test Method, с. 378), который будет проверять правильность инициализации переменных, содержащих ссылки на объект Strategy, когда они не переопределются тестом.

Кроме того, можно переопределять конкретные методы тестируемой системы с помощью *связанного с тестом подкласса* (Test-Specific Subclass, с. 591). Это возможно, когда интересующая тест логика продукта расположена в переопределяемых методах.

Причина: только для тестов (For Tests Only)

В тестируемой системе существует код, предназначенный только для тестов.

Симптомы

Некоторые методы тестируемой системы вызываются только тестами. Некоторые атрибуты являются открытыми, в то время как должны быть закрытыми.

Влияние

Программное обеспечение, которое добавляется в тестируемую систему *только для тестов* (For Tests Only), делает ее более сложной. Это может запутать потенциальных клиентов интерфейса из-за появления дополнительных методов, используемых только тестами. Такие методы тестировались только в особых ситуациях, поэтому могут отказать в типичных вариантах использования со стороны настоящего клиентского программного обеспечения.

Основная причина

Автору тестов может потребоваться добавить в класс методы, предоставляющие необходимую для тестов информацию или обеспечивающие большую степень контроля над процессом инициализации (например, для установки *тестового двойника*, Test Double). Разработка на основе тестов приводит к созданию таких методов даже в том случае, когда они не востребованы клиентами. При интеграции тестов в унаследованный код разработчику может потребоваться доступ к информации или функциям, которые пока доступны через внешний интерфейс.

Причина *только для тестов* (For Tests Only) может проявляться, когда тестируемая система используется асимметрично. Автоматизированные тесты (особенно тесты, работающие через открытый интерфейс) обычно используют программное обеспечение симметрично, а значит, могут потребовать методов, не нужных клиентскому программному обеспечению.

Возможное решение

Для обеспечения доступа тестов к закрытой информации можно воспользоваться *связанным с тестом подклассом* (Test-Specific Subclass), который предоставляет методы, открывающие доступ к интересующим тест атрибутам или логике инициализации. Для использования подобного подхода тест должен иметь возможность создавать экземпляры подкласса, а не класса самой тестируемой системы.

Если по какой-либо причине дополнительные методы невозможны перенести в *связанный с тестом подкласс* (Test-Specific Subclass), они должны быть явно обозначены как предназначенные *только для тестов* (For Tests Only). Это можно сделать с помощью соглашения об именовании (добавляя к именам методов строку FTO_).

Причина: зависимость продукта от теста (Test Dependency in Production)

Выполняемый файл продукта зависит от выполняемого файла с тестами.

Симптомы

Не удается откомпилировать только код продукта. В процессе компиляции требуется некоторая часть кода тестов. Возможен вариант, когда выполняемый файл продукта невозможно запустить, если выполняемый файл с тестами отсутствует.

Влияние

Даже если модули продукта не содержат код тестов, при появлении зависимости этих модулей от тестов могут возникать проблемы. Как минимум такая зависимость увеличивает размер выполняемого файла, даже если код тестов не используется при промышленной эксплуатации. Кроме того, появляется возможность запуска тестов во время промышленного использования.

Основная причина

Зависимость продукта от теста (Test Dependency in Production) обычно возникает из-за недостатка внимания к межмодульным зависимостям. Также подобная ситуация может возникать, когда **встроенный автотест** (built-in self-test) требует доступа к фрагментам инфраструктуры автоматизации тестов, например к *вспомогательным методам теста* (Test Utility Method, с. 610), для вывода результатов работы теста.

Возможное решение

Внимательно следите за зависимостями, чтобы код продукта не требовал даже таких безобидных фрагментов кода тестов, как определения типов.

Все, что необходимо как тесту, так и коду продукта, должно храниться в модуле продукта или в классе, доступном из обеих сред.

Причина: засорение равенства (Equality Pollution)

Еще одной причиной появления *логики теста в продукте* (Test Logic in Production) является реализация предназначенного для теста равенства в методе equals тестируемой системы.

Симптомы

Засорение равенства (Equality Pollution) сложно обнаружить после появления — можно обратить внимание, что тестируемой системе на самом деле не нужна реализация метода `equals`. В других случаях симптомы могут проявляться в поведении, например тесты завершаются неудачно при модификации метода `equals` специально для теста или при модификации метода `equals` в тестируемой системе в соответствии с новыми требованиями заказчика.

Влияние

Ненужные методы `equals` могут создаваться только для удовлетворения запросов теста. Кроме того, определение метода `equals` может измениться настолько, что оно не будет удовлетворять бизнес-требованиям к системе.

Засорение равенства (Equality Pollution) может усложнить добавления равенства в соответствии с новыми требованиями, если уже существует реализация для других тестов.

Основная причина

Причиной засорения равенства (Equality Pollution) является недостаточное понимание концепции предназначенного для тестов равенства. В некоторых ранних версиях утилит динамической генерации *подставных объектов* (Mock Object, с. 558) приходилось использовать определение метода `equals` из тестируемой системы, что приводило к *засорению равенства* (Equality Pollution).

Возможное решение

Если тесту необходима специальная реализация равенства, нужно воспользоваться *специальным утверждением* (Custom Assertion, с. 495), а не модифицировать реализацию метода `equals` в целях применения встроенного *утверждения равенства* (Equality Assertion; см. *Метод с утверждением*, Assertion Method, с. 390).

При использовании утилит динамической генерации подставных объектов (Mock Object) необходимо использовать Comparator [WWW], а не полагаться на метод `equals`, предоставленный тестируемой системой. Кроме того, метод `equals` можно реализовать в *связанном с тестом подклассе* (Test-Specific Subclass) *ожидаемого объекта* (Expected Object; см. *Проверка состояния*, State Verification, с. 484), что позволяет избежать добавления реализации непосредственно в код продукта.

Источники дополнительной информации

Запахи *только для тестов* (For Tests Only) и *засорение равенства* (Equality Pollution) впервые были описаны в докладе “Refactoring Test Code” [RTC] на конференции XP2001.

Глава 16

Запахи поведения

Запахи в этой главе:

Рулетка утверждений (Assertion Roulette)	264
Нестабильный тест (Erratic Test)	267
“Хрупкий” тест (Fragile Test)	277
Частая отладка (Frequent Debugging)	285
Ручное вмешательство (Manual Intervention)	287
Медленные тесты (Slow Tests)	289

Рулетка утверждений (Assertion Roulette)

Сложно сказать, какое из нескольких утверждений внутри тестового метода привело к неудачному завершению теста.

Симптомы

Тест завершается неудачно. Рассматривая вывод *программы запуска тестов* (Test Runner, с. 405), невозможно определить, какое утверждение оказалось ложным.

Влияние

При неудачном завершении тестов во время автоматизированной интеграции сложно определить, какое утверждение оказалось ложным. Если проблему невозможно воспроизвести на компьютере разработчика (что случается, если причина проблемы связана с особенностями среды или *оптимизмом по отношению к ресурсу*, Resource Optimism), устранение ошибки может оказаться сложным и длительным процессом.

Причины

Причина: “энергичный” тест (Eager Test)

Один тест проверяет слишком большое количество функций.

Симптомы

Тест вызывает несколько методов тестируемой системы или вызывает один и тот же метод несколько раз, перемежая вызовы созданием тестовой конфигурации и утверждениями.

```
public void testFlightMileage_asKm2() throws Exception {
    // Создание тестовой конфигурации
    // Вызов конструктора
    Flight newFlight = new Flight(validFlightNumber);
    // Проверка созданного объекта
    assertEquals(validFlightNumber, newFlight.number);
    assertEquals("", newFlight.airlineCode);
    assertNull(newFlight.airline);
    // Настройка пробега
    newFlight.setMileage(1122);
    // Вызов транслятора пробега
    int actualKilometres = newFlight.getMileageAsKm();
    // Проверка результатов
    int expectedKilometres = 1810;
    assertEquals(expectedKilometres, actualKilometres);
    // То же самое, но для отмененного рейса
    newFlight.cancel();
    try {
        newFlight.getMileageAsKm();
        fail("Expected exception");
    } catch (InvalidRequestException e) {
```

```

        assertEquals("Cannot get cancelled flight mileage",
                     e.getMessage());
    }
}

```

Еще одним симптомом может служить попытка модификации *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332) для защиты от останова при первом же ложном утверждении.

Основная причина

“Энергичный” тест (Eager Test) часто становится следствием намеренной или подсознательной минимизации количества модульных тестов, когда несколько тестовых условий проверяются внутри одного *тестового метода* (Test Method, с. 378). Хотя для выполняемых вручную тестов этот подход вполне оправдан (результаты интерпретируются человеком, способным модифицировать тесты “на лету”), для *полностью автоматизированных тестов* (Fully Automated Tests, с. 81) он просто не работает.

Еще одной причиной появления “энергичных” тестов (Eager Test) является использование пакета xUnit для автоматизации многошаговых приемочных тестов. В результате один тест вынужден проверять несколько условий. Такие тесты по определению длиннее, чем модульные тесты, но они должны быть как можно короче.

Возможное решение

“Энергичный” модульный тест разбивается на набор *тестов одного условия* (Single Condition Test, с. 99). Для этого можно воспользоваться многократным применением рефакторинга *выделение метода* (Extract Method) [Ref], что позволит выделить независимые фрагменты в отдельные *тестовые методы* (Test Method). Иногда проще клонировать тест по одному разу для каждого условия и удалить из *тестового метода* (Test Method) все лишнее (относительно данного тестового условия). Любой код, необходимый для создания тестовой конфигурации или перевода тестируемой системы в нужное состояние, может быть выделен в *метод создания* (Creation Method, с. 441). Хорошая среда разработки или компилятор подскажет, какие переменные больше не используются.

При автоматизации приемочных тестов с помощью xUnit тест может содержать слишком много операций, так как требуется сложная логика настройки тестовой конфигурации. В таком случае следует рассмотреть другие способы создания тестовой конфигурации для более поздних этапов теста. Если можно воспользоваться *настройкой через “черный ход”* (Back Door Setup; см. *Манипуляция через “черный ход”*, Back Door Manipulation, с. 359) для создания тестовой конфигурации для последней операции теста независимо от предыдущих операций, тест можно разбить на две части. Это улучшит *локализацию дефектов* (Defect Localization). Данный процесс следует повторять столько раз, сколько необходимо для получения достаточно коротких тестов, выполняющих функцию *донесения намерения* (Communicate Intent, с. 95).

Причина: отсутствующее сообщение для утверждения (Missing Assertion Message)

Симптомы

Тест завершается неудачно. Рассмотрение вывода *программы запуска тестов* (Test Runner) не позволяет определить, какое утверждение оказалось ложным.

Основная причина

Эта проблема является следствием вызовов *методов с утверждением* (Assertion Method, с. 391) с идентичными или отсутствующими *сообщениями для утверждения* (Assertion Message, с. 398). Чаще всего такая ситуация возникает при использовании *программы запуска тестов для командной строки* (Command-Line Test Runner) или программы, не интегрированной с текстовым редактором или средой разработки.

В следующем teste приводится несколько *утверждений равенства* (Equality Assertion; см. *Метод с утверждением*, Assertion Method).

```
public void testInvoice_addLineItem() {
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    LineItem actual = (LineItem)lineItems.get(0);
    assertEquals(expItem.getInv(), actual.getInv());
    assertEquals(expItem.getProd(), actual.getProd());
    assertEquals(expItem.getQuantity(), actual.getQuantity());
}
```

Как определить, какое из утверждений оказалось ложным? *Утверждение равенства* (Equality Assertion) обычно выводит ожидаемый результат и фактическое значение, но, если ожидаемые значения похожи или выводятся в неудобном формате, очень сложно определить ложное утверждение. Правильным решением является указание хотя бы минимального *сообщения для утверждения* (Assertion Message), если один и тот же *метод с утверждением* (Assertion Method) вызывается несколько раз подряд.

Возможное решение

Если проблема возникает при запуске тестов из графического интерфейса, интегрированного в среду разработки, должна существовать возможность щелкнуть на соответствующей строке вывода результата работы тестов и перехода на ложное утверждение. В противном случае можно включить отладчик и последовательно пройтись по тесту в поисках ложного утверждения.

Если проблема возникает при использовании интерфейса командной строки, можно попытаться запустить тесты из графического интерфейса, интегрированного со средой разработки. Если это не помогает, можно перейти к использованию номеров строк или найти интересующее утверждение методом исключения. Конечно, можно пересилить себя и добавить уникальное *сообщение для утверждения* (Assertion Message) в каждый вызов *метода с утверждением* (Assertion Method).

Источники дополнительной информации

Впервые *рулетка утверждений* (Assertion Roulette) и “энергичный” *тест* (Eager Test) были описаны в докладе “Refactoring Test Code” [RTC] на конференции XP2001.

Нестабильный тест (Erratic Test)

Один или несколько тестов ведут себя нестабильно; иногда они завершаются успешно, а иногда — неудачно.

Симптомы

Существует один или несколько тестов, которые запускаются, но дают разные результаты в зависимости от времени запуска и личности запускающего. В одних случаях *нестабильный тест* (Erratic Test) постоянно дает один и тот же результат при запуске одним разработчиком, но завершается неудачно при запуске кем-то другим или в другой среде. В других ситуациях *нестабильный тест* (Erratic Test) возвращает разные результаты при запуске одной и той же *программой запуска тестов* (Test Runner, с. 405).

Влияние

Возникает соблазн удалить такой тест из набора, но это приведет к намеренно *потерянному тесту* (Lost Test; см. *Ошибки в продукте*, Production Bugs, с. 303). Если *нестабильный тест* (Erratic Test) остается в наборе несмотря на отказы, известная ошибка может скрыть другие проблемы, которые также должен обнаруживать этот тест. Неудачное завершение теста вынудит разработчика пропустить дополнительные ошибки, так как значительно сложнее заметить переход от зеленого цвета к красному, чем отказ двух тестов вместо одного.

Совет по диагностике

Из-за множества возможных причин диагностика *нестабильного теста* (Erratic Test) может оказаться сложным процессом. Если причину невозможно определить сразу, имеет смысл систематически собирать данные за определенный период. Собранные данные должны позволить найти ответы на следующие вопросы. Где (в какой среде) тест завершается успешно, а где происходит отказ теста? Все ли тесты запускались или было запущено подмножество набора? Происходит ли изменение поведения при запуске набора тестов несколько раз подряд? Происходит ли изменение поведения при запуске набора несколькими *программами запуска тестов* (Test Runner) одновременно?

После получения данных можно сравнить наблюдаемые симптомы с симптомами перечисленных потенциальных причин. Так список возможных причин нестабильного теста сократится и дополнительные данные позволят сконцентрировать внимание на оставшихся в списке причинах. На рис. 16.1 показан процесс определения причин появления *нестабильного теста* (Erratic Test).

Причины

Тест может вести себя нестабильно в силу целого ряда причин. Конкретную причину обычно можно определить путем внимательного наблюдения за шаблонами неудачного завершения тестов. Некоторые из этих причин достаточно распространены и имеют собственные имена и конкретные способы устранения.

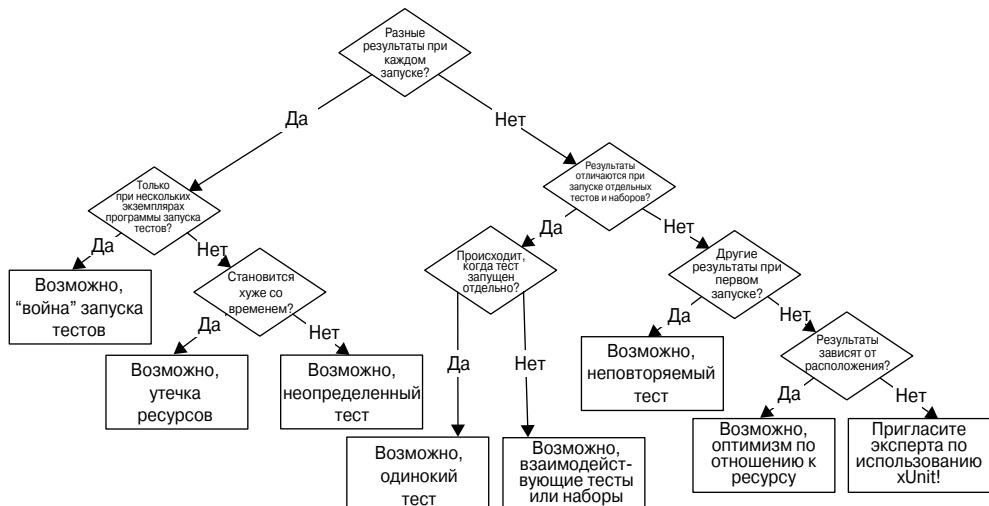


Рис. 16.1. Диагностика нестабильного теста (Erratic Test)

Причина: взаимодействующие тесты (Interacting Tests)

Одни тесты зависят от других тестов. Обратите внимание, что *взаимодействующие наборы тестов* (Interacting Test Suites) и *одинокие тесты* (Lonely Test) являются вариантами *взаимодействующих тестов* (Interacting Tests).

Симптомы

Работающий в одиночку тест вдруг завершается неудачно при следующих обстоятельствах:

- еще один тест добавлен в набор или удален из него;
- еще один тест в пределах набора завершается неудачно (или успешно);
- этот или другой тест переименован или перемещен в другой файл с исходным кодом;
- установлена новая версия *программы запуска тестов* (Test Runner).

Основная причина

Взаимодействующие тесты (Interacting Tests) обычно возникают при использовании *общей тестовой конфигурации* (Shared Fixture, с. 350), когда один тест зависит от результатов работы других тестов. Причину *появления взаимодействующих тестов* (Interacting Tests) можно описать с двух точек зрения:

- с точки зрения механизма взаимодействия;
- с точки зрения причин взаимодействия.

Механизм взаимодействия может оказаться предельно очевидным (например, тестируемая система включает в себя базу данных) или достаточно скрытым. Все, что существует дольше самого теста, может привести к взаимодействию. Может возникнуть зависимость от статических переменных, что приведет к взаимодействию тестов. Следовательно, нужно из-

бегать такой зависимости как в тестируемой системе, так и в *инфраструктуре автоматизации тестов* (Test Automation Framework, с. 332)! Пример последней проблемы приводится во врезке “Всегда есть исключения” на с. 411. Объекты Singleton [GOF] и Registry [PEAA] являются примерами сущностей, присутствия которых в тестируемой системе необходимо избегать любой ценой. Если их использование неизбежно, лучше включить механизм повторной инициализации статических переменных в начале каждого теста.

Тесты могут взаимодействовать по целому ряду причин, связанных как с проектированием, так и со случайностями:

- зависимость от тестовой конфигурации, созданной другим тестом;
- зависимость от изменений, внесенных в тестируемую систему во время работы другого теста;
- коллизия в результате взаимно исключающих действий двух тестов, запущенных одновременно.

Внезапно зависимость может оказаться неудовлетворенной, если другой тест:

- удален из набора;
- модифицирован таким образом, что больше не вносит изменения в тестируемую систему;
- завершается неудачно при попытке модификации состояния тестируемой системы;
- запускается после интересующего теста (так как был переименован или перенесен в другой *класс теста* (Testcase Class, с. 401)).

Точно так же коллизии могут возникать, когда тест:

- добавляется в набор;
- успешно завершается в первый раз;
- запускается перед зависимым тестом.

В большинстве подобных случаев тесты будут завершаться неудачно. Некоторые из них завершаются неудачно по правильной причине: поведение тестируемой системы отличается от ожидаемого поведения. Зависимые тесты завершаются неудачно по неправильной причине — они созданы так, чтобы зависеть от успешного завершения других тестов. В результате тест может давать ложное срабатывание.

В общем, зависимость от порядка выполнения тестов считается не очень умным подходом из-за перечисленных выше проблем. Большинство реализаций xUnit не гарантирует порядка вызова тестов в пределах набора. (В то же время пакет TestNG рекомендует формировать зависимости между тестами и предоставляет функции управления этими зависимостями.)

Возможное решение

Использование *новой тестовой конфигурации* (Fresh Fixture, с. 344) является рекомендованным решением проблемы *взаимодействующих тестов* (Interacting Tests). Практически всегда это приводит к решению проблемы. Если приходится использовать *общую тестовую конфигурацию* (Shared Fixture), следует рассмотреть использование *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture). Это защитит тесты от взаимодействия друг с другом через изменения в тестовой конфигурации (модифицируемые фрагменты конфигурации тестам придется создавать каждый раз заново).

Если неудовлетворенная зависимость возникает из-за отсутствия объекта или данных в базе данных, создаваемых другим тестом, рассмотрите использование “ленивой” настройки (Lazy Setup, с. 460) для создания объектов или данных в каждом из тестов. Подобный подход заставляет первый из запущенных тестов создавать необходимые объекты и данные для обоих тестов. Логику создания конфигурации можно вынести в *метод создания* (Creation Method, с. 441) и таким способом избежать дублирования тестового кода (Test Code Duplication, с. 254). Если тесты расположены в разных классах теста (Testcase Class), код создания тестовой конфигурации можно вынести во *вспомогательный класс теста* (Test Helper, с. 651).

Иногда коллизия может вызываться объектами или данными в базе данных, созданными тестом, но не удаленными после использования. В таком случае необходимо использовать *автоматическое удаление тестовой конфигурации* (Automated Fixture Teardown; см. *Автоматическая очистка*, Automated Teardown, с. 521) для безопасной и эффективной очистки.

Простым способом обнаружения зависимости тестов один от другого является их запуск в отличном от обычного порядке. Например, запуск всего набора в обратном порядке может оказаться вполне достаточным. Регулярные проверки подобного характера позволяют избежать случайного появления *взаимодействующих тестов* (Interacting Tests).

Причина: взаимодействующие наборы тестов (Interacting Test Suites)

Это специальный случай *взаимодействующих тестов* (Interacting Tests), расположенных в разных наборах.

Симптомы

Тест завершается успешно при запуске из собственного набора. При запуске из *набора наборов* (Suite of Suites, см. *Объект набора тестов*, Test Suite Object, с. 414) тест завершается неудачно.

```
Suite1.run() --> Зеленый
Suite2.run() --> Зеленый
Suite(Suite1,Suite2).run() --> Тест С в наборе Suite2 завершается неудачно
```

Основная причина

Взаимодействующие наборы тестов (Interacting Test Suites) обычно возникают, когда тесты из разных наборов пытаются создавать один и тот же ресурс. При запуске из одного набора первый завершается успешно, а второй неудачно сразу при попытке создания ресурса.

Причина проблемы может показаться очевидной при чтении сообщения о неудачном завершении теста или при знакомстве с содержимым *тестового метода* (Test Method, с. 378). Если это не так, можно попытаться по одному удалять другие тесты из успешно завершающего работу набора. Как только отказы прекращаются, последний удаленный тест рассматривается на предмет поведения, приводящего к взаимодействию с другими тестами (завершающимися неудачно). В частности, необходимо обратить внимание на все, что может подразумевать использование *общей тестовой конфигурации* (Shared Fixture), включая фрагменты инициализации переменных классов. Эти фрагменты могут находиться внутри *тестовых методов* (Test Method), внутри метода `setUp` или в любом из вызываемых *вспомогательных методов теста* (Test Utility Method, с. 610).

Внимание: Может существовать более одной пары тестов, взаимодействующей в пределах одного набора!

Взаимодействие может вызываться пересечением *настройки тестовой конфигурации набора* (Suite Fixture Setup, с. 465) или *декоратора настройки* (Setup Decorator, с. 471) нескольких классов теста (Testcase Class), а не конфликтов между фактическими *тестовыми методами* (Test Method)!

В реализациях xUnit, использующих *обнаружение классов теста* (Testcase Class Discovery; см. *Обнаружение тестов*, Test Discovery, с. 420), например в NUnit, наборы тестов не применяются. На самом деле они есть, но от разработчика тестов не ожидается использование *фабрики наборов тестов* (Test Suite Factory) для идентификации *объекта набора тестов* (Test Suite Object) для *программы запуска тестов* (Test Runner).

Возможное решение

Полностью избавиться от этой проблемы можно за счет использования *новой тестовой конфигурации* (Fresh Fixture). Если это решение недоступно, нужно попытаться воспользоваться *немодифицируемой общей тестовой конфигурацией* (Immutable Shared Fixture) для предотвращения взаимодействия тестов.

Если причиной проблемы является неудаленный объект или запись в базе данных, созданная одним из конфликтующих тестов, можно воспользоваться *автоматической очисткой* (Automated Teardown) и избежать написания сопряженного с ошибками кода очистки.

Причина: одинокий тест (Lonely Test)

Одинокий тест (Lonely Test) является специальным случаем *взаимодействующих тестов* (Interacting Tests). Тест может запускаться как часть набора, но не может работать в одиночку, поскольку зависит от элемента *общей тестовой конфигурации* (Shared Fixture), созданного другим тестом (см. *Цепочки тестов*, Chained Tests, с. 477) или логикой создания тестовой конфигурации на уровне набора (например, *декоратор настройки*, Setup Decorator).

Эта проблема решается посредством модификации теста для использования *новой тестовой конфигурации* (Fresh Fixture) или добавления логики “ленивой” *настройки* (Lazy Setup) в *одинокий тест* (Lonely Test), что позволит запускать этот тест отдельно от набора.

Причина: утечка ресурсов (Resource Leakage)

Тесты и тестируемая система потребляют ограниченный ресурс.

Симптомы

Тесты работают все медленнее и медленнее или неожиданно начинают завершаться неудачно. Повторная инициализация *программы запуска тестов* (Test Runner), тестируемой системы или “*песочницы*” с базой данных (Database Sandbox, с. 658) позволяет решить проблему, но проблема повторяется со временем.

Основная причина

Тесты и тестируемая система потребляют конечные ресурсы, выделяя ресурсы и не освобождая их после завершения работы. Подобная практика может заставить тесты работать медленнее. Со временем все ресурсы оказываются использованными и зависящими от них тесты просто завершаются неудачно.

Причиной подобных проблем могут стать ошибки одного из двух типов.

- Тестируемая система не выполняет правильной очистки ресурсов. Чем раньше обнаруживается подобное поведение, тем раньше можно исправить ошибку.
- Сам тест вызывает утечку ресурсов, выделяя их во время создания тестовой конфигурации и не освобождая их во время очистки тестовой конфигурации.

Возможное решение

Если проблема заключается в тестируемой системе, то можно считать, что тесты выполнили свою задачу и ошибку можно исправлять. Если причиной утечки ресурсов являются тесты, необходимо исключить источник утечки. Если проблема возникает из-за неправильного освобождения ресурсов после неудачного завершения, потребуется добавить в каждый тест *гарантированную встроенную очистку* (Guaranteed In-line Teardown) или модифицировать тесты для использования *автоматической очистки* (Automated Teardown).

Обычно желательно устанавливать размер пула ресурсов равным 1, что заставит тесты завершаться неудачно намного раньше, ускоряя обнаружение теста с утечкой.

Причина: оптимизм по отношению к ресурсу (Resource Optimism)

Зависящий от внешнего ресурса тест демонстрирует неопределенные результаты, зависящие от места и времени запуска.

Симптомы

Тест завершается успешно при запуске в одной среде и неудачно — при запуске в другой.

Основная причина

Ресурс, доступный в одной среде и отсутствующий в другой.

Возможное решение

По возможности необходимо преобразовать тест для использования новой тестовой конфигурации (Fresh Fixture), когда ресурс создается как часть конфигурации. Этот подход позволяет обеспечить существование ресурса при каждом запуске теста. При этом может потребоваться использование относительной адресации файлов, чтобы конкретное место в файловой системе существовало вне зависимости от места запуска тестируемой системы.

Если должен использоваться внешний ресурс, ресурсы должны находиться в хранилище исходного кода, чтобы все *программы запуска тестов* (Test Runner) работали в одной и той же среде.

Причина: неповторяемый тест (Unrepeatable Test)

Поведение теста при первом и последующих запусках отличается. Фактически тест взаимодействует сам с собой на протяжении нескольких запусков.

Симптомы

Тест или завершается успешно при первом запуске и отказывает при всех последующих, или отказывает в первый раз и завершается успешно при каждом последующем запуске. Ниже приведен пример первого процесса.

```

Suite.run() --> Зеленый
Suite.run() --> Тест С завершается неудачно
Suite.run() --> Тест С завершается неудачно
Пользователь возвращает что-то в исходное состояние
Suite.run() --> Зеленый
Suite.run() --> Тест С завершается неудачно

```

А вот пример второго процесса.

```

Suite.run() --> Тест С завершается неудачно
Suite.run() --> Зеленый
Suite.run() --> Зеленый
Пользователь возвращает что-то в исходное состояние
Suite.run() --> Тест С завершается неудачно
Suite.run() --> Зеленый

```

Учтите, что, если набор содержит несколько *неповторяемых тестов* (*Unrepeatable Test*), результаты будут выглядеть следующим образом.

```

Suite.run() --> Тест С завершается неудачно
Suite.run() --> Тест Х завершается неудачно
Suite.run() --> Тест Х завершается неудачно
Пользователь возвращает что-то в исходное состояние
Suite.run() --> Тест С завершается неудачно
Suite.run() --> Тест Х завершается неудачно

```

Тест С демонстрирует поведение “Fail-Pass-Pass”, а тест Х в то же время демонстрирует поведение “Pass-Fail-Fail”. Очень легко пропустить эту проблему, так как постоянно присутствуют неудачно завершающиеся тесты. Обнаружить разницу можно, только рассматривая каждый неудачно завершившийся тест при каждом запуске.

Основная причина

Самой распространенной причиной *неповторяемого теста* (*Unrepeatable Test*) является намеренное или случайное использование *общей тестовой конфигурации* (*Shared Fixture*). Тест может модифицировать тестовую конфигурацию таким образом, что во время последующих запусков набора тестов конфигурация будет иметь другое состояние. Хотя чаще всего проблема возникает при использовании *предварительно созданной тестовой конфигурации* (*Prebuilt Fixture*), единственным необходимым условием является существование тестовой конфигурации дольше, чем работает тест.

Использование “ *песочницы*” с базой данных (*Database Sandbox*) может изолировать тест от тестов других разработчиков, но не защитит от взаимодействия с собой или с другими тестами той же программы запуска тестов (*Test Runner*).

Использование “*ленивой*” настройки (*Lazy Setup*) для инициализации переменной класса с тестовой конфигурацией позволяет не инициализировать конфигурацию при следующих запусках этого же набора. Фактически тестовая конфигурация используется совместно всеми экземплярами тестов, запущенными одной *программой запуска тестов* (*Test Runner*).

Возможное решение

Поскольку постоянная общая тестовая конфигурация (*Shared Fixture*) является необходимым условием появления *неповторяемых тестов* (*Unrepeatable Test*), проблему можно

решить, воспользовавшись *новой тестовой конфигурацией* (Fresh Fixture) в каждом teste. Для полной изоляции необходимо обеспечить, чтобы ни один совместно используемый ресурс, например “песочница” с базой данных (Database Sandbox), не переживал отдельный тест. Одним из вариантов является замена настоящей базы данных на поддельную (Fake Database). Если приходится работать с постоянным хранилищем данных, нужно воспользоваться *отдельными сгенерированными значениями* (Distinct Generated Value, с. 728) для всех ключей в базе данных, чтобы при каждом запуске теста создавались новые объекты. Еще одной альтернативой является реализация *автоматической очистки* (Automated Teardown) для эффективного и безопасного удаления всех созданных объектов и строк.

Причина: “война” запуска тестов (Test Run War)

Если несколько разработчиков запускают testeы одновременно, testeы завершаются неудачно случайным образом.

Симптомы

Запускаемые testeы зависят от совместно используемого внешнего ресурса, например от базы данных. С точки зрения одного разработчика, картина может выглядеть следующим образом.

```
Suite.run() -> Тест 3 завершается неудачно
Suite.run() -> Тест 2 завершается неудачно
Suite.run() -> Все testeы завершаются успешно
Suite.run() -> Тест 1 завершается неудачно
```

Поделившись с коллегами результатами, разработчик узнает, что они сталкиваются с теми же проблемами одновременно с ним. Если только один разработчик запускает testeы, все testeы завершаются успешно.

Влияние

“Война” запуска testeов (Test Run War) очень мешает разработчикам, так как ее вероятность повышается с приближением к конечному сроку предоставления продукта. Это не просто закон Мерфи: проблема действительно более вероятна в это время! С приближением конечного срока разработчики вносят изменения меньшего объема с большей частотой (“исправление ошибок в последнюю минуту”). Это, в свою очередь, повышает вероятность запуска набора testeов другим разработчиком, что приводит к коллизиям между testeами, работающими одновременно.

Основная причина

“Война” запуска testeов (Test Run War) происходит только при использовании глобальной общей testeовой конфигурации (Shared Fixture), к которой получают доступ и в которую вносят модификации множество testeов. В качестве такой конфигурации может выступать файл, открываемый testeом или тестируемой системой, или записи в testeовой базе данных.

Борьба за базу данных может происходить при следующих обстоятельствах:

- попытка обновления или удаления записи одновременно с другим testeом;
- попытка обновления или удаления записи, на которую другой teste установил блокировку чтения (пессимистическая блокировка);

Борьба за файлы может вызываться попыткой получения доступа к файлу, который уже открыт другим экземпляром теста, запущенным другим разработчиком.

Возможное решение

Использование *новой тестовой конфигурации* (Fresh Fixture) является предпочтительным решением для “войны” запуска тестов (Test Run War). Еще более простым решением является предоставление каждой программе запуска тестов (Test Runner) собственно “песочницы” с базой данных (Database Sandbox). Это не потребует внесения изменений в тесты, но полностью исключит вероятность “войны” запуска тестов (Test Run War). Однако от других источников *нестабильных тестов* (Erratic Test) это не избавит, так как останется взаимодействие через общую тестовую конфигурацию (Shared Fixture) в виде “песочницы” с базой данных (Database Sandbox). Еще одно решение — перейти к использованию *немодифицируемой общей тестовой конфигурации* (Shared Fixture), заставив каждый тест создавать новый объект для внесения изменений. Такой подход потребует внесения изменений в *тестовые методы* (Test Method).

Если причиной проблемы являются оставшиеся после предыдущего запуска теста объекты или записи в базе данных, решением может стать использование *автоматической очистки* (Automated Teardown) после каждого запуска теста. Сама по себе эта мера не позволит избежать “войны” запуска тестов (Test Run War), но частоту отрицательных эффектов значительно снизит.

Причина: неопределенный тест (Nondeterministic Test)

Тест случайным образом завершается неудачно, даже когда запущена только одна программа запуска тестов (Test Runner).

Симптомы

Результат работы тестов отличается при каждом запуске.

```
Suite.run() -> Тест 3 завершается неудачно
Suite.run() -> Тест 3 завершается с ошибкой
Suite.run() -> Все тесты завершаются успешно
Suite.run() -> Тест 3 завершается неудачно
```

После сравнения записей с коллегами “война” запуска тестов (Test Run War) исключается, так как тесты запускал только один разработчик или конфигурация не используется совместно несколькими пользователями или компьютерами.

Как и в случае *неповторяемых тестов* (Unrepeatable Test), несколько *неопределенных тестов* (Nondeterministic Test) в пределах одного набора могут значительно усложнить поиск данной проблемы. Все будет выглядеть так, как будто неудачно завершаются разные тесты, а не один тест каждый раз дает новый результат.

Влияние

Отладка *неопределенного теста* (Nondeterministic Test) может потребовать очень много времени и сил, так как код работает по-разному при каждом запуске. Воспроизведение ошибки может оказаться невозможным, а поиск причины может потребовать нескольких подходов. (После определения причины происходит простая процедура замены случайног значения значением, гарантированно приводящим к проблеме.)

Основная причина

Неопределенный тест (Nondeterministic Test) возникает из-за использования разных значений при каждом запуске теста. Конечно, иногда использовать разные значения при каждом запуске имеет смысл. Например, *отдельное сгенерированное значение* (Distinct Generated Value) используется как уникальный ключ объекта, хранящегося в базе данных. Использование сгенерированных значений в качестве параметров алгоритма, если поведение тестируемой системы меняется при использовании разных значений, появляется *неопределенный тест* (Nondeterministic Test), как показано в следующих примерах.

- Целые значения, когда отрицательные (и нулевые) значения рассматриваются системой по-другому или когда существует максимально допустимое значение. Если значение генерируется случайным образом, тест может в одних случаях завершаться неудачно, а в других — успешно.
- Строковые значения, когда есть ограничения на минимальную и максимальную длину строки. Часто проблема возникает случайно при генерации случайного или уникального числового значения с последующим преобразованием в строковое представление без использования явного формата, гарантирующего постоянную длину строки.

Использование случайных значений может показаться хорошей идеей, так как позволяет увеличить покрытие тестов. К сожалению, такая тактика усложняет понимание и повторяемость тестов (что нарушает принцип *повторяемости тестов* (Repeatable Tests, с. 81)).

Еще одной потенциальной причиной является использование в тестах *условной логики теста* (Conditional Test Logic, с. 243). Ее включение может привести к появлению разных ветвей кода при каждом запуске тестов. В результате тесты становятся неопределенными. Распространенной “причиной” этого является *гибкий тест* (Flexible Test; см. *Условная логика теста*, Conditional Test Logic). Все, что делает тесты менее чем полностью определенными, является не очень умной идеей!

Возможное решение

Первым этапом получения повторяемого теста является обеспечение полностью линейного выполнения без *условной логики теста* (Conditional Test Logic). После этого случайные значения можно заменить определенными. Если это приводит к плохому покрытию тестами, можно добавить больше тестов, специально предназначенных для интересующих случаев. Хорошим способом определения лучшего множества параметров является использование граничных значений классов эквивалентности. Если их применение приводит к *дублированию тестового кода* (Test Code Duplication), можно выделить *параметризованный тест* (Parameterized Test, с. 618) или разместить параметры и ожидаемые результаты в файле, который будет читать *управляемый данными тест* (Data-Driven Test, с. 322).

“Хрупкий” тест (Fragile Test)

Тест не компилируется или не запускается при модификации тестируемой системы, даже если модификация не касается области проверки теста.

Симптомы

Один или несколько тестов завершались успешно, но сейчас они или не компилируются, или завершаются неудачно. При модификации проверяемого поведения тестируемой системы такой результат вполне ожидаем. Но если внесенное изменение не должно влиять на конкретные тесты, можно констатировать появление “хрупкого” теста (Fragile Test).

Раньше усилия по автоматизации тестов часто наталкивались на “четыре чувствительности” автоматизированных тестов. Именно такая чувствительность заставляет ранее работавшие *полностью автоматизированные тесты* (Fully Automated Tests, с. 81) неожиданно завершаться неудачно. Основные причины можно классифицировать как одну из четырех чувствительностей. Хотя каждая из них может быть следствием конкретного поведения кода теста, важно понимать природу каждой чувствительности.

Влияние

“Хрупкий” тест (Fragile Test) увеличивает стоимость обслуживания тестов, требуя модифицировать больше тестов при каждом изменении функциональности системы или тестовой конфигурации. Особенно вредным “хрупкий” тест (Fragile Test) оказывается в проектах, основанных на строгой инкрементной доставке при гибких процессах разработки (например, при экстремальном программировании).

Совет по диагностике

Необходимо отследить шаблоны неудачного завершения тестов. Задайте себе вопрос: “Что общего между всеми неудачно завершившимися тестами?” Ответ на этот вопрос позволит понять, как тесты связаны с тестируемой системой. После этого необходимо найти способы ослабления данной связи.

На рис. 16.2 показан процесс определения конкретной чувствительности.

Обычно сначала необходимо убедиться, что тест компилируется. Если это не так, то вероятной причиной является *чувствительность к интерфейсу* (Interface Sensitivity). При использовании динамических языков сообщения об ошибках несовместимости типов выводятся во время выполнения — это еще один признак *чувствительности к интерфейсу* (Interface Sensitivity).

Если тест запускается, но тестируемая система возвращает неверный результат, необходимо проверить, есть ли в коде изменения. Если изменения были, можно вернуться к предыдущей версии кода и проверить, не решает ли это проблему. Если после этого тест завершается успешно, значит, причиной является *чувствительность к поведению* (Behavior Sensitivity). (Другие тесты могут завершаться неудачно, так как удаленные последние изменения могли обеспечивать их работу. Но как минимум это позволяет обнаружить фрагменты кода, от которого зависят тесты.)



Рис. 16.2. Диагностика “хрупкого” теста

Если тест все равно завершается неудачно при отмене последних изменений кода, значит, изменилось что-то еще и, скорее всего, наблюдается *чувствительность к данным* (Data Sensitivity) или *чувствительность к контексту* (Context Sensitivity). Первый вариант возможен только при использовании *общей тестовой конфигурации* (Shared Fixture, с. 350) или при модификации кода создания тестовой конфигурации. В противном случае наблюдается *чувствительность к контексту* (Context Sensitivity).

Хотя этот метод не описывает все возможные варианты, в девяти случаях из десяти он дает правильный ответ.

Причины

“Хрупкий” тест (Fragile Test) может возникнуть по нескольким причинам. Он может быть признаком *опосредованного тестирования* (Indirect Testing; см. *Непонятный тест*, Obscure Test, с. 230), когда модифицированные объекты используются для доступа к другим объектом. Также подобные тесты могут служить признаком применения “энергичных” тестов (Eager Test; см. *Рулетка утверждений*, Assertion Roulette, с. 264), проверяющих слишком много тестовых условий одновременно. “Хрупкий” тест (Fragile Test) может служить симптомом слишком тесной связности программного обеспечения, что делает невозможным тестирование небольших фрагментов (см. *Сложный в тестировании код*, Hard-to-Test Code, с. 251), или недостаточного опыта в модульном тестировании с использованием *тестовых двойников* (Test Double, с. 538) для изоляции компонентов (*зарегулированная программа*, Overspecified Software).

Вне зависимости от основной причины “хрупкий” тест (Fragile Test) обычно проявляется как одна из четырех чувствительностей. Рассмотрим каждую из них подробнее. После этого будут приведены более детальные примеры изменения вывода теста в зависимости от причины “хрупкости” теста.

Причина: чувствительность к интерфейсу (Interface Sensitivity)

Чувствительность к интерфейсу (Interface Sensitivity) возникает, когда тест не компилируется или не запускается из-за модификации интерфейса тестируемой системы.

Симптомы

В статически типизированных языках *чувствительность к интерфейсу* (Interface Sensitivity) обычно выражается в невозможности компиляции. В динамически типизированных языках проблема проявляется только во время запуска тестов. Тест на динамически типизированном языке может приводить к ошибке во время вызова модифицированного программного интерфейса приложения (application programming interface — API). Модификации могут включать в себя изменение имени или сигнатуры метода. С другой стороны, тест может испытывать затруднения при поиске элемента пользовательского интерфейса, необходимого для взаимодействия с тестируемой системой. *Записанные тесты* (Recorded Test, с. 312), взаимодействующие с тестируемой системой через пользовательский интерфейс¹, особенно сильно страдают от этой проблемы.

Возможное решение

Обычно причина ошибки очевидна. Точка появления ошибки (во время компиляции или во время работы) обычно указывает на расположение проблемы. Очень редко тест работает после столкновения с изменением — в конце концов, именно изменение вызывает ошибку в работе теста.

Если интерфейс используется только внутри организации или приложения и для взаимодействия с автоматизированными тестами, *инкапсуляция программного интерфейса тестируемой системы* (SUT API Encapsulation; см. *Вспомогательный метод теста*, Test Utility Method, с. 610) является лучшим решением для устранения *чувствительности к интерфейсу* (Interface Sensitivity). При этом снижаются стоимость и влияние изменений программного интерфейса, а значит, разработчикам ничто не мешает вносить новые изменения. Распространенным способом реализации инкапсуляции является определение языка высокого уровня (Higher-Level Language, с. 95), на котором описываются тесты. Уровень инкапсуляции транслирует глаголы языка тестов в соответствующие вызовы методов. В результате после внесения изменений в интерфейс придется модифицировать только определение языка. “Язык тестов” можно реализовать в виде *вспомогательных методов теста* (Test Utility Method), например *методов создания* (Creation Method, с. 441) и *методов проверки* (Verification Method, с. 613), которые скрывают интерфейс тестируемой системы от тестов.

Единственным способом избежать *чувствительности к интерфейсу* (Interface Sensitivity) является строгий контроль изменений интерфейса. Если клиенты интерфейса являются внешними и анонимными (например, клиенты библиотек Windows DLL), подобная тактика может оказаться единственной возможной альтернативой, т.е. изменения должны быть обратно совместимыми. Перед удалением старые версии методов должны быть объявлены устаревшими, а устаревшие методы должны существовать некоторое время.

Причина: чувствительность к поведению (Behavior Sensitivity)

Чувствительность к поведению (Behavior Sensitivity) возникает, когда изменения в тестируемой системе заставляют тесты завершаться неудачно.

¹ Часто этот подход называется “скоблением экрана”.

Симптомы

Нормально работавший тест вдруг начинает завершаться неудачно после добавления новой функции или исправления ошибки в тестируемой системе.

Основная причина

Тест может завершаться неудачно из-за модификации проверяемой им функциональности. Такой результат не обязательно указывает на *чувствительность к поведению* (Behavior Sensitivity), поскольку именно для этого существуют регрессионные тесты. Но в следующих обстоятельствах чувствительность к поведению диагностируется однозначно.

- Модифицирована функциональность, которая используется регрессионными тестами для перевода тестируемой системы в предтестовое состояние
- Модифицирована функциональность, которая используется регрессионными тестами для проверки посттестового состояния тестируемой системы
- Модифицирован код, который используется регрессионными тестами для очистки тестовой конфигурации

Если модифицированный код не является частью тестируемой системы, то это случай *чувствительности к контексту* (Context Sensitivity), т.е. проверяется слишком большая тестируемая система. Тестируемую систему необходимо разделить на проверяемый фрагмент и компоненты, от которых он зависит.

Возможное решение

Любое некорректное предположение о поведении тестируемой системы на этапе создания тестовой конфигурации может быть скрыто в *методе создания* (Creation Method). Точно так же предположения о состоянии системы после теста могут быть скрыты в *специальном утверждении* (Custom Assertion) или *методе проверки* (Verification Method). Хотя эти меры не отменяют необходимости модифицировать код теста при изменении предположений, объем изменяемого кода тестов значительно уменьшается.

Причина: чувствительность к данным (Data Sensitivity)

Чувствительность к данным (Data Sensitivity) возникает, когда тест завершается неудачно при модификации данных, которые использовались тестируемой системой. Чаще всего такая чувствительность проявляется при изменении содержимого тестовой базы данных.

Симптомы

Раньше нормально работавший тест вдруг завершается неудачно при следующих обстоятельствах:

- в базу данных с предтестовым состоянием тестируемой системы были добавлены новые данные;
- записи в базе данных изменены или удалены;
- модифицирован код создания стандартной *тестовой конфигурации* (Standard Fixture, с. 338);

- *общая тестовая конфигурация* (Shared Fixture) модифицирована еще до ее использования первым тестом.

Во всех подобных случаях, скорее всего, используется *стандартная тестовая конфигурация* (Standard Fixture), работающая как *новая тестовая конфигурация* (Fresh Fixture, с. 344), или *общая тестовая конфигурация* (Shared Fixture) в виде *предварительно созданной тестовой конфигурации* (Prebuilt Fixture).

Основная причина

Тест может завершаться неудачно, так как логика проверки результата ищет уже несуществующие данные или применяет критерий поиска, включающий в себя новые данные. Еще одной потенциальной причиной является вызов тестируемой системы с параметрами, которые ссылаются на отсутствующие или модифицированные данные. В результате поведение системы меняется.

В любом случае тест делает предположение о существовании данных, и эти предположения оказываются ложными.

Возможное решение

Когда неудачное завершение происходит во время вызова тестируемой системы, необходимо рассмотреть предварительные условия вызываемой логики и убедиться, что она не затронута недавними изменениями в базе данных.

В большинстве случаев неудачное завершение происходит на этапе проверки результатов. Необходимо проверить логику проверки результата и убедиться, что не делаются предположения о существовании данных. Если такие предположения все-таки присутствуют, логику проверки можно модифицировать.

ЗАЧЕМ НАМ СТО КЛИЕНТОВ?

Коллега автора участвовала в одном проекте как аналитик. Однажды руководитель проекта обратился к ней с вопросом: "Почему в тестовой базе данных были созданы сто уникальных клиентов?"

Как системный аналитик коллега отвечала за помощь бизнес-аналитикам в определении требований к приемочным тестам большого и сложного проекта. Тесты планировалось автоматизировать, но для этого нужно было преодолеть препятствия. Одним из самых сложных препятствий было получение большей части данных от родительской системы — система была слишком сложна, чтобы генерировать эти данные автоматически.

Системный аналитик придумала способ генерации кода XML на основе тестов, захваченных в виде электронной таблицы. Для создания тестовой конфигурации код XML преобразовывался в сценарии QaRun (инструментарий *тестов на основе записи и воспроизведения*, Record and Playback Test), которые загружали данные в родительскую систему через пользовательский интерфейс. Поскольку работа сценариев и доставка данных в тестируемую систему требовали некоторого времени, системный аналитик запускала тесты заранее. Это значит, что стратегия на основе *новой тестовой конфигурации* (Fresh Fixture, с. 344) была недоступна. Лучшим выходом было использование *предварительно созданной тестовой конфигурации* (Prebuilt Fixture, с. 454). Пытаясь избежать появления *взаимодействующих тестов* (Interacting Tests), основанных на общей тестовой конфигурации (Shared Fixture), аналитик реализовала "песочницу" с базой данных (Database Sandbox, с. 658) с помощью *схемы разбиения базы данных* (Database

Partitioning Scheme) на основе уникального номера клиента для каждого теста. Таким образом, побочные эффекты любого из тестов не влияли на остальные тесты.

Учитывая, что требовалась автоматизация около ста тестов, аналитику потребовалось сто тестовых клиентов в базе данных. Именно это она и объяснила своему руководителю.

Неудачное завершение из-за логики проверки результата может происходить, даже если параметры тестируемой системы ссылаются на несуществующие или модифицированные данные. В таком случае придется рассмотреть состояние тестируемой системы “после теста” (и его отличия от ожидаемого состояния) с последующим поиском причин несовпадения. В результате будут обнаружены несовпадения между параметрами тестируемой системы и данными, которые существовали до начала работы теста.

Лучшим решением для того, чтобы избавиться от *чувствительности к данным* (Data Sensitivity), является обеспечение независимости тестов от текущего содержимого базы данных, т.е. использование *новой тестовой конфигурации* (Fresh Fixture). Если это невозможно, можно попытаться применить *схему разбиения базы данных* (Database Partitioning Scheme; см. “Песочница” с базой данных, Database Sandbox, с. 658) для разграничения данных, модифицируемых разными тестами (см. приведенную выше врезку “Зачем нам сто клиентов?”).

Еще одним решением является проверка правильности внесенных изменений. *Дельта-утверждение* (Delta Assertion, с. 505) сравнивает “моментальные снимки” до и после вызова тестируемой системы, игнорируя не изменившиеся данные. В результате можно не записывать знания о целой тестовой конфигурации в код проверки результатов.

Причина: чувствительность к контексту (Context Sensitivity)

Чувствительность к контексту (Context Sensitivity) проявляется, когда тест завершается неудачно из-за изменения состояния или поведения контекста тестируемой системы.

Симптомы

Нормально работавший тест по непонятным причинам завершается неудачно. В отличие от *нестабильного теста* (Erratic Test, с. 267) результаты теста повторяются при неоднократном запуске на протяжении короткого времени. Тест отличает то, что он завершается неудачно независимо от способа запуска.

Основная причина

Тест может завершаться неудачно по двум причинам:

- проверяемая функциональность каким-то образом зависит от времени или даты;
- изменилось поведение другого кода или систем (системы), от которых зависит тестируемая система.

Основным источником *чувствительности к контексту* (Context Sensitivity) является неоднозначное мнение о проверяемой системе. Помните, что тестируемой системой называется фрагмент программного обеспечения, который должен быть проверен. При модульном тестировании это очень маленькая часть всей системы или приложения. Неудачная попытка изоляции конкретного модуля (например, класса или метода) приводит

к чувствительности к контексту (Context Sensitivity), так как тестируется слишком большой компонент программного обеспечения. Опосредованный ввод, который должен контролироваться тестом, меняется случайным образом. Если кто-то модифицирует вызываемый компонент, тест будет завершаться неудачно.

Для защиты от чувствительности к контексту (Context Sensitivity) необходимо следить за изменениями и причинами изменений опосредованного ввода тестируемой системы. Если система содержит логику, зависящую от даты или времени, следует обратить пристальное внимание на такую логику. Возможно, причиной неудачного завершения является зависимость от количества дней в месяце или от других подобных факторов.

Если тестируемая система зависит от данных, передаваемых другой системой, необходимо рассмотреть эти данные и поискать недавние изменения. Журналы взаимодействия с другими системами имеет смысл сравнить с текущим сценарием, приводящим к неудачному завершению.

Если проблема появляется и исчезает, необходимо выявить закономерность. Дополнительная информация о возможных причинах чувствительности к контексту (Context Sensitivity) приводится в описании *нестабильных тестов* (Erratic Test, с. 267).

Возможное решение

Для получения полностью определенных тестов необходимо контролировать все входные параметры тестируемой системы. Если существует зависимость от данных другой системы, их также необходимо контролировать с помощью *тестовой заглушки* (Test Stub, с. 544), настраиваемой и устанавливаемой самим тестом. Если система содержит логику, зависящую от даты или времени, системные часы также должны контролироваться тестом. В этом случае может потребоваться замена системных часов *виртуальными часами* (Virtual Clock) [VCTR], позволяющими тестам устанавливать начальное время или дату, а также моделировать течение времени.

Причина: зарегулированная программа (Overspecified Software)

В этом случае тест содержит слишком подробное описание структуры или поведения программного обеспечения. Такая форма чувствительности к поведению (Behavior Sensitivity; см. “Хрупкий” тест, Fragile Test, с. 277) обычно характерна для стиля тестирования, который называется *проверкой поведения* (Behavior Verification, с. 489). Для такого стиля характерно широкое применение *подставных объектов* (Mock Object, с. 558) для создания пересекающихся уровней тестов. Основная проблема заключается в том, что тест описывает, как программа должна что-то делать, а не какой результат она должна выдавать, т.е. успешная работа теста возможна только для определенным образом реализованного программного обеспечения. Избежать этой проблемы можно, применив принцип *сначала используйте главный вход* (Use the Front Door First, с. 94). Он должен применяться везде, где только возможно, чтобы избежать размещения знаний о реализации тестируемой системы внутри тестов.

Также известен как:

Слишком связанный тест
(Overcoupled Test)

Причина: чувствительное сравнение (Sensitive Equality)

Проверяемые объекты преобразовываются в строки и сравниваются с ожидаемой строкой. Это пример чувствительности к поведению (Behavior Sensitivity), когда тест зависит от стороннего поведения. Также этот случай можно классифицировать как чувстви-

тельность к интерфейсу (Interface Sensitivity), связанную с изменением семантики интерфейса. В любом случае проблема возникает из-за конкретной реализации теста. Использование строкового представления объекта для сравнения с ожидаемым результатом рано или поздно просто привести к проблемам.

Причина: “хрупкая” тестовая конфигурация (Fragile Fixture)

При модификации *стандартной тестовой конфигурации* (Standard Fixture) в соответствии с требованиями нового теста несколько других тестов завершаются неудачно. Это вариант *чувствительности к данным* (Data Sensitivity) или *чувствительности к контексту* (Context Sensitivity) в зависимости от природы модифицируемой тестовой конфигурации.

Источники дополнительной информации

Чувствительное сравнение (Sensitive Equality) и “хрупкая” тестовая конфигурация (Fragile Fixture) впервые были описаны в статье [RTC], которая является первым формальным описанием запахов тестов и рефакторинга кода тестов. Впервые четыре чувствительности были описаны в статье [ARTRP], в которой также шла речь о том, как избежать появления “хрупких” тестов (Fragile Test) при использовании *записанных тестов* (Recorded Test).

Частая отладка (Frequent Debugging)

Для определения причин неудачного завершения большинства тестов требуется отладка вручную.

Также известен как:

*Отладка вручную
(Manual Debugging)*

Симптомы

Запуск теста приводит к ошибке или неудачному завершению. Вывод *программы запуска тестов* (Test Runner, с. 405) оказывается недостаточным для определения источника проблемы. В результате приходится использовать интерактивный отладчик (или вставлять отладочную печать в разных местах кода) и искать причину ошибки или отказа.

Если такая ситуация является исключением, опасаться нечего. Если большая часть неудачных завершений тестов требует такой отладки, ситуацию можно классифицировать как *частая отладка* (Frequent Debugging).

Причины

Частая отладка (Frequent Debugging) является результатом недостаточной *локализации дефектов* (Defect Localization, с. 78) в наборе автоматизированных тестов. Неудачно завершившиеся тесты должны указывать причину отказа либо через сообщения (см. *Сообщение для утверждения*, Assertion Message, с. 398), либо через закономерность отказов. Если этого не происходит, то:

- отсутствуют подробные модульные тесты, указывающие на логические ошибки в отдельных классах;
- отсутствуют тесты компонентов для кластеров классов (т.е. компонентов), которые указывали бы на ошибки интеграции между отдельными классами. (Такая ситуация возможна на фоне широкого применения *подставных объектов* (Mock Object, с. 558) для замены вызываемых объектов, но модульные тесты для вызываемых объектов не соответствуют поведению *подставных объектов* (Mock Object).)

Чаще всего данная проблема возникает при создании высокоуровневых (функциональных и компонентных) тестов, когда модульные тесты написаны не для всех отдельных методов. (Некоторые разработчики называют такой подход основанным на тестах историй, чтобы отличать его от разработки на основе тестов, когда любой фрагмент кода порождается отказавшим тестом.)

Также *частая отладка* (Frequent Debugging) может стать следствием *нечасто запускаемых тестов* (Infrequently Run Tests; см. *Ошибки в продукте*, Production Bugs, с. 303). Если тесты запускаются после внесения даже минимальных изменений, легко вспомнить изменения, внесенные сразу перед запуском тестов. Таким образом, при неудачном завершении теста не приходится тратить время на диагностику и поиск ошибки — разработчик помнит причину ошибки, так как только что ее внес в код!

Влияние

Отладка вручную происходит медленно и сложно. Очень легко пропустить малозаметные признаки ошибки и потратить много часов на поиск единственной логической

ошибки. *Частая отладка* (Frequent Debugging) снижает производительность труда разработчика и делает планирование менее надежным, так как единственный сеанс отладки вручную может продлить сроки разработки на полдня и дольше.

Варианты решений

Если для некоторой функциональности не написан приемочный тест и в ходе тестирования вручную была выявлена проблема, не обнаруженная автоматизированными тестами, скорее всего, наблюдается пример *не тестированного требования* (Untested Requirement; см. *Ошибки в продукте*, Production Bugs). В таком случае разработчик должен задать себе вопрос: “Какой автоматизированный тест позволил бы избежать отладки вручную?” Еще лучше после обнаружения проблемы сразу написать тест, обнаруживающий ее автоматически. После этого неудачно завершающийся тест можно использовать для исправления ошибки. Если проблема кажется распространенной, можно выделить разработчикам отдельную задачу по идентификации и написанию дополнительных тестов, закрывающих обнаруженный разрыв в покрытии.

Истинная разработка на основе тестов является наилучшим способом избежать *частой отладки* (Frequent Debugging). Начинать следует как можно ближе к оболочке приложения и вести разработку на основе тестов историй, т.е. модульные тесты должны создаваться для отдельных классов, а тесты компонентов — для коллекций классов. Все вместе они обеспечат достаточную *локализацию дефектов* (Defect Localization).

Ручное вмешательство (Manual Intervention)

При каждом запуске тест требует от разработчика определенной ручной операции.

Симптомы

От запускающего тест разработчика требуется ручная операция перед запуском теста или во время его работы. В противном случае тест завершится неудачно. Результаты работы теста также приходится проверять вручную.

Влияние

Основным назначением автоматизированных тестов является раннее предоставление обратной связи по проблемам, внесенным в программное обеспечение. Если стоимость получения обратной связи слишком высока (т.е. если требуется *ручное вмешательство*, Manual Intervention), тесты не будут запускаться, а обратная связь не будет предоставляться достаточно часто. Без этой обратной связи между запусками тестов в продукт будет вноситься множество проблем, а результатом будет *частая отладка* (Frequent Debugging, с. 285) и *высокая стоимость обслуживания тестов* (High Test Maintenance Cost, с. 300).

Также *ручное вмешательство* (Manual Intervention) исключает полностью автоматизированную интеграцию и регрессионное тестирование.

Причины

Существует множество причин *ручного вмешательства* (Manual Intervention). Ниже приводятся общие категории проблем, требующих вмешательства, но это далеко не полный список.

Причина: ручная настройка тестовой конфигурации (Manual Fixture Setup)

Симптомы

От разработчика требуется создание тестовой среды вручную перед запуском автоматизированных тестов. Эти действия могут включать в себя настройку серверов, запуск серверных процессов или запуск сценариев создания *предварительно созданной тестовой конфигурации* (Prebuilt Fixture, с. 454).

Основная причина

Данная проблема обычно вызывается недостаточным вниманием к автоматизации создания тестовой конфигурации. Кроме того, причиной может стать слишком сильная связь между компонентами тестируемой системы, что мешает тестировать большую часть кода внутри среды разработки.

Возможное решение

Убедитесь, что создаете *полностью автоматизированные тесты* (Fully Automated Tests). Это может потребовать раскрытия предназначенного для тестов программного интерфейса, позволяющего создавать тестовую конфигурацию. Если проблема связана с невозможностью запуска программного обеспечения в среде разработки, может потребоваться рефакторинг для разделения тестируемой системы и ручных операций.

Причина: ручная проверка результата (Manual Result Verification)

Симптомы

Тесты запускаются, но практически всегда завершаются успешно, даже когда известно, что тестируемая система не возвращает правильный результат.

Основная причина

Если написанные тесты не являются *самопроверяющимися* (Self-Checking Test, с. 81), у разработчика появляется ложное чувство безопасности, так как тесты завершаются неудачно только при появлении ошибки или при генерации исключения.

Возможное решение

Для превращения тестов в самопроверяющиеся можно включить в *тестовый метод* (Test Method, с. 378) логику проверки результата в виде вызовов *методов с утверждением* (Assertion Method, с. 390).

Причина: ручная генерация события (Manual Event Injection)

Симптомы

Разработчику приходится вмешиваться в работу теста и выполнять операцию вручную для продолжения тестирования.

Основная причина

Многие события тестируемой системы сложно сгенерировать программно. В качестве примера такого события можно назвать отключение сетевого кабеля, отключение базы данных и щелчок на кнопке пользовательского интерфейса.

Влияние

Если разработчику приходится что-то делать вручную, одновременно увеличиваются трудозатраты на запуск тестов и становится невозможным запуск тестов без присмотра. В результате полностью автоматизированный процесс компиляции и тестирования оказывается невозможным.

Возможное решение

Наилучшим решением является поиск способов тестирования, не требующих от разработчика выполнения операций вручную. Если события доставляются тестируемой системой через асинхронный интерфейс, *тестовый метод* (Test Method) может вызывать систему непосредственно и передавать смоделированный объект события. Если тестируемая система получает информацию в виде синхронного вызова со стороны другой подсистемы, для управления опосредованным вводом можно заменить часть тестируемой системы *тестовой заглушкой* (Test Stub, с. 544). Заглушка будет моделировать ситуацию, в которой должна находиться тестируемая система.

Источники дополнительной информации

Дополнительная информация о перехвате управления опосредованным вводом приводится в главе 11, “Использование тестовых двойников”.

Медленные тесты (Slow Tests)

Существуют тесты, которые работают слишком медленно.

Симптомы

Тесты работают достаточно медленно, чтобы разработчики не запускали их при каждой модификации тестируемой системы. Вместо этого разработчики ожидают следующего перерыва в работе и только тогда запускают тесты. Еще одним симптомом является общение с другими разработчиками (запуск одной из игр, просмотр информации в Интернете) во время работы тестов.

Влияние

С *медленным тестом* (Slow Test) связаны конкретные накладные расходы: он снижает производительность труда разработчика, запускающего набор тестов. Если разработка кода продукта ведется на основе тестов, при каждом запуске теста теряются драгоценные секунды. В момент запуска всех тестов перед включением изменений в общее хранилище ожидание оказывается еще более долгим.

Медленные тесты (Slow Test) навязывают и опосредованные накладные расходы.

- Узкое место, возникающее из-за более длительного удерживания “маркера интеграции”, так как приходится ожидать завершения всех тестов перед интеграцией внесенных изменений.
- Время, в течение которого разработчик отвлекает своих коллег, ожидая завершения собственных тестов.
- Время, потраченное на диагностику проблемы, появившейся с момента последнего запуска тестов. Чем больше времени прошло, тем меньше вероятность вспомнить, какое же изменение привело к неудачному завершению теста. Это может привести к разрыву быстрой обратной связи, обеспечиваемой автоматизированными модульными тестами.

Распространенной реакцией на появление *медленных тестов* (Slow Test) является немедленный переход к использованию *общей тестовой конфигурации* (Shared Fixture, с. 350). К сожалению, такой подход практически всегда приводит к появлению дополнительных проблем, включая *нестабильный тест* (Erratic Test, с. 267). Лучше использовать *поддельный объект* (Fake Object, с. 565) для замены медленного компонента (например, базы данных) более быстрым вариантам. Но если все другие варианты не подходят и приходится использовать *общую тестовую конфигурацию* (Shared Fixture), необходимо приложить все усилия, чтобы сделать ее неизменяемой.

Совет по диагностике

Медленные тесты (Slow Test) появляются в результате неправильного выбора способов создания и использования тестируемой системы или в результате неправильного проектирования тестов. Иногда проблема очевидна — можно спокойно сидеть и наблюдать за ростом количества успешно завершившихся тестов. Во время выполнения могут

возникать заметные паузы; в *тестовых методах* (Test Method, с. 378) могут присутствовать явные задержки. Но если причина не очевидна, можно запускать различные подмножества (или поднаборы) тестов для определения, какие из них работают быстро, а какие медленно.

Для поиска медленных мест в тестах может потребоваться инструментарий для профилирования. Конечно, инфраструктура xUnit обеспечивает базовые функции для создания собственного мини-профайлера: можно модифицировать методы `setUp` и `tearDown` *суперкласса теста* (Testcase Superclass, с. 646). При этом можно фиксировать время начала и/или завершения, имя *класса теста* (Testcase Class) и имя *тестового метода* (Test Method) в файле журнала. После завершения тестов можно импортировать этот файл в электронную таблицу, отсортировать тесты по времени работы и определить виновников задержки. Основное внимание необходимо уделить тестам с самым продолжительным временем выполнения.

Причины

Конкретная причина медленной работы теста может быть связана со способом создания тестируемой системы или с кодом самого теста. Иногда способ создания системы диктует написание медленных тестов. Такая ситуация характерна для унаследованного кода или кода, написанного до тестов.

Причина: использование медленного компонента (Slow Component Usage)

Компонент тестируемой системы генерирует слишком большие задержки.

Основная причина

Самой распространенной причиной медленной работы тестов является взаимодействие с базой данных. Если тестам приходится выполнять запись в базу данных для создания тестовой конфигурации и считывание из нее — для проверки результата (вариант *манипуляции через “черный ход”*, Back Door Manipulation, с. 359), они работают в 50 раз медленнее, чем такие же тесты, работающие со структурами данных, полностью хранящимися в памяти. Это пример более общей проблемы использования медленных компонентов.

Возможное решение

Выполнение тестов можно значительно ускорить, если заменить медленные компоненты *тестовым двойником* (Test Double, с. 538), обеспечивающим практически мгновенный ответ. Если медленным компонентом является база данных, использование *поддельной базы данных* (Fake Database) может ускорить работу тестов в 50 раз! Дополнительная информация о решении этой проблемы приводится во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351.

Причина: тестовая конфигурация общего характера (General Fixture)

Симптомы

Тесты работают медленно, поскольку каждый из них создает одну и ту же слишком большую тестовую конфигурацию.

Основная причина

Каждый тест создает большую *тестовую конфигурацию общего характера* (General Fixture) при использовании стратегии на основе *новой тестовой конфигурации* (Fresh Fixture, с. 344). Так как *тестовая конфигурация общего характера* (General Fixture) содержит намного больше объектов, чем *минимальная тестовая конфигурация* (Minimal Fixture, с. 336), возрастает продолжительность ее генерации. Учитывая, что новый экземпляр конфигурации создается для каждого *объекта теста* (Testcase Object, с. 410), умножьте “взрастает продолжительность” на количество тестов — и получите порядок замедления работы!

Возможное решение

Проще всего превратить *тестовую конфигурацию общего характера* (General Fixture) в *общую тестовую конфигурацию* (Shared Fixture) и избежать ее повторной генерации для каждого теста. Но если такая конфигурация не будет неизменяемой, это решение приведет к появлению *нестабильных тестов* (Erratic Test). Данного решения стоит избегать. Лучшим решением является сокращение объема операций, выполняемых каждым тестом для создания конфигурации.

Причина: асинхронный тест (Asynchronous Test)

Симптомы

Несколько тестов требуют слишком много времени для завершения работы; в коде тестов присутствуют явные задержки.

Основная причина

Вставленные в *тестовые методы* (Test Method) задержки значительно замедляют выполнение тестов. Необходимость медленного выполнения может быть обусловлена тестированием порожденных потоков или процессов (*асинхронный код*, Asynchronous Code; см. *Сложный в тестировании код*, Hard-to-Test Code, с. 251), когда тест должен ожидать их запуска, выполнения и завершения для проверки ожидаемых побочных эффектов. Из-за неопределенности задержек при запуске потоков или процессов тестам приходится выдерживать длинную паузу “на всякий случай”, т.е. для обеспечения гарантированного успешного завершения. Ниже приведен пример теста с задержками.

```
public class RequestHandlerThreadTest extends TestCase {
    private static final int TWO_SECONDS = 3000;
    public void testWasInitialized_Async() throws InterruptedException {
        // Настройка
        RequestHandlerThread sut = new RequestHandlerThread();
        // Вызов
        sut.start();
        // Проверка
        Thread.sleep(TWO_SECONDS);
        assertTrue(sut.initializedSuccessfully());
    }
    public void testHandleOneRequest_Async()
        throws InterruptedException {
        // Настройка
        RequestHandlerThread sut = new RequestHandlerThread();
        sut.start();
    }
}
```

```

    // Вызов
    enqueueRequest(makeSimpleRequest());
    // Проверка
    Thread.sleep(TWO_SECONDS);
    assertEquals(1, sut.getNumberOfRequestsCompleted());
    assertEquals(makeSimpleResponse(), getResponse());
}
}

```

Влияние

Двухсекундная задержка может показаться незначительной, но представьте, что будет с двумя десятками таких тестов — на их выполнение потребуется около полу-минуты. Сравните это с несколькими сотнями нормальных тестов, выполняющимися за одну секунду.

Возможное решение

Наилучшим способом решения этой проблемы является полный отказ от асинхронности в тестах. Это может потребовать применения рефакторинга *выделение тестируемого компонента* (Extract Testable Component, с. 737) для реализации *минимального выполняемого файла* (Humble Executable; см. *Минимальный объект*, Humble Object, с. 700).

Причина: слишком много тестов (Too Many Tests)

Симптомы

В системе присутствует так много тестов, что их завершение требует слишком много времени вне зависимости от скорости выполнения.

Основная причина

Очевидной причиной этой проблемы является большое количество тестов. Возможно, система достаточно велика, чтобы оправдать такой объем тестов, а возможно, тесты часто пересекаются.

Менее очевидной причиной является слишком частый запуск слишком большого количества тестов!

Возможное решение

Не обязательно каждый раз запускать все тесты! Главное, чтобы все тесты запускались регулярно. Если запуск всего набора требует слишком много времени, создайте *набор с подходящим подмножеством* (Subset Suite; см. *Именованный набор тестов*, Named Test Suite, с. 604) и запускайте его перед каждым включением изменений в общее хранилище кода. Остальные тесты можно запускать регулярно, но не так часто. Можно запланировать их запуск ночью или в любое другое удобное время. Некоторые разработчики называют такую технику “конвейером компиляции”. Дополнительная информация об этой и других идеях приводится во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351.

Если система слишком велика, имеет смысл разделить ее на несколько относительно независимых подсистем или компонентов. В результате команды разработчиков смогут работать над каждым компонентом независимо и запускать только свои

тесты. Одни из таких тестов будут выступать в роли эмуляторов поведения других подсистем; их придется постоянно обновлять в соответствии с модификациями межкомпонентного интерфейса. Это будет реализация принципа *тесты как документация* (Tests as Documentation, с. 79). При этом необходимы глобальные тесты, которые будут проверять все компоненты сразу, но запускать их перед внесением изменений в общее хранилище необязательно.

Глава 17

Запахи проектов

Запахи в этой главе:

Тест с ошибками (Buggy Test)	296
Разработчики не пишут тесты (Developers Not Writing Tests).....	298
Высокая стоимость обслуживания тестов (High Test Maintenance Cost)	300
Ошибки в продукте (Production Bugs)	303

Тест с ошибками (Buggy Test)

В автоматизированных тестах всегда можно обнаружить ошибки.

Полностью автоматизированные тесты (Fully Automated Tests, с. 81) должны выступать в роли “страховочной сети” при итеративной разработке. Но как убедиться, что страховка надежна?

Тест с ошибками (Buggy Test) указывает на уровне проекта, что не все хорошо с автоматизированными тестами.

Симптомы

Компиляция завершается неудачно, и причиной этому является неудачное завершение теста. Тщательный анализ показывает, что тестируемый код работает правильно, а ошибка содержится в самом teste.

Даже при удачном завершении тестов, проверяющих конкретные сценарии, обнаруживаются *ошибки в продукте* (Production Bugs, с. 303). Анализ причин показывает, что тест содержит ошибку, которая заставляет его пропускать ошибку в продукте.

Влияние

Вводящие в заблуждение тесты очень опасны! Тесты, завершающиеся успешно, когда должен произойти отказ (ложное несрабатывание или “здесь все нормально”), дают ложное ощущение безопасности. Тесты, завершающиеся неудачно, когда все нормально (ложное срабатывание), обесценивают идею тестирования. Они, как мальчик-пастух, кричавший “Волк!”. Несколько ложных тревог — и все игнорируют такие крики.

Причины

Существует множество причин появления *тестов с ошибками* (Buggy Test). Большинство из них проявляются в виде запахов кода или поведения. Руководитель проекта редко замечает эти запахи, если не ищет их намеренно.

Причина: “хрупкий” тест (Fragile Test)

Тест с ошибками (Buggy Test) может быть симптомом уровня проекта, соответствующим “хрупкому” тесту (Fragile Test, с. 277). В случае ложных срабатываний диагностику следует начинать с “четырех чувствительностей”: *чувствительность к интерфейсу* (Interface Sensitivity), *чувствительность к поведению* (Behavior Sensitivity), *чувствительность к данным* (Data Sensitivity) и *чувствительность к контексту* (Context Sensitivity). Каждая из них может стать причиной неудачного завершения теста после внесения изменений. Устранение чувствительности с помощью рефакторинга и применения *тестовых двойников* (Test Double, с. 538) может оказаться сложным процессом, но сделает тесты более надежными и эффективными с точки зрения трудозатрат.

Причина: непонятный тест (Obscure Test)

Частой причиной ложных несрабатываний (успешных завершений теста в ситуации, когда должен быть отказ) является *непонятный тест* (Obscure Test, с. 230). С таким тестом сложно работать — особенно при модификации существующих тестов, работа которых нарушена из-за внесения изменений в код продукта. Так как автоматизированные тесты плохо поддаются тестированию, проверка правильности тестов происходит недостаточно часто. Пока все тесты завершаются успешно, разработчик считает, что все хорошо. На самом деле некоторые тесты просто не могут завершиться неудачно.

Лучше всего подвергнуть *непонятные тесты* (Obscure Test) рефакторингу, ориентированному на читателей теста. За цель необходимо принять *тесты как документацию* (Tests as Documentation, с. 79) — цели меньшего масштаба только повышают вероятность появления *тестов с ошибками* (Buggy Test).

Причина: сложный в тестировании код (Hard-to-Test Code)

Еще одной распространенной причиной появления *тестов с ошибками* (Buggy Test), особенно при работе с “унаследованными программными продуктами” (любым программным обеспечением без полного набора автоматизированных тестов), является дизайн, несовместимый с автоматизированным тестированием. Такой *сложный в тестировании код* (Hard-to-Test Code, с. 251) может заставить использовать *опосредованное тестирование* (Indirect Testing; см. *Непонятный тест*, Obscure Test), что, в свою очередь, приводит к появлению “хрупких” тестов (Fragile Test).

Единственным способом облегчить тестирование такого кода является рефакторинг. (Более подробно такое преобразование рассматривалось в главе 6, “Стратегия автоматизации тестирования”, и в главе 11, “Использование тестовых двойников”.) Если рефакторинг невозможен, влияние изменений на код тестов можно снизить за счет использования инкапсуляции программного интерфейса тестируемой системы (SUT API Encapsulation; см. *Вспомогательный метод теста*, Test Utility Method, с. 610).

Совет по диагностике

При обнаружении *тестов с ошибками* (Buggy Test) важно задать множество вопросов. Пять “Почему?” [TPS] позволят дойти до самого корня проблемы, т.е. определить, какие запахи кода и/или поведения приводят к появлению ошибок в тестах, и выявить основную причину каждого запаха.

Варианты решений

Решение зависит от причин появления *теста с ошибками* (Buggy Test). Основные решения приводились при описании соответствующих запахов поведения и кода.

Как всегда в случае “запахов проекта”, необходимо искать причины на уровне проекта. Среди причин можно назвать недостаток времени на решение следующих задач.

- Изучение правильных подходов к написанию тестов
- Рефакторинг унаследованного кода для упрощения автоматизации тестирования
- Написание тестов перед написанием кода

Неспособность устраниТЬ эти причины на уровне проекта приводит к повторному возникновению соответствующих проблем в ближайшем будущем.

Разработчики не пишут тесты (Developers Not Writing Tests)

Разработчики не пишут автоматизированные тесты.

Симптомы

Пошел слух, что разработчики не пишут тесты. Или, обнаружив *ошибки в продукте* (Production Bugs, с. 303) и задавшись вопросом: “Почему так много ошибок проходит мимо тестов?”, мы получаем ответ: “Потому что для этой части продукта тесты не пишутся”.

Влияние

Если разработчики не пишут автоматизированные тесты для всех “потенциально нерабочающих” компонентов программного продукта, они обрекают себя на мучения в будущем. Текущие темпы разработки программного обеспечения не будут выдержаны на протяжении длительного времени, так как в системе сформируется **задолженность по тестам** (test debt). Добавление новой функциональности будет отнимать все больше и больше времени, а рефакторинг кода для улучшения дизайна будет опасен, как “танцы с волками” (а значит, будет происходить все реже и реже). Эта проблема является началом “наклонной плоскости”, ведущей к параноидальной и негибкой разработке. Если именно это и требуется, продолжайте в том же духе. В противном случае примите необходимые меры.

Причины

Причина: недостаточно времени (Not Enough Time)

Разработчики могут не успеть написать тесты в то время, которое выделено на разработку в целом. Причиной может быть слишком агрессивное расписание или руководитель группы, инструктирующий разработчиков в стиле “Не тратьте время на эти тесты”. С другой стороны, разработчики могут не иметь достаточной квалификации для быстрого написания тестов, а времени на обучение может просто не быть.

Если разработчикам нужно только время, руководитель проекта должен изменить расписание и это время выделить. Достаточно временного изменения расписания, чтобы позволить разработчикам приобрести необходимые навыки и изучить инфраструктуру автоматизации, позволяющие значительно быстрее писать необходимые тесты. Опыт показывает, что после ознакомления с процессом разработчикам удается писать тесты и код в то время, которое раньше уходило на написание кода и его отладку. Время, потраченное на создание тестов, более чем компенсируется временем, не проведенным за отладчиком.

Причина: сложный в тестировании код (Hard-to-Test Code)

Распространенной причиной отказа со стороны разработчиков писать тесты, особенно при работе с “унаследованными программными продуктами” (любым программным обеспечением без полного набора автоматизированных тестов), является дизайн, который не совместим с автоматизированным тестированием. Более подробно данная ситуация рассматривается при описании запаха *сложный в тестировании код* (Hard-to-Test Code, с. 251).

Причина: неправильная стратегия автоматизации тестов (Wrong Test Automation Strategy)

Причиной отказа писать тесты могут служить среда тестирования и стратегия автоматизации тестов, провоцирующие создание “хрупких” тестов (Fragile Test, с. 277) и непонятных тестов (Obscure Test, с. 230) и требующие слишком много времени на написание тестов. Для поиска главных причин необходимо задать пять “Почему?”. После этого можно устранить обнаруженные причины и восстановить нормальный процесс разработки.

Совет по диагностике

Такие запахи уровня проекта, как *разработчики не пишут тесты* (Developers Not Writing Tests), с большей вероятностью будут обнаружены руководителем проекта или группы, чем отдельными разработчиками. Руководитель не обязательно должен знать, как решить эту проблему, но внимание и осознание проблемы очень важны. Уникальное положение руководителя позволяет узнать у разработчиков причины отказа от написания тестов, а также обстоятельства, при которых тесты все же создаются, и время, которое на это тратится. После этого руководитель может стимулировать разработчиков для устранения всех причин, мешающих написанию необходимых тестов.

Естественно, от руководителя требуется полная поддержка разработчиков во внедрении предложенного плана улучшений. Такая поддержка может проявляться в предоставлении достаточного времени на формирование навыка и создание необходимой инфраструктуры тестирования. При этом руководитель не должен требовать решения всех проблем “уже вчера”. Возможно, стоит установить цели на каждой итерации, например “20%-ное снижение кода, не защищенного тестами” или “20%-ный рост покрытия тестами”. Такие цели должны быть оправданными и находиться на достаточно высоком уровне, чтобы стимулировать правильное поведение, а не просто давать красивые цифры. (Например, цель написать еще 205 тестов может быть достигнута и без увеличения покрытия кода. Достаточно просто разбить тесты на меньшие фрагменты или клонировать существующие тесты.)

Высокая стоимость обслуживания тестов (High Test Maintenance Cost)

На обслуживание существующих тестов тратится слишком много времени.

Код тестов требует такого же обслуживания, как и проверяемый им код продукта. С развитием приложения, скорее всего, придется регулярно пересматривать тесты при каждой модификации тестируемых классов, при каждом добавлении новой функциональности или рефакторинге тестов. *Высокая стоимость обслуживания тестов* (High Test Maintenance Cost) возникает, когда значительно усложняется понимание назначения и структуры существующих тестов.

Симптомы

Разработка новой функциональности замедляется. При каждом добавлении новых функций приходится вносить значительные изменения в существующие тесты. Разработчики могут потребовать у руководителя проекта “итерацию на рефакторинг и зачистку тестов”.

Если отслеживать время, затраченное на создание и модификацию существующих тестов, отдельно от времени, затраченного на создание самого кода, можно заметить, что большая его часть уходит на модификацию существующих тестов.

Большинство проблем в обслуживании тестов сопровождается другими запахами, например приведенными ниже.

- “Хрупкий” тест (Fragile Test, с. 277) указывает на слишком тесную связь с тестируемой системой.
- “Хрупкая” тестовая конфигурация (Fragile Fixture) указывает на зависимость слишком большого количества тестов от одной тестовой конфигурации (*стандартной тестовой конфигурации*, Standard Fixture, с. 338), что приводит к появлению *высокой стоимости обслуживания тестов* (High Test Maintenance Cost).
- *Нестабильный тест* (Erratic Test, с. 267) указывает, что причиной проблемы является использование *общей тестовой конфигурации* (Shared Fixture, с. 350).

Влияние

Производительность труда разработчиков значительно снижается, так как тесты требуют дополнительного времени на обслуживание. Со временем может появиться соблазн “вырезать” проблемные тесты из набора. В то время как код продукта писать обязательно, обслуживание тестов является полностью добровольной деятельностью (с точки зрения неискушенного наблюдателя). Если не решить эту проблему, все усилия по автоматизации тестов могут сойти на нет, когда разработчики или их руководители решат, что автоматизация тестов “просто не работает”, и откажутся от тестов вообще.

Причины

Основной причиной появления *высокой стоимости обслуживания тестов* (High Test Maintenance Cost) является недостаточное внимание к принципам, перечисленным в главе 5, “Принципы автоматизации тестирования”. Более непосредственными причинами

ми часто оказываются *дублирование тестового кода* (Test Code Duplication, с. 254) и тесты, которые слишком тесно связаны с программным интерфейсом системы.

Причина: “хрупкий” тест (Fragile Test)

Тесты, завершающиеся неудачно из-за внесения минимальных изменений в тестируемую систему, называются “хрупкими” (Fragile Test). Они приводят к росту стоимости обслуживания тестов (High Test Maintenance Cost), так как требуют постоянного внимания после любого изменения, которое вроде бы и не должно влиять на работу теста.

Основной причиной таких отказов может быть любая из “четырех чувствительностей”: *чувствительность к интерфейсу* (Interface Sensitivity), *чувствительность к поведению* (Behavior Sensitivity), *чувствительность к данным* (Data Sensitivity) и *чувствительность к контексту* (Context Sensitivity). Для сокращения стоимости обслуживания можно снизить чувствительность тестов за счет применения *тестовых двойников* (Test Double, с. 538) и рефакторинга системы с разбиением на меньшие компоненты и классы, тестируемые отдельно.

Причина: непонятный тест (Obscure Test)

Непонятные тесты (Obscure Test, с. 230) делают основной вклад в *высокую стоимость обслуживания тестов* (High Test Maintenance Cost), так как требуют больше времени на понимание при каждом просмотре. Если тест приходится модифицировать, на это требуется больше усилий и редко когда тест начинает работать “сразу же”. Это означает дополнительную отладку тестов. *Непонятные тесты* (Obscure Test) с большей вероятностью не перехватывают ошибки, для обнаружения которых они предназначены. Результатом становится появление *тестов с ошибками* (Buggy Test, с. 296).

Лучше всего подвергнуть *непонятные тесты* (Obscure Test) рефакторингу, ориентированному на читателей теста. За цель необходимо принять *тесты как документация* (Tests as Documentation, с. 79) — цели меньшего масштаба только повышают вероятность появления *тестов с ошибками* (Buggy Test).

Причина: сложный в тестировании код (Hard-to-Test Code)

“Унаследованные программные продукты” (любое программное обеспечение без полного набора автоматизированных тестов) могут оказаться сложными для тестирования, так как обычно тесты пишутся после кода. Если продукт имеет дизайн, который не совместим с автоматизированным тестированием, может возникнуть соблазн использовать *опосредованное тестирование* (Indirect Testing; см. *Непонятный тест*, Obscure Test) через неуказанные интерфейсы, что только усложнит систему. В результате такого подхода появятся “хрупкие” тесты (Fragile Test).

Придется потратить время и усилия для рефакторинга кода, чтобы сделать его более удобным для тестирования. Эти усилия не окажутся напрасными, если позволят снизить стоимость обслуживания тестов. Если рефакторинг невозможен, можно попытаться снизить количество затрагиваемых изменениями тестов через *инкапсуляцию программного интерфейса тестируемой системы* (SUT API Encapsulation; см. *Вспомогательный метод теста*, Test Utility Method, с. 610). Например, *метод создания* (Creation Method, с. 441) скрывает конструкторы, делая тест менее восприимчивым к изменениям в сигнатурах и семантике конструкторов.

Совет по диагностике

Как запах уровня проекта *высокая стоимость обслуживания тестов* (High Test Maintenance Cost) с большей вероятностью будет обнаружена руководителем проекта или группы, чем отдельными разработчиками. Руководитель не обязательно должен знать, как решить проблему, но осознать проблему очень важно. В результате руководитель может узнать у разработчиков, сколько времени тратится на обслуживание тестов, как часто это происходит и зачем это необходимо. Затем руководитель может поставить перед разработчиками задачу найти наилучшее решение, не приводящее к такой высокой стоимости обслуживания.

Конечно, разработчикам потребуется поддержка руководителя для реализации задуманного. Такая поддержка подразумевает выделение времени на расследование причин, на обучение и на фактическую реализацию решения. Руководитель может выделить время, оформив “рефакторинг тестов” как задачу, снизив скорость добавления новой функциональности или воспользовавшись другими способами. Вне зависимости от подходов к поиску дополнительного времени руководитель должен осознавать, что, если не выделить разработчикам необходимых ресурсов на решение проблемы сейчас, проблема станет еще серьезнее и потребует больших усилий для решения в будущем, когда количество тестов удвоится.

Ошибки в продукте (Production Bugs)

В процессе формального тестирования и в процессе эксплуатации в продукте обнаруживается слишком много ошибок.

Симптомы

К написанию автоматизированных тестов прилагались значительные усилия, но количество ошибок во время формального тестирования и во время эксплуатации продукта остается слишком большим.

Влияние

Диагностика и исправление ошибок, обнаруженных на этапе формального тестирования, требуют больше времени. Еще больше времени требуется на исправление ошибок, обнаруженных во время эксплуатации. Исправление ошибок и повторное тестирование может заставить отложить передачу продукта заказчику или начало промышленной эксплуатации. В такой ситуации время и усилия непосредственно отражаются на финансовых расходах и требуют ресурсов, которые можно было бы потратить на добавление новой функциональности или создание новых продуктов. Кроме того, задержка может негативно сказаться на репутации исполнителей. С низким качеством связаны и опосредованные потери, выражющиеся в снижении стоимости предоставляемого продукта или услуги.

Причины

Ошибки могут проскальзывать в готовый продукт по нескольким причинам, включая *нечасто запускаемые тесты* (Infrequently Run Tests) и *нетестированный код* (Untested Code). Вторая причина может быть результатом *отсутствующих модульных тестов* (Missing Unit Test) или *потерянных тестов* (Lost Test).

Указывая, что должно запускаться “достаточное количество” тестов, мы предполагаем адекватное покрытие тестами, а не просто некоторое количество тестов. Изменения в *нетестированном коде* (Untested Code) с большой вероятностью приводят к появлению ошибок в продукте (Production Bugs), так как не существует автоматизированных тестов, которые просигнализировали бы о появившейся проблеме. *Нетестированные требования* (Untested Requirement) не проверяются при каждом запуске теста, поэтому сказать определенно, что действительно работает, нельзя. Обе проблемы связаны с проблемой, когда *разработчики не пишут тесты* (Developers Not Writing Tests, с. 298).

Причина: нечасто запускаемые тесты (Infrequently Run Tests)

Симптомы

Руководитель проекта получает информацию о том, что разработчики запускают тесты недостаточно часто. Несколько вопросов проясняют ситуацию и показывают, что тесты работают слишком долго (*медленный тест*, Slow Tests, с. 289) или выводят слишком много ложных сообщений об ошибках (*тест с ошибками*, Buggy Test, с. 296).

В журнале интеграции присутствуют сообщения о неудачном завершении тестов. Исследование показывает, что разработчики часто включают в общее хранилище код, не запуская тесты на собственных компьютерах.

Основная причина

После знакомства с преимуществами страховой сети на основе автоматизированных тестов большинство разработчиков будут использовать тесты, пока что-то этому не помешает. Самыми распространенными помехами являются замедление работы тестов (*Slow Test*), а значит, и замедление регрессионного тестирования перед включением кода в хранилище, и *неповторяемые тесты* (*Unrepeatable Test*), заставляющие разработчиков перезапускать тестовую среду или вмешиваться в процесс (*Manual Intervention*, с. 287) до или во время работы тестов.

Возможное решение

Если основной причиной являются *неповторяемые тесты* (*Unrepeatable Test*), можно перейти к использованию стратегии на основе *новой тестовой конфигурации* (*Fresh Fixture*, с. 344), что сделает тесты более детерминированными. Если причиной являются *медленные тесты* (*Slow Test*), то необходимо направить дополнительные усилия на ускорение работы тестов.

Причина: потерянный тест (Lost Test)

Симптомы

Количество запускаемых в наборе тестов уменьшилось (или не увеличилось до прогнозируемого уровня). Это можно заметить, непосредственно просматривая счетчики тестов. С другой стороны, можно столкнуться с ошибкой, которая должна быть перехвачена заведомо существующим тестом, но разбирательство показывает, что данный тест был отключен.

Основная причина

Потерянный тест (*Lost Test*) может стать следствием отключения *тестового метода* (*Test Method*, с. 378) или *класса теста* (*Testcase Class*, с. 401). Иногда они просто не включаются в *набор всех тестов* (*AllTests Suite*; см. *Именованный набор тестов*, *Named Test Suite*, с. 604).

Случайное “выпадение” тестов из набора (а значит, и невозможность их нормального запуска) может происходить в следующих ситуациях.

- *Тестовому методу* (*Test Method*) не был назначен атрибут `[test]` или случайно использовано имя метода, не соответствующее соглашению об именовании механизма *обнаружения тестов* (*Test Discovery*, с. 420).
- При автоматизации тестов в инфраструктуре с поддержкой *перечисления тестов* (*Test Enumeration*, с. 425) был забыт вызов метода `suite.addTest`, который добавляет *тестовый метод* (*Test Method*) в *объект набора тестов* (*Test Suite Object*, с. 414).
- В процедурной реализации xUnit в *процедуру набора тестов* (*Test Suite Procedure*) забыли включить явный вызов *тестовый метод* (*Test Method*).
- Набор тестов не был добавлен в *набор наборов* (*Suite of Suites*) или *классу теста* (*Testcase Class*) не был назначен атрибут `[Test Fixture]`.

Работавшие ранее тесты могут быть отключены одним из перечисленных ниже способов.

- *Тестовый метод* (Test Method) был переименован, и новое имя не соответствует шаблону *обнаружения тестов* (Test Discovery) для включения в набор (например, соответствующие имена начинаются с “test...”).
- В реализациях xUnit, использующих атрибуты методов для обнаружения *тестовых методов* (Test Method), был добавлен атрибут `[Ignore]`.
- Был скрыт внутри комментария (или удален) код, явно добавляющий тест (или набор) в запускаемый набор.

Обычно *тест теряется* (Lost Test), когда после неудачного завершения теста разработчик отключает его, чтобы “не пробираться” через откакавшиеся тесты, работая с другими тестами. Кроме того, такая ситуация может возникать и случайно.

Возможное решение

Существует несколько способов избежать *потери тестов* (Lost Test).

Можно воспользоваться *наборами из одного теста* (Single Test Suite; см. *Именованный набор тестов*, Named Test Suite) для запуска единственного *тестового метода* (Test Method), а не отключать неудачно завершившиеся или медленные тесты. Можно воспользоваться *обозревателем набора тестов* (Test Tree Explorer; см. *Программа запуска тестов*, Test Runner, с. 405) для поиска и запуска единственного теста из набора. Оба варианта усложняются при использовании *цепочки тестов* (Chained Tests, с. 477) — намеренно реализованных *взаимодействующих тестов* (Interacting Tests). Это еще одна причина избегать подобных конструкций.

В зависимости от реализации xUnit можно воспользоваться механизмом игнорирования тестов, например, пакет NUnit позволяет запретить запуск теста, назначив атрибут `[Ignore]` *тестовому методу* (Test Method). Обычно такой механизм напоминает о количестве не запустившихся тестов, чтобы разработчики не забыли их включить. Кроме того, инструментарий *непрерывной интеграции* (continuous integration) можно настроить на неудачное завершение компиляции в случае, когда количество игнорируемых тестов превышает некоторое граничное значение.

Можно сравнить количество тестов после включения кода в хранилище с количеством, существовавшим непосредственно перед интеграцией. Достаточно убедиться, что это число увеличилось на количество добавленных тестов.

Если язык программирования поддерживает механизм интроспекции, можно реализовать или использовать существующий механизм *обнаружения тестов* (Test Discovery).

Для поиска тестов во время интеграции можно использовать другую стратегию. Некоторые утилиты автоматической компиляции (например, Ant) позволяют найти все файлы, имена которых соответствуют заданному шаблону (например, оканчивающиеся на “Test”). При использовании этой возможности можно быть уверенным, что не будут теряться целые наборы тестов.

Причина: отсутствующий модульный тест (Missing Unit Test)

Симптомы

Все модульные тесты завершаются успешно, но приемочные тесты продолжают отказывать. На каком-то этапе приемочные тесты завершаются успешно, но поведение отдельных классов не проверяет ни один из модульных тестов. При следующей модификации кода изменяется поведение одного из классов, “ломая” его функциональность.

Основная причина

Часто *модульные тесты отсутствуют* (Missing Unit Test), когда разработчики уделяют основное внимание приемочным тестам, но не применяют методику разработки на основе модульных тестов. Для успешного прохождения приемочных тестов написанной функциональности оказывается достаточно, но при последующем рефакторинге функциональность оказывается нерабочей. Модульные тесты не пропустили бы эти изменения в общее хранилище для интеграции.

Кроме того, *модульные тесты могут отсутствовать* (Missing Unit Test) и при разработке на основе тестов, когда разработчики “бегут впереди паровоза” и пишут код еще до того, как его необходимость диктуется неудачно завершившимся модульным тестом.

Возможное решение

Решение давно известно: писать больше модульных тестов. Конечно, легче сказать, чем сделать, и не всегда больше тестов позволяют решить проблему отсутствующих тестов. Строгая разработка на основе тестов — единственный способ получить все необходимые тесты, не создавая лишние.

Причина: нетестированный код (Untested Code)

Симптомы

Разработчики могут просто “знать”, что некоторая часть тестируемой системы никогда не вызывается тестами. Возможно, никто не видел, что этот код когда-то запускался, или утилиты подсчета покрытия кода полностью подтверждают это предположение. Рассмотрим следующий пример. Как определить правильность обработки исключения, генерируемого классом `timeProvider`?

```
public String getCurrentTimeAsHtmlFragment()
    throws TimeProviderEx {
    Calendar currentTime;
    try {
        currentTime = getTimeProvider().getTime();
    } catch (Exception e) {
        return e.getMessage();
    }
// и т.д.
```

Основная причина

Самой распространенной причиной появления *нетестированного кода* (Untested Code) является существование ветвей кода, специально реагирующих на поведение вызываемого компонента и не имеющих очевидного способа спровоцировать их выполнение. Обычно другой компонент вызывается синхронно и возвращает некоторые значения или генерирует исключения. Во время нормального тестирования встречается только подмножество возможных классов эквивалентности опосредованного ввода.

Еще одной распространенной причиной является неполнота набора тестов как следствие неполного описания функциональности, предоставляемой интерфейсом тестируемой системы.

Возможное решение

Если *нетестированный код* (Untested Code) является следствием невозможности управлять опосредованным вводом, самым распространенным решением является ис-

пользование *тестовой заглушки* (Test Stub, с. 544) для возврата различных типов опосредованного ввода и запуска всех ветвей кода. В противном случае достаточно будет настроить вызываемый компонент на возврат различных значений, обеспечивающих полноценное тестирование системы.

Причина: нетестированное требование (Untested Requirement)

Симптомы

Разработчики могут просто “знать”, что некоторая часть функциональности никогда не вызывается тестами. Как вариант при попытке тестировать элемент программного обеспечения не удается найти видимую функциональность, которую можно проверить через открытый интерфейс программного продукта. При этом все написанные тесты завершаются успешно.

При разработке на основе тестов известно, что для удовлетворения требования придется написать код. Но при этом не удается найти способ выразить необходимость в этом коде через операцию внутри *полностью автоматизированных тестов* (Fully Automated Tests, с. 81), как показано ниже.

```
public void testRemoveFlight() throws Exception {
    // Настройка
    FlightDto expectedFlightDto = createARegisteredFlight();
    FlightManagementFacade facade =
        new FlightManagementFacadeImpl();
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertFalse("flight should not exist after being removed",
        facade.flightExists(expectedFlightDto.
            getFlightNumber()));
}
```

Обратите внимание, что данный тест не проверяет правильность операции по занесению информации в журнал. Тест будет завершаться успешно независимо от правильности реализации журнала и даже без журнала вообще. Вот проверяемый этим тестом код и опосредованный вывод тестируемой системы. Код реализован неправильно.

```
public void removeFlight(BigDecimal flightNumber)
    throws FlightBookingException {
    System.out.println("removeFlight("+flightNumber+ ")");
    dataAccess.removeFlight(flightNumber);
    logMessage("CreateFlight", flightNumber); // Ошибка!
}
```

Если информация, захватываемая методом `logMessage`, потребуется при обслуживании приложения во время промышленной эксплуатации, как убедиться в ее правильности? Очевидно, что функциональность должна проверяться автоматизированными тестами.

Влияние

Часть необходимого поведения тестируемой системы можно случайно отключить, не заставив неудачно завершиться ни один тест. При этом программный продукт с ошибками может попасть к заказчику. Опасение, что можно внести новые ошибки, может помешать безжалостному рефакторингу или удалению предположительно ненужного кода.

Основная причина

Самой распространенной причиной *нетестированных требований* (Untested Requirement) является присутствие в тестируемой системе поведения, недоступного через открытый интерфейс. Система может иметь ожидаемые “побочные эффекты”, которые невозможно контролировать непосредственно из теста (например, запись в файл или вызов метода другого объекта либо компонента). Другими словами, система может иметь опосредованный вывод.

Если в качестве тестируемой системы выступает целое приложение, *нетестированное требование* (Untested Requirement) может стать результатом неполноты набора приемочных тестов, проверяющих все аспекты видимого поведения приложения.

Возможное решение

Если проблема заключается в отсутствии приемочных тестов, необходимо написать достаточное их количество, чтобы обеспечить правильную интеграцию компонентов. При этом может потребоваться модификация дизайна приложения с учетом тестов, включающая в себя разделение презентационного уровня и уровня бизнес-логики.

Если необходимо проверить опосредованный вывод системы, *проверка поведения* (Behavior Verification, с. 489) осуществляется через использование *подставных объектов* (Mock Object, с. 558). Более подробно тестирование опосредованного вывода рассматривалось в главе 11, “Использование тестовых двойников”.

Причина: всегда успешный тест (Neverfail Test)

Симптомы

Разработчики могут просто “знать”, что некоторые функции не работают, хотя тесты для этих функций завершаются успешно. При разработке на основе тестов для еще ненаписанной функциональности добавлен тест, но заставить его завершаться неудачно не удается.

Влияние

Если тест реагирует даже на отсутствие реализующего функциональность кода, он практически бесполезен для *локализации дефектов* (Defect Localization, с. 78).

Основная причина

Причиной проблемы могут быть неправильно запрограммированные утверждения, например `assertTrue(aVariable, true)` вместо `assertEquals(aVariable, true)` или просто `assertTrue(aVariable)`. Есть еще одна, более неприятная, причина: при запуске асинхронных тестов *программа запуска тестов* (Test Runner) может не получать уведомления о неудачном завершении в другом процессе или потоке.

Возможное решение

Можно реализовать межпотоковый механизм определения отказов, чтобы обеспечить неудачное завершение асинхронных тестов. Более правильным решением является рефакторинг кода для поддержки *минимального выполняемого файла* (Humble Executable; см. *Минимальный объект*, Humble Object, с. 700).

Часть III

Шаблоны

Глава 18

Шаблоны стратегии тестирования

Шаблоны в этой главе:

Стратегия автоматизации тестирования

Записанный тест (Recorded Test)	312
Тест на основе сценария (Scripted Test).....	319
Управляемый данными тест (Data-Driven Test).....	322
Инфраструктура автоматизации тестов (Test Automation Framework) ...	332

Стратегия создания тестовой конфигурации

Минимальная тестовая конфигурация (Minimal Fixture).....	336
Стандартная тестовая конфигурация (Standard Fixture).....	338
Новая тестовая конфигурация (Fresh Fixture).....	344
Общая тестовая конфигурация (Shared Fixture).....	350

Стратегия взаимодействия с testируемой системой

Манипуляция через “черный ход” (Back Door Manipulation)	359
Тест уровня (Layer Test).....	368

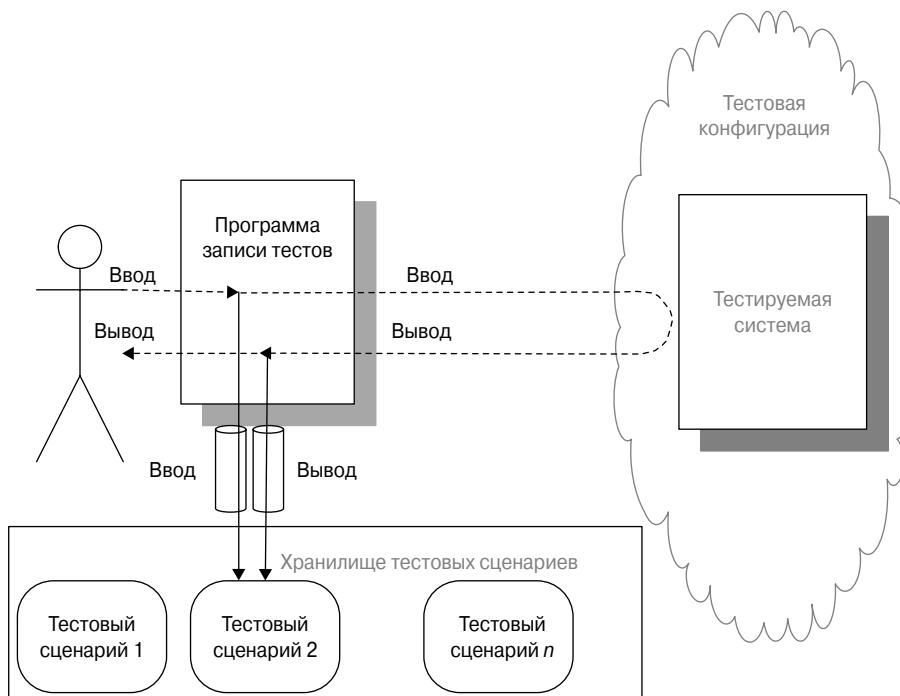
Записанный тест (Recorded Test)

Также известен как:

Тест на основе записи и воспроизведения (Record and Playback Test), Тест “роботом”-пользователем (Robot User Test), Тест вида “запись/воспроизведение” (Capture/Playback Test)

Как нужно создавать автоматизированные тесты для программного обеспечения?

Автоматизация тестов происходит в результате записи взаимодействия с приложением и воспроизведения записанных тестов специальным инструментом.



Автоматизированные тесты выполняют несколько функций. Они могут использоваться для регрессионного тестирования программного обеспечения после внесения изменений. Также они могут документировать поведение программного продукта. Кроме того, тесты описывают поведение еще до создания реализации. Способ создания сценариев для автоматизированного тестирования определяет, для выполнения каких функций можно использовать полученные тесты, как они реагируют на изменения в тестируемой системе и сколько знаний и усилий потребуется для их подготовки.

Записанные тесты (Recorded Test) позволяют быстро создавать регрессионные тесты после создания тестируемой системы, но перед внесением в нее изменений.

Как это работает

Используется утилита, отслеживающая взаимодействие пользователя с тестируемой системой. Утилита перехватывает большую часть информации, передаваемой от системы к пользователю, а также реакции пользователя на эту информацию. После завершения сеанс можно записать в файл для последующего воспроизведения. Для запуска теста включается воспроизводящая часть утилиты, которой передается записанный сеанс. Утилита запускает тестируемую систему и в ответ на вывод системы передает ей записанный ввод. Кроме того, вывод тестируемой системы во время тестирования может сравниваться с записанным выводом. Несовпадение значений может стать причиной неудачного завершения теста.

Некоторые утилиты для записи тестов позволяют изменять чувствительность сравнения записанных и фактических данных от тестируемой системы. Большинство утилит записи тестов взаимодействуют с системой через пользовательский интерфейс.

Когда это использовать

Если после создания и запуска приложения объем вносимых изменений сравнительно невелик, можно использовать *записанные тесты* (Recorded Test) для регрессионного тестирования. Кроме того, можно использовать записанные тесты при рефакторинге существующего приложения (при подготовке к изменению функциональности), когда *тесты на основе сценария* (Scripted Test, с. 319) недоступны. Обычно для одной и той же функциональности намного проще создать набор *записанных тестов* (Recorded Test), чем набор *тестов на основе сценария* (Scripted Test). Теоретически запись тестов доступна каждому, кто знает, как работать с приложением; специальные технические знания практически не требуются. На практике многие коммерческие утилиты имеют крутую кривую обучения. Кроме того, нужны специальные навыки для добавления “контрольных точек”, изменения чувствительности утилиты воспроизведения или модификации сценария теста, если была записана неправильная информация.

Большинство утилит для работы с *записанными тестами* (Recorded Test) взаимодействуют с тестируемой системой через пользовательский интерфейс. Такой подход делает тесты особенно чувствительными к развитию пользовательского интерфейса тестируемой системы (*чувствительность к интерфейсу*, Interface Sensitivity; см. “Хрупкий” тест, Fragile Test, с. 277). Даже такие небольшие изменения, как изменение внутреннего имени кнопки или поля, могут озадачить утилиту воспроизведения записанного теста. Кроме того, утилиты записывают информацию на очень низком уровне, значительно усложняя понимание тестов (*непонятный тест*, Obscure Test, с. 230). В результате тест сложно исправить вручную, если причиной сбоя является модификация тестируемой системы. Из-за этого при дальнейшем развитии тестируемой системы необходимо запланировать регулярную повторную запись тестов.

Если необходимо использовать *тесты как документацию* (Tests as Documentation, с. 79) или как движущую силу процесса разработки, стоит обратить внимание на *тесты на основе сценария* (Scripted Test). Этих целей сложно достичь при использовании коммерческих инструментов для записи тестов, так как нет возможности определить *язык высокого уровня* (Higher-Level Language, с. 95) для описания тестов. Эту проблему можно решить путем встраивания функциональности *записи тестов* (Recorded Test) в само приложение или с помощью *рефакторинга записанных тестов* (Refactored Recorded Test).

Вариант: рефакторинг записанных тестов (Refactored Recorded Test)

Использование последовательности “запись, рефакторинг, воспроизведение” (исходное название “record, refactor, playback” было придумано Адамом Герасом) порождает гибридную стратегию, основанную на извлечении “компонентов действия”, или “глаголов”, из новых *записанных тестов* (Recorded Test) с последующим переписыванием тестов для вызова этих компонентов вместо слишком подробного встроенного кода. В большинстве коммерческих инструментов записи/воспроизведения тестов предоставляется возможность превращения *точных значений* (Literal Value, с. 718) в параметры, которые передаются в “компонент действия” основным кодом теста. При смене интерфейса достаточно просто перезаписать “компонент действия”, а все существующие тесты продолжат работу, вызывая его новую реализацию. Эта стратегия практически совпадает с использованием *вспомогательных методов теста* (Test Utility Method, с. 610) для взаимодействия с тестируемой системой. В результате становится возможным использование записанных тестов после рефакторинга (Refactored Recorded Test) в качестве *языка высокого уровня* (Higher-Level Language) в *тестах на основе сценария* (Scripted Test). Такие инструменты, как Mercury Interactive BPT (business process testing — тестирование бизнес-процессов), используют подобную парадигму для тестов на основе сценария. После разработки сценариев высокого уровня и необходимых для тестов компонентов, более технически образованные пользователи могут записать или написать код вручную для отдельных компонентов.

Замечания по реализации

При использовании стратегии на основе *записанных тестов* (Recorded Test) возможны два основных варианта: использовать инструменты сторонних разработчиков, записывающие взаимодействие пользователя с приложением, или встроить механизм “записи и воспроизведения” в само приложение.

Вариант: внешняя запись тестов

На рынке доступно множество инструментов записи тестов. Каждый из них обладает собственными преимуществами и недостатками. Наилучший выбор зависит от природы пользовательского интерфейса приложения, выделенного бюджета, сложности проверяемой функциональности и ряда других факторов.

Если тесты должны “продвигать” процесс разработки, необходимо выбрать инструмент, использующий понятный и просто редактируемый вручную формат записи тестов. После записи содержимое тестов придется редактировать, так как это пример использования *тестов на основе сценария* (Scripted Test), хотя для запуска тестов используется инструмент “записи и воспроизведения”.

Вариант: встроенная запись тестов

Возможность записи тестов можно встроить в саму тестируемую систему. В таком случае “язык” записи тестов может определяться на достаточно высоком уровне — достаточно высоком для того, чтобы создавать тесты вручную еще до создания самой системы. Даже бывает мнение, что поддержка макросов VBA в электронной таблице Microsoft Excel изначально планировалась как механизм автоматизированного тестирования Excel.

Пример: встроенная запись тестов

На первый взгляд, не имеет смысла приводить пример кода для *записанных тестов* (Recorded Test), так как данный шаблон связан с созданием тестов, а не с их представлением. При воспроизведении тест по сути не отличается от *управляемого данными теста* (Data-Driven Test, с. 322). При этом рефакторинг в направлении записанных тестов (Recorded Test) происходит очень редко, так как это первая стратегия автоматизации тестирования, которую пытаются применять при разработке проекта. Несмотря на это *записанные тесты* (Recorded Test) могут вводиться в проект после *тестов на основе сценария* (Scripted Test), если обнаруживается слишком много *отсутствующих тестов* (Missing Test) из-за высокой стоимости автоматизации вручную. В таком случае существующие тесты на основе сценариев не превращаются в записанные тесты. Записываются только новые тесты.

Ниже приведен пример теста, записанного самим приложением. Этот тест использовался для регрессионного тестирования критического с точки зрения безопасности приложения после переноса с языка C и операционной системы OS/2 на язык C++ и операционную систему Windows. Обратите внимание, как записанная информация формирует *язык высокого уровня* (Higher-Level Language) для конкретной предметной области, понятный рядовому пользователю.

```
<interaction-log>
  <commands>
    <!-- more commands omitted -->
    <command seqno="2" id="Supply Create">
      <field name="engineno" type="input">
        <used-value>5566</used-value>
        <expected></expected>
        <actual status="ok"/>
      </field>
      <field name="direction" type="selection">
        <used-value>SOUTH</used-value>
        <expected>
          <value>SOUTH</value>
          <value>NORTH</value>
        </expected>
        <actual>
          <value status="ok">SOUTH</value>
          <value status="ok">NORTH</value>
        </actual>
      </field>
    </command>
    <!-- more commands omitted -->
  </commands>
</interaction-log>
```

В данном примере показан вывод в результате воспроизведения записанного теста. Элементы *actual* вставляются встроенным механизмом воспроизведения. Атрибуты *status* показывают, отличается ли значение элемента *actual* от значения элемента *expected*. К файлам были применены листы стилей для форматирования в стиле тестов Fit и цветового кодирования результатов. После этого пользователи проекта самостоятельно записывали, воспроизводили и анализировали результаты выполнения тестов.

Эта запись создана с помощью ловушек, вставленных на презентационном уровне приложения. Записывались предоставленные пользователю варианты и выбранные им пункты. Ниже приведен пример одной из таких ловушек.

```
if (playback_is_on())  (
    choice = get_choice_for_playback(dialog_id, choices_list);
) else (
    choice = display_dialog(choices_list, row, col, title, key);
)
if (recording_is_on())  (
    record_choice(dialog_id, choices_list, choice, key);
)
```

Метод `get_choice_for_playback` получает содержимое элемента `used-value` вместо запроса выбора у пользователя. Метод `record_choice` генерирует элемент `actual` и делает “утверждение” относительно элемента `expected`, записывая результат в атрибут `status` каждого элемента. Обратите внимание, что в режиме воспроизведения вызов `recording_is_on()` возвращает `true` и позволяет записать результаты теста.

Пример: коммерческий инструмент для записи и воспроизведения тестов

Практически каждая коммерческая утилита для тестирования использует метафору “записи и воспроизведения”. Кроме того, каждая утилита определяет собственный формат файла для *записанных тестов* (Recorded Test). Ниже приводится “краткая” выдержка из теста, записанного с помощью утилиты Mercury Interactive QuickTest Professional (QTP). Текст показан в режиме “Expert View”, что позволяет понять природу записанного теста: это программа на языке VbScript! Пример содержит комментарии (начинающиеся с “`@@`”), которые вставлены вручную для пояснения действий теста; эти комментарии будут утеряны, если повторно записать тест после внесения изменений в приложение.

```
@@
@@ GoToPageMaintainTaxonomy()
@@
Browser("Inf").Page("Inf").WebButton("Login").Click
Browser("Inf").Page("Inf_2").Check CheckPoint("Inf_2")
Browser("Inf").Page("Inf_2").Link("TAXONOMY LINKING").Click
Browser("Inf").Page("Inf_3").Check CheckPoint("Inf_3")
Browser("Inf").Page("Inf_3").Link("MAINTAIN TAXONOMY").Click
Browser("Inf").Page("Inf_4").Check CheckPoint("Inf_4")
@@
@@ AddTerm("A", "Top Level", "Top Level Definition")
@@
Browser("Inf").Page("Inf_4").Link("Add").Click
wait 4
Browser("Inf_2").Page("Inf").Check CheckPoint("Inf_5")
Browser("Inf_2").Page("Inf").WebEdit("childCodeSuffix").Set "A"
Browser("Inf_2").Page("Inf").
    WebEdit("taxonomyDto.descript").Set "Top Level"
Browser("Inf_2").Page("Inf").
    WebEdit("taxonomyDto.definiti").Set "Top Level Definition"
Browser("Inf_2").Page("Inf").WebButton("Save").Click
wait 4
Browser("Inf").Page("Inf_5").Check CheckPoint("Inf_5_2")
```

```

@@
@@ SelectTerm("[A] -Top Level")
@@
Browser("Inf").Page("Inf_5").
    WebList("selectedTaxonomyCode").Select "[A] -Top Level"
@@
@@ AddTerm("B", "Second Top Level", "Second Top Level Definition")
@@
Browser("Inf").Page("Inf_5").Link("Add").Click
wait 4
Browser("Inf_2").Page("Inf_2").Check CheckPoint("Inf_2_2")
    infofile_i_Inform_Alberta_21.inf_;_highlight id_;
        _Browser("Inf_2").Page("Inf_2")_:_;
@@
@@ И так далее, и тому подобное

```

Обратите внимание, как тест описывает ввод и вывод в терминах пользовательского интерфейса приложения. Данный тест страдает от двух основных проблем: *непонятный тест* (Obscure Test), причиной которого является излишняя подробность записанной информации, и *чувствительность к интерфейсу* (Interface Sensitivity), что делает этот тест “хрупким” тестом (Fragile Test).

Замечания по рефакторингу

Можно сделать тест более полезным в качестве документации или сократить его либо можно избежать *высокой стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300) и поддержать составление других тестов на основе языка *высокого уровня* (Higher-Level Language) через многократное применение рефакторинга *выделение метода* (Extract Method) [Ref].

Пример: записанный коммерческим инструментом тест после рефакторинга

Ниже приведен пример того же теста, подвергнутого рефакторингу для достижения цели *доносите намерение* (Communicate Intent, с. 95).

```

GoToPage_MaintainTaxonomy()
AddTerm("A", "Top Level", "Top Level Definition")
SelectTerm("[A] -Top Level")
AddTerm("B", "Second Top Level", "Second Top Level Definition")

```

Обратите внимание, насколько доходчивым стал тест. Выделенные *вспомогательные методы теста* (Test Utility Method) выглядят так.

```

Method GoToPage_MaintainTaxonomy()
    Browser("Inf").Page("Inf").WebButton("Login").Click
    Browser("Inf").Page("Inf_2").Check CheckPoint("Inf_2")
    Browser("Inf").Page("Inf_2").Link("TAXONOMY LINKING").Click
    Browser("Inf").Page("Inf_3").Check CheckPoint("Inf_3")
    Browser("Inf").Page("Inf_3").Link("MAINTAIN TAXONOMY").Click
    Browser("Inf").Page("Inf_4").Check CheckPoint("Inf_4")
End
Method AddTerm(code, name, description)
    Browser("Inf").Page("Inf_4").Link("Add").Click
    wait 4

```

```
Browser("Inf_2").Page("Inf").Check CheckPoint("Inf_5")
Browser("Inf_2").Page("Inf").
    WebEdit("childCodeSuffix").Set code
Browser("Inf_2").Page("Inf").
    WebEdit("taxonomyDto.descript").Set name
Browser("Inf_2").Page("Inf").
    WebEdit("taxonomyDto.definiti").Set description
Browser("Inf_2").Page("Inf").WebButton("Save").Click
wait 4
Browser("Inf").Page("Inf_5").Check CheckPoint("Inf_5_2")
end

Method SelectTerm(path)
    Browser("Inf").Page("Inf_5").
        WebList("selectedTaxonomyCode").Select path
    Browser("Inf").Page("Inf_5").Link("Add").Click
    wait 4
end
```

Этот пример создан на скорую руку, но имеет много признаков, делающих его похожим на тесты для инфраструктуры xUnit. Не пытайтесь повторить этот пример самостоятельно; скорее всего, он содержит синтаксические ошибки.

Источники дополнительной информации

В статье “Agile Regression Testing Using Record and Playback” [ARTRP] описывается авторский опыт по встраиванию механизма записи тестов в приложение, что позволило облегчить перенос на другую платформу.

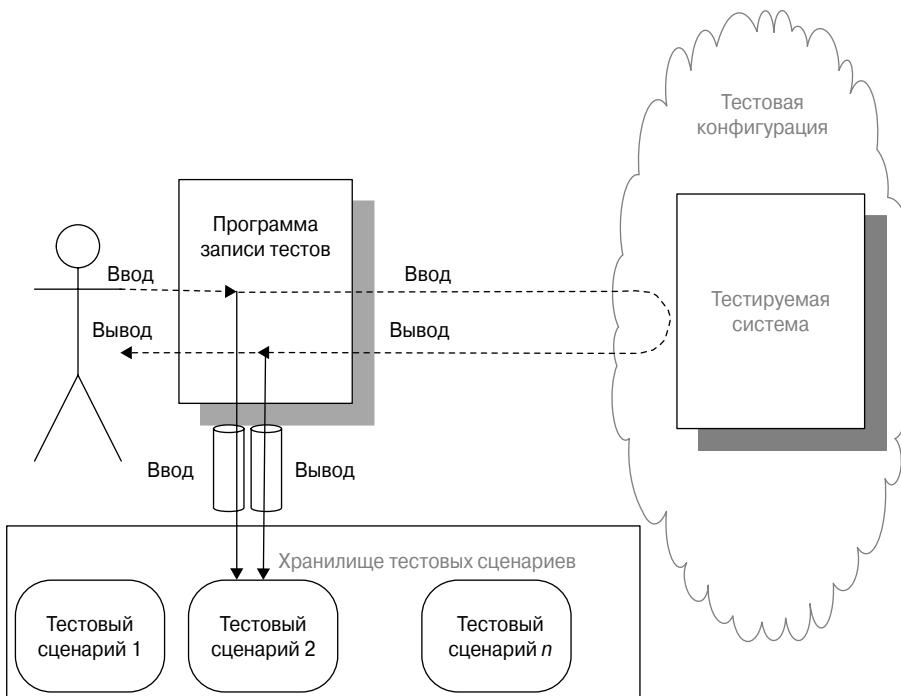
Тест на основе сценария (Scripted Test)

Как можно создать автоматизированные тесты для программного продукта?

Автоматизация тестов обеспечивается написанием тестовых программ вручную.

Также известен как:

Написанный вручную тест (Hand-Written Test), Созданный вручную сценарий теста (Hand-Scripted Test), Программный тест (Programmatic Test), Автоматизированный модульный тест (Automated Unit Test)



Автоматизированные тесты решают несколько задач. Они могут использоваться для регрессионного тестирования после внесения изменений в программный продукт, документировать поведение программного продукта, а также описывать поведение продукта до написания самого кода. Способ создания сценариев автоматизированных тестов влияет на их возможные применения, устойчивость к изменениям, вносимым в тестируемую систему, а также на объем трудозатрат, необходимых для их создания.

Тесты на основе сценария (Scripted Test) позволяют создавать тесты еще до разработки самого программного продукта. В результате они управляют процессом проектирования.

Как это работает

Для автоматизации тестов создаются тестовые программы, взаимодействующие с тестируемой системой и вызывающие ее функциональность. В отличие от *записанных тестов* (Recorded Test, с. 312) они могут быть как приемочными, так и модульными тестами. Часто тестовые программы называют тестовыми сценариями, чтобы отличать их от проверяемого кода продукта.

Когда это использовать

Практически всегда *тесты на основе сценария* (Scripted Test) используются в качестве модульных. Это связано с простотой доступа к отдельным модулям непосредственно из программы, написанной на том же языке программирования. Кроме того, данный подход позволяет выполнить все ветви кода, включая “патологические случаи”.

Приемочные тесты несколько усложняют общую картину; *тесты на основе сценария* (Scripted Test) применяются каждый раз, когда автоматизированный тест истории должен управлять разработкой программного продукта. *Записанные тесты* (Recorded Test) не выполняют эту функцию, так как сложно записать тест, если приложение еще не написано. Создание *теста на основе сценария* (Scripted Test) требует опыта в разработке программного обеспечения и в техниках тестирования. Маловероятно, что рядовой пользователь проекта согласиться учиться писать *тесты на основе сценария* (Scripted Test). Альтернативой написанию тестов на языке программирования является определение *языка высокого уровня* (Higher-Level Language, с. 95) для взаимодействия с тестируемой системой с последующей реализацией языка в виде интерпретатора *управляемых данными тестов* (Data-Driven Test, с. 322). Существует инфраструктура определения *управляемых данными тестов* (Data-Driven Test) с открытым исходным кодом, которая называется Fit. У нее есть родственный проект, FitNesse, основанный на технологии Wiki. Такой подход к тестированию также поддерживается инструментом Canoo WebTest.

При работе с унаследованным приложением (любым приложением, для которого не существует полного набора тестов, выступающего в роли страховочной сети) *записанные тесты* (Recorded Test) могут использоваться для быстрого создания набора регрессионных тестов. Такие тесты будут обеспечивать защиту на этапе рефакторинга для интеграции полноценных тестов.

Замечания по реализации

Традиционно в *тестах на основе сценария* (Scripted Test) для написания “программ” использовался специальный язык сценариев тестирования. В современных системах на основе *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332), например в xUnit, для создания тестов используется тот же язык, на котором написана тестируемая система. В такой ситуации каждая тестовая программа принимает форму *тестового метода* (Test Method, с. 378), принадлежащего *классу теста* (Test-case Class, с. 401). Для сокращения *ручного вмешательства* (Manual Intervention, с. 287) каждый тестовый метод должен содержать реализацию *самопроверяющегося теста* (Self-Checking Test, с. 81), одновременно являющегося *повторяемым тестом* (Repeatable Test, с. 81).

Пример: тест на основе сценария (Scripted Test)

Ниже приведен пример теста на *основе сценария* (Scripted Test), написанного для инфраструктуры JUnit.

```
public void testAddLineItem_quantityOne() {
    final BigDecimal BASE_PRICE = UNIT_PRICE;
    final BigDecimal EXTENDED_PRICE = BASE_PRICE;
    // Настройка тестовой конфигурации
    Customer customer = createACustomer(NO_CUST_DISCOUNT);
    Invoice invoice = createInvoice(customer);
    // Вызов тестируемой системы
    invoice.addItemQuantity(PRODUCT, QUAN_ONE);
    // Проверка результата
    LineItem expected =
        createLineItem(QUAN_ONE, NO_CUST_DISCOUNT,
                      EXTENDED_PRICE, PRODUCT, invoice);
    assertContainsExactlyOneLineItem(invoice, expected);
}
public void testChangeQuantity_severalQuantity() {
    final int ORIGINAL_QUANTITY = 3;
    final int NEW_QUANTITY = 5;
    final BigDecimal BASE_PRICE =
        UNIT_PRICE.multiply(new BigDecimal(NEW_QUANTITY));
    final BigDecimal EXTENDED_PRICE =
        BASE_PRICE.subtract(BASE_PRICE.multiply(
            CUST_DISCOUNT_PC.movePointLeft(2)));
    // Настройка тестовой конфигурации
    Customer customer = createACustomer(CUST_DISCOUNT_PC);
    Invoice invoice = createInvoice(customer);
    Product product = createAProduct(UNIT_PRICE);
    invoice.addItemQuantity(product, ORIGINAL_QUANTITY);
    // Вызов тестируемой системы
    invoice.changeQuantityForProduct(product, NEW_QUANTITY);
    // Проверка результата
    LineItem expected = createLineItem(NEW_QUANTITY,
        CUST_DISCOUNT_PC, EXTENDED_PRICE, PRODUCT, invoice);
    assertContainsExactlyOneLineItem(invoice, expected);
}
```

О названии

Автоматизированные тестовые программы традиционно называются тестовыми сценариями — изначально они писались на таких интерпретируемых языках, как Tcl.

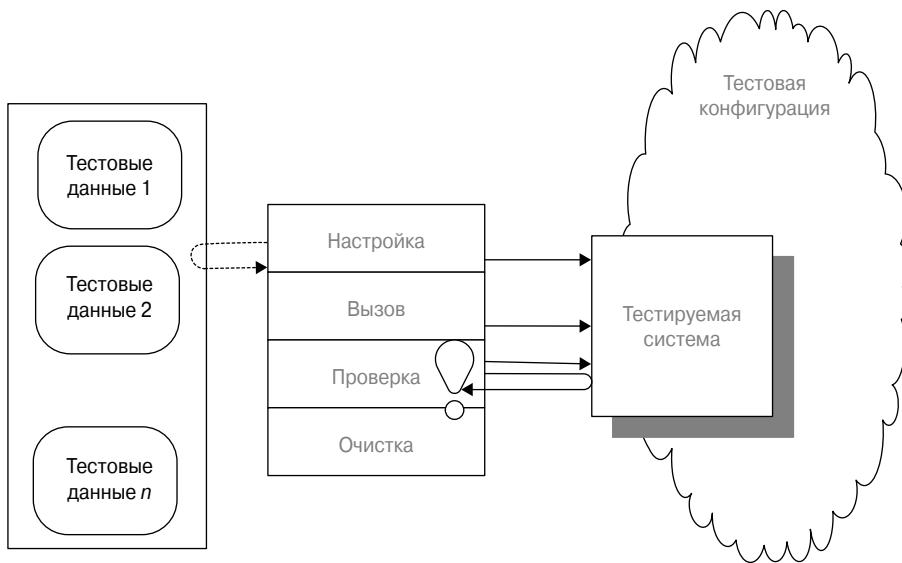
Источники дополнительной информации

Процесс написания *тестов на основе сценария* (Scripted Test) и их применение для управления дизайном тестируемой системы рассматриваются в целом ряде книг. Чтение стоит начинать с [TDD-BE] и [TDD-APG].

Управляемый данными тест (Data-Driven Test)

Как можно создавать автоматизированные тесты для программного продукта и как сократить дублирование тестового кода?

Вся информация, необходимая для каждого теста, хранится в специальном файле.
Создается интерпретатор, который читает содержимое файла и выполняет необходимые тесты.



Тестирование может быть утомительным не только потому, что одни и те же тесты запускаются много раз, но и потому, что тесты отличаются один от другого. Например, один и тот же тест должен запускаться с несколько иными входными параметрами системы и проверять, что полученный результат изменился соответственно. В такой ситуации каждый тест будет состоять из одинаковых последовательностей действий. Имея такое количество тестов, можно полностью покрыть всю функциональность; большой объем не так хорош с точки зрения обслуживания, поскольку любая модификация алгоритма может потребовать внесения изменений во все похожие тесты.

Управляемый данными тест (Data-Driven Test) является одним из способов увеличить покрытие, сократив объем написанного и обслуживаемого тестового кода.

Как это работает

Необходимо создать интерпретатор *управляемых данных тестов* (Data-Driven Test), который содержит общую для всех тестов логику. Меняющиеся данные размещаются в специальном файле. Интерпретатор читает данные из файла и выполняет необходимые тесты. Для каждого теста выполняется набор действий, составляющий *четырехфазный тест* (Four-Phase Test, с. 387). Сначала интерпретатор извлекает тестовые данные из файла и на их основе создает тестовую конфигурацию. После этого тестируемая система

вызывается с полученными из файла параметрами. Затем интерпретатор сравнивает полученный результат (возвращаемые значения, состояние после вызова и т.д.) с ожидаемым результатом, записанным в файле. Если результаты не совпадают, тест помечается как завершившийся неудачно. Если тестируемая система генерирует исключение, интерпретатор перехватывает его и соответствующим образом маркирует тест, продолжая работу. Наконец, интерпретатор выполняет необходимую очистку тестовой конфигурации и переходит к следующему тесту в файле.

В результате тест, требующий сложной последовательности действий, можно сократить до строки в файле с данными. Инфраструктура Fit является популярным средством создания управляемых данными тестов (Data-Driven Test).

Когда это использовать

Управляемый данными тест (Data-Driven Test) представляет стратегию, альтернативную *записанным тестам* (Recorded Test, с. 312) и *тестам на основе сценария* (Scripted Test, с. 319). Но он может использоваться как часть стратегии *тестов на основе сценария* (Scripted Test), а *записанный тест* (Recorded Test) в момент воспроизведения по своей сути является *управляемым данными тестом* (Data-Driven Test). *Управляемый данными тест* (Data-Driven Test) является идеальной стратегией для вовлечения пользователей продукта в процесс создания автоматизированных тестов. Сохранив простоту формата файла, можно заставить рядовых пользователей заполнить файл данными и запустить тест, не требуя от технических специалистов написания кода для каждого теста в отдельности.

Управляемый данными тест (Data-Driven Test) можно использовать как часть стратегии *теста на основе сценария* (Scripted Test), когда одна и та же последовательность операций используется для вызова тестируемой системы с множеством отличающихся наборов параметров. Обычно подобие выясняется со временем и сначала выполняется рефакторинг до *параметризованного теста* (Parameterized Test, с. 618), а затем — до *управляемого данными теста* (Data-Driven Test). Кроме того, может потребоваться организация стандартного набора действий в разных последовательностях с разными значениями данных, как в *инкрементном табличном teste* (Incremental Tabular Test). Такой подход позволяет обеспечить максимальное покрытие с минимальным объемом обслуживаемого кода. При этом остается очень простой способ добавления необходимых тестов.

Еще одним критерием использования *управляемого данными теста* (Data-Driven Test) является способ управления поведением: оно может быть фиксированным, а может управляться конфигурационными данными. Если тесты управляемого данными поведения автоматизируются через *тесты на основе сценария* (Scripted Test), тестовые программы придется обновлять при каждой модификации конфигурационных данных. Такая ситуация кажется совершенно неестественной, так как подразумевает включение в общее хранилище при каждой модификации конфигурационных данных¹. При использовании управляемых данными тестов изменения конфигурационных данных или метаобъектов управляются изменениями *управляемых данными тестов* (Data-Driven Test); такое отношение кажется значительно более естественным.

¹ Конечно, тестовые данные должны также находиться в хранилище с поддержкой управления версиями, но это тема для отдельной книги. Дополнительная информация приведена в книге [RDb].

Замечания по реализации

Варианты реализации зависят от того, насколько самостоятельной (независимой от инфраструктуры xUnit) является стратегия *управляемых данными тестов* (Data-Driven Test). Независимое использование этой стратегии обычно подразумевает применение инструментов с открытым исходным кодом (Fit) или коммерческих утилит записи тестов (QTP). Совместное использование может потребовать реализации интерпретатора управляемых данными тестов поверх инфраструктуры xUnit.

Вне зависимости от выбора стратегии следует использовать подходящую *инфраструктуру автоматизации тестов* (Test Automation Framework, с. 332), если она доступна. Таким образом, тест делится на две части: интерпретатор и файл с данными. Оба компонента должны находиться под наблюдением системы управления версиями, что позволит следить за их развитием со временем и возвращаться к любому предыдущему состоянию при неудачной модификации. Особенно важно хранить в общем хранилище файлы с данными, хотя концепция такого хранилища и может показаться незнакомой рядовым пользователям. Сохранение файлов можно сделать прозрачным, предоставив пользователям инструмент для создания файлов, подобный FitNesse. Можно настроить “дружественное к пользователем” хранилище в виде системы управления документами, которая по счастливой случайности поддерживает управление версиями.

Кроме того, данные тесты должны запускаться в процессе непрерывной интеграции, чтобы убедиться, что работавшие ранее тесты не начали вдруг завершаться неудачно. Если этого не сделать, ошибки будут проникать в продукт незамеченными и их исправление после обнаружения потребует значительных усилий на диагностику. Включение приемочных тестов в процесс непрерывной интеграции требует некоторого механизма, который будет следить за случаями успешного завершения тестов, так как не все приемочные тесты должны завершаться успешно, чтобы включить изменения в общее хранилище кода. Одним из вариантов является использование двух наборов файлов с данными. Успешно завершившиеся тесты переносятся из “полностью красной” группы в “полностью зеленую”, которая используется для регрессионного тестирования во время автоматической компиляции.

Вариант: инфраструктура управляемых данными тестов (Data-Driven Test Framework) (Fit)

При самостоятельном использовании стратегии на основе *управляемых данными тестов* (Data-Driven Test) необходимо рассмотреть возможность применения выделенной инфраструктуры. Пакет Fit был разработан Уордом Каннингэмом для включения обычных пользователей в процесс создания автоматизированных тестов. Хотя обычно инфраструктура Fit используется для автоматизации приемочных тестов, она может использоваться и для модульных тестов, когда их количество оправдывает создание необходимых тестовых конфигураций. Пакет Fit состоит из двух частей: инфраструктуры и созданной пользователем конфигурации. Инфраструктура Fit представляет собой универсальный интерпретатор *управляемых данными тестов* (Data-Driven Test), читающий содержимое файла и находящий все содержащиеся в нем таблицы. Имя класса тестовой конфигурации находится в верхней левой ячейке каждой таблицы. После обнаружения имени осуществляется поиск выполняемого файла для этого класса. Обнаружив класс, инфраструктура создает его экземпляр и передает ему управление. Получив управление, экземпляр читает каждую строку и каждый

столбец таблицы. Разработчик может переопределить методы инфраструктуры, чтобы описать, что должно происходить с каждой ячейкой в таблице. Таким образом, тестовая конфигурация Fit представляет собой адаптер, вызываемый для интерпретации таблицы с данными и обращения к методам тестируемой системы.

Кроме того, в таблице содержатся ожидаемые результаты тестируемой системы. Но в отличие от *метода с утверждением* (Assertion Method, с. 390) в инфраструктуре xUnit, инфраструктура Fit не прекращает выполнение теста при появлении первого результата, который не совпадает с ожидаемым. Вместо этого используется цветовое кодирование ячеек в таблице. Каждый ожидаемый результат, совпадающий с фактически полученным, помечается зеленым цветом. Красным цветом помечаются ошибочные или неожиданные результаты.

Инфраструктура Fit предоставляет ряд преимуществ.

- Приходится писать намного меньше кода по сравнению с реализацией собственного интерпретатора.
- Вывод понятен и рядовому пользователю, а не только разработчику.
- Тест не завершает работу после первого же ложного утверждения. Инфраструктура Fit обеспечивает вывод информации о нескольких отказах/ошибках в формате, способствующем выявлению закономерностей.
- Существует множество типов тестовой конфигурации, пригодных для наследования или использования как есть.

Тогда почему же инфраструктура Fit не используется для всех тестов вместо xUnit? Ниже приводятся основные недостатки пакета Fit.

- Необходимо хорошо изучить тестовые сценарии, прежде чем писать тестовую конфигурацию для Fit. После этого логику каждого теста необходимо описать в табличном представлении; это не всегда просто, особенно для разработчиков, обученных мыслить процедурно. Хотя в штате могут быть тестеры, способные писать конфигурации Fit для приемочных тестов, такой подход не оправдывает себя для модульных тестов (кроме случаев, когда количество тестеров примерно совпадает с количеством разработчиков).
- Тесты должны содержать одну и ту же логику взаимодействия с тестируемой системой. (Табличные данные передаются тестируемой системе на этапе создания тестовой конфигурации или вызова системы. Результаты получаются от тестируемой системы на этапе проверки результатов.) Для одновременного запуска нескольких тестов различных форматов придется создать несколько разных конфигураций (по одной для каждого формата). Создать новую конфигурацию обычно сложнее, чем написать несколько *тестовых методов* (Test Method, с. 378). Хотя существует множество типов конфигураций, доступных для использования в исходном виде или в виде базовых классов, такое их применение является еще одним навыком, который разработчикам придется приобретать. И даже в этом случае не все модульные тесты можно автоматизировать с помощью инфраструктуры Fit.
- Тесты Fit обычно не интегрируются в регрессионные тесты разработчиков, запускаемые с помощью инфраструктуры xUnit. Вместо этого тесты придется запускать отдельно — в результате с некоторой вероятностью они не будут запускаться при каждом включении кода в общее хранилище. Некоторые разработчики вклю-

чают тесты Fit в процесс непрерывной интеграции для частичного обхода этой проблемы. Известны случаи, когда успешно использовался второй “клиентский” сервер компиляции, который и запускал приемочные тесты.

Конечно, потенциально каждую из этих проблем можно решить. В целом же инфраструктура xUnit больше подходит для модульного тестирования, чем Fit. Для приемочных тестов справедливо обратное утверждение.

Вариант: простейший интерпретатор тестов (Naive xUnit Test Interpreter)

Если создано небольшое количество управляемых данными тестов (Data-Driven Test), которые необходимо запускать как часть стратегии на основе xUnit и *тестов на основе сценария* (Scripted Test), самым простым решением является создание *тестового метода* (Test Method) с циклом внутри, который будет читать набор входных данных и ожидаемых результатов из файла. Такая конструкция эквивалентна преобразованию *параметризованного теста* (Parametrized Test) и всех вызывающих его функций в *табличный тест* (Tabular Test). Как и при использовании *табличного теста* (Tabular Test), такой подход к созданию интерпретатора тестов приведет к появлению одного *объекта теста* (Testcase Object, с. 410) с большим количеством утверждений. Подобная конструкция имеет ряд особенностей.

- Весь набор управляемых данными тестов (Data-Driven Test) считается одним тестом. Таким образом, преобразование набора параметризованных тестов (Parametrized Test) в единственный управляемый данными тест (Data-Driven Test) приводит к уменьшению количества запускаемых тестов.
- Выполнение управляемого данными теста (Data-Driven Test) прекращается после первого же отказа или ошибки. Как следствие теряется результат локализации дефектов (Defect Localization, с. 78). Некоторые варианты xUnit позволяют отключить завершение теста после первого же ложного утверждения.
- Необходимо обеспечить локализацию подтеста на основе сообщения, выдаваемого ложным утверждением.

Две последние проблемы можно решить за счет добавления конструкции “try/catch” внутри цикла, окружающего логику теста. В результате появляется возможность продолжить выполнение. Несмотря на это остается проблема удобочитаемого вывода результатов работы тестов (например, “Неудачно завершились тесты 1, 3 и 6 с...”).

Можно упростить расширение интерпретатора на несколько типов тестов в одном и том же файле данных, если вставить “глагол” или “действие” в каждую запись в файле данных. После этого интерпретатор сможет выбирать необходимый *параметризованный тест* (Parametrized Test) в зависимости от конкретного действия.

Вариант: генератор объекта набора тестов (Test Suite Object Generator)

Можно обойти проблему “останова после первого отказа”, связанную с *простейшим интерпретатором тестов xUnit* (Naive xUnit Test Interpreter), заставив метод *suite фабрики наборов тестов* (Test Suite Factory; см. *Перечисление тестов*, Test Enumeration, с. 425) создавать такую же структуру *объекта набора тестов* (Test Suite Object, с. 414), какую создает встроенный *механизм обнаружения тестов* (Test Discovery, с. 420). Для этого для каждой записи в файле данных управляемого данными теста (Data-Driven Test) создается

объект теста (Testcase Object), инициализированный тестовыми данными из конкретной записи. (Такая реализация будет очень похожа на внутренний механизм *обнаружения тестовых методов* (Test Method Discovery), применяемый в xUnit. Единственным отличием является то, что вместе с именем *тестового метода* (Test Method) передаются и тестовые данные.) Такой объект будет знать, как выполнять *параметризованный тест* (Parametrized Test) с загруженными в него данными. В результате *управляемый данными тест* (Data-Driven Test) продолжит выполнение даже после неудачного завершения первого *объекта теста* (Testcase Object). После этого *программа запуска тестов* (Test Runner, с. 405) сможет самостоятельно посчитать общее количество тестов, ошибок и неудачных завершений стандартным образом.

Вариант: имитатор объекта набора тестов (Test Suite Object Simulator)

Альтернативой созданию *объекта набора тестов* (Test Suite Object) является создание *объекта теста* (Testcase Object), который ведет себя, как набор. Такой объект читает содержимое файла с данными и перебирает все тесты. Объект должен перехватывать исключения, генерированные *параметризованным тестом* (Parametrized Test), и продолжать выполнение последующих тестов. После завершения работы *объекта теста* (Testcase Object) должен возвращать количество тестов, неудачных завершений и ошибок *программе запуска тестов* (Test Runner). Кроме того, объект должен содержать реализацию всех остальных методов стандартного интерфейса, от которого зависит *программа запуска тестов* (Test Runner) (например, получение количества тестов в “наборе”, получение имени и состояния каждого теста для *обозревателя набора тестов с графическим интерфейсом* (Graphical Test Tree Explorer) и т.д.).

Мотивирующий пример

Предположим, имеется следующий набор тестов.

```
def test_extref
  sourceXml = "<extref id='abc' />"
  expectedHtml = "<a href='abc.html'>abc</a>"
  generateAndVerifyHtml(sourceXml, expectedHtml, "<extref>")
end
def test_testterm_normal
  sourceXml = "<testterm id='abc' />"
  expectedHtml = "<a href='abc.html'>abc</a>"
  generateAndVerifyHtml(sourceXml, expectedHtml, "<testterm>")
end
def test_testterm_plural
  sourceXml = "<testterms id='abc' />"
  expectedHtml = "<a href='abc.html'>abcs</a>"
  generateAndVerifyHtml(sourceXml, expectedHtml, "<plural>")
end
```

Подобная краткость теста обеспечивается следующим определением *параметризованного теста* (Parametrized Test).

```
def generateAndVerifyHtml(sourceXml, expectedHtml,
                           message, &block)
  mockFile = MockFile.new
  sourceXml.delete!("\t")
```

```

@handler = setupHandler(sourceXml, mockFile )
block.call unless block == nil
@handler.printBodyContents
actual_html = mockFile.output
assert_equal_html(expectedHtml,
                  actual_html,
                  message + "html output")
actual_html
end

```

Основным недостатком таких тестов является их реализация в виде кода, хотя единственным отличием являются используемые в качестве параметров данные.

Замечания по рефакторингу

Конечно, правильным решением является выделение общей логики *параметризованных тестов* (Parametrized Test) в интерпретатор управляемых данными тестов (Data-Driven Test) и сбор всех параметров в доступный для редактирования файл с данными. Необходимо написать “основной” тест, которому известно, из какого файла читать данные и как разбирать файл на элементы. Эта логика может вызывать существующие *параметризованные тесты* (Parametrized Test), оставляя инфраструктуре xUnit отслеживание статистики завершения тестов.

Пример: управляемый данными тест (Data-Driven Test) на основе xUnit с файлом данных в формате XML

В этом примере для представления данных используется формат XML. Каждому тесту соответствует элемент `test`, содержащий три основных компонента:

- действие, которое сообщает интерпретатору тестов, какую логику запустить (например, `crossref`);
- параметры, передаваемые в тестируемую систему; в данном случае это элемент `sourceXml`;
- код HTML, ожидаемый от тестируемой системы (в элементе `expectedHtml`).

Три компонента размещены внутри элемента `testsuite`.

```

<testsuite id="CrossRefHandlerTest">
  <test id="extref">
    <action>crossref</action>
    <sourceXml>
      <extref id='abc' />
    </sourceXml>
    <expectedHtml>
      <a href='abc.html'>abc</a>
    </expectedHtml>
  </test>
  <test id="TestTerm">
    <action>crossref</action>
    <sourceXml>
      <testterm id='abc' />
    </sourceXml>
  </test>
</testsuite>

```

```

<expectedHtml>
    <a href='abc.html'>abc</a>
</expectedHtml>
</test>
<test id="TestTerm Plural">
    <action>crossref</action>
    <sourceXml>
        <testterms id='abc' />
    </sourceXml>
    <expectedHtml>
        <a href='abc.html'>abcs</a>
    </expectedHtml>
</test>
</testsuite>

```

Кто угодно может редактировать файл XML в специализированном редакторе, не опасаясь внести ошибку в логику теста. Вся логика проверки ожидаемого результата скрыта внутри интерпретатора тестов, как и в случае *параметризованных тестов* (Parametrized Test). Для удобства просмотра можно скрыть структуру XML от пользователя, определив лист стилей. Кроме того, во многих редакторах XML для простоты редактирования на основе кода строится набор форм.

Используя формат CSV, можно избежать сложностей работы с кодом XML.

Пример: управляемый данными тест (Data-Driven Test) на основе xUnit с файлом данных в формате CSV

При использовании формата CSV тест из предыдущего примера будет выглядеть следующим образом.

ID	Action	SourceXml	ExpectedHtml
Extref	crossref	<extref id='abc' />	abc
TTerm	crossref	<testterm id='abc' />	abc
TTerms	crossref	<testterms id='abc' />	abcs

В этом случае используется достаточно простой интерпретатор, основанный на уже существующей логике *параметризованных тестов* (Parametrized Test). Эта версия читает файл CSV и с помощью функции `split` языка Ruby разбивает каждую строку на элементы.

```

def test_crossref
  executeDataDrivenTest "CrossrefHandlerTest.txt"
end
def executeDataDrivenTest filename
  dataFile = File.open(filename)
  dataFile.each_line do | line |
    desc, action, part2 = line.split(",")
    sourceXml, expectedHtml, leftOver = part2.split(",")
    if "crossref"==action.strip
      generateAndVerifyHtml sourceXml, expectedHtml, desc
    else # new "verbs" go before here as elsif's
      report_error("unknown action" + action.strip)
    end
  end
end

```

Если не изменить реализацию `generateAndVerifyHtml` для перехвата ложных утверждений и увеличения значения счетчика, тест завершит работу после первого же ложного утверждения. Хотя такое поведение может потребоваться во время регрессионного тестирования, оно не обеспечивает хорошей *локализации дефектов* (Defect Localization).

Пример: управляемый данными тест (Data-Driven Test) на основе инфраструктуры Fit

Если необходим еще больший контроль над действиями пользователя, можно создать “тестовую конфигурацию столбца” для инфраструктуры Fit со столбцами “`id`”, “`action`”, “`source XML`” и “`expected Html()`”, позволяющую пользователю редактировать Web-страницу с помощью HTML (табл. 18.1).

Таблица 18.1. Управляемый данными тест (Data-Driven Test) на основе инфраструктуры Fit

com.xunitpatterns.fit.CrossrefHandlerFixture			
id	action	source XML	expected Html()
extref	crossref	<extref id='abc' />	<a href='abc.html'?abc?
TestTerm	crossref	<testterm id='abcd' />	<a href='abc.html'?abc?
TestTermPlural	crossref	<testterm id='abc' suffix="s"/>	<a href='abc.html'?abcs?

При использовании инфраструктуры Fit интерпретатор представляет собой расширение инфраструктуры с помощью класса тестовой конфигурации.

```
public class CrossrefHandlerFixture extends ColumnFixture {
    // Столбцы параметров
    public String id;
    public String action;
    public String sourceXML;
    // Столбцы результатов
    public String expectedHtml() {
        return generateHtml(sourceXML);
    }
}
```

Методы класса тестовой конфигурации вызываются инфраструктурой Fit для каждой ячейки в каждой строке таблицы в зависимости от заголовка столбца. Простые заголовки интерпретируются как переменные экземпляра (например, “`id`” и “`source XML`”). Имена столбцов, заканчивающиеся на “()”, означают функцию, результаты вызова которой будут сравниваться с содержимым ячейки.

Результаты работы тестов показаны в табл. 18.2. Цветовое кодирование ячеек таблицы позволяет быстро получить представление о результатах работы тестов.

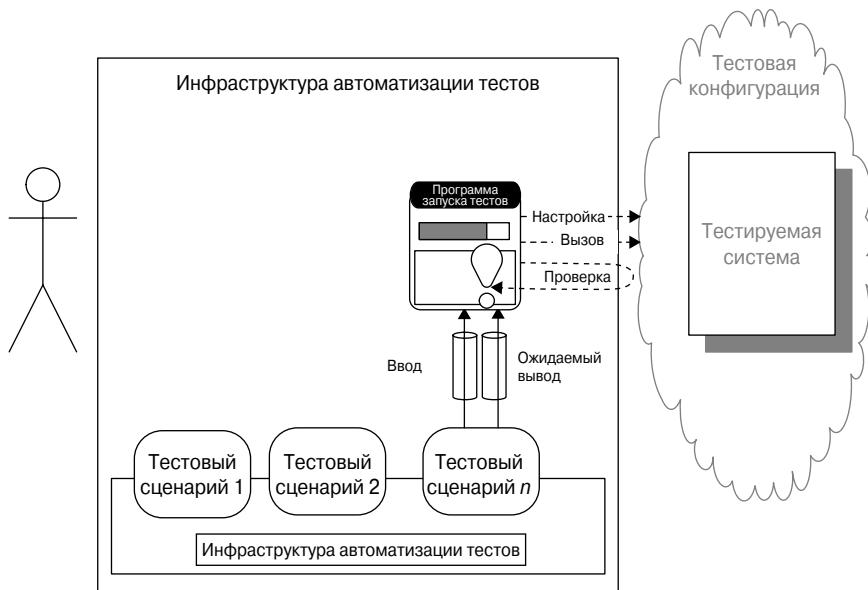
Таблица 18.2. Результаты работы теста Fit

com.xunitpatterns.fit.CrossrefHandlerFixture			
id	action	source XML	expected Html()
extref	crossref	<extref id='abc' />	<a href='abc.html'?abc?
TestTerm	crossref	<testterm id='abcd' />	<a href='abc.html'?abc?
TestTermPlural	crossref	<testterm id='abc' suffix="s"/>	<a href='abc.html'?abcs? <a href='abc.html'?abc?

Инфраструктура автоматизации тестов (Test Automation Framework)

Как упростить написание и запуск тестов разными разработчиками?

Для этого используется инфраструктура, обеспечивающая все механизмы для запуска логики тестов, а от разработчика требуется только написать эту логику.



Написание и запуск автоматизированных тестов состоит из нескольких этапов, но многие из них повторяются для множества тестов. Если каждый тест будет содержать реализацию этих этапов, создание автоматизированных тестов превратится в утомительный, долгий, содержащий ошибки и дорогостоящий процесс.

Используя *инфраструктуру автоматизации тестов* (Test Automation Framework), можно минимизировать усилия по написанию *полностью автоматизированных тестов* (Fully Automated Tests, с. 81).

Как это работает

Создается инфраструктура, реализующая все механизмы для запуска наборов тестов и записи результатов. Такие механизмы требуют возможности поиска отдельных тестов, их объединения в наборы и последовательного запуска каждого теста, проверки ожидаемых результатов, вывода информации об ошибках и неудачных завершениях, а также очистки после неудачного завершения или ошибок в тестах. Инфраструктура предоставляет способ подключения поведения тестов, описанного разработчиками.

Зачем это нужно

Создание *полностью автоматизированных тестов* (Fully Automated Tests), обладающих свойствами повторяемости и простой модифицируемости, намного сложнее создания простых сценариев, вызывающих тестируемую систему. Приходится обрабатывать успешные и ошибочные состояния, как ожидаемые, так и неожиданные. Приходится обеспечивать создание и очистку **тестовой конфигурации** (test fixture), выбирать запускаемые тесты и сообщать о результатах запуска набора тестов.

Объем трудозатрат на написание *полностью автоматизированных тестов* (Fully Automated Tests) может стать серьезным препятствием при автоматизации тестирования. Начальные затраты можно значительно снизить за счет использования инфраструктуры, в которой реализованы основные функции; в таком случае начальные затраты заключаются в ознакомлении с возможностями инфраструктуры. В свою очередь, эти затраты можно сократить, если инфраструктура содержит реализацию распространенного протокола, например xUnit. В этом случае значительно упрощается изучение второй и третьей инфраструктур после изучения первой.

Кроме того, инфраструктура позволяет изолировать реализацию необходимой для запуска тестов логики от самих тестов. Такой подход позволяет сократить *дублирование тестового кода* (Test Code Duplication, с. 254) и снизить вероятность появления *непонятных тестов* (Obscure Test, с. 230). Также написанные разными разработчиками тесты можно запускать вместе и получать результаты в одном отчете.

Замечания по реализации

Существует множество разновидностей *инфраструктур автоматизации тестов* (Test Automation Framework) как от коммерческих разработчиков, так и распространяемых с открытыми исходными текстами. Все их можно отнести к двум основным категориям: утилиты работы с “роботом”-пользователем и *тесты на основе сценария* (Scripted Test, с. 319). Вторую категорию можно разделить на семейства xUnit и *управляемые данными тесты* (Data-Driven Test, с. 322).

Вариант: инфраструктуры тестов на основе “робота”-пользователя

Множество инструментов автоматизации тестов предназначено для тестирования приложений через пользовательский интерфейс. В большинстве из них применяется метафора “записи и воспроизведения” тестов. Использование такой метафоры ведет к появлению очень соблазнительных маркетинговых материалов, поскольку автоматизация тестирования может показаться такой же простой, как выполнение тестов вручную в процессе записи. Такие инструменты для работы с “роботизированными” пользователями состоят из двух основных компонентов: устройства записи тестов, которое отслеживает и записывает взаимодействие пользователя с тестируемой системой, и устройства воспроизведения, которое воспроизводит *записанные тесты* (Recorded Test, с. 312). Большинство таких инструментов автоматизации представляет собой инфраструктуру, поддерживающую множество встраиваемых модулей “распознавания элементов управления”. В большинстве коммерческих продуктов предоставляется целый набор таких модулей.

Вариант: инфраструктуры семейства xUnit

Большинство инструментов модульного тестирования принадлежит семейству xUnit, предназначенному для автоматизации *созданных вручную сценарииев теста* (Hand-Scripted Test). Инфраструктура xUnit была перенесена (или разработана заново) на большинство современных языков программирования. Семейство xUnit состоит из нескольких основных компонентов. Самым заметным компонентом является *программа запуска тестов* (Test Runner, с. 405), которая вызывается как из командного интерпретатора, так и из графического интерфейса. После запуска программа *создает объекты теста* (Testcase Object, с. 410), собирает их в *объект набора тестов* (Test Suite Object, с. 414) и вызывает каждый *тестовый метод* (Test Method, с. 378). Еще одним основным компонентом реализации xUnit является библиотека встроенных *методов с утверждением* (Assertion Method, с. 390), используемых внутри *тестовых методов* (Test Method) для описания ожидаемого результата каждого теста.

Вариант: инфраструктура управляемых данными тестов

Такая инфраструктура обеспечивает возможность подключения интерпретаторов, знающих, как обрабатывать конкретный тип теста. Такая гибкость расширяет формат входного файла новыми “глаголами” и объектами. Также подобные инфраструктуры предоставляют программу запуска тестов, читающую содержимое файла и передающую управление подключаемому интерпретатору при обнаружении соответствующего формата. При этом программа отслеживает общую статистику выполнения тестов. Самым заметным представителем данного семейства инфраструктур является Fit, позволяющий разработчикам описывать тесты в табличном формате и “подключать” классы тестовых конфигураций, знающие, как обрабатывать конкретные форматы таблиц.

Пример: инфраструктура автоматизации тестов

Для каждого из способов автоматизации тестов *инфраструктура автоматизации тестов* (Test Automation Framework) выглядит по-своему. Примеры вариантов рассматриваются в разделах, посвященных *записанным тестам* (Recorded Test), *тестам на основе сценария* (Scripted Test) и *управляемым данными тестам* (Data-Driven Test).

Источники дополнительной информации

Среди наиболее распространенных примеров *инфраструктуры автоматизации тестов* (Test Automation Framework) можно выделить JUnit (Java), SUnit (Smalltalk), CppUnit (C++), NUnit (все языки для платформы .NET), runit (Ruby), PyUnit (Python) и VbUnit (Visual Basic). Более полный и обновленный список можно получить по адресу <http://xprogramming.com>, где также приводится перечень доступных расширений (например, HttpUnit, Cactus и т.д.).

Существуют и другие инфраструктуры автоматизации с открытым исходным кодом, включая Fit, Canoo WebTest и Watir. Среди коммерчески доступных систем можно выделить QTP, BPT и eCATT.

В книге [TDD-BE] Кент Бек демонстрирует применение разработки на основе тестов в процессе создания *инфраструктуры автоматизации тестов* (Test Automation Framework) на языке программирования Python. Основываясь на так называемом под-

ходе “нейрохирургии на собственном мозгу”, он использует уже готовые фрагменты инфраструктуры для запуска тестов, проверяющих ее новые возможности. Подобное приложение является хорошим примером как процесса разработки на основе тестов, так и самозагрузки.

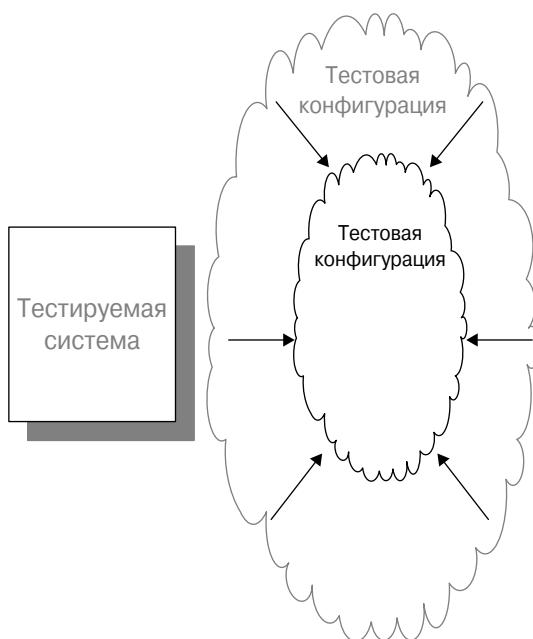
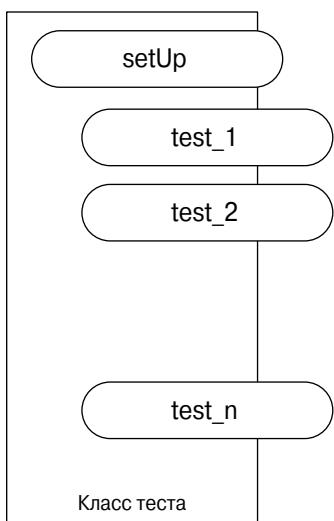
Минимальная тестовая конфигурация (Minimal Fixture)

Также известен как:

Минимальный контекст
(*Minimal Context*)

Какую стратегию тестовой конфигурации необходимо использовать?

Для каждого теста должна использоваться самая минимальная и простая возможная тестовая конфигурация.



Каждому тесту нужна тестовая конфигурация. Для понимания сути теста очень важно понимание структуры тестовой конфигурации и ее влияния на ожидаемый результат теста. Чем меньше тестовая конфигурация, тем проще понять тест.

Зачем это нужно

Минимальная тестовая конфигурация (*Minimal Fixture*) помогает использовать *тесты как документацию* (*Tests as Documentation*, с. 79) и избежать появления *медленных тестов* (*Slow Tests*, с. 289). Тесты на основе *минимальной тестовой конфигурации* (*Minimal Fixture*) всегда кажутся более понятными, чем тесты с конфигурацией, перегруженной ненужной информацией. Это справедливо как для *новой тестовой конфигурации* (*Fresh Fixture*, с. 344), так и для *общей тестовой конфигурации* (*Shared Fixture*, с. 350), хотя создание минимальной конфигурации во втором случае обычно требует больших усилий. Это связано с необходимостью учитывать потребности нескольких тестов сразу. Если конфигурация предназначена только для одного теста, ее намного проще сделать минимальной.

Замечания по реализации

При проектировании в тестовую конфигурацию должны включаться только те объекты, которые необходимы для выражения проверяемого тестом поведения. Другими словами, “если объект не нужен для понимания теста, старайтесь не включать его в тестовую конфигурацию”.

Для создания *минимальной тестовой конфигурации* (Minimal Fixture) необходимо безжалостно удалить все, что не описывает ожидаемое поведение тестируемой системы. Можно рассмотреть два варианта “минимизации”.

- Объекты могут полностью удаляться, т.е. они даже не создаются как часть конфигурации. Если объект не нужен для доказательства некоторого поведения системы, он просто не включается в конфигурацию.
- Можно скрывать ненужные атрибуты объектов, если они не участвуют в описании ожидаемого поведения.

Простым способом определения необходимости объекта является его удаление. Если после удаления тест завершается неудачно, значит, объект был необходим. Конечно, он мог использоваться в качестве аргумента ненужного метода или никогда не используемого атрибута (даже если объект, которому принадлежит атрибут, каким-то образом используется). Включение таких объектов в тестовую конфигурацию повышает вероятность появления *непонятных тестов* (Obscure Test, с. 230). Избавиться от ненужных объектов можно одним из двух способов: сокрытием или удалением зависимости за счет передачи *объекта-заглушки* (Dummy Object, с. 730) либо использования *обрезания цепочки существостей* (Entity Chain Snipping; см. *Тестовая заглушка*, Test Stub, с. 544). Но если тестируемая система действительно обращается к объекту во время работы тестируемой логики, может потребоваться сохранение объекта как части тестовой конфигурации.

Определив, что объект необходим для нормальной работы теста, нужно задать себе вопросы: “Насколько объект упрощает понимание теста?”, “Если объект инициализируется “за кулисами”, усложнит ли это понимание?”, “Приведет ли это к появлению *непонятного теста* (Obscure Test), превратив объект в *тайноменного гостя* (Mystery Guest)?” Если это так, объект должен оставаться видимым. Границы значения являются хорошим примером необходимости сохранения видимости объектов и атрибутов.

Если объект или атрибут не нужен для понимания теста, необходимо приложить все усилия для его удаления из *тестового метода* (Test Method), но необязательно из тестовой конфигурации. Часто для этого используются *методы создания* (Creation Method, с. 441). Для сокрытия не влияющих на результат теста атрибутов объектов можно воспользоваться *методами создания* (Creation Method), которые присваивают значения всем неинтересным атрибутам. Кроме того, внутри метода создания можно скрыть создание вызываемых объектов. Характерным примером такого подхода является создание тестов, принимающих в качестве параметров некорректно сформированные объекты (для тестиования системы с неправильными входными данными). В таком случае желательно не усложнять описание всеми правильными атрибутами передаваемого объекта. Вместо этого основное внимание следует уделять некорректному атрибуту. В таком случае используется шаблон *единственный дефектный атрибут* (One Bad Attribute; см. *Вычисляемое значение*, Derived Value, с. 722), создающий некорректный объект путем вызова *метода создания* (Creation Method) для генерации правильного объекта с последующей заменой значения атрибута — некорректным значением.

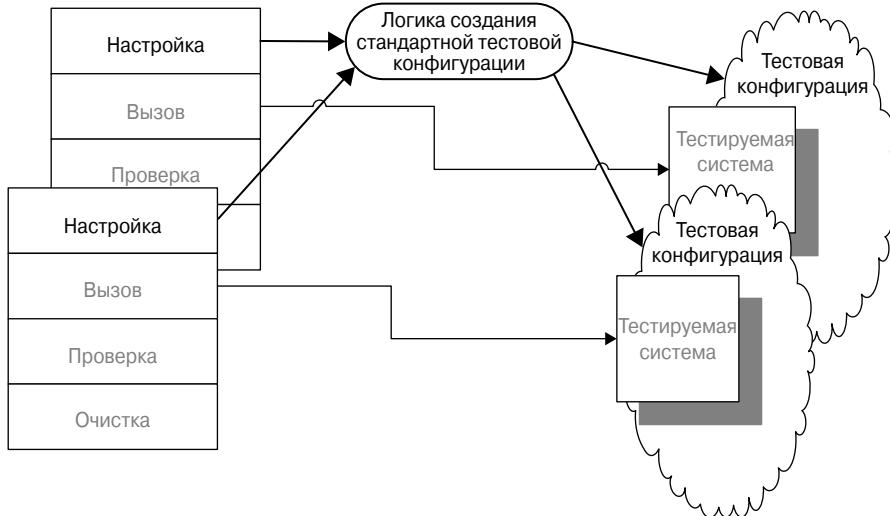
Стандартная тестовая конфигурация (Standard Fixture)

Также известен как:

Стандартный контекст
(Standard Context)

Какую стратегию тестовой конфигурации лучше использовать?

Она и та же структура тестовой конфигурации используется множеством тестов.



Для запуска автоматизированных тестов нужна понятная и полностью определенная тестовая конфигурация. Проектирование конфигурации для каждого теста требует дополнительных трудозатрат. *Стандартная тестовая конфигурация* (Standard Fixture) обеспечивает возможность повторного использования одной и той же структуры конфигурации несколькими тестами. При этом необязательно происходит совместное использование экземпляров тестовой конфигурации.

Как это работает

Стандартная тестовая конфигурация (Standard Fixture) больше касается субъектививного отношения, чем технологий. На раннем этапе должно быть принято решение о проектировании *стандартной тестовой конфигурации* (Standard Fixture), которая будет использоваться множеством тестов, а не об извлечении общей структуры тестовой конфигурации из независимо спроектированных тестов. В некотором смысле *стандартная тестовая конфигурация* (Standard Fixture) является результатом “проектирования наперед” тестовой конфигурации для всего набора тестов. После этого на основе общего дизайна конфигурации можно определять конкретные тесты.

Выбор *стандартной тестовой конфигурации* (Standard Fixture) никак не связан с выбором между *новой тестовой конфигурацией* (Fresh Fixture, с. 344) и *общей тестовой конфигурацией* (Shared Fixture, с. 350). *Общая тестовая конфигурация* (Shared Fixture) по определению является *стандартной тестовой конфигурацией* (Standard Fixture). Обратное

не всегда верно, так как *стандартная тестовая конфигурация* (Standard Fixture) основное внимание уделяет повторному использованию дизайна, а не времени создания или видимости тестовой конфигурации. После принятия решения об использовании *стандартной тестовой конфигурации* (Standard Fixture) остается решить, будет ли каждый тест создавать собственный экземпляр конфигурации или создание будет происходить один раз и конфигурация будет использоваться всеми тестами.

Когда это использовать

Просматривая ранний черновик этой книги, редактор серии Мартин Фаулер задал мне вопрос: “Неужели кто-то так делает?” Этот вопрос демонстрирует водораздел между философиями проектирования тестовой конфигурации. Опыт в гибких процессах разработки заставляет Мартина разрабатывать тесты, “вытаскивающие” тестовую конфигурацию в объективную реальность. Если нескольким тестам требуется одна и та же тестовая конфигурация, она выносится в метод `setUp` и класс делится на несколько классов теста для каждой тестовой конфигурации (Testcase Class per Fixture, с. 639). Мартину даже не приходит в голову создавать *стандартную тестовую конфигурацию* (Standard Fixture), которая будет использоваться всеми тестами. Так кто же пользуется такими конструкциями?

Стандартная тестовая конфигурация (Standard Fixture) является традиционным инструментом в подразделениях контроля качества. Часто определяется большая *стандартная тестовая конфигурация* (Standard Fixture), на основе которой и происходит тестирование. Такой подход вполне понятен и допустим при тестировании вручную (особенно при запуске большого количества приемочных тестов), так как тестерам не приходится тратить много времени на настройку тестовой среды для каждого приемочного теста. При этом несколько тестеров могут одновременно работать в одной тестовой среде. Некоторые разработчики тестов используют *стандартные тестовые конфигурации* (Standard Fixture) для определения автоматизированных приемочных тестов. По очевидным причинам такая стратегия преобладает при использовании *общей тестовой конфигурации* (Shared Fixture).

В сообществе xUnit использование *стандартной тестовой конфигурации* (Standard Fixture) только для того, чтобы избежать проектирования *минимальной тестовой конфигурации* (Minimal Fixture, с. 336) для каждого теста, считается нежелательным и получило отдельное название: *тестовая конфигурация общего характера* (General Fixture; см. *Непонятный тест*, Obscure Test, с. 230). Более приемлемым считается использование *неявной настройки* (Implicit Setup, с. 449) совместно с классами теста для каждой тестовой конфигурации (Testcase Class per Fixture), так как лишь несколько *тестовых методов* (Test Method, с. 378) совместно используют структуру тестовой конфигурации. И происходит это только потому, что им нужна одна и та же структура. Увеличение размеров и сложности *стандартной тестовой конфигурации* (Standard Fixture) делает ее более приспособленной для тестов с разными потребностями. Сохранение этой тенденции приводит к появлению “хрупкой” тестовой конфигурации (Fragile Fixture; см. “Хрупкий” тест, Fragile Test, с. 277), так как требования новых тестов могут вносить изменения, нарушающие совместимость с существующими клиентами конфигурации. Кроме того, разработчик может столкнуться с необходимостью развлекать *тайноменного гостя* (Mystery Guest), если причинно-следственная связь между конфигурацией и результатом оказывается неочевидной, так как настройка конфигурации скрыта от теста или не ясно, какие характеристи-

стики *стандартной тестовой конфигурации* (Standard Fixture) выступают в роли предварительных условий теста.

Также *стандартная тестовая конфигурация* (Standard Fixture) требует больше времени на создание, чем *минимальная тестовая конфигурация* (Minimal Fixture), из-за большего объема данных. Создание *новой тестовой конфигурации* (Fresh Fixture) для каждого *объекта теста* (Testcase Object, с. 410) может привести к появлению *медленных тестов* (Slow Tests, с. 289), особенно если в процессе принимает участие база данных (см. врезку “Правила для модульных тестов”, в которой приводится мнение о допустимом поведении модульных тестов).

ПРАВИЛА ДЛЯ МОДУЛЬНЫХ ТЕСТОВ

Майкл Фезерс на сайте Object Mentor (www.objectmentor.com) изложил следующее.

Эти правила применялись в нескольких командах. Правила стимулируют хороший дизайн и быструю обратную связь, помогая командам разработчиков избежать серьезных проблем.

Тест не является модульным, если он:

- ⇒ взаимодействует с базой данных;
- ⇒ взаимодействует с другими системами по сети;
- ⇒ взаимодействует с файловой системой;
- ⇒ не может нормально работать с другими модульными тестами;
- ⇒ требует модификации среды (редактирования конфигурационных файлов) для своего запуска.

Попадающие под это описание тесты не являются плохими. Часто их имеет смысл писать, и это можно сделать, воспользовавшись “обвязкой” для модульных тестов. Но важно четко отделить их от истинных модульных тестов, чтобы знать, какие тесты можно быстро запускать после каждой модификации кода.

По этим причинам более оправдано использование *минимальной тестовой конфигурации* (Minimal Fixture), так как она позволяет избежать дополнительных накладных расходов на настройку конфигурации при чтении объектов, необходимых другим тестам.

Замечания по реализации

Как было показано ранее, *стандартная тестовая конфигурация* (Standard Fixture) может использоваться как *новая тестовая конфигурация* (Fresh Fixture) или как *общая тестовая конфигурация* (Shared Fixture), а для настройки может использоваться как *неявная настройка* (Implicit Setup) или *делегированная настройка* (Delegated Setup, с. 437). (Использование *встроенной настройки* (In-line Setup, с. 434) в данном случае может показаться наивным — код создания *стандартной тестовой конфигурации* (Standard Fixture) пришлось бы копировать в каждый *тестовый метод* (Test Method).) При использовании в качестве *новой тестовой конфигурации* (Fresh Fixture) можно определить *вспомогательный метод теста* (Test Utility Method, с. 610) (функцию или процедуру), создающий *стандартную тестовую конфигурацию* (Standard Fixture). После этого вспомогательный метод будет вызываться из каждого теста, требующего конфигурацию с определенной структурой. Как вариант можно использовать поддержку *неявной настройки* (Implicit

`Setup`), предоставляемой инфраструктурой `xUnit`, для вынесения всей логики создания тестовой конфигурации в метод `setUp`.

При использовании *стандартной тестовой конфигурации* (Standard Fixture) в виде *общей тестовой конфигурации* (Shared Fixture) можно воспользоваться любым из доступных для общей конфигурации шаблонов создания, включая *настройку тестовой конфигурации набора* (Suite Fixture Setup, с. 465), “ленивую” настройку (Lazy Setup, с. 460) и декоратор *настройки* (Setup Decorator, с. 471).

Мотивирующий пример

Как было показано ранее, с большой вероятностью использование *стандартной тестовой конфигурации* (Standard Fixture) будет заложено в самом начале проекта — причиной такого решения обычно становится опыт одного из участников. Скорее всего, предназначенные для использования *минимальной тестовой конфигурации* (Minimal Fixture) тесты не будут перерабатываться для использования *стандартной тестовой конфигурации* (Standard Fixture), кроме случаев перехода к созданию *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture). В целях демонстрации предположим, что выполнены все операции по переходу “отсюда” “туда”. В следующем примере используется *метод создания* (Creation Method, с. 441) для создания отдельной *новой тестовой конфигурации* (Fresh Fixture) для каждого теста.

```
public void testGetFlightsByFromAirport_OneOutboundFlight_c()
    throws Exception {
    FlightDto outboundFlight = createOneOutboundFlightDto();
    // Вызов системы
    List flightsAtOrigin =
        facade.getFlightsByOriginAirport(
            outboundFlight.getOriginAirportId());
    // Проверка результата
    assertOnly1FlightInDtoList("Flights at origin",
        outboundFlight,
        flightsAtOrigin);
}
public void testGetFlightsByFromAirport_TwoOutboundFlights_c()
    throws Exception {
    FlightDto[] outboundFlights =
        createTwoOutboundFlightsFromOneAirport();
    // Вызов системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlights[0].getOriginAirportId());
    // Проверка результата
    assertExactly2FlightsInDtoList("Flights at origin",
        outboundFlights,
        flightsAtOrigin);
}
```

За счет использования *делегированной настройки* (Delegated Setup) для внесения в тестируемую систему *минимальной тестовой конфигурации* (Minimal Fixture) для каждого теста удается сохранить небольшой объем теста. Код создания конфигурации можно было бы включить в каждый метод, но это стало бы движением по направлению к *непонятному тесту* (Obscure Test).

Замечания по рефакторингу

Технически преобразование нескольких тестов для использования *стандартной тестовой конфигурации* (Standard Fixture) не является “рефакторингом”, так как приводит к изменению поведения тестов. Самой большой проблемой является проектирование *стандартной тестовой конфигурации* (Standard Fixture) таким образом, чтобы каждый *тестовый метод* (Test Method) мог найти необходимый ему фрагмент конфигурации. Это значит, что все *минимальные тестовые конфигурации* (Minimal Fixture) должны быть объединены в одну большую. Не удивительно, что переработка такого кода может оказаться нетривиальной задачей при достаточно большом количестве тестов.

Простая и механическая часть рефакторинга заключается в преобразовании логики создания конфигурации в каждом teste в вызовы *методов поиска* (Finder Method; см. *Вспомогательный метод теста*, Test Utility Method), получающие соответствующую часть *стандартной тестовой конфигурации* (Standard Fixture). Такое преобразование лучше выполнять в виде последовательности операций. Сначала необходимо выделить встроенную логику создания конфигурации из каждого *тестового метода* (Test Method) в один или несколько *методов создания* (Creation Method) с описательными именами. После этого необходимо глобально заменить фрагмент “создание” в каждом вызове фрагментом “найти”. Наконец, необходимо сгенерировать (вручную или с помощью функции среды разработки) *методы поиска* (Finder Method), что позволит откомпилировать вызовы. Внутрь каждого *метода поиска* (Finder Method) вставляется код, возвращающий соответствующую часть *стандартной тестовой конфигурации* (Standard Fixture).

Пример: стандартная тестовая конфигурация (Standard Fixture)

Ниже приведен рассмотренный ранее пример, преобразованный для использования *стандартной тестовой конфигурации* (Standard Fixture).

```
public void testGetFlightsByFromAirport_OneOutboundFlight()
    throws Exception {
    setupStandardAirportsAndFlights();
    FlightDto outboundFlight = findOneOutboundFlight();
    // Вызов системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlight.getOriginAirportId());
    // Проверка результата
    assertOnly1FlightInDtoList("Flights at origin",
        outboundFlight,
        flightsAtOrigin);
}
public void testGetFlightsByFromAirport_TwoOutboundFlights()
    throws Exception {
    setupStandardAirportsAndFlights();
    FlightDto[] outboundFlights =
        findTwoOutboundFlightsFromOneAirport();
    // Вызов системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlights[0].getOriginAirportId());
    // Проверка результата
    assertExactly2FlightsInDtoList("Flights at origin",
        outboundFlights,
        flightsAtOrigin);
}
```

Чтобы сделать использование *стандартной тестовой конфигурации* (Standard Fixture) полностью очевидным, в этом примере показано создание *новой тестовой конфигурации* (Fresh Fixture) в каждом тесте с помощью вызова одного и того же *метода создания* (Creation Method) для получения *стандартной тестовой конфигурации* (Standard Fixture) (т.е. с использованием *делегированной настройки*, Delegated Setup). Того же эффекта можно достичь за счет размещения логики создания конфигурации в методе `setUp`, таким образом воспользовавшись *неявной настройкой* (Implicit Setup). Полученный в результате тест будет идентичен тесту, использующему *общую тестовую конфигурацию* (Shared Fixture).

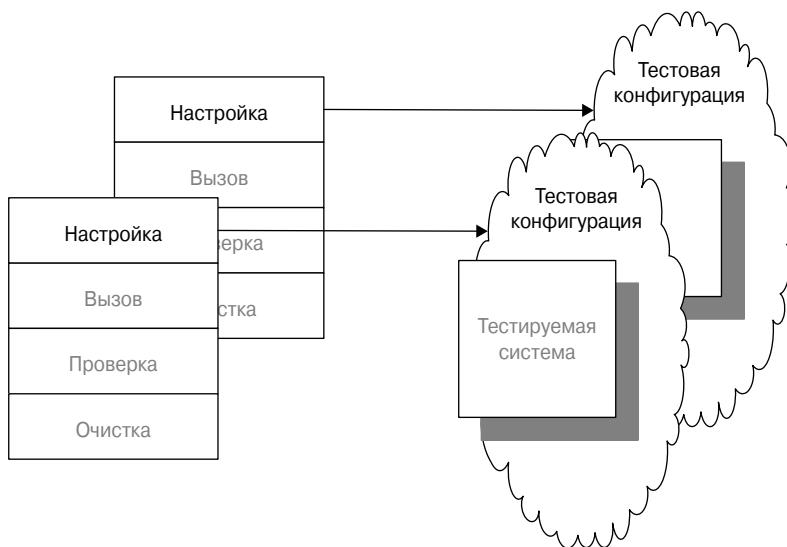
Новая тестовая конфигурация (Fresh Fixture)

Также известен как:

*Новый контекст (Fresh Context),
Закрытая тестовая конфигурация (Private Fixture)*

Какую стратегию тестовой конфигурации следует использовать?

Каждый тест создает собственную новую тестовую конфигурацию для единоличного использования.



Для каждого теста нужна тестовая конфигурация. Она определяет состояние среды до запуска теста. Выбор между созданием новой конфигурации при каждом запуске и повторным использованием созданной ранее конфигурации является ключевым решением при автоматизации тестов.

Если каждый тест создает *новую тестовую конфигурацию* (Fresh Fixture), вероятность появления *нестабильных тестов* (Erratic Test, с. 267) снижается и усилия по созданию тестов, скорее всего, приведут к использованию *тестов как документации* (Tests as Documentation, с. 79).

Как это работает

Тестовая конфигурация проектируется и создается таким образом, чтобы экземпляр использовался один раз одним тестом. Создание конфигурации может происходить во время работы теста, а очистка — после завершения работы. Остатки конфигурации после других тестов или предыдущих запусков этого теста не используются. Таким образом, каждый тест начинается и заканчивается с “чистым состоянием”.

Когда это использовать

Новая тестовая конфигурация (Fresh Fixture) должна использоваться каждый раз, когда необходимо избежать зависимостей между тестами, приводящих к появлению таких нестабильных тестов (Erratic Test), как *одинокий тест* (Lonely Test) и *взаимодействующие тесты* (Interacting Tests). Если нельзя использовать *новую тестовую конфигурацию* (Fresh Fixture) из-за замедления работы тестов, перед переходом к *общей тестовой конфигурации* (Shared Fixture) имеет смысл рассмотреть использование *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture). Обратите внимание, что использование *схемы разбиения базы данных* (Database Partitioning Scheme) для создания выделенной “песочницы” с базой данных (Database Sandbox, с. 658) для конкретного теста не считается использованием *новой тестовой конфигурации* (Fresh Fixture), так как при последующих запусках тест сможет обращаться к той же конфигурации.

Замечания по реализации

Конфигурация считается *новой тестовой конфигурацией* (Fresh Fixture), если планируется использовать ее один раз. Временное или постоянное существование конфигурации зависит от природы test-типа системы и от структуры написанных тестов (рис. 18.1). Несмотря на одно и то же назначение, реализации постоянной *новой тестовой конфигурации* (Fresh Fixture) отличаются. По большей части настройка тестовой конфигурации не затрагивается, поэтому настройка считается общей для всех типов такой конфигурации. Очистка тестовой конфигурации зависит от ее типа.

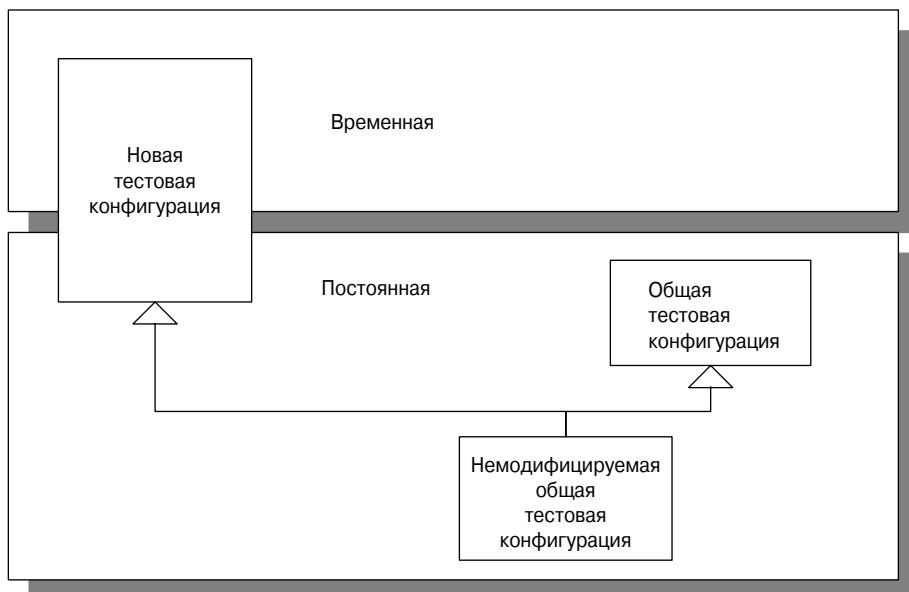


Рис. 18.1. Стратегии тестовой конфигурации. Конфигурация может быть новой, общей или скомбинированной из этих двух типов (немодифицируемой общей конфигурацией), когда некоторая часть сохраняется между запусками тестов

Почему сохраняется конфигурация

Есть две причины, по которым созданная конфигурация может продолжать существование после завершения работы *тестового метода* (Test Method, с. 378). Если конфигурация содержит состояние других объектов или компонентов, от которых зависит тестируемая система, ее сохранение зависит от сохранения этих объектов или компонентов. Одним из примеров является база данных. Если код сохранил объект конфигурации в базе, объект будет “присутствовать” длительное время после завершения работы теста. Их существование в базе данных делает возможными коллизии между несколькими экземплярами теста (*неповторяемый тест*, Unrepeatable Test). Другие тесты также могут получать доступ к этим объектам, что приводит к появлению других вариантов *нестабильных тестов* (Erratic Test), например *взаимодействующих тестов* (Interacting Tests) или “войн” запуска тестов (Test Run War). Если приходится использовать базу данных или другой способ сохранения объектов, необходимо предпринять меры для изоляции тестовой конфигурации. Кроме того, необходимо очищать конфигурацию после завершения работы каждого *тестового метода* (Test Method).

Вторая причина сохранения тестовой конфигурации заключается в механизме управления тестами, а именно, в типе переменной, которая содержит ссылку на конфигурацию. Локальные переменные естественно выходят из области видимости при завершении работы *тестового метода* (Test Method). Таким образом, любая тестовая конфигурация в локальной переменной будет удалена сборщиком мусора. Переменные экземпляров выходят из области видимости при удалении *объекта теста* (Testcase Object) и требует явной очистки, только если инфраструктура xUnit не создает повторно *объект теста* (Testcase Object) при каждом запуске теста². С другой стороны, использование переменных класса обычно приводит к появлению постоянных тестовых конфигураций, живущих дольше конкретного тестового метода или всего набора, а значит, необходимо избегать их использования совместно с *новой тестовой конфигурацией* (Fresh Fixture).

На практике конфигурация модульных тестов обычно не сохраняется³, кроме случаев, когда логика приложений тесно связана с базами данных. Сохранение конфигурации более вероятно при написании приемочных или компонентных тестов.

Настройка новой тестовой конфигурации

Процесс настройки практически не отличается для постоянных и временных конфигураций. Основным отличием является расположение настраивающего кода. *Встроенная настройка* (In-line Setup, с. 434) может использоваться для простой логики создания тестовой конфигурации. Для более сложных конфигураций предпочтительно использование *делегированной настройки* (Delegated Setup, с. 437), когда тестовые методы организованы в *класс теста для каждого класса* (Testcase Class per Class, с. 627) или *класс теста для ка-*

² В большинстве реализаций xUnit для каждого *тестового метода* (Test Method) создается отдельный *объект теста* (Testcase Object, с. 410). Но такой подход используется не во всех реализациях. Эта разница может ввести в заблуждение разработчика тестов при первом знакомстве с конкретной реализацией, так как переменные экземпляра могут неожиданно начать вести себя, как переменные класса. Подробное описание этой проблемы приводится во врезке “Всегда есть исключения” на с. 411.

³ Признаки модульных тестов приводились во врезке “Правила для модульных тестов” на с. 340.

жной функции (Testcase Class per Feature, с. 633). Если тесты организованы в виде *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639), для создания конфигурации можно использовать *неявную настройку* (Implicit Setup, с. 449).

Вариант: временная новая тестовая конфигурация (Transient Fresh Fixture)

Если на тестовую конфигурацию приходится ссылаться из нескольких мест теста, для ссылок необходимо использовать только локальные переменные или переменные экземпляров. В большинстве случаев для очистки конфигурации можно положиться на *очистку со сборкой мусора* (Garbage-Collected Teardown, с. 518), не прилагая к этому собственных усилий.

Обратите внимание, что *стандартная тестовая конфигурация* (Standard Fixture, с. 338) также может быть *новой тестовой конфигурацией* (Fresh Fixture), если она создается заново перед запуском каждого *тестового метода* (Test Method). При таком подходе повторно используется структура конфигурации, а не ее экземпляры. Такая ситуация часто наблюдается, когда используется *неявная настройка* (Implicit Setup), но тестовые методы не организованы в виде *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture).

Вариант: постоянная новая тестовая конфигурация (Persistent Fresh Fixture)

Если пришлось использовать *постоянную новую тестовую конфигурацию* (Persistent Fresh Fixture), нужно или выполнять очистку, или принимать меры для того, чтобы ее избежать. Для удаления конфигурации можно применять *встроенную очистку* (In-line Teardown, с. 527), *неявную очистку* (Implicit Teardown, с. 533), *делегированную очистку* (Delegated Teardown, с. 529) или *автоматическую очистку* (Automated Teardown, с. 521). Это позволит вернуть тестовую среду в то же состояние, в котором она была перед запуском тестов.

Для того чтобы избежать очистки для каждого уникального объекта конфигурации, можно использовать *отдельное сгенерированное значение* (Distinct Generated Value). Данная стратегия может стать основой *схемы разбиения базы данных* (Database Partitioning Scheme), изолирующей разные экземпляры тестов и программ их запуска. Такой подход позволит защититься от *утечки ресурсов* (Resource Leakage) в случае отказа процедуры очистки. Кроме того, подобный подход можно использовать в комбинации с одним из шаблонов очистки, чтобы гарантировать невозможность появления *неповторяемых тестов* (Unrepeatable Test) или *взаимодействующих тестов* (Interacting Tests).

Не удивительно, что дополнительные меры имеют и отрицательные стороны: тесты становятся сложнее в написании и медленнее в работе (Slow Tests, с. 289). Естественной реакцией является использование постоянства конфигурации через ее повторное использование несколькими тестами, а значит, избежание накладных расходов по ее созданию и очистке. К сожалению, такой подход имеет множество отрицательных сторон, так как нарушает один из основных принципов: *сохраняйте независимость тестов* (Keep Tests Independent, с. 96). Полученная в результате *общая тестовая конфигурация* (Shared Fixture) обязательно приведет к появлению *взаимодействующих тестов* (Interacting Tests) и *неповторяемых тестов* (Unrepeatable Test) (если не сразу, то через некоторое время). Не стоит двигаться по этому пути, не осознавая возможных последствий!

Мотивирующий пример

Ниже представлен пример *общей тестовой конфигурации* (Shared Fixture).

```
static Flight flight;
public void setUp() {
    if (flight == null)  ( // "Ленивая" настройка
        Airport departAirport = new Airport("Calgary", "YYC");
        Airport destAirport = new Airport("Toronto", "YYZ");
        flight = new Flight(flightNumber,
                            departAirport,
                            destAirport);
    )
}
public void testGetStatus_initial_S()  (
    // Неявная настройка
    // Вызывать тестируемую систему и проверить результат
    assertEquals(FlightState.PROPOSED, flight.getStatus());
    // Очистка
)
public void testGetStatus_cancelled()  (
    // Частично переопределенная неявная настройка
    flight.cancel();
    // Вызывать тестируемую систему и проверить результат
    assertEquals(FlightState.CANCELLED, flight.getStatus());
    // Очистка
)
```

Учитывая показанный код создания тестовой конфигурации, можно сделать вывод, что это нормальная *общая тестовая конфигурация* (Shared Fixture), но в данном случае ее можно использовать в качестве *предварительно созданной тестовой конфигурации* (Prebuilt Fixture, с. 454). В любом случае тесты могут начать взаимодействие в любой момент.

Замечания по рефакторингу

Предположим, что используется *общая тестовая конфигурация* (Shared Fixture) (один дизайн, один экземпляр) и необходим рефакторинг в сторону использования *новой тестовой конфигурации* (Fresh Fixture). Можно начать с рефакторинга теста для использования *стандартной тестовой конфигурации* (Standard Fixture) (один дизайн, несколько экземпляров). После этого можно продолжить развитие теста для использования *минимальной тестовой конфигурации* (Minimal Fixture, с. 336) за счет удаления лишней логики создания конфигурации через рефакторинг минимизации данных (Minimize Data, с. 739). Кроме того, на данном этапе можно сгруппировать *тестовые методы* (Test Method) с одинаковыми типами конфигурации в *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture) и использовать *неявную настройку* (Implicit Setup). Такое применение *стандартной тестовой конфигурации* (Standard Fixture) позволяет сократить количество минимальных конфигураций, которые необходимо придумать и создать.

Пример: новая тестовая конфигурация (Fresh Fixture)

Ниже представлен тот же тест после преобразования для использования *новой тестовой конфигурации* (Fresh Fixture) и защиты от *взаимодействия тестов* (Interacting Tests).

```
public void testGetStatus_initial()  (
    // Настройка
    Flight flight = createAnonymousFlight();
    // Вызов тестируемой системы и проверка результата
    assertEquals(FlightState.PROPOSED, flight.getStatus());
    // Очистка
    //     сборщиком мусора
)
public void testGetStatus_cancelled2()  (
    // Настройка
    Flight flight = createAnonymousCancelledFlight();
    // Вызов тестируемой системы и проверка результата
    assertEquals(FlightState.CANCELLED, flight.getStatus());
    // Очистка
    //     сборщиком мусора
)
```

Обратите внимание на использование *анонимных методов создания* (Anonymous Creation Method; см. *Метод создания*, Creation Method, с. 441) для генерации соответствующего состояния объекта Flight в каждом teste.

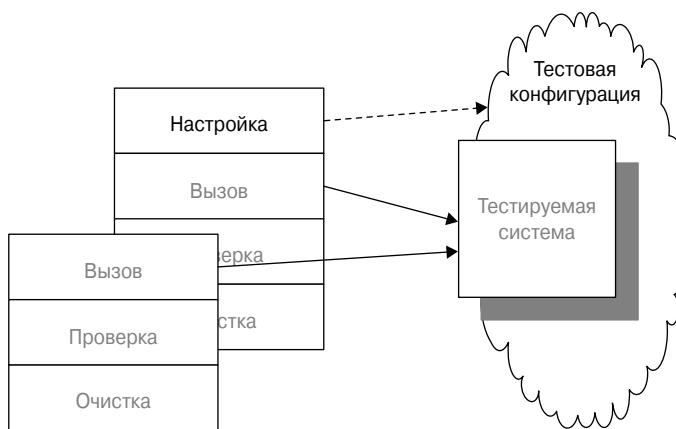
Общая тестовая конфигурация (Shared Fixture)

Также известен как:

Общий контекст (Shared Context), Оставшаяся тестовая конфигурация (Leftover Fixture), Повторно используемая тестовая конфигурация (Reused Fixture), Устаревшая тестовая конфигурация (Stale Fixture)

Как можно избежать появления медленных тестов? Какую стратегию стоит использовать?

Один и тот же экземпляр тестовой конфигурации используется несколькими тестами.



Для запуска автоматизированных тестов необходима полностью понятная и определенная тестовая конфигурация. Настройка *новой тестовой конфигурации* (Fresh Fixture, с. 344) может потребовать достаточно много времени, особенно при работе со сложными состояниями, сохраняемыми в тестовой базе данных.

Тесты можно заставить работать быстрее, используя одну конфигурацию для нескольких или многих тестов.

Как это работает

Основная концепция очень проста: создается *стандартная тестовая конфигурация* (Standard Fixture, с. 338), время жизни которой превышает время жизни *объекта теста* (Testcase Object, с. 410). Такой подход позволяет нескольким тестам использовать одну и ту же конфигурацию без ее удаления и повторного создания. Общая конфигурация может быть как *предварительно созданной* (Prebuilt Fixture), повторно используемой одним или несколькими тестами при многократном запуске, так и создаваемой тестом и используемой другими тестами при одновременном запуске. В любом случае ключевым критерием является повторное использование чужой тестовой конфигурации вместо своей. При этом в качестве конфигурации может выступать “обломок” после работы другого теста. В такой ситуации тесты работают быстрее, так как меньше времени уходит на создание конфигурации, что может сэкономить время разработчика за счет уменьшения трудозатрат на определение конфигурации для каждого теста.

Когда это использовать

Независимо от причин использования *общая тестовая конфигурация* (Shared Fixture) несет с собой багаж, требующий полного понимания еще до начала движения в этом направлении. Основной проблемой при использовании *общей тестовой конфигурации* (Shared Fixture) является взаимодействие между тестами. Потенциально это приводит к появлению *нестабильных тестов* (Erratic Test, с. 267), зависящих от результатов работы других тестов. Еще одной потенциальной проблемой является повышенная сложность конфигурации, предназначеннной для обслуживания нескольких тестов, по сравнению с *минимальной тестовой конфигурацией* (Minimal Fixture, с. 336), которая используется единственным тестом. Повышенная сложность требует больше времени на проектирование и приводит к *хрупкости* *тестовой конфигурации* (Fragile Fixture) впоследствии, когда ее приходится модифицировать.

Часто *общая тестовая конфигурация* (Shared Fixture) приводит к появлению *непонятных тестов* (Obscure Test, с. 230), так как создание конфигурации происходит за пределами теста. Данный потенциальный недостаток можно обойти за счет использования *методов поиска* (Finder Method; см. *Вспомогательный метод теста*, Test Utility Method, с. 610) с описательными именами, предоставляющими доступ к соответствующим фрагментам тестовой конфигурации.

Существуют правильные и неправильные причины использования *общей тестовой конфигурации* (Shared Fixture). Множество вариантов создавалось специально для компенсации негативных последствий использования *общей тестовой конфигурации* (Shared Fixture). Каковы же правильные причины ее использования?

Вариант: медленный тест (Slow Test)

Общая конфигурация используется в ситуациях, когда нет возможности создавать *новую тестовую конфигурацию* (Fresh Fixture) для каждого теста. Обычно это бывает, когда конфигурация требует для создания слишком большого объема вычислений, что в результате приводит к *замедлению работы тестов* (Slow Test, с. 289). Чаще всего это происходит при тестировании с настоящей базой данных, и стоимость создания записей слишком высока. Накладные расходы еще больше увеличиваются, если для создания данных используется программный интерфейс тестируемой системы, так как система часто проверяет правильность данных, а значит, читает только что записанные записи.

Ускорить работу тестов можно, полностью отказавшись от взаимодействия с базой данных. Более полное описание вариантов решения приводится в разделе о решении проблемы *медленных тестов* (Slow Test) и во врезке “Быстрые тесты без общей тестовой конфигурации”.

БЫСТРЫЕ ТЕСТЫ БЕЗ ОБЩЕЙ ТЕСТОВОЙ КОНФИГУРАЦИИ

Первой реакцией на обнаружение *медленных тестов* (Slow Tests, с. 289) является переход к использованию *общей тестовой конфигурации* (Shared Fixture, с. 350). Но существует еще несколько решений. В этой врезке описывается опыт, полученный в процессе работы над несколькими проектами.

ПОДДЕЛЬНАЯ БАЗА ДАННЫХ

В одном из ранних проектов с использованием экстремального программирования было написано множество тестов, обращающихся к базе данных. Сначала использовалась *общая тестовая конфигурация* (Shared Fixture). После обнаружения *взаимодействующих тестов* (Interacting Tests) и "войны" запуска тестов (Test Run War) мы перешли к *новой тестовой конфигурации* (Fresh Fixture, с. 344). Поскольку тестам требовался большой объем справочных данных, они работали достаточно долго. В среднем для каждой операции чтения или записи в базу данных со стороны тестируемой системы тесту приходилось выполнять еще несколько операций. Набор из нескольких сотен тестов полностью выполнялся примерно за 15 минут, что очень мешало частой и быстрой интеграции.

На тот момент для разделения кода и SQL использовался *уровень доступа к данным* (data access layer). Вскоре было обнаружено, что он позволяет заменить настоящую базу данных функционально эквивалентной *поддельной базой данных* (Fake Database; см. *Поддельный объект*, Fake Object, с. 565). Сначала для хранения объектов и ключей использовались простые хэш-таблицы. Такой подход позволил выполнять простые тесты "в памяти" без обращения к базе данных и значительно сократил время работы тестов.

Инфраструктура хранения данных поддерживала интерфейс запроса объектов. Впоследствии удалось создать интерпретатор запросов для базы данных на основе хэш-таблиц. В результате большая часть тестов работала в оперативной памяти, не обращаясь к другим ресурсам. В среднем тесты стали работать в 50 раз быстрее, чем при обращении к базе данных. Например, набор тестов, требовавший 10 минут для завершения, стал выполняться за 10 секунд.

Такой подход оказался настолько успешным, что эта инфраструктура тестирования была использована в нескольких последующих проектах. Использование поддельной инфраструктуры хранения данных позволяет отложить создание "настоящей базы данных" до момента стабилизации объектных моделей (это происходит через несколько месяцев после начала проекта).

ПОШАГОВОЕ УСКОРЕНИЕ

Тед О'Грейди и Джозеф Кинг руководят большими проектами (50 разработчиков, экспертов предметной области и тестеров) с использованием экстремального программирования. Как и многие команды, создающие приложения на основе баз данных, они страдали от *медленных тестов* (Slow Tests), но нашли способ обойти эту проблему: к концу 2005 года набор тестов при включении кода в хранилище выполнялся 8 минут вместо 8 часов при обращении тестов к базе данных. Достаточно впечатляющее ускорение. Вот подробности их истории.

На данный момент у нас используется 6700 регулярно запускающихся тестов. Было предпринято несколько попыток ускорения работы тестов, и со временем эти решения развились.

В январе 2004 года тесты непосредственно взаимодействовали с базой данных через Toplink.

В июне 2004 года приложение было модифицировано для запуска тестов, взаимодействующих с базой данных Java (HSQL), остающейся в памяти. Это в два раза сократило время работы тестов.

В августе 2004 года была создана предназначенная для тестов инфраструктура, позволяющая Toplink работать вообще без базы данных. Это сократило время работы тестов на порядок.

В июле 2005 года был создан общий сервер для запуска тестов при включении кода в хранилище, что позволяло запускать тесты удаленно. Это не привело к сокращению времени работы тестов, но впоследствии оказалось полезным.

В июле 2005 началось использование инфраструктуры кластеризации, позволившей распределить запуск тестов по сети. Это в два раза сократило время работы тестов.

В августе 2005 года тесты графического интерфейса и справочных данных были удалены из набора тестов “включения” и запускались только из Cruise Control. Это сократило время работы набора на 15–20%.

С мая 2004 года с помощью Cruise Control тесты регулярно запускались с использованием настоящей базы данных. С ростом количества тестов время на компиляцию и выполнение выросло с одного часа до восьми.

После достижения состояния, когда разработчики не могли запускать тесты часто и создавать длинные очереди на включение кода в хранилище, пришлось адаптироваться за счет экспериментирования с новыми техниками. Как правило, время работы тестов не должно превышать 5 минут. Если время превышает 8 минут, рассматривается новый способ ускорения работы.

До этого момента удавалось бороться с соблазном запускать только подмножество тестов, и вместо этого основные усилия уделялись ускорению работы тестов — хотя, как можно заметить, мы начали удалять тесты, которые должны запускаться разработчиками постоянно (например, наборы тестов Master Data и GUI, не нужные во время включения кода в общее хранилище, так как они запускаются с помощью Cruise Control, а эти области меняются недостаточно часто).

Двумя наиболее интересными решениями за последнее время (кроме инфраструктуры для тестирования в памяти) являются тестовый сервер и инфраструктура кластеризации.

Тестовый сервер (называемый здесь ящиком “включения”) на самом деле оказался очень полезным и надежным. Мы приобрели компьютер на базе процессора Opteron, который работал почти вдвое быстрее компьютеров разработчиков (самый быстрый из тех, которые можно было купить). Для каждого разработчика на сервере была заведена учетная запись. С помощью утилиты UNIX rsync рабочая среда Eclipse синхронизировалась с соответствующей учетной записью на сервере. Набор пользовательских сценариев создал на сервере базу данных для удаленной учетной записи и запускал все тесты разработчиков. После завершения тестов данные о времени работы каждого теста выводились на консоль вместе с классом MyTestSuite.java, содержащим все неудачные завершения тестов. После этого разработчик мог запустить данный класс локально и исправить все неправильно работающие тесты. Самым большим преимуществом запуска тестов на сервере было вернувшееся ощущение быстрого запуска большого количества тестов, так как разработчик мог продолжать работу во время ожидания результатов.

Инфраструктура кластеризации (на базе Condor) работала достаточно быстро, но имела недостаток — все рабочее пространство (около 11 Мбайт) должно было копироваться на все узлы сети (около 20), с чем были связаны значительные накладные расходы, особенно при использовании десятком пар. В то же время тестовый сервер использовал утилиту rsync, которая копировала только новые и модифицированные файлы из рабочей среды разработчика. Практика показала, что инфраструктура кластеризации работает менее надежно, чем решение на базе сервера, так как достаточно часто информация о результате запуска тестов терялась. Кроме того, некоторые тесты не могли надежно работать под управлением инфраструктуры кластеризации.

Источник дополнительной информации

Более полное описание данной ситуации приводится по адресу:
<http://FasterTestsPaper.gerardmeszaros.com>.

Вариант: инкрементные тесты (Incremental Tests)

Общая тестовая конфигурация (Shared Fixture) может использоваться и совместно с длинными сложными последовательностями действий, каждое из которых зависит от предыдущего. В приемочных тестах в качестве такой последовательности может рассмат-

риваться отдельный процесс. В модульных тестах это может быть последовательность вызовов методов одного и того же объекта. Такую ситуацию можно проверять с помощью единственного “энергичного” теста (Eager Test; см. *Рулетка утверждений*, Assertion Roulette, с. 264). Альтернативным вариантом является выделение каждого действия в собственный *тестовый метод* (Test Method, с. 378), который зависит от действия предыдущих тестов над *общей тестовой конфигурацией* (Shared Fixture). Такой подход является примером *цепочки тестов* (Chained Tests, с. 477). Именно так работают тестеры в подразделениях контроля качества: создается тестовая конфигурация и выполняется последовательность тестов, каждый из которых построен на основе конфигурации. Тестеры обладают одним очень важным преимуществом по сравнению с *полностью автоматизированными тестами* (Fully Automated Tests, с. 81): если тест в середине цепочки вдруг завершается неудачно, тестер может быстро сообразить, как исправить ситуацию и стоит ли вообще продолжать. С другой стороны, автоматизированные тесты продолжают работу, и многие из них завершаются неудачно или генерируют ошибки, так как не обнаруживают тестовую конфигурацию и поведение тестируемой системы отличается от ожидаемого (хотя может быть вполне правильным). Полученные результаты работы тестов могут скрыть настоящую причину проблемы в потоке ошибок. При наличии определенного опыта можно распознать последовательность отказов и выявить основную причину¹.

Для простоты диагностики в начале каждого *тестового метода* (Test Method) имеет смысл вставить одно или несколько *сторожевых утверждений* (Guard Assertion, с. 510), в которых будут документированы предположения метода о состоянии тестовой конфигурации. Если эти утверждения оказываются ложными, причину проблемы необходимо искать в другом месте — или в неудачно завершившемся тесте ранее в наборе, или в порядке запуска тестов.

Замечания по реализации

Ключевым вопросом реализации *общей тестовой конфигурации* (Shared Fixture) является вопрос: “Как тесты узнают об объектах в составе *общей тестовой конфигурации* (Shared Fixture) и возможности их (повторного) использования?” Поскольку причиной использования *общей тестовой конфигурации* (Shared Fixture) является сокращение времени работы тестов за счет использования одной конфигурации, необходимо хранить ссылку на созданную конфигурацию. Таким образом, существующую конфигурацию можно найти, а при ее создании можно уведомлять другие тесты. При использовании создаваемых во время запуска конфигураций доступно больше возможностей, так как проще “запомнить” конфигурацию, созданную кодом, чем *предварительно созданную тестовую конфигурацию* (Prebuilt Fixture, с. 454), которая генерировалась другим приложением. Хотя и можно просто вписать в код идентификаторы (например, ключи базы данных) объектов конфигурации для всех тестов, это приведет к появлению “хрупкой” *тестовой конфигурации* (Fragile Fixture). Для обхода этой проблемы необходимо сохранять ссылку на конфигурацию и предоставлять всем тестам возможность получить эту ссылку.

¹ Первый неудачно завершившийся тест не обязательно будет указывать на причину.

Вариант: одноразовая общая конфигурация (Per-Run Fixture)

Простейшей формой *общей тестовой конфигурации* (Shared Fixture) является *одноразовая общая конфигурация* (Per-Run Fixture), которая создается в начале запуска тестов и в дальнейшем используется тестами совместно. В идеальном случае конфигурация живет не дольше, чем закончит работу последний тест. Тогда не придется волноваться о взаимодействии тестов при последующих запусках. Если конфигурация сохраняется (например, в базе данных), может потребоваться явная очистка.

Если *одноразовая общая конфигурация* (Per-Run Fixture) используется совместно только в единственном *классе теста* (Testcase Class, с. 401), простейшим решением будет применение переменных класса для хранения ссылок на каждый объект тестовой конфигурации и использование “ленивой” настройки (Lazy Setup, с. 460) или настройки *тестовой конфигурации набора* (Suite Fixture Setup, с. 465) для инициализации объектов перед запуском первого теста в наборе. Если конфигурация совместно используется несколькими *классами теста* (Testcase Class), придется воспользоваться *декоратором настройки* (Setup Decorator, с. 471) для хранения методов `setUp` и `tearDown`, а также *реестром тестовых конфигураций* (Test Fixture Registry) (в качестве которого может выступать тестовая база данных) для доступа к конфигурации.

Вариант: немодифицируемая общая тестовая конфигурация (Immutable Shared Fixture)

Использование *общей тестовой конфигурации* (Shared Fixture) приводит к появлению *нестабильных тестов* (Erratic Test), если тесты ее модифицируют. Таким образом, *общая тестовая конфигурация* (Shared Fixture) нарушает принцип *независимости тестов* (Independent Tests, с. 96). Этой проблемы можно избежать, сделав *общую тестовую конфигурацию* (Shared Fixture) немодифицируемой. В результате конфигурация делится на две логические части. Первая содержит все необходимые тесту данные, но ни одним тестом не модифицируется, т.е. получается *немодифицируемая общая тестовая конфигурация* (Immutable Shared Fixture). Вторая часть содержит объекты, которые должны модифицироваться или удаляться тестами. Эта часть должна создаваться каждым тестом как *новая тестовая конфигурация* (Fresh Fixture).

Самой сложной частью применения *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture) является определение, что является изменением объекта. Вот один из ключевых критериев: если один тест рассматривает действие другого теста как модификацию объекта в *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture), это действие не должно допускаться в тестах, использующих эту конфигурацию. Чаще всего *немодифицируемая общая тестовая конфигурация* (Immutable Shared Fixture) состоит из справочных данных, необходимых собственным конфигурациям тестов. Собственная конфигурация теста может создаваться как *новая тестовая конфигурация* (Fresh Fixture) поверх *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture).

Мотивирующий пример

В следующем примере показано создание конфигурации в *классе теста* (Testcase Class) с использованием *неявной настройки* (Implicit Setup, с. 449). Каждый *тестовый метод* (Test Method) использует переменную экземпляра для доступа к содержимому конфигурации.

```

public void testGetFlightsByFromAirport_OneOutboundFlight()
    throws Exception {
    setupStandardAirportsAndFlights();
    FlightDto outboundFlight = findOneOutboundFlight();
    // Вызов системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlight.getOriginAirportId());
    // Проверка результата
    assertOnly1FlightInDtoList("Flights at origin",
        outboundFlight,
        flightsAtOrigin);
}
public void testGetFlightsByFromAirport_TwoOutboundFlights()
    throws Exception {
    setupStandardAirportsAndFlights();
    FlightDto[] outboundFlights =
        findTwoOutboundFlightsFromOneAirport();
    // Вызов системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlights[0].getOriginAirportId());
    // Проверка результата
    assertExactly2FlightsInDtoList("Flights at origin",
        outboundFlights,
        flightsAtOrigin);
}

```

Обратите внимание, что метод `setUp` вызывается по одному разу для каждого *тестового метода* (Test Method). Если настройка конфигурации включает в себя доступ к базе данных или другие сложные операции, подобное решение может привести к появлению *медленных тестов* (Slow Test).

Замечания по рефакторингу

Для преобразования *класса теста* (Testcase Class) при переходе от *стандартной тестовой конфигурации* (Standard Fixture) к *общей тестовой конфигурации* (Shared Fixture) достаточно просто заменить переменные экземпляра переменными класса. В результате конфигурация будет существовать дольше, чем *объект теста* (Testcase Object). При этом необходимо инициализировать переменные класса только один раз, чтобы избежать повторного создания каждым *тестовым методом* (Test Method). Простым решением этой задачи является использование “ленивой” настройки (Lazy Setup). Конечно, возможны и другие способы создания *общей тестовой конфигурации* (Shared Fixture), например *декоратор настройки* (Setup Decorator) или *настройка тестовой конфигурации набора* (Suite Fixture Setup).

Пример: общая тестовая конфигурация (Shared Fixture)

В этом примере показана настройка *общей тестовой конфигурации* (Shared Fixture) с помощью “ленивой” настройки (Lazy Setup).

```

protected void setUp() throws Exception {
    if (sharedFixtureInitialized) {
        return;
    }
}

```

```

facade = new FlightMgmtFacadeImpl();
setupStandardAirportsAndFlights();
sharedFixtureInitialized = true;
)
protected void tearDown() throws Exception {
    // Удалять объекты нельзя, так как не известно,
    // последний ли это тест
}
)

```

Логика “ленивой” инициализации [SBPP] в методе `setUp` обеспечивает создание *общей тестовой конфигурации* (Shared Fixture) в случае, когда переменная класса оказывается неинициализированной. *Тестовые методы* (Test Method) также были модифицированы для использования *методов поиска* (Finder Method) при получении фрагментов конфигурации.

```

public void testGetFlightsByFromAirport_OneOutboundFlight()
    throws Exception {
    FlightDto outboundFlight = findOneOutboundFlight();
    // Вызов системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlight.getOriginAirportId());
    // Проверка результата
    assertOnly1FlightInDtoList("Flights at origin",
        outboundFlight,
        flightsAtOrigin);
}
public void testGetFlightsByFromAirport_TwoOutboundFlights()
    throws Exception {
    FlightDto[] outboundFlights =
        findTwoOutboundFlightsFromOneAirport();
    // Вызов системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlights[0].getOriginAirportId());
    // Проверка результата
    assertExactly2FlightsInDtoList("Flights at origin",
        outboundFlights,
        flightsAtOrigin);
}
)

```

Подробности реализации таких *вспомогательных методов теста* (Test Utility Method), как `setupStandardAirportsAndFlights`, здесь не показаны, поскольку они не нужны для понимания сути примера. Достаточно понять, что эти методы создают аэропорты и полеты, а также хранят ссылки на них в статических переменных, что позволяет *тестовым методам* (Test Method) получать доступ к одной и той же конфигурации как непосредственно, так и через *вспомогательные методы теста* (Test Utility Method).

Пример: немодифицируемая общая тестовая конфигурация (Immutable Shared Fixture)

Ниже приведен пример “загрязнения” *общей тестовой конфигурации* (Shared Fixture).

```

public void testCancel_proposed_p() throws Exception {
    // Общая тестовая конфигурация
    BigDecimal proposedFlightId = findProposedFlight();
}
)

```

```

// вызов системы
facade.cancelFlight(proposedFlightId);
// Проверка результата
try {
    assertEquals(FlightState.CANCELLED,
                facade.findFlightById(proposedFlightId));
} finally {
    // Очистка
    // Попытка исправить повреждения в надежде,
    // что это сработает!
    facade.overrideStatus(proposedFlightId,
                          FlightState.PROPOSED);
}
)

```

Эту проблему можно обойти, сделав конфигурацию немодифицируемой, т.е. разделив конфигурацию на две логические части. Первая содержит все необходимое, но не модифицируемое ни одним из тестов — *немодифицируемую общую тестовую конфигурацию* (Immutable Shared Fixture). Вторая часть содержит объекты, модифицируемые или удаляемые тестами. Эти объекты создаются каждым тестом как часть *новой тестовой конфигурации* (Fresh Fixture).

Ниже представлен тот же тест, модифицированный для использования *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture). Просто в пределах теста создается собственный экземпляр `mutableFlight`.

```

public void testCancel_proposed() throws Exception {
    // Создание конфигурации
    BigDecimal mutableFlightId =
        createFlightBetweenInsigificantAirports();
    // Вызов системы
    facade.cancelFlight(mutableFlightId);
    // Проверка результата
    assertEquals(FlightState.CANCELLED,
                facade.findFlightById(mutableFlightId));
    // Очистка
    // не требуется, так как для каждого полета
    // система создает новые идентификаторы.
    // Со временем может потребоваться очистка
    // базы данных.
}

```

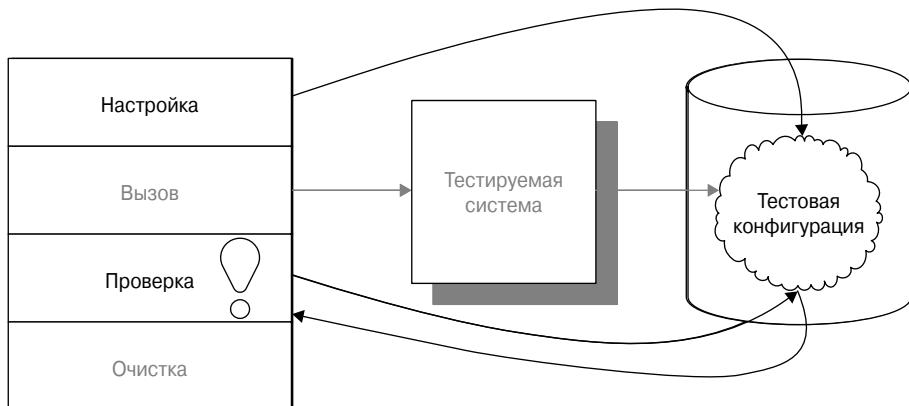
Обратите внимание, что в этой версии очистка конфигурации не нужна, так как тестируемая система использует *отдельное сгенерированное значение* (Distinct Generated Value) — разработчик номер полета не предоставляет. Кроме того, для защиты количества полетов в аэропортах других тестов используются предопределенные `dummyAirport1` и `dummyAirport2`. Таким образом, модифицируемые полеты могут без проблем накапливаться в базе данных.

Манипуляция через “черный ход” (Back Door Manipulation)

Как без стандартных тестов через открытый интерфейс проверить правильность логики?

Для создания тестовой конфигурации или проверки результата используется “черный ход” (например, непосредственный доступ к базе данных).

Также известен как:
Тест с пересечением уровней
(Layer-Crossing Test)



Каждый тест имеет начало (тестовую конфигурацию) и ожидаемую конечную точку (ожидаемый результат). “Нормальный” подход предполагает создание тестовой конфигурации и проверку результата через использование программного интерфейса тестируемой системы. В некоторых ситуациях это невозможно или нежелательно.

В таких ситуациях можно использовать *манипуляцию через “черный ход”* (Back Door Manipulation) для создания конфигурации и/или проверки состояния тестируемой системы.

Как это работает

Состояние тестируемой системы может иметь множество представлений. Оно может храниться в оперативной памяти, на диске в виде файлов или в других приложениях, с которыми взаимодействует система. Любой формат предварительных условий теста обычно требует не только известного, но и вполне конкретного состояния тестируемой системы. Точно так в конце теста *выполняется проверка состояния* (State Verification, с. 484) тестируемой системы.

При доступе к системе снаружи тест может установить предварительное состояние в обход нормального программного интерфейса через взаимодействие непосредственно с хранилищем состояния через “черный ход”. После завершения вызова тестируемой системы тест точно так может получить доступ к состоянию тестируемой системы через “черный ход” для сравнения с ожидаемым результатом. Для приемочных тестов в качестве “черного хода” чаще всего выступает тестовая база данных, но это может быть и любой другой компонент, от которого зависит тестируемая система, включая объект Registry, или даже файловая система. В случае модульных тестов в качестве “черного хода” высту-

пает класс, объект или альтернативный интерфейс тестируемой системы (или *связанный с тестом подкласс*, Test-Specific Subclass, с. 591), который предоставляет доступ к состоянию в таком виде, который не используется “нормальными” клиентами. Кроме того, можно заменить вызываемый компонент соответствующим образом настроенным *тестовым двойником* (Test Double, с. 539), что значительно упростит контроль за состоянием тестируемой системы.

Когда это использовать

Манипуляция через “черный ход” (Back Door Manipulation) может использоваться по ряду причин, которые более подробно рассматриваются ниже. Предварительным условием использования этой техники является существование “черного хода” к состоянию системы. Основным недостатком является тесное связывание тестов или *вспомогательных методов теста* (Test Utility Method, с. 610) с внутренним представлением тестируемой системы. Если представление будет меняться, могут появиться “хрупкие” тесты (Fragile Test, с. 277). Приемлемость такого решения необходимо оценивать в каждом конкретном случае. Эффект тесного связывания можно значительно ослабить, если скрыть *манипуляцию через “черный ход”* (Back Door Manipulation) внутри *вспомогательных методов теста* (Test Utility Method).

Кроме того, использование *манипуляций через “черный ход”* (Back Door Manipulation) приводит к появлению *непонятных тестов* (Obscure Test, с. 230), так как скрывается связь между результатом теста и тестовой конфигурацией. Этую проблему можно обойти, включив передаваемые через “черный ход” данные в *класс теста* (Testcase Class, с. 401), или снизить ее влияние, воспользовавшись *методами поиска* (Finder Method) для получения ссылки на объекты в составе тестовой конфигурации.

Часто *манипуляция через “черный ход”* (Back Door Manipulation) применяется для тестиования базовых операций создания, чтения, обновления и удаления над состоянием тестируемой системы. В таком случае необходимо убедиться, что информация сохранилась и восстанавливается в том же виде. Очень сложно написать нормальный тест для операции чтения, не проверив операцию создания. Точно так, невозможно проверить операции обновления и удаления без одновременного тестиирования операций создания и чтения. Конечно, нормальные тесты для стандартного интерфейса позволяют проверить эти операции, но они не обнаруживают определенные типы системных проблем, например вставку информации в неправильный столбец таблицы. Одним из решений является создание теста с *пересечением уровней* (Layer-Crossing Test), который с помощью *манипуляции через “черный ход”* (Back Door Manipulation) может непосредственно создавать или проверять содержимое в базе данных. Для теста операции чтения с помощью *настройки через “черный ход”* (Back Door Setup) в базу данных вставляется проверочная информация. Для теста операции записи система записывает некоторые объекты, а тест с помощью *проверки через “черный ход”* (Back Door Verification) проверяет их наличие в базе данных.

Вариант: настройка через “черный ход” (Back Door Setup)

Одной из причин использования *манипуляции через “черный ход”* (Back Door Manipulation) является ускорение работы тестов. Если система выполняет большой объем вычислений перед добавлением данных в хранилище, может потребоваться значительное время на создание тестовой конфигурации через программный интерфейс системы.

Один из способов ускорения работы тестов предполагает определение содержимого хранилища и создание механизма наполнения хранилища через “черный ход” в обход стандартного программного интерфейса. К сожалению, в результате такого подхода появляются новые проблемы: *настройка через “черный ход”* (Back Door Setup) обходит проверки при создании бизнес-объектов, а значит, созданные тестовые конфигурации могут оказаться нереалистичными или даже некорректными. Эта проблема может проявиться со временем в результате модификации бизнес-правил в соответствии с изменением исходных условий. В то же время такой подход позволяет создавать тестовые сценарии, недопустимые при работе через программный интерфейс тестируемой системы.

Если база данных используется совместно тестируемой системой и другим приложением, необходимо обеспечить правильность работы с базой данных и обработку всех возможных конфигураций, создаваемых другим приложением. *Настройка через “черный ход”* (Back Door Setup) позволяет создавать такие конфигурации — и это может быть единственный способ, если тестируемая система только читает данные из таблиц или записывает конкретные (и допустимые) конфигурации данных. *Настройка через “черный ход”* (Back Door Setup) позволяет просто создавать “невозможные” конфигурации и проверять поведение системы в этих условиях.

Вариант: проверка через “черный ход” (Back Door Verification)

Данная техника предполагает *проверку состояния* (State Verification) тестируемой системы после вызова через “черный ход”. Чаще всего она применяется для приемочных (или функциональных, как их еще называют) тестов. “Черный ход” обычно является альтернативным способом проверки состояния объектов в базе данных. Для этого используют стандартные интерфейсы наподобие SQL или экспорт данных, результат которого сравнивается с эталонными утилитами сравнения файлов.

Как было показано ранее, *манипуляция через “черный ход”* (Back Door Manipulation) позволяет тестам работать быстрее. Если единственным способом получения состояния системы является вызов дорогой операции (например, сложного отчета) или если такая операция вносит собственные модификации в состояние тестируемой системы, лучше использовать *манипуляцию через “черный ход”* (Back Door Manipulation).

Еще одной причиной использования данного способа могут являться предположения других систем относительно формата, в котором тестируемая система хранит свое состояние. Это один из вариантов опосредованного вывода. В такой ситуации тест через открытый интерфейс не может доказать правильность поведения тестируемой системы, так как не может обнаружить системной проблемы при наличии одинаковой ошибки в операциях записи и чтения (например, сохранение информации в неправильном столбце таблицы). В качестве решения можно использовать *тест с пересечением уровней* (Layer-Crossing Test), который непосредственно рассматривает содержимое базы данных для проверки правильности хранения информации. В случае операции записи тест запрашивает у системы запись некоторых объектов и рассматривает содержимое базы данных через “черный ход”.

Вариант: очистка через “черный ход” (Back Door Teardown)

Манипуляция через “черный ход” (Back Door Manipulation) может использоваться для очистки *новой тестовой конфигурации* (Fresh Fixture, с. 344), хранящейся в тестовой базе данных. Эта возможность оказывается особенно полезной, если одной командой можно

очищать целые таблицы, как при *очистке усечением таблиц* (Table Truncation Teardown, с. 668) или *очистке откатом транзакции* (Transaction Rollback Teardown, с. 675).

Замечания по реализации

Реализация манипуляции через “черный ход” (Back Door Manipulation) зависит от места хранения тестовой конфигурации и простоты доступа к состоянию тестируемой системы. Кроме того, одним из факторов является причина использования манипуляции через “черный ход” (Back Door Manipulation). В этом разделе показаны наиболее распространенные реализации, но существуют и другие варианты использования данного шаблона.

Вариант: сценарий наполнения базы данных (Database Population Script)

Если тестируемая система хранит состояние в базе данных, доступной во время работы, самым простым способом реализовать манипуляцию через “черный ход” (Back Door Manipulation) является непосредственная загрузка данных в базу перед вызовом тестируемой системы. Подобный подход чаще всего необходим при создании приемочных тестов, но может потребоваться и для модульных тестов, если тестируемые классы непосредственно взаимодействуют с базой данных. Сначала необходимо определить предварительные условия запуска теста и на основе этой информации идентифицировать данные, которые должны входить в тестовую конфигурацию. После этого определяется сценарий базы данных, вставляющий соответствующие записи непосредственно в базу данных в обход логики тестируемой системы. Данный сценарий наполнения базы данных (Database Population Script) используется каждый раз, когда необходимо настроить тестовую конфигурацию, — это решение зависит от выбранной стратегии тестовой конфигурации. (Дополнительная информация по данной теме приводится в главе 6, “Стратегия автоматизации тестирования”.)

Выбрав использование сценария наполнения базы данных (Database Population Script) при модификации структуры хранилища тестируемой системы или семантики данных, придется обслуживать как сценарий, так и файлы, которые используются им в качестве входных параметров. Это требование может увеличить стоимость обслуживания тестов.

Вариант: загрузчик данных (Data Loader)

Загрузчик данных (Data Loader) представляет собой программу, загружающую данные в хранилище тестируемой системы. Она отличается от сценария наполнения базы данных (Database Population Script) тем, что написана не на языке базы данных, а на другом языке программирования. Это обеспечивает дополнительную гибкость и позволяет использовать загрузчик данных (Data Loader), даже если состояние системы хранится не в реляционной базе данных.

Если хранилище данных находится за пределами тестируемой системы, например в реляционной базе данных, загрузчик данных (Data Loader) может выступать в роли “еще одного приложения”, которое записывает данные в хранилище. Загрузчик будет использовать базу данных так же, как тестируемая система, но входные параметры будет получать из файла, а не из стандартного источника тестируемой системы (например, от других приложений). Если для доступа к данным из тестируемой системы используется инструмент объектно-реляционного отображения, проще всего для создания загрузчика данных (Data Loader) воспользоваться теми же объектами и отображениями. Необходи-

мые объекты просто создаются в памяти и через объектно-реляционный модуль сохраняются в базе данных.

Если тестируемая система хранит данные во внутренних структурах данных (т.е. в памяти), загрузчик данных (Data Loader) может потребовать доступа к интерфейсу, предоставляемому тестируемой системой. Следующие характеристики отличают его от нормальной функциональности тестируемой системы:

- используется только тестами;
- читает данные из файла, а не из стандартного хранилища данных тестируемой системы;
- пропускает большинство проверок действительности введенных данных, которые выполняются тестируемой системой.

В качестве входных файлов могут использоваться простые плоские файлы, в которых в качестве разделителей полей используется запятая или символ табуляции. Точно так же файлы могут быть структурированы с использованием формата XML. DbUnit является расширением инфраструктуры JUnit, реализующим загрузчик данных (Data Loader) для настройки тестовой конфигурации.

Вариант: сценарий экспорта из базы данных (Database Extraction Script)

Когда тестируемая система сохраняет состояние в базе данных, доступной во время работы, можно воспользоваться такой структурой и осуществить проверку через “черный ход” (Back Door Verification). Достаточно воспользоваться сценарием базы данных для извлечения данных из тестовой базы и проверки их правильности через сравнение с предварительно подготовленным “эталоном” или через проверку правильности результата выполнения конкретных запросов.

Вариант: получатель данных (Data Retriever)

Получатель данных (Data Retriever) является аналогом загрузчика данных (Data Loader), так как получает состояние тестируемой системы во время проверки через “черный ход” (Back Door Verification). Как верный пес, он “приносит” данные, которые можно сравнить с ожидаемым результатом внутри теста. DbUnit является расширением инфраструктуры JUnit, реализующим получатель данных (Data Retriever) для проверки результата.

Вариант: тестовый двойник (Test Double) в качестве “черного хода”

Пока что все описанные здесь техники реализации предполагали взаимодействие с вызываемым компонентом тестируемой системы для настройки или очистки тестовой конфигурации или для проверки ожидаемого результата. Возможно, самой распространенной формой манипуляции через “черный ход” (Back Door Manipulation) является замена вызываемого компонента тестовым двойником (Test Double). Одним из вариантов является использование поддельного объекта (Fake Object, с. 565), в который данные загружены таким образом, как будто тестируемая система уже с ним взаимодействовала. Такая стратегия позволяет избежать использования тестируемой системы для настройки ее состояния. Еще одним вариантом является использование одного из типов настраиваемого тестового двойника (Configurable Test Double, с. 571), например подставного объекта (Mock Object, с. 558) или тестовой заглушки (Test Stub, с. 544).

В любом случае можно полностью избежать появления *непонятных тестов* (Obscure Test), сделав состояние *тестового двойника* (Test Double) видимым внутри *тестового метода* (Test Method, с. 378).

Как только потребуется выполнить *проверку поведения* (Behavior Verification, с. 489) обращений тестируемой системы к вызываемому компоненту, можно будет воспользоваться *тестом с пересечением уровней* (Layer-Crossing Test), который заменит вызываемый компонент *тестовым агентом* (Test Spy, с. 552) или *подставным объектом* (Mock Object). Если необходимо убедиться, что тестируемая система ведет себя определенным образом при получении опосредованного ввода от вызываемого компонента (или при переходе в определенное состояние), вызываемый компонент можно заменить *тестовой заглушкой* (Test Stub).

Мотивирующий пример

Следующий тест, работающий через стандартный интерфейс, проверяет базовую функциональность удаления полета через основной интерфейс тестируемой системы. Но он не проверяет опосредованный вывод системы, а именно, обязательный вызов функции записи в журнал при каждом удалении полета с занесением времени, даны и идентификатора пользователя, отправившего запрос. В большинстве систем такая ситуация служит примером “поведения с пересечением уровней”. Функция записи в журнал является частью уровня универсальной инфраструктуры, а тестируемая система относится к поведению на уровне приложения.

```
public void testRemoveFlight() throws Exception {
    // Настройка
    FlightDto expectedFlightDto = createARegisteredFlight();
    FlightManagementFacade facade = new FlightManagementFacadeImpl();
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertFalse("flight should not exist after being removed",
        facade.flightExists(expectedFlightDto,
            getFlightNumber()));
}
```

Замечания по рефакторингу

Данный тест можно преобразовать для использования *проверки через “черный ход”* (Back Door Verification), добавив код проверки результата для доступа и проверки функции записи состояния в журнал. Это можно сделать как через чтение состояния из базы данных, в которую пишет данная функция, так и через замену функции *тестовым агентом* (Test Spy), который будет сохранять состояние в хранилище с более простым доступом.

Пример: проверка результата через “черный ход” с помощью тестового агента (Test Spy)

Ниже приведен тот же тест, преобразованный для использования *тестового агента* (Test Spy) для доступа к послетестовому состоянию функции записи в журнал.

```

public void testRemoveFlightLogging_recordingTestStub()
    throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnUnregFlight();
    FlightManagementFacade facade =
        new FlightManagementFacadeImpl();
    // Настройка тестового двойника (Test Double)
    AuditLogSpy logSpy = new AuditLogSpy();
    facade.setAuditLog(logSpy);
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertEquals("number of calls", 1,
        logSpy.getNumberofCalls());
    assertEquals("action code",
        Helper.REMOVE_FLIGHT_ACTION_CODE,
        logSpy.getActionCode());
    assertEquals("date", helper.getTodaysDateWithoutTime(),
        logSpy.getDate());
    assertEquals("user", Helper.TEST_USER_NAME,
        logSpy.getUser());
    assertEquals("detail",
        expectedFlightDto.getFlightNumber(),
        logSpy.getDetail());
}

```

Такой подход позволяет лучше проверять выполнение записи в журнал, когда база данных содержит так много записей, что проверка новых записей с помощью *дельта-утверждений* (Delta Assertion, с. 505) становится непрактичной.

Пример: настройка тестовой конфигурации через “черный ход”

В следующем примере показана настройка тестовой конфигурации с использованием базы данных в качестве “черного хода” к тестируемой системе. Тест вставляет запись в таблицу EmailSubscription и запрашивает у тестируемой системы поиск этой записи. После этого делаются предположения относительно различных полей объекта, возвращаемого тестируемой системой. Это позволяет проверить правильность чтения записей.

```

static final String TABLE_NAME = "EmailSubscription";
static final BigDecimal RECORD_ID = new BigDecimal("111");
static final String LOGIN_ID = "Bob";
static final String EMAIL_ID = "bob@foo.com";
public void setUp() throws Exception {
    String xmlString =
        "<?xml version='1.0' encoding='UTF-8'?>" +
        "<dataset>" +
        "  <" + TABLE_NAME + " " +
        "    EmailSubscriptionId='" + RECORD_ID + "' " +
        "    UserLoginId='" + LOGIN_ID + "' " +
        "    EmailAddress='" + EMAIL_ID + "' " +
        "    RecordVersionNum='62' " +
        "    CreateByUserId='MappingTest' " +
        "    CreateDateTime='2004-03-01 00:00:00.0' " +
        "    LastModByUserId='MappingTest' " +
        "    LastModDateTime='2004-03-01 00:00:00.0'/>" +

```

```

        "</dataset>";
    insertRowsIntoDatabase(xmlString);
}

public void testRead_Login() throws Exception {
    // вызов
    EmailSubscription subs =
        EmailSubscription.findInstanceWithId(RECORD_ID);
    // проверка
    assertNotNull("Email Subscription", subs);
    assertEquals("User Name", LOGIN_ID, subs.getUserName());
}
public void testRead_Email() throws Exception {
    // вызов
    EmailSubscription subs =
        EmailSubscription.findInstanceWithId(RECORD_ID);
    // проверка
    assertNotNull("Email Subscription", subs);
    assertEquals("Email Address",
        EMAIL_ID,
        subs.getEmailAddress());
}

```

Документ XML для наполнения базы данных создается внутри *класса теста* (Testcase Class), чтобы избежать появления *тайного гостя* (Mystery Guest), который будет создан, если воспользоваться внешним файлом для загрузки базы данных (данный подход рассматривается в описании рефакторинга *встраивания ресурса*, In-line Resource, с. 738). Для того чтобы сделать тест более понятным, вызываются методы с описательными именами, скрывающие подробности использования DbUnit для загрузки и очистки базы данных после завершения теста с помощью *очистки усечением таблиц* (Table Truncation Teardown). Ниже приведен текст *вспомогательных методов теста* (Test Utility Method) из этого примера.

```

private void insertRowsIntoDatabase(String xmlString)
    throws Exception {
    IDataset dataSet = new FlatXmlDataSet(new StringReader(xmlString));
    DatabaseOperation.CLEAN_INSERT.
        execute(getDbConnection(), dataSet);
}
public void tearDown() throws Exception {
    emptyTable(TABLE_NAME);
}
public void emptyTable(String tableName) throws Exception {
    IDataset dataSet = new DefaultDataSet(new DefaultTable(tableName));
    DatabaseOperation.DELETE_ALL.
        execute(getDbConnection(), dataSet);
}

```

Конечно, реализация этих методов характерна для DbUnit, но ее придется заменить, если будет использоваться другая реализация xUnit.

Вот некоторые наблюдения относительно этих тестов: во избежание появления “энергичных” тестов (Eager Test; см. *Рулетка утверждений*, Assertion Roulette, с. 264) утверждения для каждого поля располагаются в отдельных тестах. Такая структура приводит к появлению медленных тестов (Slow Tests, с. 289), поскольку тесты взаимодействуют

с базой данных. Можно воспользоваться “ленивой” настройкой (Lazy Setup, с. 460) или настройкой тестовой конфигурации набора (Suite Fixture Setup, с. 465), чтобы избежать многократной настройки конфигурации, но получившаяся общая тестовая конфигурация (Shared Fixture, с. 350) не должна модифицироваться другими тестами. (Данная проблема не будет рассматриваться подробнее, чтобы не усложнять пример.)

Источник дополнительной информации

Примеры использования “черного хода” в качестве основного интерфейса приводятся во врезке “База данных как программный интерфейс тестируемой системы?”.

БАЗА ДАННЫХ КАК ПРОГРАММНЫЙ ИНТЕРФЕЙС ТЕСТИРУЕМОЙ СИСТЕМЫ?

Распространенным способом настройки тестовых конфигураций является *настройка через “черный ход”* (Back Door Setup; см. *Манипуляция через “черный ход”*, Back Door Manipulation, с. 359). Для проверки результата популярным решением является использование *проверки через “черный ход”* (Back Door Verification). Но когда тест, непосредственно взаимодействующий с базой данных в обход тестируемой системы, не считается работающим через “черный ход”?

В одном из недавних проектов знакомые автора задавались именно этим вопросом, хотя его осознание пришло не сразу. Один из аналитиков (одновременно бывший опытным пользователем) слишком большое внимание уделял схеме базы данных. Сначала причиной такого повышенного внимания считался опыт работы с пакетом Powerbuilder, и были предприняты попытки отучить его от подобной привычки. Ничего не получилось. Аналитик уперся еще сильнее. Разработчики пытались объяснить, что в проектах с гибким процессом разработки важно не пытаться определять всю схему данных в начале проекта, поскольку схема развивается вместе с реализацией основных требований.

Конечно, аналитик жаловался при каждой модификации схемы базы данных, так как изменения “ломали” все его запросы. С развитием проекта остальные разработчики начали понимать, что аналитику действительно нужна стабильная база данных для выполнения запросов. Это был его способ проверки правильности данных, генерированных системой.

После выделения этого требования разработчики стали рассматривать схему запросов как формальный интерфейс разрабатываемой системы. На основе этого интерфейса были разработаны приемочные тесты, и от разработчиков требовалось нормальное завершение этих тестов при каждой модификации базы данных. Для снижения эффекта от рефакторинга базы данных были определены несколько представлений, реализующих описанный интерфейс. В результате появилась возможность выполнять рефакторинг базы данных при появлении такой необходимости.

В какой момент подобная ситуация может повториться? В любой момент, когда клиент запускает утилиты генерации отчетов (например, Crystal Reports) на основе вашей базы данных, может возникнуть требование обеспечить стабильный интерфейс для генерации отчетов. Точно так же, если клиент использует сценарии (DTS или SQL) для загрузки информации в базу данных, может потребоваться стабильный интерфейс загрузки данных.

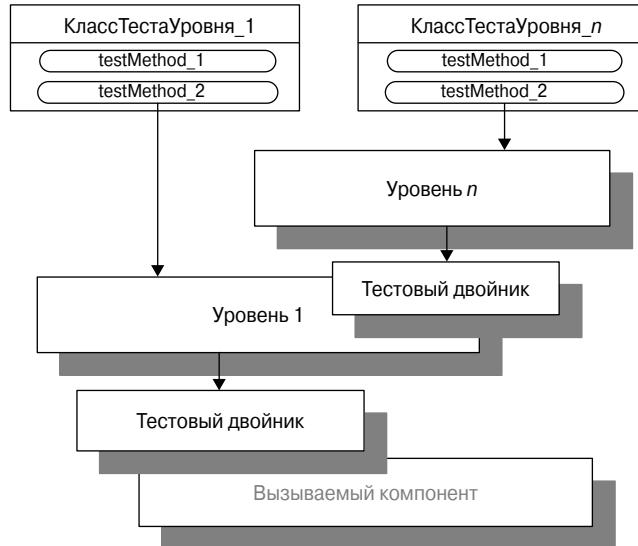
Тест уровня (Layer Test)

Также известен как:

Тест одного уровня (Single Layer Test), Тестирование по уровням (Testing by Layers), Одноуровневый тест (Layered Test)

Как независимо проверить логику, являющуюся частью многоуровневой архитектуры?

Для каждого уровня архитектуры создаются отдельные тесты.



Сложно обеспечить хорошее покрытие тестами при тестировании всего приложения в направлении “сверху вниз”. При этом некоторые фрагменты приложения будут *тестироваться опосредованно* (Indirect Testing; см. *Непонятный тест*, Obscure Test, с. 230). Для создания многих приложений используется многоуровневая архитектура, позволяющая разделить основные компоненты. Многие приложения состоят из так называемого презентационного уровня (пользовательского интерфейса), уровня бизнес-логики или уровня предметной области и уровня хранения данных. В некоторых архитектурах применяется еще больше уровней.

Приложение на основе многоуровневой архитектуры можно тестировать более эффективно, если тестировать каждый уровень изолированно.

Как это работает

Тестируемая система проектируется на основе уровней, отделяющих презентационную логику от бизнес-логики и механизмов сохранения данных или интерфейсов к другим системам⁵. Вся бизнес-логика размещается на служебном уровне, который пре-

⁵ Не вся презентационная логика относится к пользовательскому интерфейсу. Она также может обеспечивать обмен сообщениями с другими приложениями.

доставляет функциональность приложения презентационному уровню в виде программного интерфейса. Каждый уровень архитектуры рассматривается как отдельная тестируемая система. Для каждого уровня независимо создаются компонентные тесты, не зависящие от других уровней. Таким образом, при n -уровневой архитектуре тесты размещаются на уровне $n+1$, а на уровне $n-1$ может устанавливаться *тестовый двойник* (Test Double, с. 538).

Когда это использовать

Тест уровня (Layer Test) можно использовать при наличии многоуровневой архитектуры, когда необходимо обеспечить хорошее покрытие тестами логики на каждом из уровней. Намного проще проверять каждый уровень независимо, а не проверять все уровни сразу. Это особенно справедливо при создании защитного кода обработки значений, возвращаемых через границы уровней. В правильно работающем программном обеспечении такие ошибки “никогда не должны происходить”, но в реальной жизни это случается. Для правильной обработки таких ошибок “невозможные” значения можно вставлять под видом опосредованного ввода тестируемого уровня.

Тест уровня (Layer Test) может оказаться очень полезным при разделении команды проекта на несколько команд меньшего размера, специализирующихся на разных технологиях. Каждый уровень архитектуры требует различных знаний и часто основан на разных технологиях. Таким образом, границы уровней служат естественными разделителями между командами. *Тест уровня* (Layer Test) является хорошим средством фиксирования и описания семантики интерфейса уровня.

Даже после выбора стратегии на основе *тестов уровня* (Layer Test) имеет смысл добавить несколько тестов для направления “сверху вниз”, которые будут проверять правильность интеграции различных уровней. Такие тесты должны проверять только один или два базовых сценария; нет смысла проверять каждое бизнес-условие, так как все они уже проверены *тестами уровня* (Layer Test).

Большинство вариантов данного шаблона зависят от уровня, который тестируется независимо от других уровней.

Вариант: тест презентационного уровня (Presentation Layer Test)

Можно написать целую книгу о тестах презентационного уровня. Конкретный шаблон зависит от природы технологии презентационного уровня (например, это может быть графический интерфейс пользователя, традиционный Web-интерфейс, “умный” Web-интерфейс, Web-службы). Независимо от технологии ключевым условием является тестируемость презентационной логики независимо от бизнес-логики, что защищает тесты от изменений в нижележащей бизнес-логике. (Тесты презентационного уровня и так сложно автоматизировать!)

Еще одним условием является проектирование презентационного уровня с учетом тестируемости независимо от презентационной инфраструктуры. *Минимальный диалог* (Humble Dialog; см. *Минимальный объект*, Humble Object, с. 700) является ключевым элементом проектирования этого уровня с учетом последующего тестируемости. Фактически определяются подуровни внутри презентационного уровня. Уровень с *минимальным диалогом* (Humble Dialog) является презентационным графическим уровнем, а тестируемый уровень — презентационным поведенческим уровнем. Такое разделение уровней позво-

ляет проверять активизацию кнопок, деактивизацию пунктов меню и т.д. без создания экземпляров настоящих графических объектов.

Вариант: тест служебного уровня (Service Layer Test)

Служебный уровень (Service Layer) традиционно содержит большую часть модульных и компонентных тестов. Тестирование бизнес-логики с помощью приемочных тестов представляет собой достаточно сложную задачу, так как тестирование служебного уровня через презентационный подразумевает *опосредованное тестирование* (Indirect Testing) и *чувствительное сравнение* (Sensitive Equality; см. “Хрупкий” тест, Fragile Test, с. 277). И то, и другое приводят к появлению “хрупких” тестов (Fragile Test) и *высокой стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300). Непосредственное тестирование служебного уровня позволяет избежать всех этих проблем.

Для защиты от появления *медленных тестов* (Slow Tests, с. 289) обычно уровень хранения данных заменяется *поддельной базой данных* (Fake Database; см. *Поддельный объект*, Fake Object, с. 565) и только после этого запускаются тесты. На самом деле основной причиной использования многоуровневой архитектуры является изоляция этого кода от других сложных в тестировании уровней. Алистер Коуберн развел эту идею, описав шестиугольную архитектуру <http://alistair.cockburn.us> [WWW].

Служебный уровень может оказаться полезным и в других случаях. Он может использоваться для запуска приложений без “интерфейса” (без презентационного уровня), например как при использовании Microsoft Excel для автоматизации часто выполняемых задач с помощью макросов.

Вариант: тест уровня хранения (Persistence Layer Test)

Уровень хранения данных также требует тестирования. Если только приложение использует хранилище данных, то обычных тестов через открытый интерфейс должно быть достаточно. Но такие тесты не перехватывают один тип программных ошибок: когда данные случайно сохраняются не в тех столбцах. Пока типы данных перепутанных столбцов совпадают и при чтении допускается та же ошибка, тест через открытый интерфейс будет завершаться успешно! Такая ошибка не повлияет на работу приложения, но может усложнить обслуживание в будущем и обязательно приведет к проблемам при взаимодействии с другими приложениями.

Если и другие приложения используют хранилище данных, желательно реализовать несколько тестов с пересечением уровней, которые проверяют правильность размещения данных в столбцах таблицы. Для настройки содержимого базы данных или проверки состояния после теста можно воспользоваться *манипуляцией через “черный ход”* (Back Door Manipulation).

Вариант: “подкожный” тест (Subcutaneous Test)

“Подкожный” тест (Subcutaneous Test) является вырожденной формой *теста уровня* (Layer Test). Он пропускает презентационный уровень системы и взаимодействует непосредственно со служебным уровнем. В большинстве случаев служебный уровень не изолирован от нижележащих уровней, а значит, тестируется вся система, кроме презентационного уровня. “Подкожный” тест (Subcutaneous Test) не требует такого строгого разделения, как *тест служебного уровня* (Service Layer Test), что значительно упрощает

интеграцию тестов в существующее приложение. “*Подкожный*” *тест* (Subcutaneous Test) применяется при создании приемочных тестов приложения. Вероятность нарушения работы этих тестов в результате изменений в приложении значительно ниже⁶, так как он не взаимодействует с приложением через презентационный интерфейс. В результате на него не влияет целая категория изменений.

Вариант: тест компонента (Component Test)

Тест компонента (Component Test) является наиболее общей формой *теста уровня* (Layer Test), поскольку уровни можно рассматривать как набор отдельных компонентов, выступающих в роли “микроуровней”. *Тесты компонентов* (Component Test) являются хорошим способом документирования поведения отдельных компонентов при разработке на основе компонентов, когда некоторые компоненты должны быть модифицированы или переписаны.

Замечания по реализации

Собственные тесты уровня (Layer Test) могут использовать открытый интерфейс, а могут пересекать уровни. Каждый подход имеет свои преимущества. На практике обычно комбинируются оба стиля тестирования. Тесты для открытого интерфейса проще написать (предполагается, что существует подходящий *поддельный объект* (Fake Object) для уровня n-1). Но при проверке логики обработки ошибок на уровне n приходится использовать тесты с пересечением уровня.

Тесты для открытого интерфейса

Хорошей отправной точкой для *теста уровня* (Layer Test) может служить тест для открытого интерфейса, так как его достаточно для большинства *простых тестов успешности* (Simple Success Test; см. *Тестовый метод*, Test Method, с. 378). Эти тесты можно написать таким образом, чтобы они не зависели от неполной изоляции интересующего уровня от нижележащих уровней. При этом можно использовать настоящие компоненты или заменить их *поддельными объектами* (Fake Object). Второй вариант особенно полезен, когда база данных или асинхронные механизмы на нижних уровнях приводят к появлению *медленных тестов* (Slow Test).

Управление опосредованным вводом

Нижние уровни системы можно заменить *тестовой заглушкой* (Test Stub, с. 544), которая будет возвращать “законсервированный” результат в зависимости от запросов клиентского уровня (например, клиент 0001 является действительным, клиент 0002 является отложенным, а у клиента 0003 три учетные записи). Таким образом можно тестировать клиентскую логику с понятным опосредованным вводом от нижележащего уровня. Данный подход хорошо подходит при автоматизации *тестов на ожидаемое исключение* (Expected Exception Test) или при вызове поведения, основанного на данных

⁶ Значительно ниже, чем при использовании теста, вызывающего логику через презентационный интерфейс.

от внешней системы⁷. Альтернативным решением является использование *манипуляции через “черный ход”* (Back Door Manipulation) для настройки опосредованного ввода.

Проверка опосредованного вывода

Если необходимо проверить опосредованный вывод интересующего уровня, можно воспользоваться *подставным объектом* (Mock Object, с. 558) или *тестовым агентом* (Test Spy, с. 552) для замены компонента на уровнях ниже тестируемой системы. При этом можно сравнивать фактические обращения к вызываемому компоненту с ожидаемыми вызовами. Альтернативным вариантом является использование *манипуляции через “черный ход”* (Back Door Manipulation) для проверки опосредованного вывода тестируемой системы после его формирования.

Мотивирующий пример

При попытке тестирования всех уровней приложения одновременно приходится проверять бизнес-логику через презентационный уровень. Следующий тест является простым примером тестирования тривиальной бизнес-логики через тривиальный пользовательский интерфейс.

```
private final int LEGAL_CONN_MINS_SAME = 30;
public void testAnalyze_sameAirline_LessThanConnectionLimit()
throws Exception {
    // Настройка
    FlightConnection illegalConn =
        createSameAirlineConn(LEGAL_CONN_MINS_SAME - 1);
    // Вызов
    FlightConnectionAnalyzerImpl sut =
        new FlightConnectionAnalyzerImpl();
    String actualHtml =
        sut.getFlightConnectionAsHtmlFragment(
            illegalConn.getInboundFlightNumber(),
            illegalConn.getOutboundFlightNumber());
    // Проверка
    StringBuffer expected = new StringBuffer();
    expected.append("<span class='boldRedText'>");
    expected.append("Connection time between flight ");
    expected.append(illegalConn.getInboundFlightNumber());
    expected.append(" and flight ");
    expected.append(illegalConn.getOutboundFlightNumber());
    expected.append(" is ");
    expected.append(illegalConn.getActualConnectionTime());
    expected.append(" minutes.</span>");
    assertEquals("html", expected.toString(), actualHtml);
}
```

Тест содержит знание о функциональности бизнес-уровня (что делает подключение недопустимым) и презентационного уровня (как выглядит недопустимое подключение). Кроме того, он зависит от базы данных, так как *FlightConnection* извлекается из другого компонента. Если любой из этих элементов изменится, тест придется переделывать.

⁷ Обычно эти данные вставляются непосредственно в совместно используемую базу данных или вставляются с помощью “импорта данных”.

Замечания по рефакторингу

Этот тест можно разделить на два: один из них будет проверять бизнес-логику (что собой представляет недопустимое подключение?), а второй — презентационный уровень (как недопустимое подключение должно выглядеть со стороны пользователя?). Обычно для этого необходимо продублировать весь *класс теста* (Testcase Class, с. 401), исключая логику проверки презентационного уровня из *тестовых методов* (Test Method) бизнес-уровня и заменяя заглушками объекты бизнес-уровня в *тестовых методах* (Test Method) презентационного уровня.

В процессе рефакторинга может оказаться, что как минимум в одном *классе теста* (Testcase Class) можно сократить количество тестов, поскольку для этого уровня существует меньше тестовых условий. Данный пример начинается с четырех тестов (комбинаций одинаковых/разных авиалиний и периодов времени), каждый из которых проверяет и бизнес-, и презентационный уровень. В итоге получается четыре теста для бизнес-уровня (непосредственно тестируемые исходные комбинации) и два теста для презентационного уровня (форматирование допустимых и недопустимых подключений)⁸. Таким образом, только два последних теста должны учитывать подробности форматирования строк, и при неудачном завершении теста сразу ясно, на каком уровне находится ошибка.

Рефакторинг можно продолжить, воспользовавшись *заменой зависимости тестовым двойником* (Replace Dependency with Test Double, с. 740) для превращения этого “*подкожного*” теста (Subcutaneous Test) в настоящий тест служебного уровня.

Пример: тест презентационного уровня

В следующем примере представлен предыдущий тест, переработанный для проверки поведения презентационного уровня в ситуации запроса недопустимого подключения. Тест заменяет заглушкой объект FlightConnAnalyzer и настраивает его на возврат недопустимого подключения в HTMLFacade. Такой способ обеспечивает полное управление опосредованным вводом тестируемой системы.

```
public void testGetFlightConnAsHtml_illegalConnection()
throws Exception {
    // Настройка
    FlightConnection illegalConn = createIllegalConnection();
    Mock analyzerStub = mock(IFlightConnAnalyzer.class);
    analyzerStub.expects(once()).method("analyze")
        .will(returnValue(illegalConn));
    HTMLFacade htmlFacade =
        new HTMLFacade((IFlightConnAnalyzer)analyzerStub.proxy());
    // Вызов
    String actualHtmlString =
        htmlFacade.getFlightConnectionAsHtmlFragment(
            illegalConn.getInboundFlightNumber(),
            illegalConn.getOutboundFlightNumber());
    // Проверка
    StringBuffer expected = new StringBuffer();
    expected.append("<span class='boldRedText'>");
    expected.append("Connection time between flight ");
}
```

⁸ Для простоты обсуждения здесь не рассматриваются различные тесты для обработки ошибок, но обратите внимание, что *тест уровня* (Layer Test) упрощает вызов логики обработки ошибок.

```

expected.append(illegalConn.getInboundFlightNumber());
expected.append(" and flight ");
expected.append(illegalConn.getOutboundFlightNumber());
expected.append(" is ");
expected.append(illegalConn.getActualConnectionTime());
expected.append(" minutes.</span>");
assertEquals("returned HTML",
            expected.toString(),
            actualHtmlString);
)

```

Приходится сравнивать строковые представления кода HTML для определения правильности сгенерированного ответа. К счастью, только два таких теста проверяют базовое поведение этого компонента.

Пример: “подкожный” тест (Subcutaneous Test)

Ниже приведен исходный тест, превращенный в “подкожный” тест (Subcutaneous Test) и пропускающий презентационный уровень для проверки правильности расчета информации о подключении. Обратите внимание на отсутствие операций работы со строками внутри теста.

```

private final int LEGAL_CONN_MINS_SAME = 30;
public void testAnalyze_sameAirline_LessThanConnectionLimit()
throws Exception {
    // Настройка
    FlightConnection expectedConnection =
        createSameAirlineConn(LEGAL_CONN_MINS_SAME - 1);
    // Вызов
    IFlightConnAnalyzer theConnectionAnalyzer =
        new FlightConnAnalyzer();
    FlightConnection actualConnection =
        theConnectionAnalyzer.getConn(
            expectedConnection.getInboundFlightNumber(),
            expectedConnection.getOutboundFlightNumber());
    // Проверка
    assertNotNull("actual connection", actualConnection);
    assertFalse("IsLegal", actualConnection.isLegal());
}

```

Хотя презентационный уровень был пропущен, служебный уровень от нижележащих уровней не изолируется. Это может привести к появлению *медленных* (Slow Tests) или *нестабильных тестов* (Erratic Test, с. 267).

Пример: тест бизнес-уровня

В следующем примере показан тот же тест, преобразованный в тест служебного уровня с полной изоляцией от нижележащих уровней. Для замены этих компонентов *подставными объектами* (Mock Object) использовался пакет JMock. Подставные объекты проверяют правильность запросов на поиск рейса и вставляют в тестируемую систему соответствующие рейсы.

```
public void testAnalyze_sameAirline_EqualsConnectionLimit()
throws Exception {
    // Настройка
    Mock flightMgntStub = mock(FlightManagementFacade.class);
    Flight firstFlight = createFlight();
    Flight secondFlight = createConnectingFlight(
        firstFlight, LEGAL_CONN_MINS_SAME);
    flightMgntStub.expects(once()).method("getFlight")
        .with(eq(firstFlight.getFlightNumber()))
        .will(returnValue(firstFlight));
    flightMgntStub.expects(once()).method("getFlight")
        .with(eq(secondFlight.getFlightNumber()))
        .will(returnValue(secondFlight));
    // Вызов
    FlightConnAnalyzer theConnectionAnalyzer = new FlightConnAnalyzer();
    theConnectionAnalyzer.facade =
        (FlightManagementFacade)flightMgntStub.proxy();
    FlightConnection actualConnection =
        theConnectionAnalyzer.getConn(
            firstFlight.getFlightNumber(),
            secondFlight.getFlightNumber());
    // Проверка
    assertNotNull("actual connection", actualConnection);
    assertTrue("IsLegal", actualConnection.isLegal());
}
```

Этот тест работает очень быстро, так как служебный уровень полностью изолирован от более низких уровней. Кроме того, он работает намного стабильнее, поскольку содержит меньше кода.

Глава 19

Базовые шаблоны xUnit

Шаблоны в этой главе:

Определение теста

Тестовый метод (Test Method).....	378
Четырехфазный тест (Four-Phase Test).....	387
Метод с утверждением (Assertion Method)	390
Сообщение для утверждения (Assertion Message).....	398
Класс теста (Testcase Class).....	401

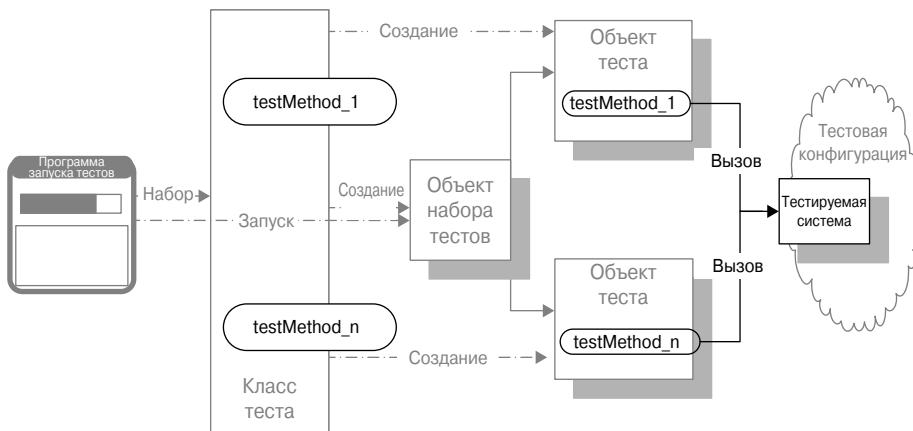
Выполнение теста

Программа запуска тестов (Test Runner)	405
Объект теста (Testcase Object).....	410
Объект набора тестов (Test Suite Object).....	414
Обнаружение тестов (Test Discovery)	420
Перечисление тестов (Test Enumeration).....	425
Выбор тестов (Test Selection).....	429

Тестовый метод (Test Method)

Куда следует вставить код теста?

Каждый тест описывается как **тестовый метод (Test Method)** некоторого класса.



Полностью автоматизированные тесты (Fully Automated Tests, с. 81) состоят из логики теста. Логика должна где-то находиться до компиляции и запуска.

Как это работает

Каждый тест определяется в виде метода, процедуры или функции, реализующей четыре фазы (см. *Четырехфазный тест*, Four-Phase Test, с. 387), необходимые для нормальной работы *полностью автоматизированных тестов* (Fully Automated Tests). Основным элементом **тестового метода (Test Method)** являются утверждения. Без них тест не будет *самопроверяющимся* (Self-Checking Test, с. 81).

Тестовая логика организовывается в соответствии с одним из шаблонов *тестовых методов (Test Method)*, чтобы тип теста легко распознавался во время чтения кода. В *простом teste успешности* (Simple Success Test) используется простой линейный поток управления от создания тестовой конфигурации до вызова тестируемой системы и проверки результата. В *тесте на ожидаемое исключение* (Expected Exception Test) языковые структуры передают управление коду обработки ошибок. Если этот код достигнут, тест считается успешным, если нет — тест завершился неудачно. В *тесте конструктора* (Constructor Test) просто создается экземпляр объекта и делаются утверждения относительно его атрибутов.

Зачем это нужно

Тестовая логика должна где-то храниться. В процедурном мире каждый тест реализуется в виде процедуры, расположенной в файле или модуле. В объектно-ориентированных языках программирования предпочтительным местом хранения тестов являются методы подчиняющего класса *теста* (Testcase Class, с. 401). На этапе выполнения методы превращаются

ются в *объекты теста* (Testcase Object, с. 410) с помощью механизмов *обнаружения тестов* (Test Discovery, с. 420) или *перечисления тестов* (Test Enumeration, с. 425).

Стандартные шаблоны используются для получения как можно более простых *тестовых методов* (Test Method). Это значительно повышает их эффективность в качестве системной документации за счет более простого описания базового поведения тестируемой системы. Намного проще распознать описываемое поведение, если только *тест на ожидаемое исключение* (Expected Exception Test) содержит такие конструкции обработки ошибок, как `try/catch`.

Замечания по реализации

Необходим механизм запуска всех *тестовых методов* (Test Method) в конкретном *классе теста* (Testcase Class). Одним из решений является определение статического метода *класса теста* (Testcase Class), который вызывает каждый из тестовых методов. Конечно, придется подсчитывать тесты и вести учет успешным и неудачным тестам. Так как эта функциональность необходима набору тестов, простым решением является создание *объекта набора тестов* (Test Suite Object, с. 414) для хранения каждого *тестового метода* (Test Method) (описание исключений из этого правила приводится во врезке “Всегда есть исключения” на с. 411). Это просто реализовать, если создать экземпляр *класса теста* (Testcase Class) для каждого *тестового метода* (Test Method) с помощью *обнаружения тестов* (Test Discovery) и *перечисления тестов* (Test Enumeration).

В статически типизированных языках Java и C# предложение `throws` может потребоваться в декларации *тестового метода* (Test Method), чтобы компилятор не жаловался на отсутствие обработки исключений, потенциально генерируемых тестируемой системой. Фактически компилятору сообщается, что *программа запуска тестов* (Test Runner, с. 405) будет заниматься исключениями.

Конечно, различные типы функциональности требуют разных *тестовых методов* (Test Method). Несмотря на это практически все тесты можно свести к трем базовым типам.

Вариант: простой тест успешности (Simple Success Test)

Большая часть программного обеспечения содержит очевидный успешный сценарий (или “счастливый маршрут”). *Простой тест успешности* (Simple Success Test) проверяет этот сценарий простым и легко узнаваемым способом.

Создается экземпляр тестируемой тестируемой системы и вызываются методы, которые необходимо проверить. После этого делается утверждение о получении ожидаемого результата. Другими словами, выполняется нормальная последовательность *четырехфазного теста* (Four-Phase Test). Исключения, которые могут произойти, не перехватываются. Их перехватом занимается *инфраструктура автоматизации тестов* (Test Automation Framework, с. 332). В противном случае получается *непонятный тест* (Obscure Test, с. 230), который вводит читателя тестов в заблуждение, заставляя думать, что исключения ожидались. Причиной этого подхода является использование *тестов как документации* (Tests as Documentation).

Еще одним положительным эффектом отказа от конструкций `try/catch` является простота отслеживания причины ошибки, так как *инфраструктура автоматизации тестов* (Test Automation Framework) показывает место появления ошибки глубоко внутри тестируемой системы, а не в точке вызова *метода с утверждением* (Assertion Method,

с. 390) (например, `fail` или `assertTrue`). Ошибки такого типа намного проще диагностировать, чем ложные утверждения.

Вариант: тест на ожидаемое исключение (Expected Exception Test)

Нет ничего сложного в создании программного обеспечения, успешно проходящего *простой тест успешности* (Simple Success Test). Большинство дефектов в программном обеспечении обнаруживается в различных альтернативных путях выполнения, особенно в коде обработки ошибок, так как эти сценарии обычно представляют собой *нетестированные требования* (Untested Requirement; см. *Ошибки в продукте*, Production Bugs, с. 303) или *нетестированный код* (Untested Code). *Тест на ожидаемое исключение* (Expected Exception Test) позволяет убедиться, что обработка ошибок написана правильно. Сначала создается тестовая конфигурация и тестируемая система вызывается всеми способами, приводящими к ошибке. Для перехвата ошибки используется любая доступная конструкция языка. Если ошибка произошла, управление будет передаваться в блок обработки ошибок. Этого перенаправления может быть достаточно для успешного завершения теста, но если тип (или содержимое сообщения об ошибке) важен (например, когда сообщение будет выводиться пользователю), для его проверки можно воспользоваться *утверждением равенства* (Equality Assertion). Если ошибки не происходит, вызывается метод `fail`, чтобы сообщить о неудачной попытке сгенерировать ошибку внутри тестируемой системы.

Тест на ожидаемое исключение (Expected Exception Test) должен создаваться для каждого исключения, которое должно генерироваться внутри тестируемой системы. Система может генерировать ошибку, так как клиент (т.е. тест) требует от нее выполнения недопустимой операции. Кроме того, система может транслировать или переводить ошибку другого используемого компонента. Не стоит писать *тест на ожидаемое исключение* (Expected Exception Test) для исключений, которые тестируемая система потенциально может сгенерировать, но которые нельзя спровоцировать принудительно, поскольку эти ошибки должны проявляться как неудачные завершения *простых тестов успешности* (Simple Success Test). Чтобы убедиться в том, что такие ошибки обрабатываются правильно, необходимо найти способ их принудительной генерации. Самым распространенным решением для этой цели является использование *тестовой заглушки* (Test Stub, с. 544) для управления опосредованным вводом тестируемой системы и генерации поддающихся ошибок внутри *тестовой заглушки* (Test Stub).

О тестах исключений очень интересно писать, так как существуют разные способы их определения в реализациях xUnit. Пакет JUnit 3.x предоставляет специальный класс `ExpectedException`. Но этот класс требует создания *класса теста* (Testcase Class) для каждого *тестового метода* (Test Method), поэтому никакой экономии за счет отсутствия блоков `try/catch` нет и результатом является большое количество очень маленьких *классов теста* (Testcase Class). В последних версиях JUnit и NUnit (для .NET) предоставляется специальный атрибут метода (в Java он называется *аннотацией*) `ExpectedException`, который вынуждает инфраструктуру неудачно завершать тест, если исключение не генерируется. Этот атрибут метода позволяет включать текст сообщения, чтобы проверять не только ожидаемый тип, но и текст исключения.

В языках с поддержкой блоков (например, Smalltalk и Ruby) обеспечиваются специальные утверждения, которым передаются блоки кода и ожидаемое исключение/объект ошибки. В *методе с утверждением* (Assertion Method) реализуется логика обработки ошибки, необходимая для подтверждения ее появления. Это значительно упрощает *тест-*

тестовый метод (Test Method) несмотря на необходимость внимательнее отслеживать имена утверждений для определения типа теста.

Вариант: тест конструктора (Constructor Test)

Если каждый тест будет проверять правильность создания объекта на этапе создания тестовой конфигурации, это приведет к значительному *дублированию тестового кода* (Test Code Duplication, с. 254). Для того чтобы избежать такого развития событий конструктор, структура которого сложнее, чем простая последовательность присвоений значений полям объекта, проверяется отдельно от остальных *тестовых методов* (Test Method). Такие *тесты конструктора* (Constructor Test) обеспечивают лучшую *локализацию дефектов* (Defect Localization, с. 78), чем включение логики проверки конструкторов в другие тесты. Для каждой сигнатуры конструктора может потребоваться один или больше тестов. Большинство *тестов конструктора* (Constructor Test) основано на шаблоне *простого теста успешности* (Simple Success Test), но можно воспользоваться *тестом на ожидаемое исключение* (Expected Exception Test) для проверки логики обработки некорректных параметров.

Необходимо проверять каждый атрибут объекта или структуры данных независимо от того, будут ли они инициализированы. Чтобы проверить инициализируемые атрибуты, можно воспользоваться *утверждением равенства* (Equality Assertion) для описания правильного значения. Для проверки неинициализируемых атрибутов можно воспользоваться *утверждением с заявленным результатом* (Stated Outcome Assertion) в соответствии с типом атрибута (например, `assertNull(anObjectReference)` для объектных переменных или указателей). Обратите внимание, что при организации тестов в виде *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639) каждое утверждение можно разместить в отдельном *тестовом методе* (Test Method) для получения оптимальной *локализации дефектов* (Defect Localization).

Вариант: тест инициализации зависимости (Dependency Initialization Test)

Если существует объект с заменяемой зависимостью, необходимо убедиться, что атрибут со ссылкой на вызываемый компонент инициализирован правильной ссылкой. *Тест инициализации зависимости* (Dependency Initialization Test) является *тестом конструктора* (Constructor Test), утверждающим правильность инициализации конкретного атрибута. Часто это делается в отдельном *тестовом методе* (Test Method) для улучшения видимости.

Пример: простой тест успешности (Simple Success Test)

В следующем примере показан тест, содержащий код перехвата вероятных исключений (эти исключения могли наблюдаваться разработчиком во время отладки кода).

```
public void testFlightMileage_asKm() throws Exception {
    // Настройка тестовой конфигурации
    Flight newFlight = new Flight(validFlightNumber);
    try {
        // Вызов тестируемой системы
        newFlight.setMileage(1122);
        // Проверка результата
        int actualKilometres = newFlight.getMileageAsKm();
        int expectedKilometres = 1810;
        // Проверка результата
    }
```

```

        assertEquals( expectedKilometres, actualKilometres);
    } catch (IllegalArgumentException e) {
        fail(e.getMessage());
    } catch (ArrayStoreException e) {
        fail(e.getMessage());
    }
}

```

Большая часть кода просто не нужна и скрывает назначение теста. К счастью, можно полностью отказаться от обработки исключений. В инфраструктуре xUnit присутствует встроенная поддержка перехвата неожиданных исключений. Можно удалить весь код обработки исключений и позволить *инфраструктуре автоматизации тестов* (Test Automation Framework) перехватить все неожиданные исключения. Неожиданные исключения считаются ошибками тестов, так как тест завершается непредсказуемо. Это полезная информация, и такая ситуация не считается более опасной, чем неудачное завершение теста.

```

public void testFlightMileage_askKm() throws Exception {
    // Настройка тестовой конфигурации
    Flight newFlight = new Flight(validFlightNumber);
    newFlight.setMileage(1122);
    // Вызов транслятора пробега
    int actualKilometres = newFlight.getMileageAsKm();
    // Проверка результатов
    int expectedKilometres = 1810;
    assertEquals( expectedKilometres, actualKilometres);
}

```

Этот пример написан на языке Java (статически типизированном языке), поэтому приходится объявлять возможность генерации исключения в сигнатуре *тестового метода* (Test Method).

Пример: тест на ожидаемое исключение (Expected Exception Test) на основе блока try/catch

В следующем примере показан почти завершенный тест для проверки варианта исключения. Неопытный разработчик создал правильные тестовые условия для генерации ошибки внутри тестируемой системы.

```

public void testSetMileage_invalidInput() throws Exception {
    // Настройка тестовой конфигурации
    Flight newFlight = new Flight(validFlightNumber);
    // Вызов тестируемой системы
    newFlight.setMileage(-1122); // некорректный параметр
    // Как убедиться в генерации исключения?
}

```

Поскольку *инфраструктура автоматизации тестов* (Test Automation Framework) перехватит исключение и зафиксирует неудачное завершение теста, *программа запуска тестов* (Test Runner) не покажет зеленый индикатор состояния даже при правильном поведении тестируемой системы. Можно вставить блок обработки ошибок вокруг точки вызова системы и воспользоваться им для инвертирования критерия успешности (завершать успешно при генерации исключения; завершать неудачно, если исключение не появилось). Ниже показана проверка генерации исключения, принятая в пакете JUnit 3.x.

```
public void testSetMileage_invalidInput() throws Exception {
    // Настройка тестовой конфигурации
    Flight newFlight = new Flight(validFlightNumber);
    try {
        // Вызов тестируемой системы
        newFlight.setMileage(-1122);
        fail("Should have thrown InvalidInputException");
    } catch(InvalidArgumentException e) {
        // Проверка результатов
        assertEquals("Flight mileage must be positive",
                    e.getMessage());
    }
}
```

Такой вариант применения `try/catch` может использоваться только в языках, поддерживающих перечисление перехватываемых исключений. Если необходимо перехватить универсальное или генерируемое *методом с утверждением* (Assertion Method) `fail` исключение, такая конструкция не сработает, так как оно передает управление в фрагмент `catch`. В подобных случаях необходимо использовать такой же тип *теста на ожидаемое исключение* (Expected Exception Test), как и в тестах *специальных утверждений* (Custom Assertion, с. 495).

```
public void testSetMileage_invalidInput2() throws Exception {
    // Настройка тестовой конфигурации
    Flight newFlight = new Flight(validFlightNumber);
    try {
        // Вызов тестируемой системы
        newFlight.setMileage(-1122);
        // Если система генерирует исключение,
        // здесь нельзя вызывать fail()
    } catch(AssertionFailedError e) {
        // Проверка результатов
        assertEquals("Flight mileage must be positive",
                    e.getMessage());
        return;
    }
    fail("Should have thrown InvalidInputException");
}
```

Пример: тест на ожидаемое исключение (Expected Exception Test) на основе атрибутов метода

В пакете NUnit предоставляется атрибут метода, позволяющий создать *тест на ожидаемое исключение* (Expected Exception Test) без явного использования блока `try/catch`.

```
[Test]
[ExpectedException(typeof(InvalidArgumentException),
                   "Flight mileage must be > zero")]
public void testSetMileage_invalidInput_AttributeWithMessage()
{
    // Настройка тестовой конфигурации
    Flight newFlight = new Flight(validFlightNumber);
    // Вызов тестируемой системы
    newFlight.setMileage(-1122);
}
```

Такой подход значительно уменьшает тест, но не позволяет указывать ничего, кроме типа и текста исключения. Если необходимо проверить утверждения относительно содержимого исключения (чтобы избежать *чувствительного сравнения* (Sensitive Equality); см. “Хрупкий” тест, Fragile Test, с. 277), придется использовать конструкцию `try/catch`.

Пример: тест на ожидаемое исключение (Expected Exception Test) на основе блочной конструкции

В пакете SUnit для языка Smalltalk обеспечивается еще один механизм получения такого же результата.

```
testSetMileageWithInvalidInput
self
    should: [Flight new mileage: -1122]
    raise: RuntimeError new 'Should have raised error'
```

Поскольку Smalltalk поддерживает блочные конструкции, методу `should:raise:` передаются выполняемый блок кода и ожидаемый объект `Exception`. В пакете Test::Unit для языка Ruby применяется такой же подход.

```
def testSetMileage_invalidInput
  flight = Flight.new();
  assert_raises( RuntimeError, "Should have raised error" ) do
    flight.setMileage(-1122)
  end
end
```

Код между `do/end` является конструкцией, которая выполняется методом `assert_raises`. Если не генерируется экземпляр первого аргумента (класса `RuntimeError`), тест завершается неудачно и выводит указанное сообщение об ошибке.

Пример: тест конструктора (Constructor Test)

В этом примере необходимо создать полет для проверки преобразования единиц полетной дальности (из миль в километры). Сначала необходимо убедиться, что полет создан правильно.

```
public void testFlightMileage_asKm2() throws Exception {
    // Настройка тестовой конфигурации
    // Вызов конструктора
    Flight newFlight = new Flight(validFlightNumber);
    // Проверка созданного объекта
    assertEquals(validFlightNumber, newFlight.number);
    assertEquals("", newFlight.airlineCode);
    assertNull(newFlight.airline);
    // Установка полетной дальности
    newFlight.setMileage(1122);
    // Вызов преобразователя дальности
    int actualKilometres = newFlight.getMileageAsKm();
    // Проверка результата
    int expectedKilometres = 1810;
    assertEquals(expectedKilometres, actualKilometres);
```

```
// То же самое, но с отмененным полетом
newFlight.cancel();
try {
    newFlight.getMileageAsKm();
    fail("Expected exception");
} catch (InvalidRequestException e) {
    assertEquals("Cannot get cancelled flight mileage",
                e.getMessage());
}
}
```

Этот тест не является *тестом одного условия* (Single Condition Test, с. 99), так как в нем рассматриваются и создание объекта и поведение преобразователя расстояния. Если создание объекта завершается неудачно, причины отказа не будут известны, пока не начнется отладка теста.

Желательно разделить этот “энергичный” тест (Eager Test; см. *Рулетка утверждений*, Assertion Roulette, с. 264) на два *теста одного условия* (Single Condition Test). Проще всего сделать это через клонирование *тестового метода* (Test Method) и переименование каждой копии в соответствии с реализованной функциональностью. Ненужный код должен быть удален.

Ниже приведен пример простого *теста конструктора* (Constructor Test).

```
public void testFlightConstructor_OK() throws Exception {
    // Настройка тестовой конфигурации
    // Вызов тестируемой системы
    Flight newFlight = new Flight(validFlightNumber);
    // Проверка результатов
    assertEquals(validFlightNumber, newFlight.number);
    assertEquals("", newFlight.airlineCode);
    assertNull(newFlight.airline);
}
```

Продолжая работу над этим тестом, можно указать, что должно происходить при передаче конструктору некорректного аргумента. Для этого в качестве шаблона имеет смысл воспользоваться *тестом на ожидаемое исключение* (Expected Exception Test).

```
public void testFlightConstructor_badInput() {
    // Настройка тестовой конфигурации
    BigDecimal invalidFlightNumber = new BigDecimal(-1023);
    // Вызов тестируемой системы
    try {
        Flight newFlight = new Flight(invalidFlightNumber);
        fail("Didn't catch negative flight number!");
    } catch (IllegalArgumentException e) {
        // Проверка результатов
        assertEquals("Flight numbers must be positive",
                    e.getMessage());
    }
}
```

Полностью проверив логику конструктора, можно перейти к созданию *простого теста успешности* (Simple Success Test) для функции преобразования дальности. Обратите внимание, насколько проще стал тест, когда основное внимание стало уделяться бизнес-логике.

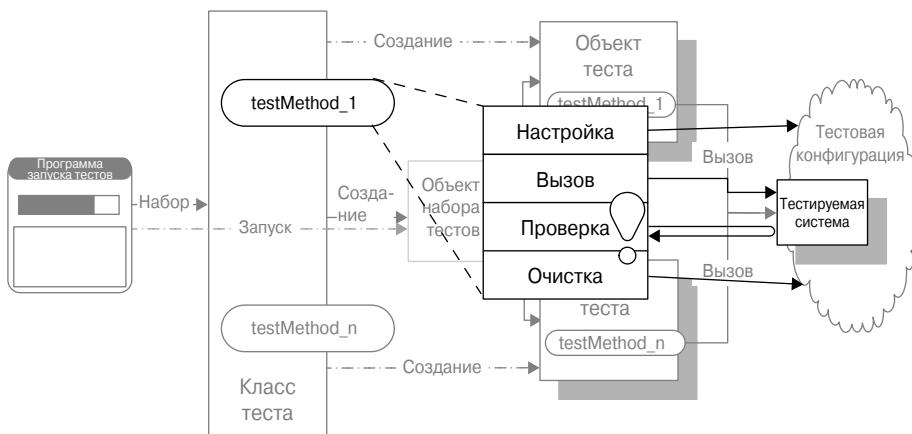
```
public void testFlightMileage_asKm() throws Exception {  
    // Настройка тестовой конфигурации  
    Flight newFlight = new Flight(validFlightNumber);  
    newFlight.setMileage(1122);  
    // Вызов транслятора дальности  
    int actualKilometres = newFlight.getMileageAsKm();  
    // Проверка результатов  
    int expectedKilometres = 1810;  
    assertEquals(expectedKilometres, actualKilometres);  
}
```

Но что же произойдет, если логика конструктора содержит ошибку? Тест завершится неудачно, так как его вывод зависит от значения, передаваемого конструктору. Тест конструктора также завершится неудачно. Этот результат заставит, в первую очередь, смотреть на логику конструктора. После решения обнаруженной проблемы оба теста завершатся успешно. Если это не произойдет, придется исправлять и метод `getMileageAsKm`. Это хороший *пример локализации дефектов* (Defect Localization).

Четырехфазный тест (Four-Phase Test)

Как структурировать тестовую логику, чтобы назначение теста было очевидным?

Каждый тест составляется из четырех отдельных фрагментов, выполняемых последовательно.



Как это работает

В результате проектирования каждый тест состоит из четырех отдельных фаз, выполняемых последовательно: настройка тестовой конфигурации, вызов тестируемой системы, проверка результата и очистка тестовой конфигурации.

- Во время первой фазы создаются тестовая конфигурация (картина “до запуска”), которая необходима тестируемой системе для проявления ожидаемого поведения, а также все необходимо для наблюдения за фактическим результатом (например, через использование *тестового двойника*, Test Double, с. 538).
 - Во время второй фазы происходит взаимодействие с тестируемой системой.
 - Во время третьей фазы проверяется наступление ожидаемого результата.
 - Во время четвертой фазы тестовая конфигурация удаляется, чтобы вернуть среду выполнения в исходное состояние.

Зачем это нужно

Читатель теста должен иметь возможность быстро определить, какое поведение проверяет тест. Все очень усложняется, если один тест проверяет несколько аспектов поведения тестируемой системы (когда одна проверка создает конфигурацию, другая — взаимодействует с системой, а еще несколько — проверяют состояние тестируемой системы после взаимодействия). Явное выделение четырех фаз делает назначение теста очевидным.

Во время фазы создания тестовой конфигурации настраивается состояние тестируемой системы перед началом теста, что составляет один из важных входных параметров. Во время фазы вызова тестируемой системы выполняется фактический запуск тестируемого программного обеспечения. При чтении теста должно быть понятно, какое программное обеспечение запускается. Во время фазы проверки результатов описывается ожидаемый результат. Последняя фаза, очистка тестовой конфигурации, обеспечивает удаление оставшегося мусора. Не стоит скрывать важную логику теста командами очистки, так как с точки зрения *тестов как документации* (Tests as Documentation, с. 79) очистка не несет никакой полезной нагрузки.

Важно преодолеть соблазн проверить как можно больше функций внутри одного *тестового метода* (Test Method, с. 378), так как это приводит к появлению *непонятных тестов* (Obscure Test, с. 230). На самом деле лучше иметь много небольших *тестов одного условия* (Single Condition Test, с. 99). Комментарии на границах четырех фаз хорошо дисциплинируют разработчика, так как быстро делают очевидным появление нескольких условий в составе одного теста (когда несколько фаз вызова тестируемой системы перемежаются несколькими фазами проверки результатов). Такой тест будет работать, но его ценность с точки зрения *локализации дефектов* (Defect Localization, с. 78) значительно снизится по сравнению с *несколькими тестами одного условия* (Single Condition Test).

Замечания по реализации

Существует несколько вариантов реализации *четырехфазных тестов* (Four-Phase Test). В простейшем случае каждый тест работает независимо и все четыре фазы теста описываются внутри *тестового метода* (Test Method). Такая структура предполагает использование *встроенной настройки* (In-line Setup, с. 434) и *очистки сборкой мусора* (Garbage-Collected Teardown, с. 518) или *встроенной очистки* (In-line Teardown, с. 527). Это наиболее подходящий вариант при структурировании тестов в виде *класса теста для каждого класса* (Testcase Class per Class, с. 627) или *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639).

Еще один вариант основан на поддержке *неявной настройки* (Implicit Setup, с. 449) и *неявной очистки* (Implicit Teardown, с. 533) со стороны *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332). В таком случае общая логика настройки тестовой конфигурации выносится в методы *setUp* и *tearDown* *класса теста* (Testcase Class, с. 401). В результате внутри *тестового метода* (Test Method) остаются только фазы вызова тестируемой системы и проверки результата. Подобный подход оправдан при организации в виде *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture). Кроме того, данная конструкция может применяться для создания общего фрагмента тестовой конфигурации при организации в виде *класса теста для каждого класса* (Testcase Class per Class) или *класса теста для каждой функции* (Testcase Class per Feature), а также для удаления тестовой конфигурации при использовании *автоматической очистки* (Automated Teardown, с. 521).

Пример: четырехфазный тест (Four-Phase Test) со встроенной настройкой

Ниже приведен пример теста с явно выделенными фазами.

```
public void testGetFlightsByOriginAirport_NoFlights_inline()
throws Exception {
```

```
// Настройка тестовой конфигурации
NonTxFlightMngtFacade facade = new NonTxFlightMngtFacade();
BigDecimal airportId = facade.createTestAirport("1OF");
try {
    // Вызов тестируемой системы
    List flightsAtDestination1 =
        facade.getFlightsByOriginAirport(airportId);
    // Проверка результата
    assertEquals(0, flightsAtDestination1.size());
} finally {
    // Очистка тестовой конфигурации
    facade.removeAirport(airportId);
}
}
```

Все четыре фазы теста присутствуют в коде метода. Так как вызов *метода с утверждением* (Assertion Method, с. 390) может привести к генерации исключения, код очистки тестовой конфигурации необходимо заключить в конструкцию `try/finally` для обеспечения его безусловного запуска.

Пример: четырехфазный тест (Four-Phase Test) с неявной настройкой

Ниже представлен тот же тест, в котором логика создания и очистки тестовой конфигурации вынесена за пределы *тестового метода* (Test Method).

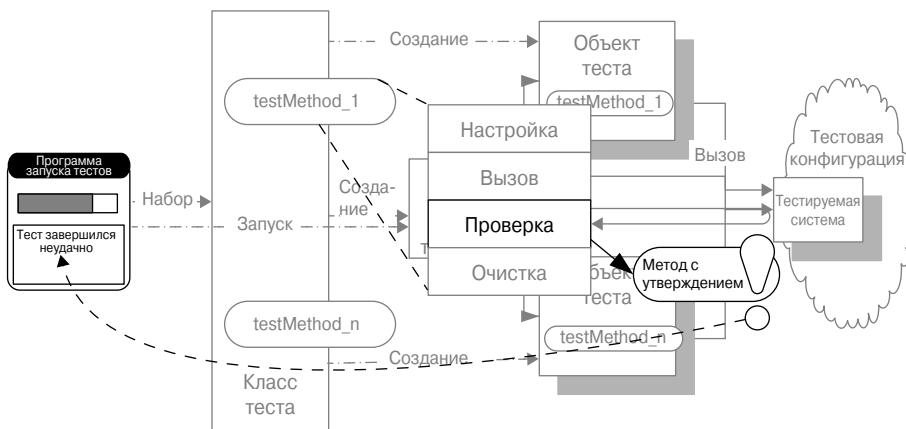
```
NonTxFlightMngtFacade facade = new NonTxFlightMngtFacade();
private BigDecimal airportId;
protected void setUp() throws Exception {
    // Настройка тестовой конфигурации
    super.setUp();
    airportId = facade.createTestAirport("1OF");
}
public void testGetFlightsByOriginAirport_NoFlights_implicit()
    throws Exception {
    // Вызов тестируемой системы
    List flightsAtDestination1 =
        facade.getFlightsByOriginAirport(airportId);
    // Проверка результата
    assertEquals(0, flightsAtDestination1.size());
}
protected void tearDown() throws Exception {
    // Настройка тестовой конфигурации
    facade.removeAirport(airportId);
    super.tearDown();
}
```

Поскольку метод `tearDown` вызывается автоматически даже в случае неудачного завершения теста, конструкция `try/finally` внутри тестового метода не нужна. С другой стороны, ссылки на тестовую конфигурацию приходится хранить в переменных экземпляра, а не в локальных переменных.

Метод с утверждением (Assertion Method)

Как разработать самопроверяющийся тест?

Вызывается вспомогательный метод, проверяющий достижение желаемого результата.



Ключевым свойством *полностью автоматизированных тестов* (Fully Automated Tests, с. 81) является их способность к *самопроверке* (Self-Checking Test, с. 81), которая позволяет отказаться от проверки правильности результата при каждом запуске. Данная стратегия предполагает описание ожидаемого результата таким образом, чтобы его можно было проверить автоматически средствами самого теста.

Метод с утверждением (Assertion Method) позволяет выразить ожидаемый результат в формате, понятном как компьютеру, так и человеку, который может использовать *тесты как документацию* (Tests as Documentation, с. 79).

Как это работает

Ожидаемый результат теста описывается последовательностью утверждений. Тест завершается успешно, если все утверждения являются истинными. Утверждения реализованы в виде *методов с утверждением* (Assertion Method), содержащих механизм неудачного завершения теста. *Метод с утверждением* (Assertion Method) может предоставляться *инфраструктурой автоматизации тестов* (Test Automation Framework, с. 332) или описываться разработчиком в виде *специального утверждения* (Custom Assertion, с. 495).

Зачем это нужно

Описание ожидаемого результата с помощью *условной логики теста* (Conditional Test Logic, с. 243) приводит к росту объема кода и делает тесты сложными для чтения и понимания. Кроме того, повышается вероятность *дублирования тестового кода* (Test Code Duplication, с. 254) и появления *тестов с ошибками* (Buggy Test, с. 296). Методы с утверждением (Assertion Method) позволяют обойти эти проблемы за счет вынесения сложных

конструкций в повторно используемые *вспомогательные методы теста* (Test Utility Method, с. 610). После этого такие методы можно проверить с помощью тестов *вспомогательных методов теста* (Test Utility Test).

Замечания по реализации

Хотя во всех реализациях xUnit предоставляется набор *методов с утверждением* (Assertion Method), предоставленный инструментарием может значительно отличаться в зависимости от реализации. Ниже приведены ключевые различия между разными реализациями:

- способ вызова *метода с утверждением* (Assertion Method);
- способ выбора подходящего *метода с утверждением* (Assertion Method);
- объем информации, включаемой в *сообщение для утверждения* (Assertion Message, с. 398).

Вызов встроенных методов с утверждением

Способ вызова *метода с утверждением* (Assertion Method) из *тестового метода* (Test Method) зависит от используемых языка и инфраструктуры. Языковые возможности определяют пределы возможного и предпочтительного, а инфраструктура предоставляет варианты выбора. Имена *методов с утверждением* (Assertion Method) выбирались под влиянием способа запуска. Ниже приведены наиболее распространенные способы получения доступа к *методу с утверждением* (Assertion Method).

- *Метод с утверждением* (Assertion Method) наследуется от *суперкласса теста* (Testcase Superclass, с. 646), предоставленного инфраструктурой. Такие методы могут вызываться как локальные относительно *класса теста* (Testcase Class, с. 401). Например, в оригинальной версии Java JUnit этот подход используется для предоставления *суперкласса теста* (Testcase Superclass), наследующего класс Assert (и содержащего фактическую реализацию *методов с утверждением*, Assertion Method).
- *Метод с утверждением* (Assertion Method) предоставляется через глобально доступный класс или модуль. Имя модуля или класса используется для полной квалификации имени метода с утверждением (Assertion Method). Такой подход используется в NUnit (например, `Assert.IsTrue(x)`). Инфраструктура JUnit позволяет вызывать утверждения в виде статических методов класса Assert (например, `Assert.AssertTrue(x)`), но обычно это не требуется, так как они наследуются через *суперкласс теста* (Testcase Superclass).
- *Метод с утверждением* (Assertion Method) предоставляется в виде “миксинов” или макросов. Например, `Test::Unit` в языке Ruby предоставляет *методы с утверждением* (Assertion Method) в модуле Assert, который можно включать в любой класс, таким образом позволяя использовать методы с утверждением как определенные в этом *классе теста* (Testcase Class) (например, `assert_equal(a, b)`). (Такой подход особенно полезен при создании *подставных объектов* (Mock Object, с. 558), так как они находятся за пределами *класса теста* (Testcase Class), но требуют возможности вызывать *метод с утверждением* (Assertion Method).) С другой стороны, инфраструктура CppUnit определяет *методы с утверждением* (Assertion Method) в виде макросов, которые разворачиваются до передачи кода компилятору.

Сообщение для утверждения

Обычно *метод с утверждением* (Assertion Method) принимает необязательное *сообщение для утверждения* (Assertion Message) в виде текстового параметра. Значение параметра выводится, когда утверждение оказывается ложным. Подобная структура позволяет разработчику теста описать, почему *метод с утверждением* (Assertion Method) завершился неудачно и что должно было произойти. Обнаруженную тестом ошибку будет проще диагностировать, если *метод с утверждением* (Assertion Method) предоставляет больше информации. Выбор подходящего *метода с утверждением* (Assertion Method) значительно приближает к этой цели, так как многие встроенные *методы с утверждением* (Assertion Method) обеспечивают полезную диагностическую информацию о значениях аргументов. Это особенно справедливо для *утверждений равенства* (Equality Assertion).

Одним из основных отличий между реализациями инфраструктуры xUnit является расположение необязательного *сообщения для утверждения* (Assertion Message) в списке аргументов. В большинстве реализаций этот параметр указывается в конце списка. Но в пакете JUnit *сообщение для утверждения* (Assertion Message) является первым аргументом.

Выбор правильного утверждения

Вызывая *метод с утверждением* (Assertion Method) из *тестового метода* (Test Method), мы преследуем две цели:

- неудачно завершить тест, если результат отличается от ожидаемого;
- документировать правильное поведение тестируемой системы (т.е. использовать *тесты как документацию*, Tests as Documentation).

Для достижения этих целей необходимо стараться использовать наиболее подходящий *метод с утверждением* (Assertion Method). Хотя синтаксис и соглашения об именовании меняются от реализации к реализации xUnit, в большинстве случаев предоставляется базовый набор утверждений, попадающих в следующие категории.

- *Утверждение с единственным результатом* (Single Outcome Assertion), например `fail`. Не принимает аргументов, так как всегда ведет себя одинаково.
- *Утверждение с заявленным результатом* (Stated Outcome Assertion), например `assertNotNull(anObjectReference)` и `assertTrue(aBooleanExpression)`. Сравнивает единственный аргумент с результатом, подразумеваемым в названии метода.
- *Утверждение об ожидаемом исключении* (Expected Exception Assertion), например `assert_raises(expectedError) { codeToExecute }`. Такие утверждения оценивают блок кода и единственный аргумент в виде ожидаемого исключения.
- *Утверждение равенства* (Equality Assertion), например `assertEqual(expected, actual)`. Проверяет равенство двух объектов или значений.
- *Нечеткое утверждение равенства* (Fuzzy Equality Assertion), например `assertEqual(expected, actual, tolerance)`. Определяет “достаточно ли близки” два значения. В качестве дополнительных аргументов принимает “допуск” и “маску сравнения”.

Вариант: утверждение равенства (Equality Assertion)

Утверждение равенства (Equality Assertion) является наиболее распространенным примером *метода с утверждением* (Assertion Method). Такие утверждения используются для сравнения фактического результата с ожидаемым, выраженным в виде *точного значения* (Literal Value, с. 718) или *ожидаемого объекта* (Expected Object; см. *Проверка состояния*, State Verification, с. 484). По соглашению ожидаемое значение указывается первым, а за ним следует фактическое. Генерируемое инфраструктурой диагностическое сообщение имеет смысл только при таком порядке передачи параметров. Равенство двух объектов обычно определяется через вызов метода equals ожидаемого объекта. Если определение метода equals в тестируемой системе не подходит для использования в тестах, можно вызывать *утверждение равенства* (Equality Assertion) относительно отдельных полей объекта или реализовать специфичное для теста равенство в *связанном с тестом подклассе* (Test-Specific Subclass, с. 591) класса *ожидаемого объекта* (Expected Object).

Вариант: нечеткое утверждение равенства (Fuzzy Equality Assertion)

Если невозможно гарантировать полное равенство из-за разной точности вычислений или прогнозируемого изменения значений, можно воспользоваться *нечетким утверждением равенства* (Fuzzy Equality Assertion). Обычно такие утверждения принимают форму *утверждения равенства* (Equality Assertion) с дополнительным параметром “допуска” (или “маски сравнения”), описывающим, насколько точно фактический аргумент должен совпадать с ожидаемым. Наиболее распространенным примером *нечеткого утверждения равенства* (Fuzzy Equality Assertion) является сравнение чисел с плавающей точкой, при котором необходимо учитывать ограниченную точность. Для этого в сравнение передается “допуск” (максимальное допустимое расстояние между двумя значениями).

Такой же подход используется при сравнении документов в формате XML, когда непосредственное сравнение строк может привести к неудачному завершению, поскольку в некоторых полях содержатся непредсказуемые значения. В таком случае используется “схема сравнения” с описанием полей, которые необходимо сравнивать или игнорировать. Данное *утверждение равенства* (Equality Assertion) похоже на утверждение соответствия строки регулярному выражению или другой форме описания шаблонов.

Вариант: утверждение с заявлением результатом (Stated Outcome Assertion)

Утверждение с заявлением результатом (Stated Outcome Assertion) позволяет указать ожидаемый результат без его передачи в качестве аргумента. Результат должен быть достаточно распространенным, чтобы заслужить отдельный *метод с утверждением* (Assertion Method). Ниже приведены наиболее типичные примеры.

- `assertTrue(aBooleanExpression)`. Завершает тест неудачно, если выражение принимает значение FALSE.
- `assertNotNull(anObjectReference)`. Завершает тест неудачно, если objectReference не ссылается на действительный объект.

Утверждение с заявлением результатом (Stated Outcome Assertion) часто используется в виде *сторожевого утверждения* (Guard Assertion, с. 510), чтобы избежать написания *условной логики теста* (Conditional Test Logic).

Вариант: утверждение об ожидаемом исключении (Expected Exception Assertion)

В языках с поддержкой блочных конструкций можно воспользоваться вариантом *утверждения с заявленным результатом* (Stated Outcome Assertion), принимающим дополнительный параметр с ожидаемым исключением. Такое *утверждение об ожидаемом исключении* (Expected Exception Assertion) позволяет сказать: “Запусти этот блок кода и убедись, что генерируется следующее исключение”. Такой формат дает более компактный результат по сравнению с конструкциями на основе `try/catch`. Ниже приведены некоторые типичные примеры таких утверждений:

- `should: [aBlockToExecute] raise: expectedException` — Smalltalk SUnit;
- `assert_raises(expectedError) { codeToExecute }` — Ruby Test::Unit.

Вариант: утверждение с единственным результатом (Single Outcome Assertion)

Утверждение с единственным результатом (Single Outcome Assertion) всегда ведет себя одинаково. Чаще всего из утверждений такого типа используется утверждение `fail`, заставляющее тест завершаться неудачно. Обычно оно используется в двух ситуациях.

- В качестве *утверждения незаконченного теста* (Unfinished Test Assertion, с. 514), когда тест идентифицирован и реализован в виде практически пустого *тестового метода* (Test Method). Вставив вызов `fail`, можно заставить *программу запуска тестов* (Test Runner, с. 405) напомнить о необходимости завершить реализацию теста.
- Как часть блока `try/catch` (или эквивалентной конструкции) в *тесте на ожидаемое исключение* (Expected Exception Test). Вызов `fail` вставляется внутри блока `try` сразу после вызова, который должен сгенерировать исключение. Если никаких утверждений относительно сгенерированного исключения не планируется, можно избежать использования пустого блока `catch` и вставить утверждение `success`, показывающее, что это ожидаемый результат.

Одним из мест, где *утверждение с единственным результатом* (Single Outcome Assertion) использоваться не должно, является *условная логика теста* (Conditional Test Logic). Практически не существует причин для использования условной логики внутри *тестового метода* (Test Method), так как существуют более осмысленные способы обработки ситуации с помощью других *методов с утверждением* (Assertion Method). Например, *сторожевое утверждение* (Guard Assertion) делает тесты проще для понимания и защищает от получения некорректных результатов.

Мотивирующий пример

В следующем примере показан тип кода, который был бы необходим для проверки результатов без *методов с утверждением* (Assertion Method).

```
if (x.equals(y)) {
    throw new AssertionFailedError(
        "expected: <" + x.toString() +
        "> but found: <" + y.toString() + ">");
} else { // Отлично, продолжаем
    // ...
}
```

К сожалению, такой код приведет к генерации исключения `NullPointerException`, если `x` имеет значение `null`. В то же время отличить это исключение от ошибки в тестируемой системе будет крайне сложно. Таким образом, придется вставить охранные конструкции вокруг этого кода, чтобы всегда генерировалось только исключение `AssertionFailedException`.

```
if (x == null) { // Нельзя использовать null.equals(null)
    if (y == null) { // Оба равны null, а значит, равны друг другу
        return;
    } else {
        throw new AssertionFailedError(
            "expected null but found: <" + y.toString() + ">") ;
    }
} else if (!x.equals(y)) { // Сравнимы, но не равны!
    throw new AssertionFailedError(
        "expected: <" + x.toString() +
        "> but found: <" + y.toString() + ">") ;
} // Равно
```

Ой! Все так запуталось. И все это придется повторить для каждого проверяемого атрибута. Это не очень хорошо, и должен существовать лучший способ.

Замечания по рефакторингу

К счастью, создатели xUnit распознали эту проблему и воспользовались рефакторингом *выделение метода* (Extract Method) для создания библиотеки *методов с утверждением* (Assertion Method). Достаточно заменить путаницу операторов `if` вызовом соответствующего *метода с утверждением* (Assertion Method). В следующем примере кода используется вызов `assertEquals` из пакета JUnit. Хотя назначение этого примера совпадает с назначением показанного выше кода, он был переписан в терминах охранных конструкций, обнаруживающих равенство сущностей.

```
/***
 * Подтвердить равенство двух объектов. Если они не равны,
 * генерируется исключение AssertionFailedError и указанное
 * сообщение.
 */
static public void assertEquals(String message,
                               Object expected,
                               Object actual) {
    if (expected == null && actual == null)
        return;
    if (expected != null && expected.equals(actual))
        return;
    failNotEquals(message, expected, actual);
}
```

Метод `failNotEquals` является *вспомогательным методом теста* (Test Utility Method), который завершает тест неудачно и выводит диагностическое сообщение.

Пример: утверждение равенства (Equality Assertion)

Ниже представлена та же логика, переписанная для использования *утверждения равенства* (Equality Assertion) из пакета JUnit.

```
assertEquals(x, y);
```

А вот то же утверждение на языке C#. Обратите внимание на использование квалификатора имени класса и отличие в имени метода.

```
Assert.AreEqual(x, y);
```

Пример: нечеткое утверждение равенства (Fuzzy Equality Assertion)

Для сравнения двух чисел с плавающей точкой (которые на самом деле очень редко бывают равны) необходимо указать допустимую разницу с помощью *нечеткого утверждения равенства* (Fuzzy Equality Assertion).

```
assertEquals(3.1415, diameter/2/radius, 0.001);
assertEquals(expectedXml, actualXml, elementsToCompare);
```

Пример: утверждение с заявленным результатом (Stated Outcome Assertion)

Для принудительного получения конкретного результата используется *утверждение с заявленным результатом* (Stated Outcome Assertion).

```
assertNotNull(a);
assertTrue(b > c);
assertNonZero(b);
```

Пример: утверждение об ожидаемом исключении (Expected Exception Assertion)

Ниже приведен пример проверки генерации правильного исключения. Данный пример написан с использованием Smalltalk Sunit.

```
self
  should: [Flight new mileage: -1122]
  raise: RuntimeError new 'Should have raised error'
```

Ключевое слово `should:` указывает на запускаемый блок кода (заключенный в квадратные скобки), а `raise:` описывает ожидаемый объект исключения. В языке Ruby этот же пример выглядит так.

```
assert_raises(RuntimeError,
  "Should have raised error")
  {flight.setMileage(-1122)}
```

Кроме того, синтаксис языка Ruby позволяет разделять блоки кода ключевыми словами `do/end` вместо фигурных скобок.

```
assert_raises(RuntimeError, "Should have raised error") do
  flight.setMileage(-1122)
end
```

Пример: утверждение с единственным результатом (Single Outcome Assertion)

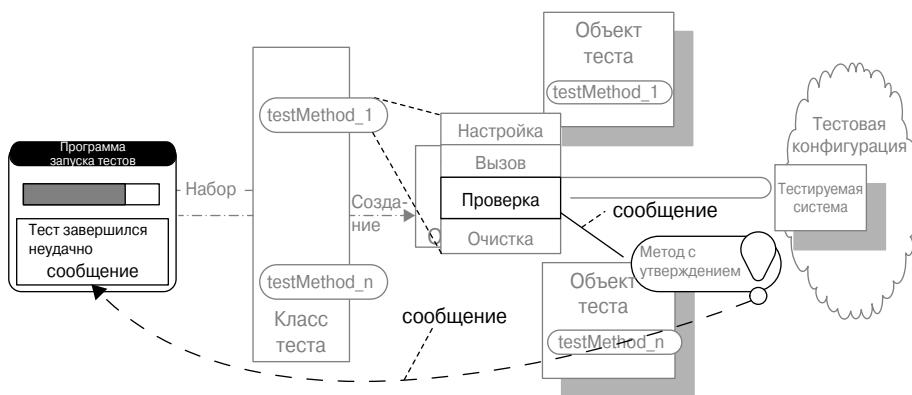
Для принудительного неудачного завершения теста используется *утверждение с единственным результатом* (Single Outcome Assertion).

```
fail("Expected an exception");  
unfinishedTest();
```

Сообщение для утверждения (Assertion Message)

Как структурировать тестовую логику для идентификации ложного утверждения?

В каждый вызов метода с утверждением добавляется описательный строковый аргумент.



Самопроверяемость тестов (Self-Checking Test, с. 81) достигается через включение вызовов *методов с утверждением* (Assertion Method, с. 390), описывающих ожидаемый результат. При неудачном завершении теста *программа запуска тестов* (Test Runner, с. 405) вносит соответствующую запись в журнал тестирования.

Правильно составленное *сообщение для утверждения* (Assertion Message) значительно упрощает поиск ложного утверждения и определение симптомов на момент неудачного завершения.

Как это работает

Каждый *метод с утверждением* (Assertion Method) принимает необязательный строковый параметр, который включается в журнал тестирования. Если утверждение оказывается ложным, *сообщение для утверждения* (Assertion Message) выводится в журнале *программы запуска тестов* (Test Runner) вместе с остальным выводом метода с утверждением.

Когда это использовать

По этому вопросу мнения разделились. Разработчики, придерживающиеся принципа “одно утверждение на один тестовый метод”, верят, что *сообщение для утверждения* (Assertion Message) является лишней сущностью, так как только одно утверждение может оказаться ложным, а значит, его несложно найти. При этом *метод с утверждением* (Assertion Method) получает обязательные аргументы, но не получает текст сообщения.

С другой стороны, разработчики методов с несколькими или многими утверждениями вынуждены всегда вставлять сообщение, указывающее на конкретное утверждение. Та-

кая информация особенно полезна, если тесты часто запускаются с помощью *программы запуска тестов для командной строки* (Command-Line Test Runner), которые редко предоставляют информацию о точке неудачного завершения.

Замечания по реализации

Очень просто указать на необходимость сообщения для каждого вызова метода с утверждением. Но что вписать в это сообщение? Иногда полезно поставить себя на место читателя журнала тестирования и задать себе вопрос: “Какую информацию необходимо извлечь из этого журнала?”

Вариант: сообщение, идентифицирующее утверждение (Assertion-Identifying Message)

Если несколько утверждений одного типа расположены в одном и том же *тестовом методе* (Test Method, с. 378), значительно сложнее определить, какое из них привело к неудачному завершению метода. Включив уникальный текст в каждое *сообщение для утверждения* (Assertion Message), можно значительно упростить идентификацию ложного утверждения. Чаще всего в качестве текста сообщения используются имена переменной или атрибута. Это очень простое и не требующее обдумывания решение. Кроме того, утверждения можно нумеровать. Эта информация также будет уникальной, но менее интуитивно понятной, поскольку в поисках ложного утверждения придется рассматривать исходный код.

Вариант: сообщение, описывающее ожидания (Expectation-Describing Message)

При неудачном завершении теста известно, что произошло на самом деле. Но остается вопрос: “Что должно было произойти?” Существует несколько способов документирования ожидаемого поведения. Например, можно вставить комментарии в код тестов. Более правильным решением считается описание ожидаемого поведения внутри *сообщения для утверждения* (Assertion Message). Хотя в случае *утверждений равенства* (Equality Assertion) это делается автоматически, для любого *утверждения с заявленным результатом* (Stated Outcome Assertion) эту информацию придется вносить самостоятельно.

Вариант: сообщение с описанием аргументов (Argument-Describing Message)

Некоторые типы *методов с утверждением* (Assertion Method) выводят менее полезные сообщения о неудачном завершении. Среди наиболее бесполезных можно выделить *утверждение с заявлением результатом* (Stated Outcome Assertion), например `assertTrue(aBooleanExpression)`. Если такое утверждение оказывается ложным, известно только то, что заявленный результат не был получен. В таких случаях в составе *сообщения для утверждения* (Assertion Message) можно указать вычисляемое выражение (с фактическими значениями). После этого разработчик может просмотреть журнал тестирования и определить, что привело к неудачному завершению теста.

Мотивирующий пример

```
assertTrue(a > b);
assertTrue(b > c);
```

Этот код приводит к появлению сообщения о неудачном завершении, которое выглядит как “Assertion failed”. Из этого сообщения даже не ясно, какое утверждение оказалось ложным. Не очень полезно, не правда ли?

Замечания по рефакторингу

Проблема решается добавлением одного параметра к каждому вызову *метода с утверждением* (Assertion Method). В данном случае необходимо показать, что “*a*” должно быть больше “*b*”. Конечно, при этом полезно видеть фактические значения “*a*” и “*b*”. Вся эта информация может быть добавлена в *сообщение для утверждения* (Assertion Message) с помощью подходящей операции конкатенации строк.

Пример: сообщение, описывающее ожидания

Ниже приведен тот же тест с добавлением *сообщения с описанием аргументов* (Argument-Describing Message).

```
assertTrue("Expected a > b but a was '" + a.toString() +  
          "' and b was '" + b.toString() + "'",  
          a.gt(b));  
assertTrue("Expected b > c but b was '" + b.toString() +  
          "' and c was '" + c.toString() + "'",  
          b > c);
```

В результате будет выводиться полезное сообщение о неудачном завершении.

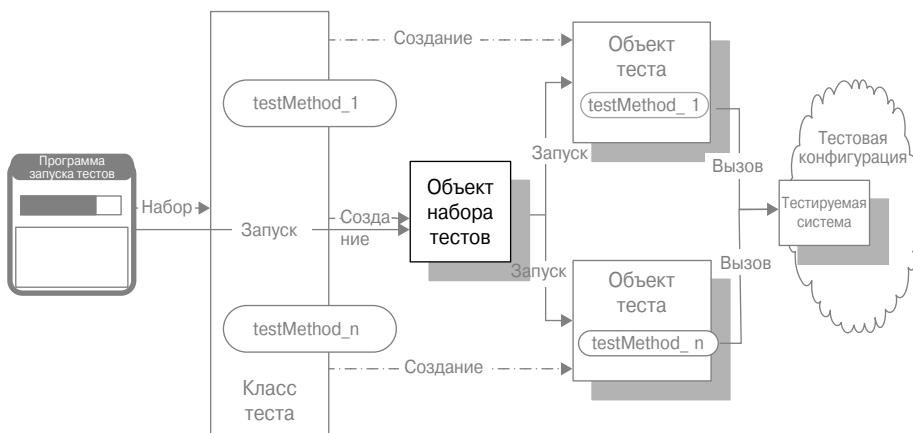
```
Assertion Failed. Expected a > b but a was '17' and b was '19'.
```

Конечно, вывод теста будет еще более осмысленным, если переменные будут иметь описательные имена!

Класс теста (Testcase Class)

Куда вписывать код тестов?
Наборы связанных тестовых методов (Test Method) группируются в один класс теста (Testcase Class).

Также известен как:
Тестовая конфигурация (Test Fixture)



Логика тестов находится внутри *тестовых методов* (Test Method), которые, в свою очередь, должны быть связаны с классом. *Класс теста* (Testcase Class) обеспечивает место для хранения этих методов с последующим преобразованием в *объект теста* (Testcase Object, с. 410).

Как это работает

Все связанные тестовые методы собираются в специальный класс — *класс теста* (Testcase Class). Во время выполнения *класс теста* (Testcase Class) выступает в роли *фабрики наборов тестов* (Test Suite Factory; см. *Перечисление тестов*, Test Enumeration, с. 425), которая создает *объект теста* (Testcase Object) для каждого *тестового метода* (Test Method). Все эти объекты добавляются в *объект набора тестов* (Test Suite Object, с. 414), который будет использоваться *программой запуска тестов* (Test Runner, с. 405).

Зачем это нужно

В объектно-ориентированных языках *тестовые методы* (Test Method) размещаются в классах, а не создаются в виде глобальных функций или процедур (даже если это разрешено). Сделав их методами экземпляров *класса теста* (Testcase Class), можно получить *объект теста* (Testcase Object) для каждого теста, создав экземпляр *класса теста* (Testcase Class) для каждого *тестового метода* (Test Method).

Такая стратегия позволяет управлять *тестовыми методами* (Test Method) на этапе выполнения.

ДУАЛИЗМ КЛАССА И ЭКЗЕМПЛЯРА

В старших классах на уроках физики рассказывают о “дуализме” света. Иногда свет ведет себя, как частица (например, при проходе через апертурную решетку), а иногда — как волна (например, при появлении радуги). Поведение *класса теста* (Testcase Class, с. 401) напоминает такую же концепцию. Рассмотрим, почему.

Новички в тестировании с помощью xUnit часто спрашивают: “Почему класс, подкласс которого создается, называется *Testcase*, если он имеет несколько *тестовых методов* (Test Method)? Не должен ли он называться *TestSuite*?“ Такие вопросы имеют смысл, когда основное внимание уделяется классу при написании тестового кода (в отличие от процесса запуска кода).

При написании тестового кода основное внимание уделяется *тестовым методам* (Test Method). *Класс теста* (Testcase Class) просто является местом хранения этих методов. В терминах объектов мы думаем только при использовании *неявной настройки* (Implicit Setup, с. 449), когда необходимо создать поля (переменные экземпляра) для хранения объектов между вызовом метода *setUp* и их использованием внутри *тестовых методов* (Test Method). Обычно первые тесты с использованием инфраструктуры xUnit создаются на основе примеров. С помощью существующего примера можно быстро получить работающий результат, но это не всегда дает полное понимание происходящего.

В процессе выполнения инфраструктура xUnit обычно создает один экземпляр *класса теста* (Testcase Class) для каждого *тестового метода* (Test Method). *Класс теста* (Testcase Class) выступает в роли *фабрики наборов тестов* (Test Suite Factory; см. *Перечисление тестов*, Test Enumeration, с. 425), создающей *объект набора тестов* (Test Suite Object, с. 414) со всеми собственными экземплярами (по одному на каждый тестовый метод). Проблема в том, что не часто статический метод класса возвращает экземпляр другого класса, содержащий несколько собственных экземпляров. Если даже это поведение не кажется слишком странным, использование в инфраструктуре xUnit имени *тестового метода* (Test Method) для описания неудачных завершений может скрыть от большинства разработчиков существование “*объекта внутри*”.

Если рассмотреть отношения между объектами на этапе выполнения, все становится немного понятнее. *Объект набора тестов* (Test Suite Object), который возвращается *фабрикой наборов тестов* (Test Suite Factory), содержит один или несколько *объектов теста* (Testcase Object, с. 410). Каждый из этих объектов является экземпляром *класса теста* (Testcase Class). Каждый экземпляр настроен на запуск одного *тестового метода* (Test Method). Очень важно понять, что каждый из них запускает собственный тестовый метод. (Более подробно происходящее рассматривается при описании *обнаружения тестов* (Test Discovery, с. 420).) Иначе говоря, каждый экземпляр *класса теста* (Testcase Class) по сути является *отдельным тестом* (test case). *Тестовый метод* (Test Method) просто указывает каждому экземпляру, что он должен тестировать.

Источник дополнительной информации

Мартин Фаулер описал эту проблему в своем сетевом журнале, в статье “JUnit New Instance”.

Конечно, каждый *тестовый метод* (Test Method) можно реализовать в отдельном классе, но это приведет к дополнительным накладным расходам и засорению пространства имен классов. Кроме того, это значительно усложнит (если не сделает невозможным) повторное использование функциональности несколькими тестами.

Замечания по реализации

В основном сложность тестов связана с написанием *тестовых методов* (Test Method): что вписать в тестовый метод, а что выделить во *вспомогательный метод теста* (Test Utility Method, с. 610), как *изолировать тестируемую систему* (Isolate the SUT, с. 97) и т.д.

Настоящая “магия” вокруг *класса теста* (Testcase Class) происходит во время выполнения и описывается в *объекте теста* (Testcase Object) и *программе запуска тестов* (Test Runner). От разработчика требуется написать *тестовые методы* (Test Method), содержащие логику тестов, а *программа запуска тестов* (Test Runner) обеспечит все “магические” манипуляции. Можно избежать *дублирования тестового кода* (Test Code Duplication, с. 254), используя рефакторинг *выделение метода* (Extract Method) для создания *вспомогательных методов теста* (Test Utility Method). Эти методы можно оставить внутри *класса теста* (Testcase Class) или переместить в суперкласс *абстрактного теста* (Abstract Testcase; см. *Суперкласс теста*, Testcase Superclass, с. 646), во *вспомогательный класс теста* (Test Helper, с. 651) или во *вспомогательный “миксин”* (Test Helper Mixin; см. *Суперкласс теста* (Testcase Superclass).

Пример: класс теста (Testcase Class)

Ниже приведен пример простого *класса теста* (Testcase Class).

```
public class TestScheduleFlight extends TestCase {
    public void testUnscheduled_shouldEndUpInScheduled() throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInUnscheduledState();
        flight.schedule();
        assertTrue("isScheduled()", flight.isScheduled());
    }
    public void testScheduledState_shouldThrowInvalidRequestEx()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInScheduledState();
        try {
            flight.schedule();
            fail("not allowed in scheduled state");
        } catch (InvalidRequestException e) {
            assertEquals("InvalidRequestException.getRequest ()",
                "schedule",
                e.getRequest());
            assertTrue("isScheduled()", flight.isScheduled());
        }
    }
    public void testAwaitingApproval_shouldThrowInvalidRequestEx()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInAwaitingApprovalState();
        try {
            flight.schedule();
            fail("not allowed in schedule state");
        } catch (InvalidRequestException e) {
            assertEquals("InvalidRequestException.getRequest ()",
                "schedule",
                e.getRequest());
        }
    }
}
```

```
        e.getRequest());
    assertTrue("isAwaitingApproval()",  
               flight.isAwaitingApproval());
}
}
```

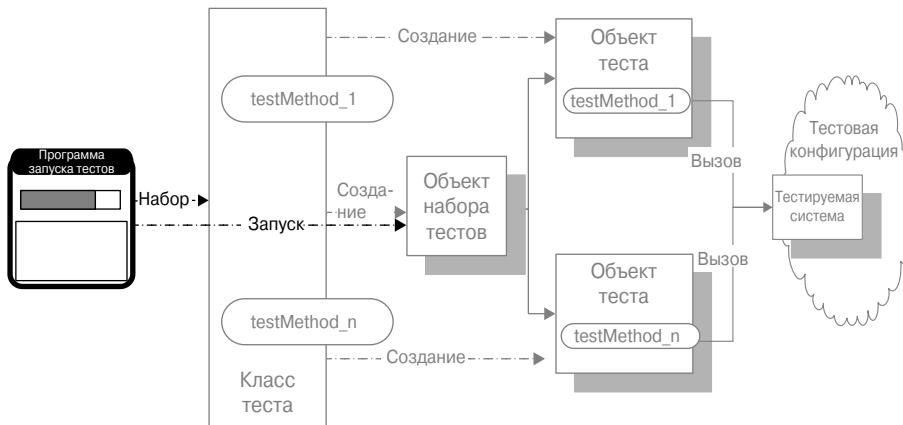
Дополнительная информация

В некоторых реализациях xUnit (в частности, в VbUnit и NUnit) *класс теста* (Testcase Class) называется тестовой конфигурацией. Не путайте его с тестовой конфигурацией (или контекстом теста), состоящим из всего, что необходимо для вызова тестируемой системы (предварительные условия теста). Ни тот, ни другой варианты не связаны с конфигурацией в понимании инфраструктуры Fit, где используется адаптер, взаимодействующий с таблицей Fit и реализующий интерпретатор *управляемого данными теста* (Data-Driven Test, с. 322).

Программа запуска тестов (Test Runner)

Как запустить написанные тесты?

Пишется приложение, которое создает *объект набора тестов* (Test Suite Object) и выполняет все содержащиеся в нем *объекты теста* (Testcase Object).



Предположим, что *тестовые методы* (Test Method, с. 378) уже определены и находятся в одном или нескольких *классах теста* (Testcase Class, с. 401). Как заставить *инфраструктуру автоматизации тестов* (Test Automation Framework, с. 332) запускать имеющиеся тесты?

Как это работает

Каждая реализация *инфраструктуры автоматизации тестов* (Test Automation Framework) семейства xUnit предоставляет приложение с текстовым или графическим интерфейсом, которое можно использовать для запуска автоматизированных тестов и создания отчета о результатах. *Программа запуска тестов* (Test Runner) использует *перечисление тестов* (Test Enumeration, с. 425), *обнаружение тестов* (Test Discovery, с. 420) или *выбор тестов* (Test Selection, с. 429) для создания тестового объекта Composite. Это может быть единственный *объект теста* (Testcase Object), *объект набора тестов* (Test Suite Object, с. 414) или набор тестов Composite (*набор наборов*, Suite of Suites). Так как все эти объекты реализуют один и тот же интерфейс, *программе запуска тестов* (Test Runner) не нужно знать, с каким типом набора она работает. *Программа запуска тестов* (Test Runner) следит за количеством запущенных тестов, за количеством ложных утверждений и за количеством ошибок или исключений. После этого полученная статистика выводится в отчете.

Зачем это нужно

Каждый разработчик тестов не должен разрабатывать собственный механизм запуска созданных тестов. Такая ситуация помешала бы одновременному запуску автоматизированных тестов несколькими разработчиками. Стандартная *программа запуска тестов*

(Test Runner) помогает разработчикам создавать тесты, которые смогут запускать другие люди. Кроме того, обеспечивается несколько способов запуска одних и тех же тестов.

Замечания по реализации

Существует несколько вариантов *программы запуска тестов* (Test Runner). Наиболее распространенные варианты запускают тесты из среды разработки или из интерпретатора командной строки. Все эти варианты основаны на реализации стандартного интерфейса всеми *объектами теста* (Testcase Object).

Стандартный интерфейс тестов

Статически типизированные языки (например, Java и C#) обычно позволяют определить тип интерфейса (полностью абстрактный класс), описывающий интерфейс, который должен быть реализован всеми *объектами теста* (Testcase Object) и *объектами набора тестов* (Test Suite Object). В некоторых языках (например, C# и Java 5.0) реализация “подмешивается” через использование атрибутов классов или аннотаций в *классе теста* (Testcase Class). В динамически типизированных языках такой интерфейс может явно не существовать. Вместо этого каждый класс просто реализует методы интерфейса. Обычно стандартный интерфейс теста включает в себя методы подсчета доступных тестов и запуска тестов. Если инфраструктура поддерживает *перечисление тестов* (Test Enumeration), каждый *класс теста* (Testcase Class) и класс набора тестов должен содержать реализацию метода *фабрики наборов тестов* (Test Suite Factory; см. *Перечисление тестов*, Test Enumeration, с. 425), который обычно называется suite.

Вариант: программа запуска тестов с графическим интерфейсом (Graphical Test Runner)

Программа запуска тестов с графическим интерфейсом (Graphical Test Runner) обычно является отдельным приложением или элементом среды разработки. Как минимум одна реализация (JUnit) поддерживает запуск из Web-обозревателя. Наиболее распространенной особенностью программы запуска тестов является индикатор текущего состояния. Обычно индикатор показывает количество запущенных тестов, количество неудачных завершений и количество ошибок. Кроме того, одним из элементов является полоса состояния, которая до запуска тестов имеет зеленый цвет и меняет цвет на красный при неудачном завершении или ошибке теста. В некоторых реализациях xUnit предоставляется *обозреватель набора тестов* (Test Tree Explorer), позволяющий выбрать и запустить единственный тест из всего набора.

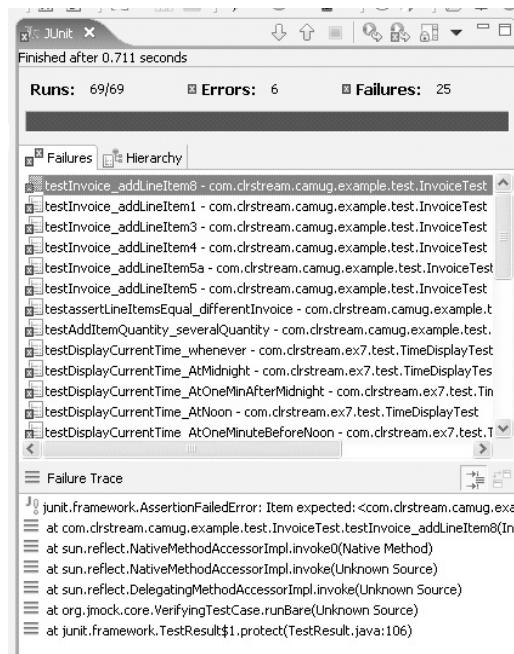
Ниже приведен пример *программы запуска тестов с графическим интерфейсом* (Graphical Test Runner) для модуля JUnit среды разработки Eclipse.

Красная полоса в верхней части окна показывает, что как минимум один тест завершился неудачно. На верхней панели перечислены неудачные завершения и ошибки тестов. На нижней панели показано состояние стека для неудачно завершившегося теста из верхней панели.

Вариант: программа запуска тестов для командной строки (Command-Line Test Runner)

Программа запуска тестов для командной строки (Command-Line Test Runner) проектируется для запуска из командного интерпретатора или из сценариев пакетной обработ-

ки. Программа с таким интерфейсом очень удобна для запуска на удаленном компьютере, а также из сценариев автоматической компиляции make, Ant или из систем **непрерывной интеграции** (continuous integration), например Cruise Control.



Ниже показан пример запуска `runit` (одна из реализаций xUnit для языка программирования Ruby).

```
>ruby testrunner.rb c:/examples/tests/SmellHandlerTest.rb
Loaded suite SmellHandlerTest
Started
....
Finished in 0.016 seconds.
5 tests, 6 assertions, 0 failures, 0 errors
>Exit code: 0
```

Первая строка содержит приглашение командного интерпретатора и ввод команды. В этом примере запускаются тесты, определенные в единственном *классе теста* (TestCase Class) — `SmellHandlerTest`. В следующих двух строках показан стандартный вывод в начале теста. Стока из точек показывает текущее состояние процесса, по одной точке на каждый завершенный тест. Данная конкретная *программа запуска тестов для командной строки* (Command-Line Test Runner) заменяет точку символом “E” или “F”, если происходит ошибка или тест завершается неудачно. В последних трех строках приводится итоговая статистика запуска. Обычно код завершения равен общему количеству ошибочных и неудачно завершившихся тестов. В результате код завершения позволяет быстро определить успешность запуска тестов с помощью инструментов автоматической компиляции.

Вариант: программа запуска тестов из файловой системы (File System Test Runner)

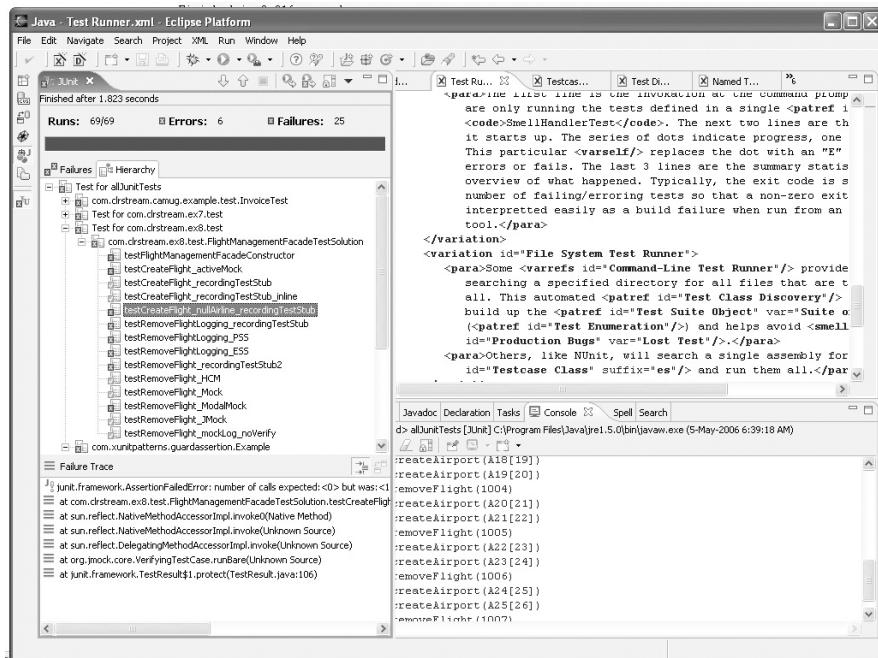
Некоторые программы запуска тестов для командной строки (Command-Line Test Runner) позволяют указать каталог, в котором расположены файлы с тестами. Программа просматривает все файлы и запускает все найденные тесты. Подобный автоматизированный механизм обнаружения классов теста (Testcase Class Discovery) позволяет избежать необходимости создавать набор наборов (Suite of Suites) и появления потерянных тестов (Lost Test; см. *Ошибки в продукте*, Production Bugs, с. 303).

Кроме того, существуют утилиты поиска файлов по заданному шаблону имени. Найденные файлы могут передаваться в программу запуска тестов (Test Runner) из утилиты компиляции.

Вариант: обозреватель набора тестов (Test Tree Explorer)

В реализациях xUnit, превращающих каждый *тестовый метод* (Test Method) в *объект теста* (Testcase Object), значительно упрощается управление тестами. Во многих реализациях доступно графическое представление *набора наборов* (Suite of Suites), и пользователь может выбрать для запуска целый *объект набора тестов* (Test Suite Object) или *объект теста* (Testcase Object). В результате отпадает необходимость в создании *набора из одного теста* (Single Test Suite; см. *Именованный набор тестов*, Named Test Suite, с. 604) для запуска единственного теста.

Ниже приведен пример *обозревателя набора тестов* (Test Tree Explorer) для модуля JUnit в среде разработки Eclipse.

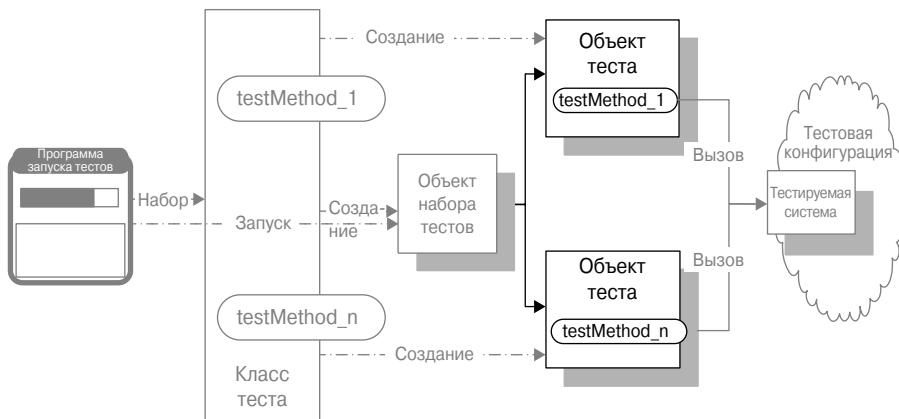


На левой панели среды разработки показан модуль JUnit. Стока состояния показана в верхней части панели. В верхней части панели показан *обозреватель набора тестов* (Test Tree Explorer), а в нижней части — состояние стека для текущего теста. Обратите внимание, что некоторые *объекты набора тестов* (Test Suite Object) в *обозревателе набора тестов* (Test Tree Explorer) “открыты” и демонстрируют свое содержимое, а некоторые закрыты. Разноцветный текст аннотаций возле каждого *объекта теста* (Testcase Object) показывает его состояние. Аннотации для каждого *объекта набора тестов* (Test Suite Object) показывают, были ли ошибки или неудачные завершения тестов в пределах набора. *Объект набора тестов* (Test Suite Object) с названием “Test for com.clrstream.ex8.test” является *набором наборов* (Suite of Suites) для пакета “com.clrstream.ex8.test”. Объект “Test for allJUnitTests” является набором наборов верхнего уровня для запуска всех тестов.

Объект теста (Testcase Object)

Как запустить созданные тесты?

Для каждого теста создается объект **Command**, и его метод `run` вызывается каждый раз, когда необходимо запустить тест.



Программе запуска тестов (Test Runner, с. 405) нужен механизм обнаружения и запуска соответствующих *тестовых методов* (Test Method, с. 378), а также вывода результатов для пользователя. Многие программы запуска тестов с графическим интерфейсом позволяют пользователю просматривать полный список тестов и выбирать отдельные тесты для запуска. Для обеспечения такой функциональности *программа запуска тестов* (Test Runner) должна иметь возможность просматривать список тестов во время выполнения и управлять им.

Как это работает

Объект Command создается для каждого запускаемого *тестового метода* (Test Method). При этом *класс теста* (Testcase Class, с. 401) используется в виде *фабрики наборов тестов* (Test Suite Factory) для создания *объекта набора тестов* (Test Suite Object, с. 414), содержащего все *объекты теста* (Testcase Object) для конкретного *класса теста* (Testcase Class). Для создания *объектов теста* (Testcase Object) можно использовать *обнаружение тестов* (Test Discovery, с. 420) или *перечисление тестов* (Test Enumeration).

Зачем это нужно

Обработка тестов как полноценных объектов обеспечивает много новых возможностей, недоступных при интерпретации тестов как обычных процедур. *Программе запуска тестов* (Test Runner) в составе *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332) проще управлять тестами в виде объектов. Их можно хранить в коллекциях (*объектах набора тестов*, Test Suite Object), перебирать итеративно, вызывать их методы и т.д.

В большинстве реализаций xUnit создается отдельный *объект теста* (Testcase Object) для каждого теста. Это позволяет изолировать тесты друг от друга в соответствии с принципом независимости тестов. К сожалению, всегда существуют исключения, и пользователи нестандартных инфраструктур автоматизации тестов (Test Automation Framework) должны внимательно следить за поведением инструментария.

Замечания по реализации

Каждый *объект теста* (Testcase Object) реализует стандартный интерфейс теста для защиты *программы запуска тестов* (Test Runner) от особенностей реализации. Такая схема позволяет каждому *объекту теста* (Testcase Object) выступать в роли объекта Command. В результате можно создавать коллекции *объектов теста* (Testcase Object), содержимое которых можно перебирать итеративно для подсчета, запуска вывода результатов и выполнения других операций.

В большинстве языков программирования для определения поведения *объектов теста* (Testcase Object) необходимо создать класс. Можно создать отдельный *класс теста* (Testcase Class) для каждого *объекта теста* (Testcase Object). Но намного удобнее хранить несколько *тестовых методов* (Test Method) в одном *классе теста* (Testcase Class), так как подобная стратегия позволяет сократить количество классов и стимулирует повторное использование *вспомогательных методов теста* (Test Utility Method, с. 610). Такой подход требует, чтобы каждый *объект теста* (Testcase Object) внутри *класса теста* (Testcase Class) имел возможность определить вызываемый *тестовый метод* (Test Method). Наиболее распространенным решением является *подключаемое поведение* (Pluggable Behavior). Конструктор *класса теста* (Testcase Class) принимает имя вызываемого метода в качестве параметра и хранит это имя в переменной экземпляра. Когда *программа запуска тестов* (Test Runner) вызывает метод *run* *объекта теста* (Testcase Object), с помощью механизма интроспекции вызывается метод, имя которого хранится в переменной экземпляра.

ВСЕГДА ЕСТЬ ИСКЛЮЧЕНИЯ

Как в правилах склонения глаголов в иностранных языках, так и в шаблонах создания программного обеспечения всегда бывают исключения!

Одно из наиболее заметных исключений в семействе xUnit относится к использованию *объекта теста* (Testcase Object, с. 410) для представления каждого *тестового метода* (Test Method, с. 378) на этапе выполнения. Это ключевой элемент xUnit, позволяющий обеспечить независимость тестов. Существуют две реализации xUnit, не поддерживающие такую схему: TestNG и NUnit (версии 2.x). По описанным ниже причинам создатели NUnit 2.0 решили отказаться от создания *объекта теста* (Testcase Object) для каждого тестового метода (Test Method) и создают единственный экземпляр *класса теста* (Testcase Class, с. 401). Они называют этот экземпляр **тестовой конфигурацией** (test fixture), повторно используемой каждым тестовым методом. Один из авторов NUnit 2.0, Джеймс Ньюкирк, пишет следующее.

Я думаю, что самой большой ошибкой при написании NUnit v2.0 был отказ от создания нового экземпляра класса тестовой конфигурации для каждого тестового метода. Готов признать, что это была именно моя ошибка, так как я не совсем понимал причины, по которым в JUnit для каждого метода создается новая конфигурация. Оглядываясь назад,

я понимаю, что использование одного экземпляра допускало сохранение данных в переменной экземпляра одним классом и использование этих данных другим тестом. В результате появлялись зависимости от порядка запуска, а это антишаблон для данного типа тестирования. Намного лучше изолировать тестовые методы друг от друга. Для этого новый объект должен создаваться для каждого тестового метода.

К сожалению, это привело к интересным (и нежелательным) последствиям для тех, кто знаком с поведением JUnit по созданию отдельного *объекта теста* (Testcase Object) для каждого метода. Так как объект используется повторно, любые объекты, ссылки на которые хранятся в переменных экземпляра, становятся доступными всем последующим тестам. В результате появляются неявная общая *тестовая конфигурация* (Shared Fixture, с. 350) и все варианты *нестабильных тестов* (Erratic Test, с. 267), с ней связанные. Джеймс продолжает свою мысль так.

Поскольку на данном этапе сложно изменить поведение NUnit, а жалоб достаточно много, все переменные классов тестовой конфигурации сделаны статическими. Существует только один экземпляр переменной, независимо от количества созданных объектов. Если переменная является статической, незнакомые с особенностями NUnit разработчики не будут предполагать, что новая переменная создается перед каждым запуском теста. Это максимальное приближение к особенностям работы JUnit без модификации механизма вызова методов, принятого в NUnit.

Мартин Фаулер считал это исключение настолько важным, что даже написал статью о правильности подхода JUnit, доступную по адресу <http://martinfowler.com/bliki/JunitNewInstance.html>.

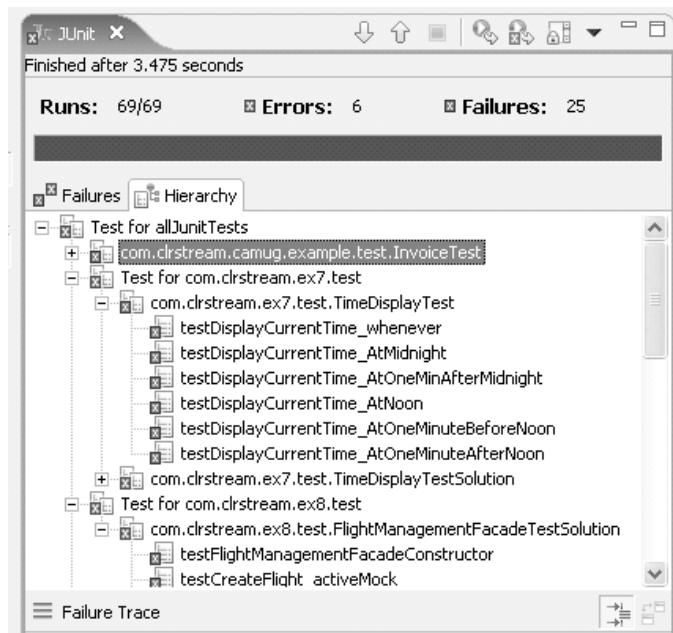
Пример: объект теста (Testcase Object)

Основное доказательство существования *объекта теста* (Testcase Object) можно увидеть в *обозревателе набора тестов* (Test Tree Explorer) при “разворачивании” *объекта набора тестов* (Test Suite Object) для просмотра содержащихся в нем *объектов теста* (Testcase Object). Рассмотрим пример программы запуска тестов JUnit для среды разработки Eclipse. Ниже приведен список объектов, составленный на основе примера кода из *класса теста* (Testcase Class).

```
TestSuite("...flightstate.featuretests.AllTests")
    TestSuite("...flightstate.featuretests.TestApproveFlight")
        TestApproveFlight("testScheduledState_shouldThrowIn..ReEx")
        TestApproveFlight("testUnscheduled_shouldEndUpInAwai..oval")
        TestApproveFlight("testAwaitingApproval_shouldThrow..stEx")
        TestApproveFlight("testWithNullArgument_shouldThrow..ntEx")
        TestApproveFlight("testWithInvalidApprover_shouldTh..ntEx")
    TestSuite("...flightstate.featuretests.TestDescheduleFlight")
        TestDescheduleFlight("testScheduled_shouldEndUpInSc..tate")
        TestDescheduleFlight("testUnscheduled_shouldThrowIn..stEx")
        TestDescheduleFlight("testAwaitingApproval_shouldTh..stEx")
    TestSuite("...flightstate.featuretests.TestRequestApproval")
        TestRequestApproval("testScheduledState_shouldThrow..stEx")
        TestRequestApproval("testUnscheduledState_shouldEndU..oval")
        TestRequestApproval("testAwaitingApprovalState_shou..stEx")
    TestSuite("...flightstate.featuretests.TestScheduleFlight")
        TestScheduleFlight("testUnscheduled_shouldEndUpInSc..uled")
        TestScheduleFlight("testScheduledState_shouldThrowI..stEx")
        TestScheduleFlight("testAwaitingApproval_shouldThro..stEx")
```

Перед скобками указано имя класса. Стока внутри скобок — имя объекта, экземпляра этого класса. По соглашению имя запускаемого *тестового метода* (Test Method) (часть имени заменена на “..” в соответствии с ограничениями формата книжной страницы) используется в качестве имени *объекта теста* (Testcase Object) и в качестве имени *объекта набора тестов* (Test Suite Object) используется любая строка, переданная как параметр конструктору *объекта набора тестов* (Test Suite Object). В этом примере в качестве такой строки использовалось полное имя пакета и класса от *класса теста* (Testcase Class).

Вот так будет выглядеть подобная схема при просмотре в *обозревателе набора тестов* (Test Tree Explorer).



Объект набора тестов (Test Suite Object)

Как запускать несколько тестов одновременно?

Определяется класс коллекции, реализующий стандартный интерфейс теста. Этот класс используется для запуска набора объектов теста (Testcase Object).



Когда *тестовые методы* (Test Method, с. 378), содержащие логику тестов, уже созданы и размещены внутри *класса теста* (Testcase Class, с. 401) для создания *объекта теста* (Testcase Object) для каждого теста, нужен механизм запуска всех тестов одной командой.

Как это работает

Определяется *объект теста* (Testcase Object) (реализующий шаблон Composite), который называется *объектом набора тестов* (Test Suite Object) и содержит коллекцию отдельных *объектов теста* (Testcase Object). Для одновременного запуска всех тестов набора *программа запуска тестов* (Test Runner, с. 405) отправляет соответствующий запрос *объекту набора тестов* (Test Suite Object).

Зачем это нужно

Обработка наборов тестов как полноценных объектов значительно упрощает манипулирование тестами средствами *программы запуска тестов* (Test Runner) и *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332). С *объектом набора тестов* (Test Suite Object) или без него *программа запуска тестов* (Test Runner) должна хранить коллекцию *объектов теста* (Testcase Object) для их перебора, подсчета, запуска и т.д. Если коллекция становится “умной”, очень легко добавить новые конструкции, например *набор наборов* (Suite of Suites).

Вариант: набор классов тестов (Testcase Class Suite)

Для запуска всех *тестовых методов* (Test Method) в одном *классе теста* (Testcase Class) достаточно создать *объект набора тестов* (Test Suite Object) для класса теста

(Testcase Class) и добавить один *объект теста* (Testcase Object) для каждого *тестового метода* (Test Method). Это позволит запустить все *тестовые методы* (Test Method) в *классе теста* (Testcase Class), передав *программе запуска тестов* (Test Runner) имя класса теста.

Вариант: набор наборов (Suite of Suites)

Собирая древовидные структуры из небольших наборов тестов, можно создавать большие *именованные наборы тестов* (Named Test Suite, с. 604). Шаблон Composite делает такую организацию тестов прозрачной для *программы запуска тестов* (Test Runner), позволяя рассматривать *набор наборов* (Suite of Suites) точно так, как простой *набор классов тестов* (Testcase Class Suite) или единственный *объект теста* (Testcase Object).

Замечания по реализации

Как объект Composite каждый *объект набора тестов* (Test Suite Object) реализует такой же интерфейс, как и простой *объект теста* (Testcase Object). Таким образом, и *программа запуска тестов* (Test Runner), и *объект набора тестов* (Test Suite Object) могут не знать, что они работают со ссылкой на целый набор, а не на отдельный тест. В результате значительно упрощается реализация всех операций, подразумевающих итерацию по набору тестов с последующим подсчетом, запуском или выводом результатов.

Объект набора тестов (Test Suite Object) необходимо создать. Можно воспользоваться одним из доступных механизмов.

- *Обнаружение тестов* (Test Discovery, с. 420). Позволяет *инфраструктуре автоматизации тестов* (Test Automation Framework) самостоятельно обнаруживать *классы теста* (Testcase Class) и *тестовые методы* (Test Method).
- *Перечисление тестов* (Test Enumeration, с. 425). Позволяет написать код, который будет перечислять *тестовые методы* (Test Method) для включения в *объект набора тестов* (Test Suite Object). Обычно при этом создается *фабрика наборов тестов* (Test Suite Factory).
- *Выбор тестов* (Test Selection, с. 429). Позволяет указать подмножество *объектов теста* (Testcase Object), которые должны быть включены в существующий *объект набора тестов* (Test Suite Object).

Вариант: процедура набора тестов (Test Suite Procedure)

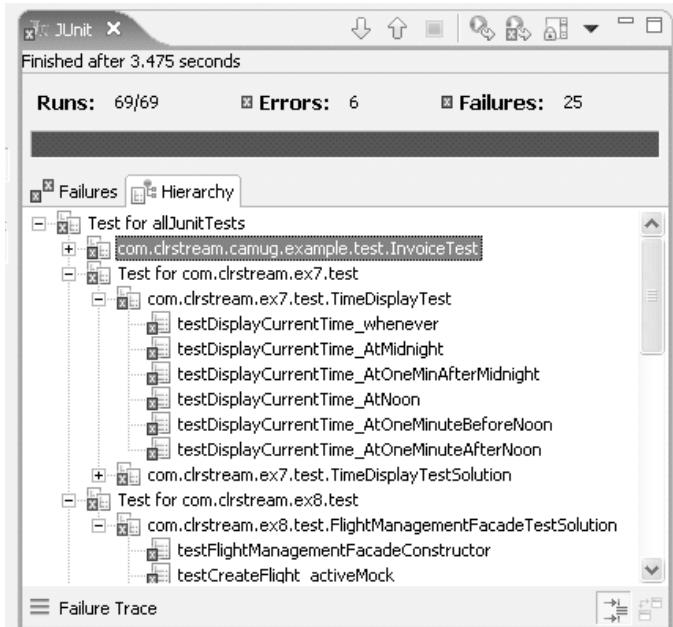
Иногда код приходится писать на языках программирования без поддержки объектов. При этом *программа запуска тестов* (Test Runner) должна получить список уже *написанных тестовых методов* (Test Method). *Процедура набора тестов* (Test Suite Procedure) позволяет перечислять все запускаемые тесты, перебирая их по очереди. Вызов каждого теста жестко записан в тело *объекта набора тестов* (Test Suite Object). Конечно, *процедура набора тестов* (Test Suite Procedure) может вызывать другие *процедуры набора тестов* (Test Suite Procedure) для получения *набора наборов* (Suite of Suites).

Основным недостатком такого подхода является принудительное *перечисление тестов* (Test Enumeration), что повышает затраты на написание тестов и вероятность появления *потерянных тестов* (Lost Test). Поскольку код не может рассматриваться как “данные”, теряется возможность манипуляции кодом на этапе выполнения. В результате сложнее

создать программу запуска тестов с графическим интерфейсом и иерархическим представлением набора наборов (Suite of Suites).

Пример: объект набора тестов (Test Suite Object)

В большинстве реализаций xUnit используется *обнаружение тестов* (Test Discovery), поэтому примеров *объектов набора тестов* (Test Suite Object) довольно мало. Существование такого объекта можно наблюдать в *обозревателе набора тестов* (Test Tree Explorer) при просмотре списка *объектов теста* (Testcase Object). Ниже приведен пример для пакета JUnit, поддержка которого реализована в среде разработки Eclipse.



Пример: создание набора наборов (Suite of Suites) с помощью перечисления тестов (Test Enumeration)

Ниже представлен пример использования *перечисления тестов* (Test Enumeration) для создания набора наборов (Suite of Suites).

```
public class AllTests {
    public static Test suite() {
        TestSuite suite = new TestSuite("Test for allJUnitTests");
        suite.addTestSuite(
            com.clrstream.camug.example.test.InvoiceTest.class);
        suite.addTest(com.clrstream.ex7.test.AllTests.suite());
        suite.addTest(com.clrstream.ex8.test.AllTests.suite());
        suite.addTestSuite(com.xunitpatterns.guardassertion.Example.class);
        return suite;
    }
}
```

Первые и последние строки добавляют *объекты набора тестов* (Test Suite Object), созданные на основе одного *класса теста* (Testcase Class). Каждая из средних строк вызывает *фабрику наборов тестов* (Test Suite Factory) для еще одного *набора наборов* (Suite of Suites). Возвращаемый *объект набора тестов* (Test Suite Object) имеет как минимум тройную вложенность.

1. Созданный и наполненный перед возвращением *объект набора тестов* (Test Suite Object).
2. *Объект набора тестов* (Test Suite Object) AllTests, который возвращается двумя вызовами методов фабрики.
3. *Объект набора тестов* (Test Suite Object) для каждого *класса теста* (Testcase Class), собранного в *объекты набора тестов* (Test Suite Object).

Пример описанной конструкции показан ниже.

```
TestSuite("Test for allJunitTests");
TestSuite("com.clrstream.camug.example.test.InvoiceTest")
    TestCase("testInvoice_addLineItem")
    ...
    TestCase("testRemoveLineItemsForProduct_oneOfTwo")
TestSuite("com.clrstream.ex7.test.AllTests")
    TestSuite("com.clrstream.ex7.test.TimeDisplayTest")
        TestCase("testDisplayCurrentTime_AtMidnight")
        TestCase("testDisplayCurrentTime_AtOneMinAfterMidnight")
        TestCase("testDisplayCurrentTime_AtOneMinuteBeforeNoon")
        TestCase("testDisplayCurrentTime_AtNoon")
        ...
    TestSuite("com.clrstream.ex7.test.TimeDisplaySolutionTest")
        TestCase("testDisplayCurrentTime_AtMidnight")
        TestCase("testDisplayCurrentTime_AtOneMinAfterMidnight")
        TestCase("testDisplayCurrentTime_AtOneMinuteBeforeNoon")
        TestCase("testDisplayCurrentTime_AtNoon")
        ...
TestSuite("com.clrstream.ex8.test.AllTests")
    TestSuite("com.clrstream.ex8.FlightMgmtFacadeTest")
        TestCase("testAddFlight")
        TestCase("testAddFlightLogging")
        TestCase("testRemoveFlight")
        TestCase("testRemoveFlightLogging")
        ...
TestSuite("com.xunitpatterns.guardassertion.Example")
    TestCase("testWithConditionals")
    TestCase("testWithoutConditionals")
    ...

```

Обратите внимание, что класс не является подклассом. Класс TestSuite и другие необходимые классы импортируются для использования в качестве *фабрик наборов тестов* (Test Suite Factory).

Пример: процедура набора тестов (Test Suite Procedure)

На ранних этапах гибкой разработки программного обеспечения, еще до появления средств управления такими процессами, автор создал набор электронных таблиц Excel

для управления задачами и пользовательскими историями. Часто выполняемые задачи (например, сортировка всех историй по номеру версии и итерации, сортировка задач по итерации и состоянию и т.д.) были автоматизированы. Со временем был написан целый макрос (программа), который вычислял оценки и фактические затраты для всех задач каждой истории. На этом этапе код стал сложнее и требовал больших усилий по сопровождению. В частности, после удаления одного из используемых макросом сортировки именованных диапазонов макрос выводил ошибку.

К сожалению, на тот момент не существовало реализации инфраструктуры xUnit для языка VBA, поэтому все обслуживание выполнялось без *тестов как страховки* (Tests as Safety Net, с. 79). Ниже приведен текст основного макроса. Весь вывод записывается на новый лист книги.

```
'Main Macro
Sub summarizeActivities()
    Call VerifyVersionCompatibility
    Call initialize
    Call SortByActivity
    For row = firstTaskDataRow To lastTaskDataRow
        If numberOfNumberlessTasks < MaxNumberlessTasks Then
            thisActivity =
                ActiveSheet.Cells(row, TaskActivityColumn).Value
            If thisActivity <> currentActivity Then
                Call finalizeCurrentActivityTotals
                currentActivity = thisActivity
                Call initializeCurrentActivityTotals
            End If
            Call accumulateActivityTotals(row)
        Else
            lastTaskDataRow = row      ' end the For loop right away
        End If
    Next row
    Call cleanUp
End Sub
```

Не имея тестов или *инфраструктуры автоматизации тестов* (Test Automation Framework) пришлось самостоятельно придумывать способ ввода регрессионных тестов. В данном случае достаточным условием был успешный запуск всех макросов. Успешное завершение работы макросов было лучшим индикатором отсутствия ошибок, чем отказ от запуска макросов вообще. Поскольку VBA основан на языке Visual Basic 5, классы в языке не поддерживались. Таким образом, не было никаких *классов теста* (Testcase Class) и *объектов теста* (Testcase Object) на этапе выполнения. Ниже приведен пример различных *процедур набора тестов* (Test Suite Procedure) и *тестовых методов* (Test Method).

```
Sub TestAll()
    Call TestAllStoryMacros
    Call TestAllTaskMacros
    Call TestReportingMacros
    Call TestToolbarMenus 'All The Same
End Sub
Sub TestAllStoryMacros()
    Call TestActivitySorting
    Call TestStoryHiding
    Call ReportSuccess("All Story Macros")
```

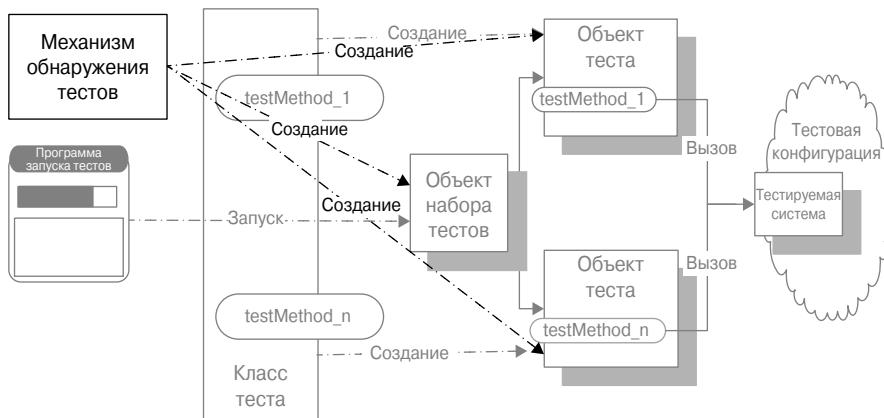
```
End Sub
Sub TestActivitySorting()
    Call SortStoriesbyAreaAndNumber
    Call SortActivitiesByIteration
    Call SortActivitiesByIterationAndOrder
    Call SortActivitiesByNumber
    Call SortActivitiesByPercentDone
End Sub
Sub TestReportingMacros()
    Call summarizeActivities
End Sub
```

Первая *процедура набора тестов* (Test Suite Procedure) является *набором наборов* (Suite of Suites). Вторая *процедура набора тестов* (Test Suite Procedure) является эквивалентом одного *объекта набора тестов* (Test Suite Object). Третья подпрограмма является *тестовым методом* (Test Method) для запуска всех макросов сортировки. В последней подпрограмме вызывается макрос `summarizeActivities` с использованием *предварительно созданной тестовой конфигурации* (Prebuilt Fixture, с. 454). (Внимательный читатель мог заметить, что в teste отсутствует фаза проверки результата. На самом деле это не *самопроверяющийся тест* (Self-Checking Test) и не *тест одного условия* (Single Condition Test).)

Обнаружение тестов (Test Discovery)

Откуда программа запуска тестов (Test Runner) знает, какие тесты запускать?

Инфраструктура автоматизации тестов (Test Automation Framework)
автоматически обнаруживает тесты, принадлежащие набору.



После написания множества *тестовых методов* (Test Method, с. 378) в одном или нескольких *классах теста* (Testcase Class, с. 401) *программе запуска тестов* (Test Runner, с. 405) необходимо обеспечить возможность найти эти тесты. Механизм *обнаружения тестов* (Test Discovery) решает большинство проблем, связанных с *перечислением тестов* (Test Enumeration, с. 425).

Как это работает

Инфраструктура автоматизации тестов (Test Automation Framework, с. 332) использует механизм интроспекции на этапе выполнения (или информацию, полученную на этапе компиляции) для обнаружения всех *тестовых методов* (Test Method), принадлежащих набору тестов и/или всем *объектам набора тестов* (Test Suite Object, с. 414), входящим в *набор наборов* (Suite of Suites). После этого создаются *объекты набора тестов* (Test Suite Object), содержащие соответствующие *объекты теста* (Testcase Object, с. 410) и другие *объекты набора тестов* (Test Suite Object).

Когда это использовать

Механизм *обнаружения тестов* (Test Discovery) должен использоваться во всех случаях, когда существует его поддержка в *инфраструктуре автоматизации тестов* (Test Automation Framework). Данный шаблон сокращает усилия, необходимые для автоматизации тестов, и значительно снижает вероятность появления *потерянных тестов* (Lost Test). Механизм *перечисления тестов* (Test Enumeration) должен использоваться только в двух случаях.

1. Когда *инфраструктура автоматизации тестов* (Test Automation Framework) не поддерживает механизм *обнаружения тестов* (Test Discovery).
2. Когда необходимо определить *именованный набор тестов* (Named Test Suite, с. 604), состоящий из подмножества тестов (хорошим примером может служить **контрольный набор тестов**, Smoke Test) из других наборов, и *инфраструктура автоматизации тестов* (Test Automation Framework) не поддерживает *выбор тестов* (Test Selection, с. 429).

Достаточно часто *перечисление наборов тестов* (Test Suite Enumeration) используется совместно с *обнаружением тестовых методов* (Test Method Discovery). Обратная комбинация встречается намного реже.

Замечания по реализации

Набор наборов (Suite of Suites) для запуска с помощью *программы запуска тестов* (Test Runner) создается в два этапа. Во-первых, необходимо найти все *тестовые методы* (Test Method) для включения в каждый *объект набора тестов* (Test Suite Object). Во-вторых, необходимо найти все *объекты набора тестов* (Test Suite Object), которые должны быть запущены. Последовательность этапов может меняться. Каждый из них можно выполнить вручную с помощью *перечисления тестовых методов* (Test Method Enumeration) и *перечисления наборов тестов* (Test Suite Enumeration) или автоматически с помощью *обнаружения тестовых методов* (Test Method Discovery) и *обнаружения классов теста* (Testcase Class Discovery).

Вариант: обнаружение классов теста (Testcase Class Discovery)

Обнаружение классов теста (Testcase Class Discovery) представляет собой процесс *обнаружения классов теста* (Testcase Class) средствами *инфраструктуры автоматизации тестов* (Test Automation Framework). Одно из решений предполагает пометку каждого *класса теста* (Testcase Class) через наследование *суперкласса теста* (Testcase Superclass, с. 646) или через реализацию Marker Interface. Еще один вариант используется в языках .NET и в более новых версиях JUnit. Он основан на применении атрибутов класса (например, [Test Fixture]) или аннотации (например, @Testcase) для идентификации каждого *класса теста* (Testcase Class). Следующее решение — размещение всех *классов теста* (Testcase Class) в одном каталоге и передача имени каталога *программе запуска тестов* (Test Runner) в качестве параметра. Последнее решение заключается в следовании соглашению об именовании *классов теста* (Testcase Class) и использовании внешней программы для поиска всех файлов, соответствующих этому соглашению. В любом случае после обнаружения *класса теста* (Testcase Class) можно переходить к *обнаружению тестовых методов* (Test Method Discovery) или *перечислению тестовых методов* (Test Method Enumeration).

Вариант: обнаружение тестовых методов (Test Method Discovery)

Механизм *обнаружения тестовых методов* (Test Method Discovery) предполагает предоставление *инфраструктуре автоматизации тестов* (Test Automation Framework) способа обнаружения *тестовых методов* (Test Method), находящихся внутри *классов теста* (Testcase Class). Существует два основных способа обозначить метод *класса теста* (Testcase Class) как *тестовый метод* (Test Method). Более традиционный подход предполагает ис-

пользование соглашения об именовании *тестовых методов* (Test Method) (например, имена методов могут начинаться на “test”). После этого *инфраструктура автоматизации тестов* (Test Automation Framework) перебирает все методы *класса теста* (Testcase Class), выбирает методы, начинающиеся на “test” (например, `testCounters`), и вызывает конструктор с одним аргументом для создания *объекта теста* (Testcase Object) для каждого *тестового метода* (Test Method). Альтернативный подход, используемый в языках .NET и новых версиях JUnit, предполагает использование атрибутов методов (например, `[Test]`) или аннотаций (например, `@Test`) для идентификации каждого *тестового метода* (Test Method).

Мотивирующий пример

Ниже приведен пример кода, который потребуется для *перечисления тестовых методов* (Test Method Enumeration) в отсутствие механизма *обнаружения тестов* (Test Discovery).

```
public:
    static CppUnit::Test *suite()
    {
        CppUnit::TestSuite *suite =
            new CppUnit::TestSuite("ComplexNumberTest");
        suite->addTest(
            new CppUnit::TestCaller<ComplexNumberTest>(
                "testEquality",
                &ComplexNumberTest::testEquality));
        suite->addTest(
            new CppUnit::TestCaller<ComplexNumberTest>(
                "testAddition",
                &ComplexNumberTest::testAddition));
        return suite;
    }
```

Этот пример взят из учебника по более ранней версии пакета CppUnit. В новых версиях данный подход больше не требуется.

Замечания по рефакторингу

К счастью для пользователей существующих реализаций пакета xUnit, создатели этой инфраструктуры осознали важность механизма *обнаружения тестов* (Test Discovery). Таким образом, достаточно последовать их советам по идентификации тестовых методов. Если разработчики используемой версии xUnit остановились на соглашении об именовании, для обеспечения возможности обнаруживать тесты может потребоваться рефакторинг *переименование метода* (Rename Method). Если для идентификации методов применяются атрибуты, достаточно добавить соответствующий атрибут ко всем *тестовым методам* (Test Method).

Пример: обнаружение тестовых методов (Test Method Discovery) на основе соглашения об именовании методов и макроса компилятора

Если язык программирования допускает управление тестами как объектами и вызов методов, но не поддерживает обнаружение всех тестовых методов, его можно простилизировать для решения возникшей проблемы. В новых версиях CppUnit предоставляется макрос, обнаруживающий все *тестовые методы* (Test Method) на этапе компиляции

и генерирующий код для создания набора тестов, показанный в предыдущем примере. Следующий фрагмент кода запускает механизм *обнаружения тестовых методов* (Test Method Discovery).

```
CPPUNIT_TEST_SUITE_REGISTRATION( FlightManagementFacadeTest );
```

Данный макрос основан на соглашении об именовании методов. Все методы с подходящими именами превращаются в *объект теста* (Testcase Object). Для этого методы заключаются в объект TestCaller, как и в показанном выше примере.

Пример: обнаружение тестовых методов (Test Method Discovery) на основе соглашения об именовании методов

Ценность следующих примеров — в отсутствующем, а не в показанном коде. Обратите внимание, что код добавления *тестовых методов* (Test Method) в *объект набора тестов* (Test Suite Object) отсутствует.

В этом примере на языке Java инфраструктура автоматически запускает все не имеющие аргументов тестовые методы с именами, начинающимися на “test”.

```
public class TimeDisplayTest extends TestCase {
    public void testDisplayCurrentTime_AtMidnight()
        throws Exception {
        // Настройка тестируемой системы
        TimeDisplay theTimeDisplay = new TimeDisplay();
        // Вызов тестируемой системы
        String actualTimeString =
            theTimeDisplay.getCurrentTimeAsHtmlFragment();
        // Проверка результата
        String expectedTimeString =
            "<span class=\"tinyBoldText\">Midnight</span>";
        assertEquals("Midnight",
            expectedTimeString,
            actualTimeString);
    }
    public void testDisplayCurrentTime_AtOneMinuteAfterMidnight()
        throws Exception {
        // Настройка тестируемой системы
        TimeDisplay actualTimeDisplay = new TimeDisplay();
        // Вызов тестируемой системы
        String actualTimeString =
            actualTimeDisplay.getCurrentTimeAsHtmlFragment();
        // Проверка результата
        String expectedTimeString =
            "<span class=\"tinyBoldText\">12:01 AM</span>";
        assertEquals("12:01 AM",
            expectedTimeString,
            actualTimeString);
    }
}
```

Пример: обнаружение тестовых методов (Test Method Discovery) с помощью атрибутов методов

В данном примере на языке C# тесты помечены атрибутом метода [Test]. Этот способ идентификации *тестовых методов* (Test Method) используется как в CsUnit, так и в NUnit.

```
[Test]
public void testFlightMileage_asKm()
{
    // Настройка тестовой конфигурации
    Flight newFlight = new Flight(validFlightNumber);
    newFlight.setMileage(1122);
    // Вызов преобразователя пробега
    int actualKilometres = newFlight.getMileageAsKm();
    int expectedKilometres = 1810;
    // Проверка результатов
    Assert.AreEqual(expectedKilometres, actualKilometres);
}
[Test]
[ExpectedException(typeof(InvalidArgumentException))]
public void testSetMileage_invalidInput_attribute()
{
    // Настройка тестовой конфигурации
    Flight newFlight = new Flight(validFlightNumber);
    // Вызов тестируемой системы
    newFlight.setMileage(-1122);
}
```

Пример: обнаружение классов теста (Testcase Class Discovery) с помощью атрибутов классов

В данном примере атрибут класса используется для идентификации *класса теста* (Testcase Class) (в NUnit он называется тестовой конфигурацией).

```
[TestFixture]
public class SampleTestcase
{
}
```

Пример: обнаружение классов теста (Testcase Class Discovery) с помощью общего расположения и суперкласса теста (Testcase Superclass)

В следующем примере на языке Ruby выполняется поиск всех файлов с расширением .rb в каталоге tests с последующим подключением найденных файлов с помощью команды require. В результате инфраструктура Test::Unit будет запускать тесты в каждом из файлов, так как содержащиеся в них *классы теста* (Testcase Class) расширяют класс Test::Unit::TestCase.

```
Dir['tests/*.rb'].each do |each|
    require each
end
```

Команда Dir['tests/*.rb'] возвращает коллекцию файлов. Коллекция перебирается методом each, который над каждой записью выполняет команду require each, обеспечивая *обнаружение классов теста* (Testcase Class Discovery). Интерпретатор языка Ruby и инфраструктура Test::Unit завершают процесс, выполняя *обнаружение тестовых методов* (Test Method Discovery) внутри каждого класса.

Перечисление тестов (Test Enumeration)

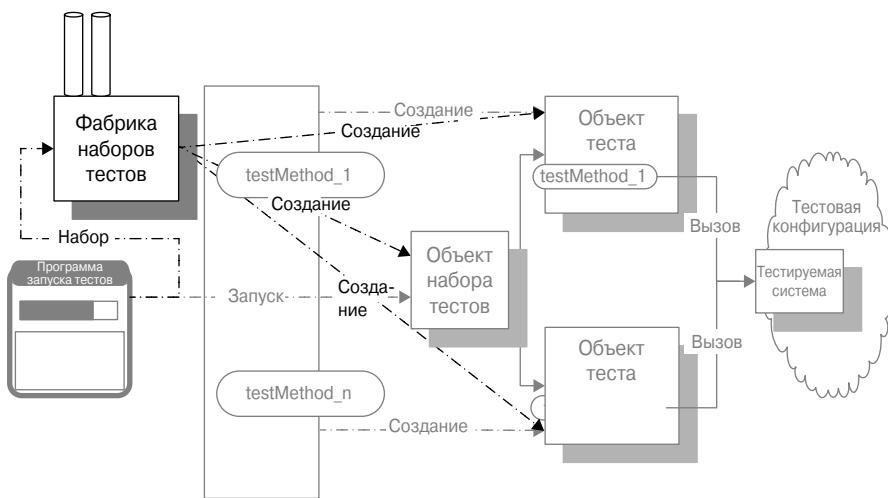
Откуда программа запуска тестов (Test Runner)

знает, какие тесты запускать?

Разработчик тестов вручную пишет код,
перечисляющий все тесты набора.

Также известен как:

*Фабрика наборов тестов
(Test Suite Factory)*



При наличии *тестовых методов* (Test Method, с. 378) в одном или нескольких *классах теста* (Testcase Class, с. 401) программа запуска тестов (Test Runner, с. 405) должна найти эти тесты. *Перечисление тестов* (Test Enumeration) может использоваться при недоступности механизма обнаружения тестов (Test Discovery, с. 420).

Как это работает

Автор тестов вручную создает код перечисления всех *тестовых методов* (Test Method) и их добавления в один или несколько *объектов набора тестов* (Test Suite Object, с. 414), принадлежащих *набору наборов* (Suite of Suites). Обычно это достигается через реализацию метода `suite` в классе *теста* (Testcase Class) для *перечисления тестовых методов* (Test Method Enumeration) или в *фабрике наборов тестов* (Test Suite Factory) для *перечисления наборов тестов* (Test Suite Enumeration).

Когда это использовать

Перечисление тестов (Test Enumeration) используется в тех случаях, когда *инфраструктура автоматизации тестов* (Test Automation Framework) не поддерживает механизм обнаружения тестов (Test Discovery). Кроме того, если инфраструктура не поддерживает механизм выбора тестов (Test Selection), перечисление тестов может использоваться для определения *именованного набора тестов* (Named Test Suite, с. 604), состоящего из подмножества тестов из других наборов.

Во многих реализациях xUnit поддерживается *обнаружение тестов* (Test Discovery) на уровне *тестовых методов* (Test Method), но на уровне *классов теста* (Testcase Class) приходится использовать *перечисление тестов* (Test Enumeration).

Замечания по реализации

Набор наборов (Suite of Suites) для выполнения *программой запуска тестов* (Test Runner) создается в два этапа. Во-первых, необходимо найти все *тестовые методы* (Test Method) для включения в каждый *объект набора тестов* (Test Suite Object). Во-вторых, необходимо найти все *объекты набора тестов* (Test Suite Object), которые должны быть запущены. Последовательность этапов может меняться. Каждый из них можно выполнить вручную с помощью *перечисления тестовых методов* (Test Method Enumeration) и *перечисления наборов тестов* (Test Suite Enumeration) или автоматически с помощью *обнаружения тестовых методов* (Test Method Discovery) и *обнаружения классов теста* (Testcase Class Discovery).

Вариант: перечисление наборов тестов (Test Suite Enumeration)

Во многих реализациях xUnit разработчик должен предоставить *фабрику наборов тестов* (Test Suite Factory), которая возвращает *набор наборов* (Suite of Suites) верхнего уровня (часто называемый AllSuites), что позволяет указать, какие *объекты набора тестов* (Test Suite Object) следует запускать в конкретный момент. Для этого создается метод класса фабрики. В большинстве реализаций этот *метод фабрики* (Factory Method) называется suite. Он использует вызовы метода addTest для добавления вложенных *объектов набора тестов* (Test Suite Object) в создаваемый набор.

Хотя подобный подход обеспечивает достаточную гибкость, он может стать причиной *потери тестов* (Lost Test). Альтернативным решением является использование инструментов разработки для автоматического создания *набора всех тестов* (AllTests Suite) или *программы запуска тестов* (Test Runner) для поиска всех наборов тестов в указанном каталоге файловой системы. Например, пакет NUnit обеспечивает встроенный механизм *обнаружения классов теста* (Testcase Class Discovery). Кроме того, для поиска всех *классов теста* (Testcase Class) в структуре каталогов можно использовать сторонние утилиты (например, Ant).

Даже в статически типизированных языках, например в Java, *фабрика наборов тестов* (Test Suite Factory) не должна быть подклассом конкретного класса или реализовывать конкретный интерфейс. Вместо этого зависимости связаны с возвращаемым *объектом набора тестов* (Test Suite Object) и с *классами теста* (Testcase Class) или *фабриками наборов тестов* (Test Suite Factory), у которых запрашиваются вложенные наборы.

Вариант: перечисление тестовых методов (Test Method Enumeration)

Во многих реализациях xUnit поддерживается *обнаружение тестовых методов* (Test Method Discovery). Если используется версия без поддержки такого механизма, придется найти все *тестовые методы* (Test Method) в *классе теста* (Testcase Class), превратить их в *объекты теста* (Testcase Object, с. 410) и добавить их в *объекты набора тестов* (Test Suite Object). *Перечисление тестовых методов* (Test Method Enumeration) реализуется через метод класса suite, принадлежащий *классу теста* (Testcase Class).

Способность создавать объект, вызывающий произвольный метод, обычно наследуется от *инфраструктуры автоматизации тестов* (Test Automation Framework) через *суперкласс теста* (Testcase Superclass, с. 646) или подключается с помощью атрибута класса или директивы `Include`. В некоторых реализациях xUnit возможность *подключения поведения* (Pluggable Behavior) обеспечивается отдельным классом (см. приведенный ниже пример для CppUnit).

Вариант: непосредственный вызов метода теста (Direct Test Method Invocation)

В чистом процедурном мире *тестовый метод* (Test Method) нельзя рассматривать как объект или элемент данных. В таком случае ничего не остается, как вручную создать *процедуру набора тестов* (Test Suite Procedure) для каждого набора. Такая процедура вызывает каждый *тестовый метод* (Test Method) (или другие *процедуры набора тестов*, Test Suite Procedure).

Пример: перечисление тестовых методов (Test Method Enumeration) в пакете CppUnit

В ранних версиях всех реализаций xUnit разработчик тестов должен был добавлять каждый *тестовый метод* (Test Method) вручную. Реализации без поддержки интроспекции сохранили это требование. Ниже приведен пример, основанный на ранней версии пакета CppUnit.

```
public:
    static CppUnit::Test *suite()
    {
        CppUnit::TestSuite *suite =
            new CppUnit::TestSuite("ComplexNumberTest");
        suite->addTest(
            new CppUnit::TestCaller<ComplexNumberTest>(
                "testEquality",
                &ComplexNumberTest::testEquality));
        suite->addTest(
            new CppUnit::TestCaller<ComplexNumberTest>(
                "testAddition",
                &ComplexNumberTest::testAddition));
        return suite;
    }
```

Здесь показано, как в CppUnit каждый *тестовый метод* (Test Method) заключается в экземпляр класса `TestCaller` для получения *объекта теста* (Testcase Object).

Пример: вызов тестового метода (Test Method) вручную

Следующий пример взят из набора тестов для приложения, написанного на VBA (Visual Basic for Applications — язык макросов для продуктов в составе Microsoft Office), в котором объекты не поддерживаются.

```
Sub TestAllStoryMacros()
    Call TestActivitySorting
    Call TestStoryHiding
    Call ReportSuccess("All Story Macros")
End Sub
```

Пример: перечисление наборов тестов (Test Suite Enumeration)

Данный механизм используется в случаях, когда *инфраструктура автоматизации тестов* (Test Automation Framework) не поддерживает *обнаружение тестов* (Test Discovery) или необходимо определить *именованный набор тестов* (Named Test Suite), включающий только подмножество тестов.

Основным недостатком использования *перечисления наборов тестов* (Test Suite Enumeration) для запуска всех тестов является высокая вероятность появления *потерянных тестов* (Lost Test), если забыть включить новый набор в *набор всех тестов* (AllTests Suite). Данный риск можно сократить, если следить за количеством тестов, запускаемых при получении кода из хранилища. Количество тестов, запускаемых при включении кода обратно в хранилище, должно увеличиваться на количество добавленных тестов.

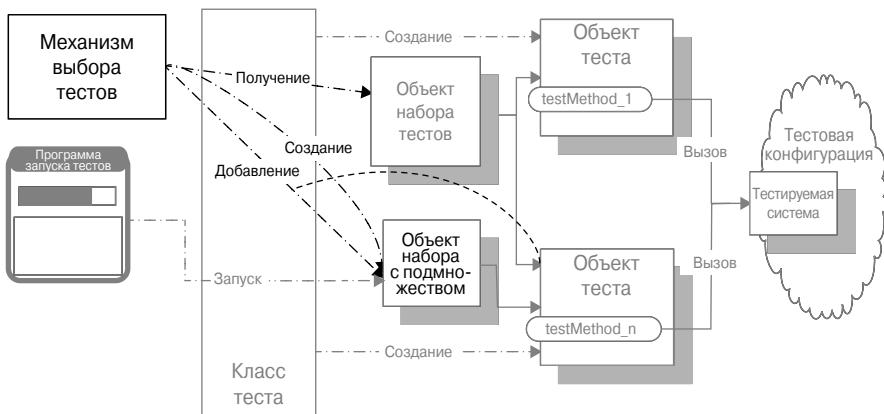
```
public class AllTests {
    public static Test suite() {
        TestSuite suite = new TestSuite("Test for allJUnitTests");
        //JUnit-BEGIN$
        suite.addTestSuite(
            com.clrstream.camug.example.test.InvoiceTest.class);
        suite.addTest(com.clrstream.ex7.test.AllTests.suite());
        suite.addTest(com.clrstream.ex8.test.AllTests.suite());
        suite.addTestSuite(
            com.xunitpatterns.guardassertion.Example.class);
        //JUnit-END$
        return suite;
    }
}
```

В этом примере используется способность среды разработки генерировать (возможно, повторно) набор всех тестов. (Среда Eclipse может по запросу повторно генерировать код между двумя маркерами в комментариях.) Время от времени нужно генерировать набор повторно, но этот подход позволяет хоть как-то защититься от появления *потерянных тестов* (Lost Test) в отсутствие механизма *обнаружения тестов* (Test Discovery).

Выбор тестов (Test Selection)

Откуда программа запуска тестов (Test Runner) знает, какие тесты запускать?

*Инфраструктура автоматизации тестов (Test Automation Framework) выбирает запускаемые **тестовые методы** (Test Method) во время выполнения в соответствии с атрибутами тестов.*



При наличии *тестовых методов* (Test Method, с. 378) в одном или нескольких *классах теста* (Testcase Class, с. 401) программа запуска тестов (Test Runner, с. 405) должна найти эти тесты. Механизм *выбора тестов* (Test Selection) позволяет динамически указать подмножество запускаемых тестов.

Как это работает

Разработчик выбирает подмножество запускаемых тестов, указывая критерий выбора. Критерий может быть основан на явных или неявных атрибутах *классов теста* (Testcase Class) или *тестовых методов* (Test Method).

Когда это использовать

Механизм *выбора тестов* (Test Selection) должен использоваться, когда необходимо запустить подмножество тестов, но нежелательно хранить отдельную структуру, созданную с помощью *перечисления тестов* (Test Enumeration, с. 425). Распространенным вариантом применения этого механизма является **контрольный набор тестов** (Smoke Test). Другие применения описываются в разделе, посвященном *именованному набору тестов* (Named Test Suite, с. 604).

Замечания по реализации

Выбор тестов (Test Selection) может быть реализован через создание *набора с подмножеством* (Subset Suite) на основе существующего *объекта набора тестов* (Test Suite Object,

с. 414) или через пропуск некоторых тестов из *объекта набора тестов* (Test Suite Object) при запуске *объектов теста* (Testcase Object, с. 410).

Как и в случае с *обнаружением тестов* (Test Discovery, с. 420) и *перечислением тестов* (Test Enumeration), *выбор тестов* (Test Selection) может применяться на двух разных уровнях: для выбора *классов теста* (Testcase Class) и для выбора *тестовых методов* (Test Method). Механизм *выбора тестов* (Test Selection) может быть встроен в *инфраструктуру автоматизации тестов* (Test Automation Framework, с. 332), а может быть реализован более грубо как часть процесса компиляции.

Вариант: выбор классов теста (Testcase Class Selection)

Существует несколько способов выбора *классов теста* (Testcase Class) для поиска *тестовых методов* (Test Method). Самым грубым способом *выбора классов теста* (Testcase Class Selection) является размещение *классов теста* (Testcase Class) в тестовых пакетах в соответствии с некоторым критерием. К сожалению, эта стратегия работает только для единственной схемы классификации тестов и с большой вероятностью снижает ценность *тестов как документации* (Tests as Documentation, с. 79). Несколько более гибким подходом является использование соглашения об именовании, например “содержит ‘WebServer’”, для выбора только тех классов, которые проверяют поведение некоторых фрагментов системы. Этот способ также имеет ограниченную применимость.

Наиболее гибким способом является реализация *выбора тестов* (Test Selection) внутри *инфраструктуры автоматизации тестов* (Test Automation Framework). Можно использовать атрибуты классов (.NET) или аннотации (Java) для обозначения характеристик *классов теста* (Testcase Class). Этот же способ может применяться на уровне *тестовых методов* (Test Method).

Вариант: выбор тестовых методов (Test Method Selection)

При реализации в качестве функции *инфраструктуры автоматизации тестов* (Test Automation Framework) *выбор тестовых методов* (Test Method Selection) может основываться на указании “категории” (или категорий), которой принадлежит *тестовый метод* (Test Method). Обычно это требует поддержки атрибутов методов (.NET) или аннотаций (Java) со стороны языка программирования. Кроме того, данный механизм может использовать схему именования методов, хотя это решение обладает меньшей гибкостью и приводит к более тесной связи с *программой запуска тестов* (Test Runner).

Пример: выбор классов теста (Testcase Class Selection) на основе атрибутов класса

Ниже приведен пример *выбора классов теста* (Testcase Class Selection) при использовании инфраструктуры NUnit. Атрибут класса *Category ("FastSuite")* указывает, что все тесты в составе *класса теста* (Testcase Class) должны быть включены (или исключены), когда категория “FastSuite” указывается *программой запуска тестов* (Test Runner).

```
[TestFixture]
[Category("FastSuite")]
public class CategorizedTests
{
```

```
[Test]
public void testFlightConstructor_OK()
// Методы опущены
}
```

Пример: выбор тестовых методов (Test Method Selection) на основе атрибутов методов

Этот пример *выбора тестовых методов* (Test Method Selection) основан на пакете NUnit. Атрибут метода `Category("SmokeTest")` указывает, что данный *тестовый метод* (Test Method) должен быть включен (или исключен) при указании категории “SmokeTest” *программой запуска тестов* (Test Runner).

```
[Test]
[Category("SmokeTests")]
public void testFlightMileage_asKm()
{
    // Настройка тестовой конфигурации
    Flight newFlight = new Flight(validFlightNumber);
    newFlight.setMileage(1122);
    // Вызов преобразователя пробега
    int actualKilometres = newFlight.getMileageAsKm();
    int expectedKilometres = 1810;
    // Проверка результата
    Assert.AreEqual(expectedKilometres, actualKilometres);
}
```


Глава 20

Шаблоны настройки тестовой конфигурации

Шаблоны в этой главе:

Настройка новой тестовой конфигурации

Встроенная настройка (In-line Setup)	434
Делегированная настройка (Delegated Setup).....	437
Метод создания (Creation Method)	441
Неявная настройка (Implicit Setup).....	449

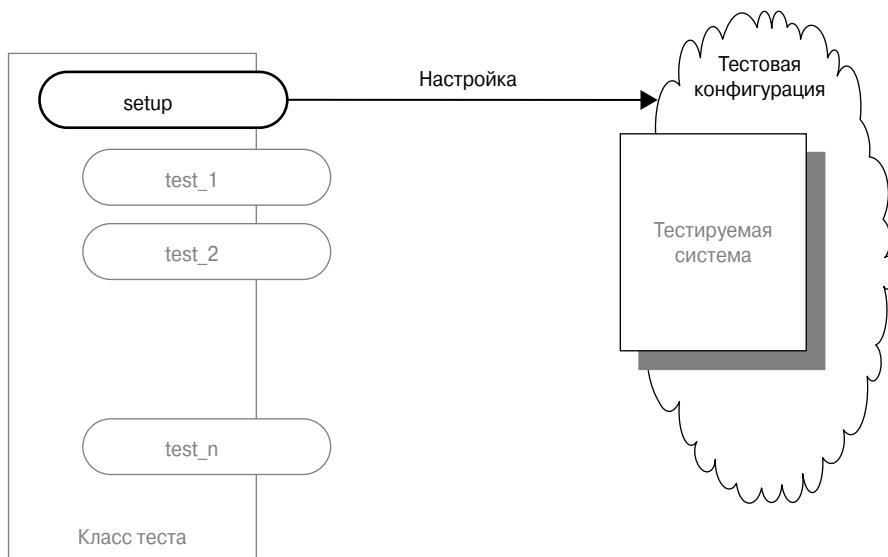
Создание общей тестовой конфигурации

Предварительно созданная тестовая конфигурация (Prebuilt Fixture)	454
“Ленивая” настройка (Lazy Setup)	460
Настройка тестовой конфигурации набора (Suite Fixture Setup)	465
Декоратор настройки (Setup Decorator).....	471
Цепочки тестов (Chained Tests).....	477

Встроенная настройка (In-line Setup)

Как создается новая тестовая конфигурация (Fresh Fixture)?

Каждый тестовый метод (Test Method) создает собственную новую тестовую конфигурацию (Fresh Fixture), вызывая подходящие методы.



Для запуска автоматизированного теста нужна понятная и полностью определенная тестовая конфигурация. Для создания *минимальной тестовой конфигурации* (Minimal Fixture, с. 336) можно воспользоваться *новой тестовой конфигурацией* (Fresh Fixture, с. 344). Настройка конфигурации внутри самого теста является наиболее очевидным способом ее создания.

Как это работает

Каждый тестовый метод (Test Method, с. 378) создает собственную тестовую конфигурацию, непосредственно вызывая соответствующий код в тестируемой системе. Код создания тестовой конфигурации соответствует первой фазе четырехфазного теста (Four-Phase Test, с. 387) и располагается в начале каждого *тестового метода* (Test Method).

Когда это использовать

Встроенная настройка (In-line Setup) может использоваться для очень простой логики настройки. Как только логика становится сложнее, возникает необходимость рассмотреть использование *делегированной настройки* (Delegated Setup, с. 437) или *неявной настройки* (Implicit Setup, с. 449) для фрагмента или всей тестовой конфигурации.

Кроме того, *встроенная настройка* (In-line Setup) может использоваться при создании первого наброска тестов, когда неизвестно, какая часть тестовой конфигурации будет повторяться от теста к тесту. Это пример применения шаблона процесса “Red-Green-Refactor” к самим тестам. Несмотря на это необходимо соблюдать осторожность при рефакторинге тестов, так как повреждения тестов могут оказаться незамеченными.

Еще один пример использования *встроенной настройки* (In-line Setup) — рефакторинг сложного кода создания тестовой конфигурации. Первым этапом может стать использование рефакторинга *встраивание метода* (In-line Method) [Ref] для всех *методов создания* (Creation Method, с. 441) и метода `setUp`. После этого можно попытаться повторно применить рефакторинг *выделение метода* (Extract Method) [Ref] для определения нового набора *методов создания* (Creation Method) с более понятным назначением и возможностью повторного использования.

Замечания по реализации

На практике большая часть логики настройки тестовой конфигурации будет содержать комбинацию стилей, например *встроенная настройка* (In-line Setup) поверх *неявной настройки* (Implicit Setup) или *делегированная настройка* (Delegated Setup), перемежающаяся *встроенной настройкой* (In-line Setup).

Пример: встроенная настройка (In-line Setup)

Ниже приведен пример простой встроенной настройки. Все необходимое включено в тело *тестового метода* (Test Method).

```
public void testStatus_initial() {
    // Встроенная настройка (In-line Setup)
    Airport departureAirport = new Airport("Calgary", "YYC");
    Airport destinationAirport = new Airport("Toronto", "YYZ");
    Flight flight = new Flight(flightNumber,
        departureAirport,
        destinationAirport);
    // Вызов тестируемой системы и проверка результата
    assertEquals(FlightState.PROPOSED, flight.getStatus());
    // Очистка:
    //     сборкой мусора
}
public void testStatus_cancelled() {
    // Встроенная настройка (In-line Setup)
    Airport departureAirport = new Airport("Calgary", "YYC");
    Airport destinationAirport = new Airport("Toronto", "YYZ");
    Flight flight = new Flight(flightNumber,
        departureAirport,
        destinationAirport);
    flight.cancel(); // Все еще часть настройки
    // Вызов тестируемой системы и проверка результата
    assertEquals(FlightState.CANCELLED, flight.getStatus());
    // Очистка:
    //     сборкой мусора
}
```

Замечания по рефакторингу

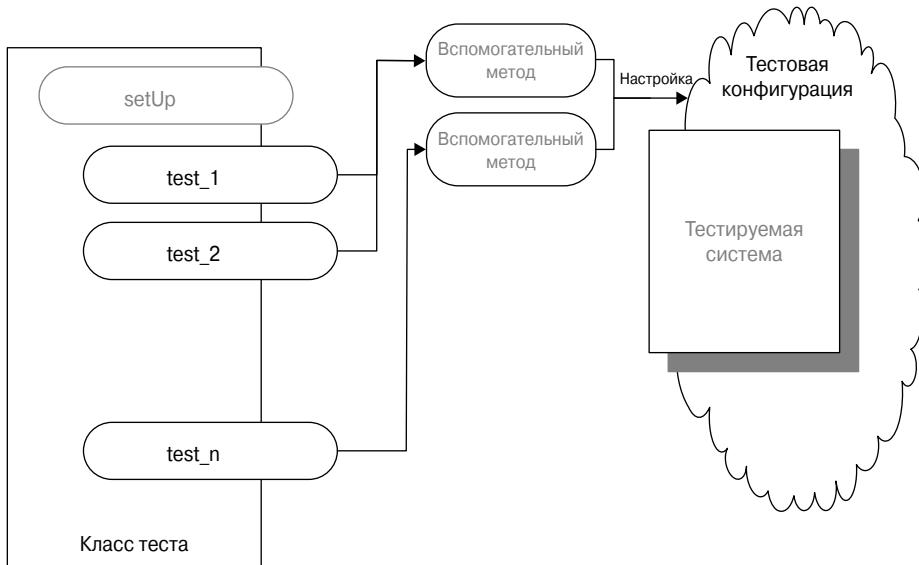
Обычно *встроенная настройка* (In-line Setup) является отправной точкой, а не целью рефакторинга. Но иногда встречаются очень сложные для понимания тесты, так как все в них происходит “за кулисами”, что является одним из проявлений *тайновенного гостя* (Mystery Guest; см. *Непонятный тест*, Obscure Test, с. 230). В других случаях внутри теста может модифицироваться ранее созданная тестовая конфигурация.

Обе ситуации показывают, что нужно разбить класс теста на несколько классов в зависимости от используемой конфигурации. Сначала рефакторинг *встраивание метода* (In-line Method) позволит организовать *встроенную настройку* (In-line Setup). После этого с помощью рефакторинга *выделение класса* (Extract Class) [Ref] можно реорганизовать тесты. Наконец, многократное повторение рефакторинга *выделение метода* (Extract Method) позволит получить более понятный набор методов настройки тестовой конфигурации.

Делегированная настройка (Delegated Setup)

Как создать новую тестовую конфигурацию (Fresh Fixture)?

Каждый тестовый метод (Test Method) создает собственную новую тестовую конфигурацию (Fresh Fixture), вызывая методы создания (Creation Method).



Для запуска автоматизированного теста нужна понятная и полностью определенная тестовая конфигурация. Для создания *минимальной тестовой конфигурации* (Minimal Fixture, с. 336) можно воспользоваться *новой тестовой конфигурацией* (Fresh Fixture, с. 344). При этом желательно избежать *дублирования тестового кода* (Test Code Duplication, с. 254).

Делегированная настройка (Delegated Setup) позволяет повторно использовать код для настройки тестовой конфигурации без отказа от *тестов как документации* (Tests as Documentation, с. 79).

Как это работает

Каждый *тестовый метод* (Test Method, с. 378) создает собственную конфигурацию, вызывая один или несколько *методов создания* (Creation Method, с. 441). Для получения *тестов как документации* (Tests as Documentation) для создания *минимальной тестовой конфигурации* (Minimal Fixture) используются *методы создания* (Creation Method). Методы генерируют полноценные объекты, готовые для использования тестом. При этом предпринимается попытка описать “общую картину” с помощью вызовов методов, а читатель теста будет видеть только те значения, которые касаются поведения тестируемой системы.

Когда это использовать

Делегированная настройка (Delegated Setup) используется в тех случаях, когда необходимо избежать *дублирования тестового кода* (Test Code Duplication) при создании похожих тестовых конфигураций в нескольких тестах. При этом удается сохранить видимость природы тестовой конфигурации изнутри *тестового метода* (Test Method). Неплохим решением является инкапсуляция важных, но не связанных непосредственно этапов настройки тестовой конфигурации и сохранение только значений и операций, описывающих смысл теста внутри *тестового метода* (Test Method). Подобная схема позволяет использовать *тесты как документацию* (Tests as Documentation) через сокрытие лишнего кода *встроенной настройки* (In-line Setup, с. 434), чтобы он не скрывал назначения теста. Кроме того, решается проблема *таинственного гостя* (Mystery Guest), поскольку описательные имена *методов создания* (Creation Method) остаются внутри *тестового метода* (Test Method).

Более того, *делегированная настройка* (Delegated Setup) позволяет использовать любую схему организации *тестовых методов* (Test Method). В частности, не обязательно помещать *тестовые методы* (Test Method) с одинаковой тестовой конфигурацией в один *класс теста* (Testcase Class, с. 401) для повторного использования метода `setUp`, как в случае применения *неявной настройки* (Implicit Setup, с. 449). Также *делегированная настройка* (Delegated Setup) позволяет избежать появления “хрупких” тестов (Fragile Test, с. 277) за счет вынесения ненужного взаимодействия с тестируемой системой из большого количества *тестовых методов* (Test Method) в значительно меньшее количество *методов создания* (Creation Method), что существенно упрощает их обслуживание.

Замечания по реализации

Современные инструменты рефакторинга часто позволяют создать первый набросок *метода создания* (Creation Method) через применение рефакторинга *выделение метода* (Extract Method). Создавая несколько тестов с помощью способа “клонирования и модификации”, необходимо следить за *дублированием тестового кода* (Test Code Duplication) в области настройки тестовой конфигурации. Для каждого проверяемого объекта генерируется *метод создания* (Creation Method), принимающий в качестве аргументов только те параметры объекта, от которых зависит результат теста.

В начале *методы создания* (Creation Method) можно оставить в составе *класса теста* (Testcase Class). Если они должны использоваться совместно с другим классом, их можно вынести в класс *абстрактного теста* (Abstract Testcase; см. *Суперкласс теста*, Testcase Superclass, с. 646) или во *вспомогательный класс теста* (Test Helper, с. 651).

Мотивирующий пример

Предположим, что тестируется модель состояний класса `Flight`. В каждом teste требуется объект полета в подходящем состоянии. Поскольку полет должен связывать как минимум два аэропорта, перед созданием полета необходимо создать аэропорты. Конечно, обычно аэропорты связаны с городами, штатами или провинциями. Для простоты предположим, что для создания аэропорта необходимо только название города и код аэропорта.

```

public void testStatus_initial() {
    // Встроенная настройка (In-line Setup)
    Airport departureAirport = new Airport("Calgary", "YYC");
    Airport destinationAirport = new Airport("Toronto", "YYZ");
    Flight flight = new Flight(flightNumber,
        departureAirport,
        destinationAirport);
    // Вызов тестируемой системы и проверка результата
    assertEquals(FlightState.PROPOSED, flight.getStatus());
    // Очистка:
    //     // сборкой мусора
}
public void testStatus_cancelled() {
    // Встроенная настройка (In-line Setup)
    Airport departureAirport = new Airport("Calgary", "YYC");
    Airport destinationAirport = new Airport("Toronto", "YYZ");
    Flight flight = new Flight(flightNumber,
        departureAirport,
        destinationAirport);
    flight.cancel(); // Все еще часть настройки
    // Вызов тестируемой системы и проверка результата
    assertEquals(FlightState.CANCELLED, flight.getStatus());
    // Очистка:
    //     // сборкой мусора
}

```

В данном случае очевидно *дублирование тестового кода* (Test Code Duplication).

Замечания по рефакторингу

Преобразование логики настройки тестовой конфигурации можно начинать с применения рефакторинга *выделение метода* (Extract Method) и вынесения часто повторяющихся последовательностей кода во вспомогательные методы с описательными именами. Но эти вызовы должны оставаться внутри *тестовых методов* (Test Method), чтобы читателю было ясно, что происходит. В тело вспомогательного теста выносится не относящаяся к назначению теста логика, необходимая для выполнения поставленной задачи. Если *делегированная настройка* (Delegated Setup) должна использоваться совместно с другим классом теста (Testcase Class), можно воспользоваться рефакторингом *подъем метода* (Pull up method) для переноса методов в *суперкласс теста* (Testcase Superclass) или *перемещение метода* (Move method) для их переноса во *вспомогательный класс теста* (Test Helper).

Пример: делегированная настройка (Delegated Setup)

В данной версии теста используется метод, скрывающий необходимость создания двух аэропортов и создающий их за пределами *тестового метода* (Test Method). Такую версию можно получить с помощью рефакторинга, а можно написать заново.

```

public void testGetStatus_initial() {
    // Настройка
    Flight flight = createAnonymousFlight();
    // Вызов тестируемой системы и проверка результата
    assertEquals(FlightState.PROPOSED, flight.getStatus());
}

```

```

    // Очистка
    // сборкой мусора
}
public void testGetStatus_cancelled2() {
    // Настройка
    Flight flight = createAnonymousCancelledFlight();
    // Вызов тестируемой системы и проверка результата
    assertEquals(FlightState.CANCELLED, flight.getStatus());
    // Очистка
    // сборкой мусора
}

```

Простота этих тестов основана на следующем *методе создания* (Creation Method), скрывающем “необходимую, но не относящуюся к назначению теста” функциональность от читателя теста.

```

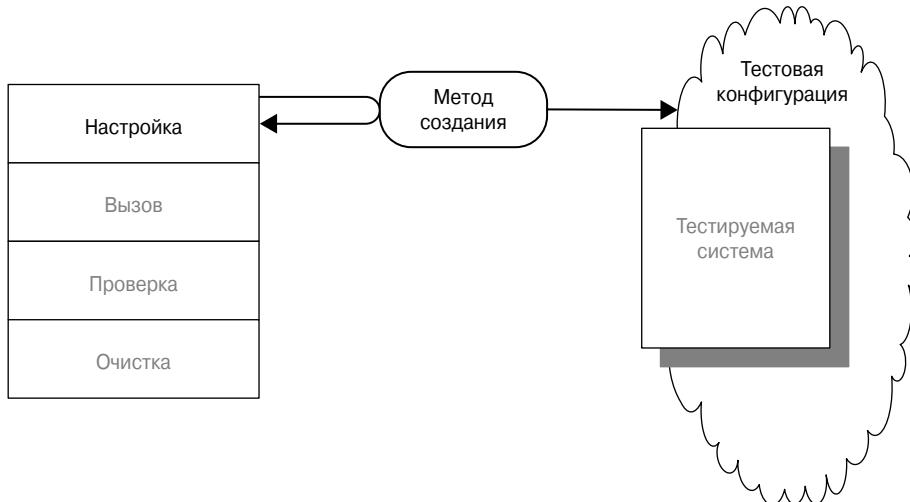
private int uniqueFlightNumber = 2000;
public Flight createAnonymousFlight() {
    Airport departureAirport = new Airport("Calgary", "YYC");
    Airport destinationAirport = new Airport("Toronto", "YYZ");
    Flight flight =
        new Flight(new BigDecimal(uniqueFlightNumber++),
                  departureAirport,
                  destinationAirport);
    return flight;
}
public Flight createAnonymousCancelledFlight() {
    Flight flight = createAnonymousFlight();
    flight.cancel();
    return flight;
}

```

Метод создания (Creation Method)

Как создать новую тестовую конфигурацию (Fresh Fixture)?

Для генерации тестовой конфигурации вызываются методы с описательными именами, скрывающие механизм создания готовых объектов.



Настройка тестовой конфигурации обычно предполагает создание нескольких объектов. В большинстве случаев особенности этих объектов (например, значения атрибутов) не важны, но они должны быть указаны для вызова конструктора. Добавление ненужной сложности в фазу настройки тестовой конфигурации теста может привести к появлению *непонятного теста* (Obscure Test, с. 230) и не помешает использованию *тестов как документации* (Tests as Documentation, с. 79)!

Как создать правильно инициализированный объект без усложнения теста *встроенной настройкой* (In-line Setup, с. 434)? Конечно, ответом является инкапсуляция сложности. *Делегированная настройка* (Delegated Setup, с. 437) позволяет вынести механизм создания конфигураций в другие методы, но контроль и координация процесса остаются внутри теста. Но куда делегировать настройку? *Метод создания* (Creation Method) является одним из способов инкапсуляции механизма создания объектов, не отвлекающим читателя тестов на ненужные детали.

Как это работает

При написании тестов не нужно интересоваться, существует ли необходимая вспомогательная функция. Ее нужно просто использовать! (Можно представить, что рядом сидит помощник, который быстро заполняет тела вызываемых функций, если они еще не существуют.) Тесты пишутся в терминах магических функций с описательными именами с передачей в качестве параметров только тех данных, которые будут проверяться утверждениями или влиять на результат теста.

После создания теста в таком стиле необходимо реализовать все магические функции. Функции генерации объектов называются *методами создания* (Creation Method). Они скрывают сложности процесса создания объектов. Простые методы вызывают подходящий конструктор, передавая соответствующие принятые по умолчанию значения в качестве параметров. Если параметром конструктора является другой объект, *метод создания* (Creation Method) сначала создает эти объекты и только после этого вызывает конструктор.

Метод создания (Creation Method) может храниться там же, где и *вспомогательные методы теста* (Test Utility Method, с. 610). Как обычно, решение принимается на основе ожидаемой области повторного использования и зависимости *метода создания* (Creation Method) от программного интерфейса тестируемой системы. Родственным шаблоном является *инкубатор объектов* (Object Mother; см. *Вспомогательный класс теста*, Test Helper, с. 651), который представляет собой комбинацию *метода создания* (Creation Method), *вспомогательного класса теста* (Test Helper) и необязательной *автоматической очистки* (Automated Teardown, с. 521).

Когда это использовать

Методы создания (Creation Method) должны использоваться каждый раз, когда генерация *новой тестовой конфигурации* (Fresh Fixture, с. 344) имеет сложную структуру, мешающую использованию *тестов как документации* (Tests as Documentation). Еще одним индикатором необходимости *методов создания* (Creation Method) является инкрементный процесс создания системы и предположение о частом изменении программного интерфейса (и особенно конструкторов объектов). Инкапсуляция знаний о создании объекта конфигурации является специальным случаем *инкапсуляции программного интерфейса тестируемой системы* (SUT API Encapsulation), которая позволяет избежать появления “хрупких” тестов (Fragile Test, с. 277) и *непонятных тестов* (Obscure Test).

Основным недостатком *методов создания* (Creation Method) является создание еще одного программного интерфейса, который придется изучать разработчикам. Это не проблема для разработчиков первой версии тестов, так как именно они создают этот интерфейс, но это “еще один интерфейс” для новичков в команде разработки. Даже в этом случае данный интерфейс изучить достаточно просто, так как это просто **методы фабрик** (factory method), организованные определенным образом.

При использовании предварительно созданной *тестовой конфигурации* (Prebuilt Fixture, с. 454) для обнаружения существующих объектов должны применяться *методы поиска* (Finder Method; см. *Вспомогательный метод теста*, Test Utility Method). В то же время *метод создания* (Creation Method) может использоваться для генерации модифицируемых объектов, добавляемых к *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture).

Имеет смысл рассмотреть несколько вариантов *методов создания* (Creation Method).

Вариант: параметризованный метод создания (Parameterized Creation Method)

Несмотря на возможность (и даже желательность) использования *метода создания* (Creation Method) без параметров, многие тесты могут потребовать модификации создаваемого объекта. Параметризованный метод создания позволяет тестам передавать некоторые атрибуты создаваемого объекта. В таком случае должны передаваться только атри-

буты, влияющие (или не влияющие, если необходимо доказать их несвязанность) на результат теста. В противном случае неизбежно скатывание по наклонной плоскости к *непонятным тестам* (Obscure Test).

Вариант: анонимный метод создания (Anonymous Creation Method)

Анонимный метод создания (Anonymous Creation Method) автоматически создает и использует *отдельное сгенерированное значение* (Distinct Generated Value; см. *Сгенерированное значение*, Generated Value, с. 726) в качестве уникального идентификатора создаваемого объекта, даже если получаемые аргументы не уникальны. Такое поведение незаменимо при попытках избежать появления *неповторяемых тестов* (Unrepeatable Test), поскольку каждый создаваемый объект уникален (даже в пределах нескольких запусков одного теста). Если тесту нужны конкретные атрибуты создаваемого объекта, он может передать их в качестве параметров *метода создания* (Creation Method). Такое поведение превращает *анонимный метод создания* (Anonymous Creation Method) в *параметризованный анонимный метод создания* (Parameterized Anonymous Creation Method).

Вариант: параметризованный анонимный метод создания (Parameterized Anonymous Creation Method)

Параметризованный анонимный метод создания (Parameterized Anonymous Creation Method) представляет собой комбинацию нескольких вариантов *метода создания* (Creation Method) и позволяет передавать некоторые атрибуты создаваемого объекта в качестве параметров и в то же время генерировать для объекта уникальный идентификатор. Кроме того, *метод создания* (Creation Method) может вообще не принимать параметры, если тест не заинтересован ни в одном из аргументов объекта.

Вариант: метод, достигающий указанного состояния (Named State Reaching Method)

Некоторые тестируемые системы не имеют состояния, т.е. можно вызывать любой метод в любое время. С другой стороны, если система имеет внутреннее состояние и допустимость или поведение методов зависит от текущего состояния, важно проверять каждый метод во всех начальных состояниях. Набор таких тестов можно собрать в один *тестовый метод* (Test Method, с. 378), но подобный подход приведет к появлению “энергичного” теста (Eager Test). Для этих целей лучше воспользоваться набором *тестов одного условия* (Single Condition Test, с. 99). К сожалению, остается проблема установки начального состояния для каждого теста без *дублирования тестового кода* (Test Code Duplication, с. 254).

Одним из очевидных решений является помещение всех тестов, зависящих от одного начального состояния, в один *класс теста* (Testcase Class) и создание тестируемой системы с необходимым состоянием в методе *setUp* с помощью *неявной настройки* (Implicit Setup, с. 449). Это организация тестов в виде *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639). Альтернативное решение предполагает использование *делегированной настройки* (Delegated Setup) через вызов *достигающего указанного состояния метода* (Named State Reaching Method). При этом остается возможность другой организации *классов теста* (Testcase Class).

В любом случае проще понять короткий код настройки тестируемой системы. Именно для этого может понадобиться *достигающий указанного состояния метод*

(Named State Reaching Method). Сокрытие логики создания объектов с правильным состоянием в одном месте (в *классе теста*, Testcase Class, или во *вспомогательном классе теста*, Test Helper) сокращает объем обновляемого кода при изменениях в механизме достижения состояния.

Вариант: подключаемый метод (Attachment Method)

Предположим, что тестируемый объект уже существует и его необходимо немного модифицировать. Эта операция встречается в таком количестве тестов, что возникла необходимость реализовать ее в одном месте. Решением проблемы является *подключаемый метод* (Attachment Method). Основным его отличием от *метода создания* (Creation Method) является передача модифицируемого объекта (созданного с помощью *метода создания*, Creation Method) в качестве параметра.

Замечания по реализации

Большинство *методов создания* (Creation Method) появляются в результате рефакторинга *выделение метода* (Extract Method) над фрагментами существующих тестов. При написании тестов в направлении “извне внутрь” предполагается, что *методы создания* (Creation Method) уже существуют, а тела методов заполняются позднее. Фактически определяется *язык высокого уровня* (Higher-Level Language, с. 95) для описания тестовых конфигураций. Но существует и совершенно другой способ определения *методов создания* (Creation Method).

Вариант: повторное использование теста для настройки тестовой конфигурации (Reuse Test for Fixture Setup)

Тестовую конфигурацию можно создавать, вызывая другой *тестовый метод* (Test Method). Предполагается, что можно получить доступ к конфигурации, созданной другим тестом. Это можно сделать как через объект Registry [PEAA], так и через переменные экземпляра *объекта теста* (Testcase Object, с. 410).

Такая реализация *методов создания* (Creation Method) может подойти, когда уже существуют тесты, зависящие от других тестов для создания конфигурации, но необходимо сократить вероятность неудачного завершения тестов при изменении порядка запуска. Не забывайте, что тесты будут работать медленнее, так как каждый из них будет запускать тест, от которого он зависит. В результате некоторые тесты будут запускаться неоднократно. Замедление будет не таким заметным, если заменить все медленные компоненты, например базу данных, *поддельными объектами* (Fake Object, с. 565).

Заключение *тестового метода* (Test Method) в *метод создания* (Creation Method) все же лучше, чем непосредственный вызов другого *тестового метода* (Test Method), так как в большинстве случаев имена методов основаны на проверяемых условиях, а не на типе создаваемой тестовой конфигурации. *Метод создания* (Creation Method) позволяет вставить описательное имя между методом-клиентом и *тестовым методом* (Test Method), содержащим необходимую реализацию. Кроме того, таким образом решается проблема *одиноких тестов* (Lonely Test), поскольку другие тесты будут явно запускаться из вызывающего теста, а не полагаться на предположения о его успешном завершении. Подобная схема снижает “хрупкость” тестов и упрощает понимание, хотя и не решает проблему *взаимодействующих тестов* (Interacting Tests): если вызываемый тест завершается не-

удачно и оставляет после себя конфигурацию в неопределенном состоянии, вызывающий тест также завершится неудачно, хотя проверяемая им функциональность может работать правильно.

Мотивирующий пример

В следующем примере тест `testPurchase` требует объект `Customer` для использования в качестве покупателя (`buyer`). Имя и фамилия покупателя не влияют на процедуру покупки, но являются обязательными параметрами конструктора объекта `Customer`. При этом тест проверяет кредитный рейтинг (“G”) и состояние объекта `Customer`.

```
public void testPurchase_firstPurchase_ICC() {
    Customer buyer =
        new Customer(17, "FirstName", "LastName", "G", "ACTIVE");
    // ...
}
public void testPurchase_subsequentPurchase_ICC() {
    Customer buyer =
        new Customer(18, "FirstName", "LastName", "G", "ACTIVE");
    // ...
}
```

Использование конструкторов внутри теста может быть затруднено, особенно при инкрементном создании приложения. При каждой модификации параметров конструктора придется пересмотреть множество тестов или сохранить обратную совместимость сигнатур конструктора специально для обеспечения нормальной работы тестов.

Замечания по рефакторингу

Рефакторинг *выделение метода* (Extract Method) позволяет избавиться от непосредственного вызова конструктора. Новый *метод создания* (Creation Method) можно назвать `createCustomer`, что хорошо описывает назначение метода.

Пример: анонимный метод создания (Anonymous Creation Method)

В следующем примере вместо непосредственного вызова конструктора `Customer` используется *метод создания* (Creation Method). Обратите внимание на разрыв тесной связи между кодом настройки тестовой конфигурации и конструктором. Если в конструктор `Customer` добавить еще один параметр (например, номер телефона), придется модифицировать только *метод создания* (Creation Method), который будет вставлять принятое по умолчанию значение. Код настройки конфигурации останется неизменным, так как изменения будут скрыты в методе создания.

```
public void testPurchase_firstPurchase_ACM() {
    Customer buyer = createAnonymousCustomer();
    // ...
}
public void testPurchase_subsequentPurchase_ACM() {
    Customer buyer = createAnonymousCustomer();
    // ...
}
```

Этот шаблон называется *анонимным методом создания* (Anonymous Creation Method), поскольку “личность” покупателя не играет роли. *Анонимный метод создания* (Anonymous Creation Method) может выглядеть так, как показано ниже.

```
public Customer createAnonymousCustomer() {
    int uniqueid = getUniqueCustomerId();
    return new Customer(uniqueid,
        "FirstName" + uniqueid,
        "LastName" + uniqueid,
        "G", "ACTIVE");
}
```

Обратите внимание на использование *отдельного сгенерированного значения* (Distinct Generated Value), которое обеспечивает отличие каждого анонимного объекта Customer от других таких же объектов.

Пример: параметризованный метод создания (Parameterized Creation Method)

Если необходимо передать некоторые атрибуты объекта Customer в качестве параметров, можно определить *параметризованный метод создания* (Parameterized Creation Method).

```
public void testPurchase_firstPurchase_PCM() {
    Customer buyer =
        createCreditworthyCustomer("FirstName", "LastName");
    // ...
}
public void testPurchase_subsequentPurchase_PCM() {
    Customer buyer = createCreditworthyCustomer("FirstName", "LastName");
    // ...
}
```

Ниже показано определение соответствующего *параметризованного метода создания* (Parameterized Creation Method).

```
public Customer createCreditworthyCustomer(
    String firstName, String lastName) {
    int uniqueid = getUniqueCustomerId();
    Customer customer =
        new Customer(uniqueid, firstName, lastName, "G", "ACTIVE");
    customer.setCredit(CreditRating.EXCELLENT);
    customer.approveCredit();
    return customer;
}
```

Пример: подключаемый метод (Attachment Method)

Ниже приведен пример теста, использующего *подключаемый метод* (Attachment Method) для связывания двух покупателей и проверки получения ими максимальных скидок.

```
public void testPurchase_relatedCustomerDiscount_AM() {
    Customer buyer = createCreditworthyCustomer("Related", "Buyer");
    Customer discountHolder =
```

```

        createCreditworthyCustomer("Discount", "Holder");
createRelationshipBetweenCustomers(buyer, discountHolder);
// ...
}

```

За кулисами *подключаемый метод* (Attachment Method) выполняет все необходимые операции для установки этой связи.

```

private void createRelationshipBetweenCustomers(
    Customer buyer,
    Customer discountHolder) {
    buyer.addToRelatedCustomersList(discountHolder);
    discountHolder.addToRelatedCustomersList(buyer);
}

```

Хотя это достаточно простой пример, вызов данного метода понять проще, чем прочитать оба вызова, из которых он состоит.

Пример: повторное использование теста для настройки тестовой конфигурации (Reuse Tests For Fixture Setup)

Для настройки конфигурации можно повторно использовать другие тесты. Ниже показано, как не нужно использовать тесты.

```

private Customer buyer;
private AccountManager sut = new AccountManager();
private Account account;
public void testCustomerConstructor_SRT() {
    // Вызов
    buyer = new Customer(17, "First", "Last", "G", "ACTIVE");
    // Проверка
    assertEquals("First", buyer.firstName(), "first");
    // ...
}
public void testPurchase_SRT() {
    testCustomerConstructor_SRT(); // Оставляет в поле "buyer"
    account = sut.createAccountForCustomer(buyer);
    assertEquals(buyer.name, account.customerName, "cust");
    // ...
}

```

Здесь наблюдается двойная проблема. Во-первых, имя вызываемого *тестового метода* (Test Method) описывает проверяемое условие, а не конечный результат вызова (например, объект Customer в поле buyer). Во-вторых, тест не возвращает объект Customer; он просто оставляет его в переменной экземпляра. Такая схема работает только потому, что повторно используемый *тестовый метод* (Test Method) расположен в том же *классе теста* (Testcase Class). Если бы он располагался в другом классе, для доступа к значению buyer пришлось бы пару раз прыгнуть выше головы. Проще достичь этой цели через инкапсуляцию вызова внутри *метода создания* (Creation Method).

```

private Customer buyer;
private AccountManager sut = new AccountManager();
private Account account;
public void testCustomerConstructor_RTCM() {

```

```
// Вызов
buyer = new Customer(17, "First", "Last", "G", "ACTIVE");
// Проверка
assertEquals("First", buyer.firstName(), "first");
// ...
}
public void testPurchase_RTCM() {
    buyer = createCreditworthyCustomer();
    account = sut.createAccountForCustomer(buyer);
    assertEquals(buyer.name, account.customerName, "cust");
    // ...
}
public Customer createCreditworthyCustomer() {
    testCustomerConstructor_RTCM();
    return buyer;
    // ...
}
```

Обратите внимание, насколько более простым для чтения стал этот тест. Происхождение поля `buyer` становится очевидным. Все оказалось очень просто, так как оба *тестовых метода* (Test Method) находятся в одном классе. В случае разных классов *методу создания* (Creation Method) пришлось бы создавать экземпляр другого *класса теста* (Testcase Class) перед запуском теста, а затем искать способ доступа к переменной экземпляра, чтобы вернуть ее содержимое в вызывающий тестовый метод.

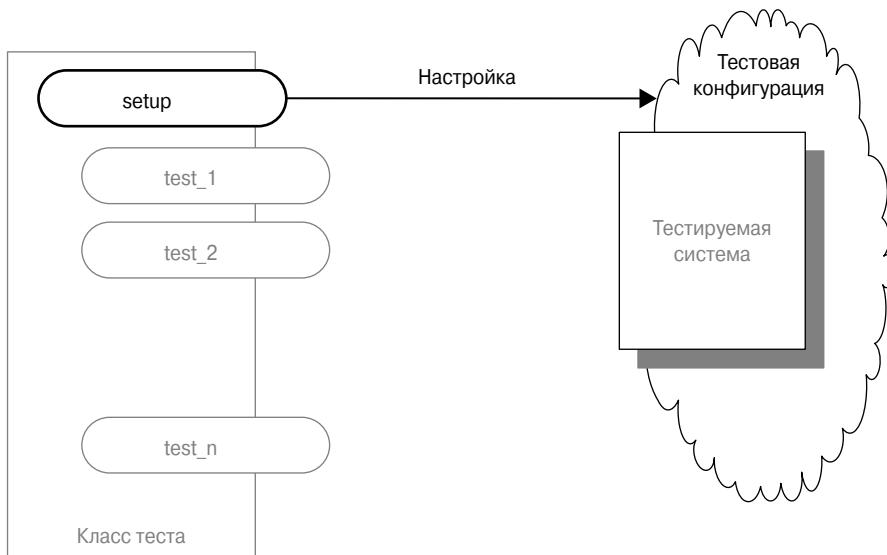
Неявная настройка (Implicit Setup)

Как создать новую тестовую конфигурацию (Fresh Fixture)?

Общая для нескольких тестов конфигурация создается в методе `setUp`.

Также известен как:

Настойка через “ловушки” (Hooked Setup), Вызываемая инфраструктурой настройка (Framework-Invoked Setup), Общий метод настройки (Shared Setup Method)



Для запуска автоматизированного теста необходима полностью понятная и определенная тестовая конфигурация. Подход на основе *новой тестовой конфигурации* (Fresh Fixture, с. 344) позволяет создавать *минимальную тестовую конфигурацию* (Minimal Fixture, с. 336) для запускаемых тестов.

Неявная настройка (Implicit Setup) позволяет повторно использовать код настройки тестовой конфигурации для всех *тестовых методов* (Test Method, с. 378) в *классе теста* (Testcase Class, с. 401).

Как это работает

Все тесты в составе *класса теста* (Testcase Class) создают одинаковые *новые тестовые конфигурации* (Fresh Fixture) с помощью специального метода класса `setUp`. Этот метод вызывается автоматически *инфраструктурой автоматизации тестов* (Test Automation Framework, с. 332) перед вызовом *тестового метода* (Test Method). В результате удается повторно использовать код создания тестовой конфигурации без повторного использования ее экземпляров. Такой подход называется “неявной” настройкой, так как вызов логики настройки тестовой конфигурации не виден внутри *тестового метода* (Test

Method), в отличие от встроенной настройки (In-line Setup, с. 434) и делегированной настройки (Delegated Setup, с. 437).

Когда это использовать

Неявная настройка (Implicit Setup) может использоваться, когда несколько *тестовых методов* (Test Method) в одном *классе теста* (Testcase Class) нуждаются в одной и той же *новой тестовой конфигурации* (Fresh Fixture). Если всем *тестовым методам* (Test Method) нужна одна конфигурация, то вся *минимальная тестовая конфигурация* (Minimal Fixture) может создаваться внутри метода *setUp*. Такая форма организации *тестовых методов* (Test Method) называется *классом теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639).

Когда *тестовым методам* (Test Method) нужны разные конфигурации и используется организация тестов *класс теста для каждой функции* (Testcase Class per Feature, с. 633) или *класс теста для каждого класса* (Testcase Class per Class, с. 627), воспользоваться *неявной настройкой* (Implicit Setup) и создать при этом *минимальную тестовую конфигурацию* (Minimal Fixture) оказывается сложнее. Метод *setUp* можно использовать для создания только той части конфигурации, которая не приводит к проблемам в других тестах. Разумный компромисс заключается в использовании *неявной настройки* (Implicit Setup) для создания важных, но не относящихся к смыслу тестов, фрагментов конфигурации, а и в создании критических (и отличающихся от теста к тесту) элементов в отдельных *тестовых методах* (Test Method). Примером “важных, но не относящихся к смыслу” элементов можно назвать инициализацию переменных с “неиспользуемыми” значениями, а также настройку скрытых механизмов, например, подключения к базе данных. Логика создания конфигурации, непосредственно влияющая на состояние тестируемой системы, должна оставаться внутри отдельных *тестовых методов* (Test Method), кроме случаев, когда все *тестовые методы* (Test Method) нуждаются в одном и том же начальном состоянии.

Очевидными альтернативами создания *новой тестовой конфигурации* (Fresh Fixture) являются *встроенная настройка* (In-line Setup), при которой вся логика создания конфигурации расположена внутри *тестового метода* (Test Method), и *делегированная настройка* (Delegated Setup), при которой общий код настройки конфигурации выносится в *методы создания* (Creation Method), вызываемые из *тестовых методов* (Test Method).

Неявная настройка (Implicit Setup) позволяет избавиться от большей части *дублирования тестового кода* (Test Code Duplication, с. 254) и избежать появления “хрупких” тестов (Fragile Test, с. 277), так как побочное взаимодействие с тестируемой системой выносится из большого количества тестов в намного меньшее количество методов, что значительно упрощает процесс обслуживания. Но существует опасность появления *непонятных тестов* (Obscure Test, с. 230), так как *таинственный гость* (Mystery Guest) делает используемые тестовые конфигурации менее очевидными. Кроме того, если на самом деле тесты не нуждаются в идентичных конфигурациях, такой подход к настройке может привести к появлению “хрупкой” *тестовой конфигурации* (Fragile Fixture).

Замечания по реализации

Ниже перечислены основные особенности реализации *неявной настройки* (Implicit Setup).

- Как обеспечить вызов метода `setUp`?
- Как обеспечить очистку тестовой конфигурации?
- Как *тестовые методы* (Test Method) получают доступ к конфигурации?

Вызов кода настройки

Метод `setUp` является наиболее распространенным средством запуска *неявной настройки* (Implicit Setup). *Инфраструктура автоматизации тестов* (Test Automation Framework) автоматически вызывает метод `setUp` перед каждым *тестовым методом* (Test Method). Строго говоря, метод `setUp` не является единственным средством неявной настройки конфигурации. Например, *настройка тестовой конфигурации набора* (Suite Fixture Setup, с. 465) используется для настройки и очистки *общей тестовой конфигурации* (Shared Fixture, с. 350), повторно используемой *тестовыми методами* (Test Method) в пределах *класса теста* (Testcase Class). Кроме того, *декоратор настройки* (Setup Decorator, с. 471) перемещает метод `setUp` в объект `Decorator`, установленный между *объектом набора тестов* (Test Suite Object, с. 414) и *программой запуска тестов* (Test Runner, с. 405). И тот, и другой способы являются вариантами *неявной настройки* (Implicit Setup), так как логика `setUp` расположена за пределами *тестового метода* (Test Method).

Очистка тестовой конфигурации

Парным для неявной настройки методом очистки является *неявная очистка* (Implicit Teardown, с. 533). Все, что создано в методе `setUp` и не удалено с помощью *автоматической очистки* (Automated Teardown, с. 521) или сборщиком мусора, должно быть очищено в методе `tearDown`.

Доступ к тестовой конфигурации

Тестовые методы (Test Method) должны иметь возможность доступа к конфигурации, созданной в методе `setUp`. При создании в пределах метода было достаточно локальных переменных. Для передачи между методом `setUp` и *тестовым методом* (Test Method) локальные переменные приходится заменять переменными экземпляра. Очень важно не использовать переменные класса, так как это создает потенциал для появления *общей тестовой конфигурации* (Shared Fixture). Во врезке “Всегда есть исключения” на с. 411 показаны примеры, когда переменные экземпляра не обеспечивают такой уровень изоляции.)

Мотивирующий пример

В следующем примере каждому тесту необходимо создать перелет между парой аэропортов.

```
public void testStatus_initial() {
    // Встроенная настройка (In-line Setup)
    Airport departureAirport = new Airport("Calgary", "YYC");
    Airport destinationAirport = new Airport("Toronto", "YYZ");
```

```

Flight flight = new Flight(flightNumber,
                           departureAirport,
                           destinationAirport);
// Вызвать тестируемую систему и проверить результат
assertEquals(FlightState.PROPOSED, flight.getStatus());
// Очистка
// сборкой мусора
}
public void testStatus_cancelled() {
// Встроенная настройка (In-line Setup)
Airport departureAirport = new Airport("Calgary", "YYC");
Airport destinationAirport = new Airport("Toronto", "YYZ");
Flight flight = new Flight(flightNumber,
                           departureAirport,
                           destinationAirport);
flight.cancel(); // Все еще часть настройки
// Вызвать тестируемую систему и проверить результат
assertEquals(FlightState.CANCELLED, flight.getStatus());
// Очистка
// сборкой мусора
}

```

Замечания по рефакторингу

В показанных тестах присутствует достаточно большая доля *дублирования тестового кода* (Test Code Duplication). От дублирования можно избавиться, проведя рефакторинг *класса теста* (Testcase Class) в направлении использования *неявной настройки* (Implicit Setup). В данном случае возможны два варианта рефакторинга.

В первом случае, обнаружив, что тесты выполняют одинаковые действия по созданию тестовой конфигурации, но не используют совместно метод `setUp`, можно воспользоваться рефакторингом *выделение метода* (Extract Method) для создания метода `setUp` на основе логики одного из тестов. Кроме того, придется преобразовать локальные переменные в переменные экземпляра, чтобы хранить ссылки на конфигурацию до ее использования *тестовым методом* (Test Method).

Во втором случае *класс теста* (Testcase Class) уже может использовать метод `setUp` для создания конфигурации, но содержащиеся в нем тесты нуждаются в разных конфигурациях. В такой ситуации можно воспользоваться рефакторингом *выделение класса* (Extract Class) для перемещения всех *тестовых методов* (Test Method) с другими требованиями к методу `setUp` в отдельный класс. Убедитесь, что все переменные экземпляра, доносящие знание о тестовой конфигурации из метода `setUp` в *тестовые методы* (Test Method), переносятся в другой класс вместе с методом `setUp`. Иногда проще клонировать *класс теста* (Testcase Class) и удалить тесты и переменные экземпляра из одной или другой копии.

Пример: неявная настройка (Implicit Setup)

Ниже показан модифицированный пример, в котором весь общий код создания тестовой конфигурации вынесен в метод `setUp` *класса теста* (Testcase Class). В результате код не повторяется в каждом teste и тесты стали значительно короче.

```
Airport departureAirport;
Airport destinationAirport;
Flight flight;
public void setUp() throws Exception{
    super.setUp();
    departureAirport = new Airport("Calgary", "YYC");
    destinationAirport = new Airport("Toronto", "YYZ");
    BigDecimal flightNumber = new BigDecimal("999");
    flight = new Flight(flightNumber, departureAirport,
                        destinationAirport);
}
public void testGetStatus_initial() {
    // Неявная настройка (Implicit Setup)
    // Вызвать тестируемую систему и проверить результат
    assertEquals(FlightState.PROPOSED, flight.getStatus());
}
public void testGetStatus_cancelled() {
    // Частично переопределенная неявная настройка
    flight.cancel();
    // Вызвать тестируемую систему и проверить результат
    assertEquals(FlightState.CANCELLED, flight.getStatus());
}
```

Подобный подход обладает рядом недостатков, связанных с отсутствием организации *тестовых методов* (Test Method) в структуру *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture). (В данном случае используется организация *класса теста для каждой функции* (Testcase Class per Feature).) Все *тестовые методы* (Test Method) в *классе теста* (Testcase Class) должны быть в состоянии работать с одной и той же тестовой конфигурацией (как минимум в начале работы), что продемонстрировано на примере частично переопределенной тестовой конфигурации во втором тесте. Кроме того, из текста теста невозможно понять, какая тестовая конфигурация используется. Откуда берется содержимое *flight*? В чем особенности этого значения? Невозможно даже переименовать переменную экземпляра, чтобы описать природу перелета, так как в переменной хранятся перелеты с различными характеристиками для разных тестов.

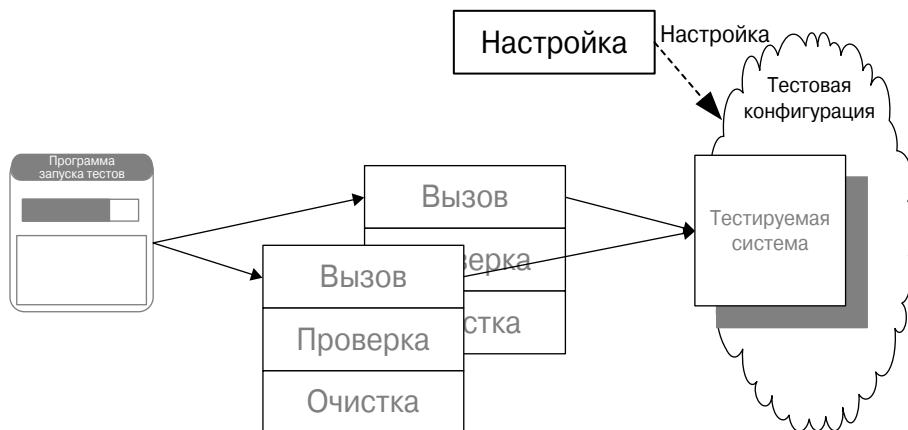
Предварительно созданная тестовая конфигурация (Prebuilt Fixture)

Также известен как:

Предварительно созданный контекст (Prebuilt Context), Тестовый стенд (Test Bed)

Как создать общую тестовую конфигурацию (Shared Fixture) до запуска использующих ее тестовых методов?

Общая тестовая конфигурация (Shared Fixture) создается отдельно от запуска тестов.



Если из удобства или по необходимости было выбрано использование *общей тестовой конфигурации* (Shared Fixture, с. 350), ее необходимо создавать до начала использования.

Как это работает

Тестовая конфигурация создается в какой-то момент до начала ее использования. Конфигурацию можно создавать несколькими способами, которые рассматриваются ниже. Наиболее важным моментом является отсутствие необходимости создавать тестовую конфигурацию при каждом запуске набора тестов, так как конфигурация существует дольше механизма ее создания и тестов, ее использующих.

Когда это использовать

Создавая конфигурацию время от времени, можно сократить накладные расходы по созданию *общей тестовой конфигурации* (Shared Fixture) при каждом запуске набора тестов. Такой подход оправдан, когда стоимость создания *общей тестовой конфигурации* (Shared Fixture) исключительно высока или процесс не поддается автоматизации.

Из-за *ручного вмешательства* (Manual Intervention, с. 287), необходимого для создания (или воссоздания) конфигурации перед запуском тестов, одна и та же конфигурация будет использоваться несколько раз, что может привести к появлению *нестабильных тестов* (Erratic Test, с. 267) из-за нарушения целостности конфигурации. Этых проблем можно избежать, рассматривая *предварительно созданную тестовую конфигурацию* (Prebuilt Fixture) как немодифицируемую *общую тестовую конфигурацию* (Immutable Shared Fixture).

и создавая *новую тестовую конфигурацию* (Fresh Fixture, с. 344) для любого модифицируемого элемента конфигурации.

Альтернативой *предварительно созданной тестовой конфигурации* (Prebuilt Fixture) является *общая тестовая конфигурация* (Shared Fixture), создаваемая при каждом запуске тестов, а также *новая тестовая конфигурация* (Fresh Fixture). Общие конфигурации можно создавать с помощью настройки *тестовой конфигурации набора* (Suite Fixture Setup, с. 465), “ленивой” настройки (Lazy Setup, с. 460) или декоратора настройки (Setup Decorator, с. 471). *Новая тестовая конфигурация* (Fresh Fixture) создается с помощью *встроенной настройки* (In-line Setup, с. 434), *неявной настройки* (Implicit Setup, с. 449) или *делегированной настройки* (Delegated Setup, с. 437).

Вариант: глобальная тестовая конфигурация (Global Fixture)

Это специальный случай *предварительно созданной тестовой конфигурации* (Prebuilt Fixture), когда конфигурация используется совместно несколькими разработчиками тестов. Ключевым отличием является глобальная видимость конфигурации, а не “принадлежность” конкретному пользователю. Данный шаблон чаще всего применяется при использовании единственной “песочницы” с базой данных (Database Sandbox, с. 658) без применения *схемы разбиения базы данных* (Database Partitioning Scheme).

Сами тесты и логика настройки могут не отличаться от использующих *предварительно созданной тестовой конфигурации* (Prebuilt Fixture). Основное отличие заключается в возможных проблемах при использовании глобальной конфигурации. Так как конфигурация совместно используется несколькими разработчиками и каждый из них запускает отдельную *программу запуска тестов* (Test Runner, с. 405) на выделенном процессоре, могут возникать различные проблемы межпроцессного взаимодействия. Наиболее распространенной из них является “война” запуска тестов (Test Run War) со случайными результатами. Этой проблемы можно избежать, используя *схему разбиения базы данных* (Database Partitioning Scheme) или *отдельные сгенерированные значения* (Distinct Generated Value) для всех полей с ограничением уникальности ключа.

Замечания по реализации

Сами тесты выглядят как тесты, использующие *общую тестовую конфигурацию* (Shared Fixture). Отличаются они способом создания тестовой конфигурации. Читатель не сможет найти признаки логики настройки ни в *классе теста* (Testcase Class, с. 401), ни в *декораторе настройки* (Setup Decorator) или методе *настройки тестовой конфигурации набора* (Suite Fixture Setup). Вместо этого настройка конфигурации, скорее всего, выполняется вручную, с помощью некоторой операции копирования базы данных, с помощью *загрузчика данных* (Data Loader) или через использование *сценария наполнения базы данных* (Database Population Script). В этих примерах *настройки через “черный ход”* (Back Door Setup) мы обходим тестируемую систему и взаимодействуем с базой данных непосредственно. (См. врезку “База данных как программный интерфейс тестируемой системы?” на с. 367 для знакомства с примерами, когда “черный ход” является основным интерфейсом.) Еще одно решение — использование *теста с настройкой тестовой конфигурации* (Fixture Setup Testcase; см. *Цепочки тестов*, Chained Tests, с. 477), запускаемого вручную или по расписанию из *программы запуска тестов* (Test Runner).

Еще одним отличием является реализация *методов поиска* (Finder Method). Нельзя просто сохранить результат создания объектов в переменной класса или в хранящемся в памяти *реестре тестовых конфигураций* (Test Fixture Registry), так как конфигурация не создается во время работы тестов. Существует два наиболее распространенных варианта решения:

- 1) хранение уникальных идентификаторов, сгенерированных на этапе создания конфигурации, в постоянном *реестре тестовых конфигураций* (Test Fixture Registry) (в виде файла), чтобы *методы поиска* (Finder Method) могли найти элементы конфигурации позднее;
- 2) фиксация идентификаторов в коде *методов поиска* (Finder Method).

Объекты/записи, соответствующие критериям *методов поиска* (Finder Method), можно искать во время выполнения, но такой подход может привести к появлению *неопределенных тестов* (Nondeterministic Test), так как при каждом запуске теста будут использоваться разные объекты/записи из *предварительно созданной тестовой конфигурации* (Prebuilt Fixture). Такая стратегия может оказаться оправданной, если при каждом запуске объекты модифицируются и больше не соответствуют критериям. Несмотря на это отладка неудачно завершившегося теста может оказаться достаточно сложной, особенно если неудачное завершение происходит время от времени из-за изменения того или иного атрибута выбранного объекта.

Мотивирующий пример

В следующем примере показано создание *общей тестовой конфигурации* (Shared Fixture) с помощью “ленивой” настройки (Lazy Setup). (Конечно, существуют и другие способы создания *общей тестовой конфигурации* (Shared Fixture), например с помощью декоратора настройки (Setup Decorator) и настройки тестовой конфигурации набора (Suite Fixture Setup).)

```
protected void setUp() throws Exception {
    if (sharedFixtureInitialized) {
        return;
    }
    facade = new FlightMgmtFacadeImpl();
    setupStandardAirportsAndFlights();
    sharedFixtureInitialized = true;
}
protected void tearDown() throws Exception {
    // Невозможно удалить объекты, так как
    // неизвестно, последний ли это тест
}
```

Обратите внимание на вызов `setupStandardAirports` в методе `setUp`. Тесты используют такую конфигурацию через *методы поиска* (Finder Method), возвращающие объекты конфигурации, соответствующие определенным критериям.

```
public void testGetFlightsByFromAirport_OneOutboundFlight()
    throws Exception {
    FlightDto outboundFlight = findOneOutboundFlight();
    // Вызов системы
```

```

List flightsAtOrigin = facade.getFlightsByOriginAirport(
    outboundFlight.getOriginAirportId());
// Проверка результата
assertOnly1FlightInDtoList("Flights at origin",
    outboundFlight,
    flightsAtOrigin);
}
public void testGetFlightsByFromAirport_TwoOutboundFlights()
throws Exception {
FlightDto[] outboundFlights =
findTwoOutboundFlightsFromOneAirport();
// Вызов системы
List flightsAtOrigin = facade.getFlightsByOriginAirport(
    outboundFlights[0].getOriginAirportId());
// Проверка результата
assertExactly2FlightsInDtoList("Flights at origin",
    outboundFlights,
    flightsAtOrigin);
}

```

Замечания по рефакторингу

Одним из способов преобразовать *класс теста* (Testcase Class) в направлении от использования *стандартной тестовой конфигурации* (Standard Fixture, с. 338) к использованию *предварительно созданной тестовой конфигурации* (Prebuilt Fixture) является применение рефакторинга выделение класса (Extract Class). В результате конфигурация будет создаваться в одном классе, а *тестовые методы* (Test Method, с. 378) будут находиться в другом. Конечно, *методы поиска* (Finder Method) должны иметь возможность определить, какие объекты или записи существуют в структуре, так как невозможно гарантировать, что любая переменная класса или экземпляра закроет разрыв между созданием тестовой конфигурации и ее использованием.

Пример: тест с использованием предварительно созданной тестовой конфигурации (Prebuilt Fixture Test)

Ниже показан полученный в результате рефакторинга *класс теста* (Testcase Class), содержащий *тестовые методы* (Test Method). Обратите внимание, что он практически совпадает с тестом, использующим *общую тестовую конфигурацию* (Shared Fixture).

```

public void testGetFlightsByFromAirport_OneOutboundFlight()
throws Exception {
FlightDto outboundFlight = findOneOutboundFlight();
// Вызов системы
List flightsAtOrigin = facade.getFlightsByOriginAirport(
    outboundFlight.getOriginAirportId());
// Проверка результата
assertOnly1FlightInDtoList("Flights at origin",
    outboundFlight,
    flightsAtOrigin);
}
public void testGetFlightsByFromAirport_TwoOutboundFlights()
throws Exception {
FlightDto[] outboundFlights =

```

```

        findTwoOutboundFlightsFromOneAirport();
        // Вызов системы
        List flightsAtOrigin = facade.getFlightsByOriginAirport(
                outboundFlights[0].getOriginAirportId());
        // Проверка результата
        assertExactly2FlightsInDtoList("Flights at origin",
                outboundFlights,
                flightsAtOrigin);
    }
}

```

Основное отличие заключается в настройке тестовой конфигурации и реализации *методов поиска* (Finder Method).

Пример: тест с настройкой тестовой конфигурации (Fixture Setup Testcase)

Иногда удобно создавать *предварительно созданную тестовую конфигурацию* (Prebuilt Fixture) средствами инфраструктуры xUnit. Это просто, если уже существуют подходящие *методы создания* (Creation Method, с. 441) или конструкторы и существует способ сохранения объектов в “*лесочнице*” с базой данных (Database Sandbox). В следующем примере из метода `setUp` вызывается тот же метод, что и в предыдущем примере, но на этот раз метод располагается в методе `setUp теста с настройкой тестовой конфигурации` (Fixture Setup Testcase). Этот тест запускается каждый раз, когда необходимо перегенерировать *предварительно созданную тестовую конфигурацию* (Prebuilt Fixture).

```

public class FlightManagementFacadeSetupTestcase
    extends AbstractFlightManagementFacadeTestCase {
    public FlightManagementFacadeSetupTestcase(String name) {
        super(name);
    }
    protected void setUp() throws Exception {
        facade = new FlightMgmtFacadeImpl();
        helper = new FlightManagementTestHelper();
        setupStandardAirportsAndFlights();
        saveFixtureInformation();
    }
    protected void tearDown() throws Exception {
        // Оставить конфигурацию для последующего использования
    }
}

```

Обратите внимание, что в этом *классе теста* (Testcase Class) отсутствуют *тестовые методы* (Test Method) и метод `tearDown` пуст. От класса требуется только создание конфигурации — и больше ничего.

После создания объектов информация сохраняется в базе данных с помощью вызова `saveFixtureInformation`. Этот метод сохраняет объекты и сохраняет различные ключи в файле для последующей загрузки и использования другими тестами. При таком подходе не приходится жестко описывать конфигурацию внутри *тестовых методов* (Test Method) или *вспомогательных методов теста* (Test Utility Method). Для экономии места здесь не описывается поиск “грязных” объектов и сохранение информации о ключах. Существует больше одного варианта решения этой проблемы, и любой из них подойдет.

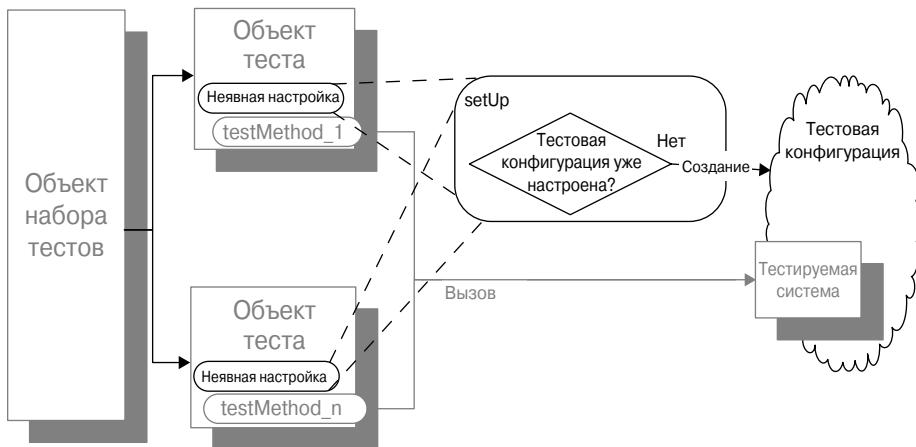
Пример: тестовая конфигурация, предварительно созданная с помощью сценария наполнения базы данных

Существует множество способов генерации *предварительно созданной тестовой конфигурации* (Prebuilt Fixture) с помощью сценариев в пределах “песочницы” с базой данных (Database Sandbox) — от сценария SQL до программ на языках Perl и Ruby. Сценарии могут содержать данные в своем коде или читать их из набора файлов. Можно даже скопировать содержимое “эталонной” базы данных в “песочницу” с базой данных (Database Sandbox). Выбор наиболее подходящего варианта остается за разработчиком.

“Ленивая” настройка (Lazy Setup)

Как создать общую тестовую конфигурацию (Shared Fixture) до запуска использующих ее тестовых методов?

Для этого используется “ленивая” инициализация конфигурации первым использующим ее тестом.



Общая тестовая конфигурация (Shared Fixture, с. 350) часто используется для ускорения работы тестов за счет экономии времени на создании конфигурации для каждого теста. К сожалению, тест, зависящий от других тестов для создания тестовой конфигурации, не может запускаться отдельно; это *одинокий тест* (Lonely Test; см. *Нестабильный тест*, Erratic Test, с. 267).

Этой проблемы можно избежать, заставив каждый тест использовать “ленивую” настройку (Lazy Setup) для создания отсутствующей конфигурации.

Как это работает

“Ленивая” инициализация (Lazy Initialization) [SBPP] предполагает создание тестовой конфигурации первым же тестом, которому она нужна. После этого ссылка на конфигурацию хранится в переменной класса, доступной каждому тесту. Все последующие тесты обнаруживают существующую конфигурацию и используют ее повторно, не пытаясь создавать ее заново.

Когда это использовать

“Ленивую” настройку (Lazy Setup) можно использовать каждый раз, когда необходимо создать общую тестовую конфигурацию (Shared Fixture), но каждый тест должен иметь возможность выполняться самостоятельно. Кроме того, “ленивая” настройка (Lazy Setup) используется вместо других способов генерации конфигурации, например вместо декоратора настройки (Setup Decorator, с. 471) или настройки тестовой конфигурации набора

(Suite Fixture Setup, с. 465), когда не требуется обязательная очистка тестовой конфигурации. Например, “ленивая” настройка (Lazy Setup) применяется, когда тестовая конфигурация удаляется с помощью очистки со сборкой мусора (Garbage-Collected Teardown, с. 518). Кроме того, “ленивая” настройка (Lazy Setup) может использоваться вместе с *отдельными сгенерированными значениями* (Distinct Generated Value) для всех ключей базы данных, когда никого не заботят оставшиеся записи после каждого теста. Также в подобной ситуации можно использовать *дельта-утверждения* (Delta Assertion, с. 505).

Основной недостаток “ленивой” настройки (Lazy Setup) связан со сложностью определения последнего теста, после которого должна происходить очистка конфигурации. В большинстве реализаций xUnit отсутствуют механизмы определения последнего теста (кроме использования *декоратора настройки* (Setup Decorator) для всего набора тестов). В некоторых реализациях поддерживается *настройка тестовой конфигурации набора* (Suite Fixture Setup), например в NUnit, VbUnit и JUnit 4.0, через методы `setUp/tearDown` для *класса теста* (Testcase Class, с. 402). К сожалению, эта возможность недоступна в тестах на языках Ruby, Python и PL/SQL.

В некоторых средах разработки и *программах запуска тестов* (Test Runner, с. 405) происходит автоматическая перезагрузка классов при каждом запуске набора тестов. В результате исходные переменные классов выходят за пределы видимости и тестовая конфигурация удаляется сборщиком мусора до запуска следующей версии класса. В таком случае “ленивая” настройка (Lazy Setup) может не иметь нежелательных последствий.

Предварительно созданная тестовая конфигурация (Prebuilt Fixture, с. 454) является еще одним вариантом создания общей тестовой конфигурации (Shared Fixture) перед запуском тестов. Ее применение может привести к появлению *неповторяемых тестов* (Unrepeatable Test), если конфигурация оказывается поврежденной одним из тестов.

Замечания по реализации

Поскольку “ленивая” настройка (Lazy Setup) имеет смысл только для *общих тестовых конфигураций* (Shared Fixture), с ней связаны те же трудности, что и с общей конфигурацией.

Обычно “ленивая” настройка (Lazy Setup) используется для создания *общей тестовой конфигурации* (Shared Fixture) для единственного *класса теста* (Testcase Class). Ссылка на конфигурацию хранится в переменной класса. Все становится сложнее, если конфигурацию приходится использовать в нескольких *классах теста* (Testcase Class). Можно вынести логику “ленивой” инициализации и переменную класса в *суперкласс теста* (Testcase Superclass, с. 646), но не во всех языках поддерживается наследование переменных класса. Еще одним решением является выделение логики и переменных во *вспомогательный класс теста* (Test Helper, с. 651).

Конечно, для определения, все ли *тестовые методы* (Test Method, с. 378) вызывались, можно воспользоваться подсчетом ссылок или аналогичным способом. Останется только узнать, сколько *объектов теста* (Testcase Object, с. 410) хранится в *объекте набора тестов* (Test Suite Object, с. 414), чтобы сравнить это количество с количеством вызовов метода `tearDown`. Автору реализация такого решения не встречалась, поэтому назвать его шаблоном нельзя! Добавление в *программу запуска тестов* (Test Runner) логики вызова метода `tearDown` на уровне *объекта набора тестов* (Test Suite Object) будет шагом в сторону реализации *настройки тестовой конфигурации набора* (Suite Fixture Setup).

Мотивирующий пример

В этом примере создается новая конфигурация для каждого *объекта теста* (Testcase Object).

```
public void testGetFlightsByFromAirport_OneOutboundFlight()
    throws Exception {
    setupStandardAirportsAndFlights();
    FlightDto outboundFlight = findOneOutboundFlight();
    // Вызов тестируемой системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlight.getOriginAirportId());
    // Проверка результата
    assertOnly1FlightInDtoList("Flights at origin",
        outboundFlight,
        flightsAtOrigin);
}
public void testGetFlightsByFromAirport_TwoOutboundFlights()
    throws Exception {
    setupStandardAirportsAndFlights();
    FlightDto[] outboundFlights =
        findTwoOutboundFlightsFromOneAirport();
    // Вызов тестируемой системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlights[0].getOriginAirportId());
    // Проверка результата
    assertExactly2FlightsInDtoList("Flights at origin",
        outboundFlights,
        flightsAtOrigin);
}
```

Нет ничего удивительного в низкой скорости работы этих тестов, так как создание аэропортов и перелетов требует обращения к базе данных. Можно попытаться переработать тесты для настройки конфигурации в методе `setUp` (см. *Неявная настройка*, Implicit Setup, с. 449).

```
protected void setUp() throws Exception {
    facade = new FlightMgmtFacadeImpl();
    helper = new FlightManagementTestHelper();
    setupStandardAirportsAndFlights();
    oneOutboundFlight = findOneOutboundFlight();
}

protected void tearDown() throws Exception {
    removeStandardAirportsAndFlights();
}
public void testGetFlightsByOriginAirport_NoFlights_td()
    throws Exception {
    // Настройка конфигурации
    BigDecimal outboundAirport = createTestAirport("1OF");
    try {
        // Вызов тестируемой системы
        List flightsAtDestination1 =
            facade.getFlightsByOriginAirport(outboundAirport);
        // Проверка результата
        assertEquals(0, flightsAtDestination1.size());
    }
```

```

    } finally {
        facade.removeAirport(outboundAirport);
    }
}
public void testGetFlightsByFromAirport_OneOutboundFlight()
    throws Exception {
    // Вызов тестируемой системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        oneOutboundFlight.getOriginAirportId());
    // Проверка результата
    assertOnly1FlightInDtoList("Flights at origin",
        oneOutboundFlight,
        flightsAtOrigin);
}
public void testGetFlightsByFromAirport_TwoOutboundFlights()
    throws Exception {
    FlightDto[] outboundFlights =
        findTwoOutboundFlightsFromOneAirport();
    // Вызов тестируемой системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlights[0].getOriginAirportId());
    // Проверка результата
    assertExactly2FlightsInDtoList("Flights at origin",
        outboundFlights,
        flightsAtOrigin);
}
}

```

Это не приведет к ускоренному выполнению тестов, так как *инфраструктура автоматизации тестов* (Test Automation Framework) вызывает методы `setUp` и `tearDown` для каждого *объекта теста* (Testcase Object). Код настройки был просто перемещен в другое место. Необходимо найти способ однократного создания тестовой конфигурации для всех тестов.

Замечания по рефакторингу

Можно сократить количество запусков настройки конфигурации, преобразовав тесты для использования “ленивой” настройки (Lazy Setup). Поскольку настройка тестовой конфигурации уже обрабатывается методом `setUp`, достаточно вставить в этот метод логику “ленивой” инициализации, которая будет выполняться только при первом обращении теста. Не забывайте удалить очистку из метода `tearDown`, так как “ленивая” инициализация оказывается бесполезной, если конфигурация удаляется после завершения работы каждого *тестового метода* (Test Method)! К сожалению, если реализация xUnit не поддерживает *настройку тестовой конфигурации набора* (Suite Fixture Setup), логику очистки придется просто удалить.

Пример: “ленивая” настройка (Lazy Setup)

Ниже приведен пример теста, преобразованного для использования “ленивой” настройки (Lazy Setup).

```

protected void setUp() throws Exception {
    if (sharedFixtureInitialized) {
        return;
    }
}

```

```

    }
    facade = new FlightMgmtFacadeImpl();
    setupStandardAirportsAndFlights();
    sharedFixtureInitialized = true;
}
protected void tearDown() throws Exception {
    // Нельзя удалять объекты, так как неизвестно,
    // последний ли это тест
}

```

Хотя метод `tearDown` в классе `AirportFixture` существует, неизвестно, когда его вызывать! Это основное следствие использования “ленивой” настройки (Lazy Setup). Так как используются статические переменные, они не выйдут из области видимости. В результате конфигурация не будет удалена сборщиком мусора, пока класс не будет выгружен из памяти и загружен заново.

Тесты ничем не отличаются от варианта для *неявной настройки* (Implicit Setup).

```

public void testGetFlightsByFromAirport_OneOutboundFlight()
    throws Exception {
    FlightDto outboundFlight = findOneOutboundFlight();
    // Вызов тестируемой системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlight.getOriginAirportId());
    // Проверка результата
    assertOnly1FlightInDtoList("Flights at origin",
        outboundFlight,
        flightsAtOrigin);
}

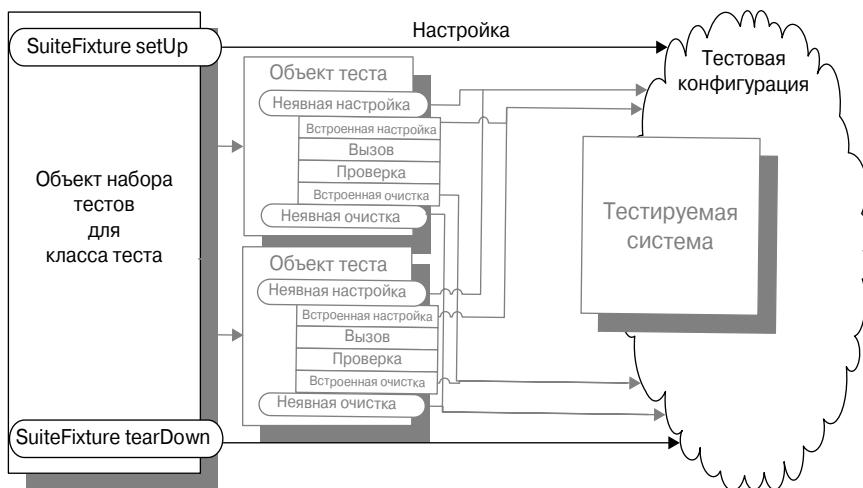
public void testGetFlightsByFromAirport_TwoOutboundFlights()
    throws Exception {
    FlightDto[] outboundFlights =
        findTwoOutboundFlightsFromOneAirport();
    // Вызов тестируемой системы
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        outboundFlights[0].getOriginAirportId());
    // Проверка результата
    assertExactly2FlightsInDtoList("Flights at origin",
        outboundFlights,
        flightsAtOrigin);
}

```

Настройка тестовой конфигурации набора (Suite Fixture Setup)

Как создать общую тестовую конфигурацию (Shared Fixture) до запуска использующих ее тестовых методов?

Общая тестовая конфигурация создается и/или удаляется в специальных методах, вызываемых инфраструктурой автоматизации тестов (Test Automation Framework) перед запуском первого и после завершения работы последнего теста.



Общая тестовая конфигурация (Shared Fixture, с. 350) часто используется для экономии времени за счет отказа от создания конфигурации для каждого теста в отдельности. Совместное использование тестовой конфигурации требует от разработчика дополнительных усилий, так как приходится создавать конфигурацию и обеспечивать методы ее поиска каждым тестом. Независимо от способа доступа конфигурацию необходимо создать до начала использования.

Настройка тестовой конфигурации набора (Suite Fixture Setup) является одним из способов инициализации конфигурации, если все использующие ее *тестовые методы* (Test Method, с. 378) определены в одном *классе теста* (Testcase Class, с. 401).

Как это работает

Инфраструктура автоматизации тестов (Test Automation Framework, с. 332) вызывает пару методов, реализованных или переопределенных разработчиком тестов. Название или аннотация методов зависит от используемой реализации xUnit, но механизм везде работает одинаково: инфраструктура вызывает метод *настройки тестовой конфигурации набора* (Suite Fixture Setup) перед вызовом метода `setUp` для первого *тестового метода* (Test Method). Метод *очистки тестовой конфигурации набора* (Suite Fixture Teardown) вызывается после вызова метода `tearDown` для последнего *тестового метода* (Test Method). (Хотелось бы написать, что методы вызываются для первого и/или последнего

объекта теста (Testcase Object), но это не так: пакет NUnit создает единственный *объект теста* (Testcase Object). Дополнительная информация по этой теме приводится во врезке “Всегда есть исключения” на с. 411.)

Когда это использовать

Настройка тестовой конфигурации набора (Suite Fixture Setup) может применяться для создания тестовой конфигурации, которая будет совместно использоваться всеми *тестовыми методами* (Test Method) одного *класса теста* (Testcase Class), если используемая реализация xUnit поддерживает данную функцию. Данный шаблон особенно полезен, если необходимо удалять конфигурацию после завершения работы последнего теста. Во время написания этой книги только пакеты VbUnit, NUnit и JUnit 4.0 поддерживали функцию *настройки тестовой конфигурации набора* (Suite Fixture Setup). Но нет ничего сложного в добавлении этой функции в большинство реализаций xUnit.

Если тестовая конфигурация должна использоваться более широко, придется вернуться к применению *предварительно созданной тестовой конфигурации* (Prebuilt Fixture, с. 454), *декоратора настройки* (Setup Decorator, с. 471) или “ленивой” настройки (Lazy Setup, с. 460). Если совместно должен использоваться не экземпляр тестовой конфигурации, а код ее настройки, можно обратиться к *неявной настройке* (Implicit Setup, с. 449) или *делегированной настройке* (Delegated Setup, с. 437).

Основной причиной использования *общей тестовой конфигурации* (Shared Fixture) и *настройки тестовой конфигурации набора* (Suite Fixture Setup) является попытка преодолеть проблему *медленных тестов* (Slow Tests, с. 289), вызванную большим количеством объектов, создаваемых для каждого теста. Конечно, использование *общей тестовой конфигурации* (Shared Fixture) может привести к появлению *взаимодействующих тестов* (Interacting Tests) или даже к “войне” запуска тестов (Test Run War). Другие варианты решения этой проблемы рассматриваются во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351.

Замечания по реализации

Для нормальной работы логики *настройки тестовой конфигурации набора* (Suite Fixture Setup) необходимо обеспечить сохранение тестовой конфигурации между вызовами *тестовых методов* (Test Method). Этот критерий неявно предполагает использование переменной класса, объекта Registry [PEAA] или объекта Singleton [GOF] для хранения ссылки на конфигурацию (кроме пакета NUnit, особенности которого рассматриваются во врезке “Всегда есть исключения” на с. 411). Конкретная реализация зависит от используемого пакета xUnit. Ниже перечислены возможные варианты реализации.

- В пакете VbUnit в *классе теста* (Testcase Class) реализуется интерфейс *IFixtureFrame*, заставляя *инфраструктуру автоматизации тестов* (Test Automation Framework) вызывать метод *IFixtureFrame_Create* перед вызовом первого *тестового метода* (Test Method) и метод *IFixtureFrame_Destroy* после завершения работы последнего тестового метода.
- В пакете NUnit атрибуты *[TestFixtureSetUp]* и *[TestFixtureTearDown]* используются внутри тестовой конфигурации для перечисления вызываемых ме-

тодов. Эти методы вызываются перед первым тестом в пределах конфигурации и после завершения работы всех тестов.

- В JUnit 4.0 и в более поздних версиях атрибут @BeforeClass используется для идентификации метода, вызываемого перед первым *тестовым методом* (Test Method). Метод с атрибутом @AfterClass вызывается после завершения работы последнего тестового метода. Пакет JUnit допускает наследование и переопределение этих методов. Методы подкласса вызываются между выполнением методов суперкласса.

Не стоит переносить всю логику создания конфигурации в механизм *настройки тестовой конфигурации набора* (Suite Fixture Setup) только потому, что для создания и очистки тестовой конфигурации используется форма *неявной настройки* (Implicit Setup). *Методы создания* (Creation Method, с. 441) могут содержать сложную логику настройки и вызываться из метода *настройки тестовой конфигурации набора* (Suite Fixture Setup). Таким образом, сложная логика может выноситься в компоненты, которые проще проверять и использовать повторно, например в *суперкласс теста* (Testcase Superclass, с. 646) или во *вспомогательный класс теста* (Test Helper, с. 651).

Мотивирующий пример

Предположим, что существует следующий тест.

```
[SetUp]
protected void setUp() {
    helper.setupStandardAirportsAndFlights();
}

[TearDown]
protected void tearDown() {
    helper.removeStandardAirportsAndFlights();
}

[Test]
public void testGetFlightsByOriginAirport_2OutboundFlights() {
    FlightDto[] expectedFlights =
        helper.findTwoOutboundFlightsFromOneAirport();
    long originAirportId = expectedFlights[0].OriginAirportId;
    // Вызов тестируемой системы
    IList flightsAtOrigin =
        facade.GetFlightsByOriginAirport(originAirportId);
    // Проверка результата
    AssertExactly2FlightsInDtoList(
        expectedFlights[0], expectedFlights[1],
        flightsAtOrigin, "Flights at origin");
}

[Test]
public void testGetFlightsByOriginAirport_OneOutboundFlight() {
    FlightDto expectedFlight = helper.findOneOutboundFlight();
    // Вызов тестируемой системы
    IList flightsAtOrigin = facade.GetFlightsByOriginAirport(
        expectedFlight.OriginAirportId);
    // Проверка результата
    AssertOnly1FlightInDtoList( expectedFlight,
        flightsAtOrigin, "Outbound flight at origin");
}
```

На рис. 20.1 показан вывод модифицированной версии этих тестов.

```
-----
setUp
  setupStandardAirportsAndFlights
testGetFlightsByOriginAirport_OneOutboundFlight
tearDown
  removeStandardAirportsAndFlights
-----
setUp
  setupStandardAirportsAndFlights
testGetFlightsByOriginAirport_TwoOutboundFlights
tearDown
  removeStandardAirportsAndFlights
-----
```

Рис. 20.1. Последовательность вызовов тестовых методов (Test Method) при неявной настройке (Implicit Setup). Метод `setupStandardAirportsAndFlights` вызывается перед каждым тестовым методом. Горизонтальные линии показывают границы тестовых методов

Замечания по рефакторингу

Предположим, этот пример необходимо изменить для использования *общей тестовой конфигурации* (Shared Fixture). Если удаление конфигурации после завершения работы тестов не требуется, можно воспользоваться “ленивой” настройкой (Lazy Setup). В противном случае этот пример можно преобразовать с использованием стратегии *настройки тестовой конфигурации набора* (Suite Fixture Setup), вынеся код из методов `setUp` и `tearDown` в методы `suiteFixtureSetUp` и `suiteFixtureTearDown` соответственно.

В пакете NUnit с целью идентификации таких методов для *инфраструктуры автоматизации тестов* (Test Automation Framework) используются атрибуты `[TestFixtureSetUp]` и `[TestFixtureTearDown]`. Если методы `setUp/tearDown` остаются пустыми, атрибуты `[Setup]` и `[TearDown]` можно просто заменить атрибутами `[TestFixtureSetUp]` и `[TestFixtureTearDown]` соответственно.

Пример: настройка тестовой конфигурации набора (Suite Fixture Setup)

Ниже приведен результат рефакторинга в направлении *настройки тестовой конфигурации набора* (Suite Fixture Setup).

```
[TestFixtureSetUp]
protected void suiteFixtureSetUp()
{
    helper.setupStandardAirportsAndFlights();
}
[TestFixtureTearDown]
protected void suiteFixtureTearDown() {
    helper.removeStandardAirportsAndFlights();
}
[SetUp]
protected void setUp() {
}
[TearDown]
```

```

protected void tearDown() {
}
[Test]
public void testGetFlightsByOrigin_TwoOutboundFlights() {
    FlightDto[] expectedFlights =
        helper.findTwoOutboundFlightsFromOneAirport();
    long originAirportId = expectedFlights[0].OriginAirportId;
    // Вызов тестируемой системы
    IList flightsAtOrigin =
        facade.GetFlightsByOriginAirport(originAirportId);
    // Проверка результата
    AssertExactly2FlightsInDtoList(
        expectedFlights[0], expectedFlights[1],
        flightsAtOrigin, "Flights at origin");
}

[Test]
public void testGetFlightsByOrigin_OneOutboundFlight() {
    FlightDto expectedFlight = helper.findOneOutboundFlight();
    // Вызов тестируемой системы
    IList flightsAtOrigin = facade.GetFlightsByOriginAirport(
        expectedFlight.OriginAirportId);
    // Проверка результата
    AssertOnly1FlightInDtoList(expectedFlight,
        flightsAtOrigin, "Outbound flight at origin");
}

```

Теперь, после вызова различных методов *класса теста* (Testcase Class), вывод будет выглядеть следующим образом (рис. 20.2).

```

suiteFixtureSetUp
    setupStandardAirportsAndFlights
-----
setUp
testGetFlightsByOriginAirport_OneOutboundFlight
tearDown
-----
setUp
testGetFlightsByOriginAirport_TwoOutboundFlights
tearDown
-----
suiteFixtureTearDown
removeStandardAirportsAndFlights

```

Рис. 20.2. Последовательность вызова тестовых методов при настройке тестовой конфигурации набора (Suite Fixture Setup). Метод setupStandardAndAirportsAndFlights вызывается один раз для всего класса теста (Testcase Class), а не перед каждым тестовым методом (Test Method). Горизонтальные линии показывают границы тестовых методов

Метод setUp все еще вызывается перед каждым *тестовым методом* (Test Method). Но при этом вызывается и метод suiteFixtureSetUp, в котором вызывается метод setupStandardAirportsAndFlights для создания тестовой конфигурации. Данная конфигурация практически не отличается от использования “ленивой” настройки (Lazy Setup).

Единственным отличием является вызов `removeStandardAirportsAndFlights` после всех тестовых методов для очистки конфигурации.

Выбор названия

Выбор названия для рассматриваемого шаблона оказался сложной процедурой, поскольку в различных реализациях xUnit он имеет различные названия. Еще более усложняет ситуацию вкладывание большего смысла в понятие “тестовая конфигурация” среди сторонников продуктов компании Microsoft по сравнению со сторонниками Java/Pearl/Ruby и т.д. Выбор названия *настройка тестовой конфигурации набора* (Suite Fixture Setup) основан на области применения *общей тестовой конфигурации* (Shared Fixture). Конфигурация совместно используется тестами в пределах набора из единственного класса *теста* (Testcase Class), порождающего единственный объект набора тестов (Test Suite Object, с. 414). Конфигурация, созданная для *объекта набора тестов* (Test Suite Object), может называться тестовой конфигурацией набора.

Источники дополнительной информации

Дополнительная информация о реализации *настройки тестовой конфигурации набора* (Suite Fixture Setup) в пакете VbUnit приводится по адресу:

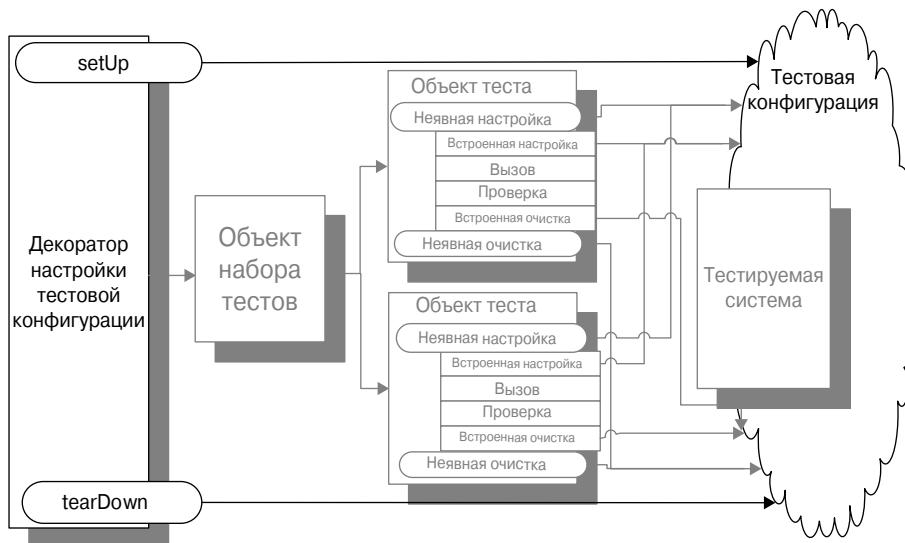
<http://www.vbunit.com/doc/Advanced.htm>

Дополнительная информация о реализации этого шаблона в пакете NUnit приводится на сайте nunit.org.

Декоратор настройки (Setup Decorator)

Как создать общую тестовую конфигурацию (Shared Fixture) до запуска использующих ее тестовых методов?

Набор тестов заключается в декоратор, который настраивает общую тестовую конфигурацию до вызова тестовых методов и удаляет ее после завершения работы последнего теста.



Если принято решение использовать *общую тестовую конфигурацию* (Shared Fixture, с. 350), но по каким-то причинам *предварительно созданная тестовая конфигурация* (Prebuilt Fixture, с. 454) не подходит, необходимо обеспечить создание конфигурации перед вызовом первого тестового метода. Одной из возможных стратегий является использование *“ленивой” настройки* (Lazy Setup), обеспечивающей создание конфигурации как раз перед ее использованием. Но что если удаление конфигурации после завершения работы последнего теста является обязательным требованием? Как узнать, что все тесты завершили работу?

Как это работает

Декоратор настройки (Setup Decorator) “огораживает” выполнение всего набора тестов парой методов `setUp` и `tearDown`. Для построения такого решения используется шаблон Decorator. Созданный *декоратор настройки* (Setup Decorator) содержит ссылку на необходимый *объект набора тестов* (Test Suite Object, с. 414), а сам передается в *программу запуска тестов* (Test Runner, с. 405). Когда приходит время запускать тесты, *программа запуска тестов* (Test Runner) вызывает метод `run` *декоратора настройки* (Setup Decorator), а не фактического *объекта набора тестов* (Test Suite Object). *Декоратор настройки* (Setup Decorator) выполняет создание конфигурации и только после этого вызы-

вает метод `xrun` объекта набора тестов (Test Suite Object). После завершения работы метода `xrun` выполняется удаление созданной тестовой конфигурации.

Когда это использовать

Декоратор настройки (Setup Decorator) можно использовать в ситуациях, когда *общая тестовая конфигурация* (Shared Fixture) должна создаваться перед каждым запуском набора тестов и удаляться после завершения их работы. Необходимость такого поведения может потребоваться для тестов, использующих *фиксированные значения* (Hard-Coded Value), которые приводят к неудачному завершению тестов при повторном запуске без очистки конфигурации (см. *Неповторяемый тест*, Unrepeatable Test). Такое поведение может потребоваться и для защиты от переполнения ограниченного ресурса, например от разрастания базы данных в результате повторных запусков теста.

Кроме того, *декоратор настройки* (Setup Decorator) может использоваться для модификации глобальных параметров перед вызовом тестируемой системы и восстановления значения параметра после завершения работы. Одним из примеров такой ситуации может служить замена базы данных *поддельной базой данных* (Fake Database) в попытке избежать появления *медленных тестов* (Slow Tests, с. 289). В качестве еще одного примера можно назвать установку глобальных параметров в соответствии с конкретной конфигурацией. *Декоратор настройки* (Setup Decorator) устанавливается во время выполнения, поэтому ничто не мешает использовать несколько различных декораторов для одного набора тестов в разные моменты времени (или даже одновременно).

Если необходимо только совместное использование конфигурации тестами набора в пределах одного *класса теста* (Testcase Class, с. 401), в качестве альтернативы *декоратору настройки* (Setup Decorator) можно использовать *настройку тестовой конфигурации набора* (Suite Fixture Setup, с. 465) (если используемая реализация xUnit поддерживает такое поведение). Если удаление тестовой конфигурации после каждого запуска не требуется, можно воспользоваться “ленивой” *настройкой* (Lazy Setup).

Замечания по реализации

Декоратор настройки (Setup Decorator) состоит из объекта, который создает конфигурацию, делегирует выполнение запускаемому набору тестов и вызывает код очистки конфигурации. Для поддержки единообразия имен код создания конфигурации размещается в методе `setUp`, а код очистки конфигурации — в методе `tearDown`. В таком случае метод `xrun` декоратора *настройки* (Setup Decorator) будет выглядеть следующим образом.

```
void run() {
    setup();
    decoratedSuite.run();
    teardown();
}
```

Существует несколько способов создания *декоратора настройки* (Setup Decorator).

Вариант: абстрактный декоратор настройки (Abstract Setup Decorator)

Во многих реализациях xUnit предоставляется суперкласс, реализующий *декоратор настройки* (Setup Decorator). Обычно в этом классе методы `setUp/run/tearDown` реа-

лизованы в виде Template Method. Разработчику остается наследовать этот класс и реализовать методы `setUp` и `tearDown`, как в обычном *классе теста* (Testcase Class). При создании экземпляра класса *декоратора настройки* (Setup Decorator) декорируемый *объект набора тестов* (Test Suite Object) передается в качестве аргумента конструктора.

Вариант: фиксированный декоратор настройки (Hard-Coded Setup Decorator)

Если реализация *декоратора настройки* (Setup Decorator) создается “с нуля”, “простейшим работающим решением” является фиксация имени интересующего класса в коде метода `suite` *декоратора настройки* (Setup Decorator). В результате *декоратор настройки* (Setup Decorator) может выступать в роли *фабрики наборов тестов* (Test Suite Factory) для получения необходимого набора.

Вариант: параметризованный декоратор настройки (Parameterized Setup Decorator)

Если *декоратор настройки* (Setup Decorator) должен использоваться для нескольких наборов, его конструктор можно параметризовать запускаемым *объектом набора тестов* (Test Suite Object). Это значит, что логика настройки и очистки может быть реализована внутри *декоратора настройки* (Setup Decorator) и не требовать отдельного *вспомогательного класса теста* (Test Helper, с. 651) только для повторного использования логики настройки в разных тестах.

Вариант: “ленивая” настройка с декоратором (Decorated Lazy Setup)

Одним из недостатков использования *декоратора настройки* (Setup Decorator) является невозможность запуска тестов по отдельности, так как они зависят от *декоратора настройки* (Setup Decorator), создающего тестовую конфигурацию. Это требование можно обойти, дополнив декоратор “ленивой” *настройкой* (Lazy Setup) в методе `setUp`, чтобы запущенный самостоятельно *объект теста* (Testcase Object, с. 410) мог создать необходимую конфигурацию. Кроме того, *объект теста* (Testcase Object) может помнить о создании конфигурации и удалять ее в методе `tearDown`. Такая функциональность может быть реализована в *суперклассе теста* (Testcase Superclass, с. 646). В таком случае писать и тестировать реализацию придется один раз.

Единственной альтернативой является использование *декоратора нижнего уровня* (Pushdown Decorator), но это полностью отменяет ускорение работы тестов за счет использования *общей тестовой конфигурации* (Shared Fixture). Поэтому данный *декоратор нижнего уровня* (Pushdown Decorator) имеет смысл использовать только в случаях, когда использование *декоратора настройки* (Setup Decorator) не было продиктовано необходимости создавать *общую тестовую конфигурацию* (Shared Fixture).

Вариант: декоратор нижнего уровня (Pushdown Decorator)

Одним из основных недостатков использования *декоратора настройки* (Setup Decorator) является невозможность самостоятельного запуска тестов, так как они зависят от декоратора, создающего тестовую конфигурацию. Одним из способов обойти это ограничение является смещение декоратора на уровень отдельных тестов с уровня целого набора. Такое решение требует нескольких модификаций класса набора тестов, позволяющих передавать *декоратор настройки* (Setup Decorator) в механизм создания *объектов*

теста (Testcase Object), работающий в процессе *обнаружения тестов* (Test Discovery, с. 420). Так как каждый объект создается на основе *тестового метода* (Test Method, с. 378), он заключается в *декоратор настройки* (Setup Decorator) перед добавлением в коллекцию *объекта набора тестов* (Test Suite Object).

Конечно, это отменяет один из основных источников ускорения, созданных с помощью *декоратора настройки* (Setup Decorator), так как тестовая конфигурация будет создаваться для каждого теста. Другие способы ускорения работы тестов приводятся во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351.

Мотивирующий пример

В приведенном ниже примере набор тестов использует “*ленивую*” *настройку* (Lazy Setup) для создания *общей тестовой конфигурации* (Shared Fixture) и *методы поиска* (Finder Method) для получения объектов конфигурации. Проверка показала, что остатки конфигурации приводят к появлению *неповторяемых тестов* (Unrepeatable Test), поэтому после запуска последнего теста конфигурацию необходимо удалить.

```
protected void setUp() throws Exception {
    if (sharedFixtureInitialized) {
        return;
    }
    facade = new FlightMgmtFacadeImpl();
    setupStandardAirportsAndFlights();
    sharedFixtureInitialized = true;
}
protected void tearDown() throws Exception {
    // Невозможно удалить объекты, так как неизвестно,
    // последний ли это тест
}
```

Поскольку не существует простого способа решить эту задачу в пределах “*ленивой*” *настройки* (Lazy Setup), необходимо сменить стратегию настройки тестовой конфигурации. Одним из вариантов является использование *декоратора настройки* (Setup Decorator).

Замечания по рефакторингу

При создании *декоратора настройки* (Setup Decorator) можно повторно использовать уже имеющуюся логику создания конфигурации. Просто она вызывается в другой момент. Таким образом, рефакторинг, в основном, состоит из перемещения вызова логики создания конфигурации из метода *setUp класса теста* (Testcase Class) в метод *setUp* класса *декоратора настройки* (Setup Decorator). Предположим, что можно наследовать доступный *абстрактный декоратор настройки* (Abstract Setup Decorator). Создадим новый подкласс и конкретную реализацию методов *setUp* и *tearDown*.

Если используемая реализация xUnit не обеспечивает непосредственную поддержку *декоратора настройки* (Setup Decorator), можно создать собственный суперкласс и воспользоваться параметром конструктора и переменной экземпляра для хранения запускаемого набора тестов. После этого с помощью рефакторинга Extract Superclass [Ref] можно создать суперкласс для повторного использования.

Пример: фиксированный декоратор настройки (Hard-Coded Setup Decorator)

В этом примере вся логика создания конфигурации вынесена в метод `setUp` *декоратора настройки* (Setup Decorator), который наследует базовую функциональность от *абстрактного декоратора настройки* (Abstract Setup Decorator). Логика очистки тестовой конфигурации вписана в метод `tearDown`.

```
public class FlightManagementTestSetup extends TestSetup {
    private FlightManagementTestHelper helper;
    public FlightManagementTestSetup() {
        // Создать объект набора тестов (Test Suite Object)
        // и передать его в суперкласс
        // абстрактного декоратора настройки (Abstract Setup Decorator)
        super( SafeFlightManagementFacadeTest.suite() );
        helper = new FlightManagementTestHelper();
    }
    public void setUp() throws Exception {
        helper.setupStandardAirportsAndFlights();
    }
    public void tearDown() throws Exception {
        helper.removeStandardAirportsAndFlights();
    }
    public static Test suite() {
        // Вернуть экземпляр этого класса декоратора
        return new FlightManagementTestSetup();
    }
}
```

Так как это *фиксированный декоратор настройки* (Hard-Coded Setup Decorator), вызов *фабрики наборов тестов* (Test Suite Factory) для создания *объекта набора тестов* (Test Suite Object) зафиксирован в конструкторе. Метод `suite` просто вызывает конструктор.

Пример: параметризованный декоратор настройки (Parameterized Setup Decorator)

Для обеспечения повторного использования *декоратора настройки* (Setup Decorator) разными наборами тестов, необходимо воспользоваться рефакторингом *введение параметра* (Introduce parameter) по отношению к вызову *фабрики наборов тестов* (Test Suite Factory) внутри конструктора.

```
public class ParameterizedFlightManagementTestSetup extends TestSetup {
    private FlightManagementTestHelper helper =
        new FlightManagementTestHelper();
    public ParameterizedFlightManagementTestSetup(
        Test testSuiteToDecorate) {
        super(testSuiteToDecorate);
    }
    public void setUp() throws Exception {
        helper.setupStandardAirportsAndFlights();
    }
    public void tearDown() throws Exception {
        helper.removeStandardAirportsAndFlights();
    }
}
```

Для более простого создания набора тестов средствами *программы запуска тестов* (Test Runner) необходимо создать *фабрику наборов тестов* (Test Suite Factory), которая вызывает конструктор *декоратора настройки* (Setup Decorator) и передает ему *объект набора тестов* (Test Suite Object) в качестве параметра.

```
public class DecoratedFlightManagementFacadeTestFactory {
    public static Test suite() {
        // Вернуть новый объект набора тестов внутри декоратора
        return new ParameterizedFlightManagementTestSetup(
            SafeFlightManagementFacadeTest.suite());
    }
}
```

Для каждого набора тестов потребуется одна такая *фабрика наборов тестов* (Test Suite Factory). Но даже это — небольшая цена за повторное использование *декоратора настройки* (Setup Decorator).

Пример: абстрактный класс декоратора (Abstract Decorator Class)

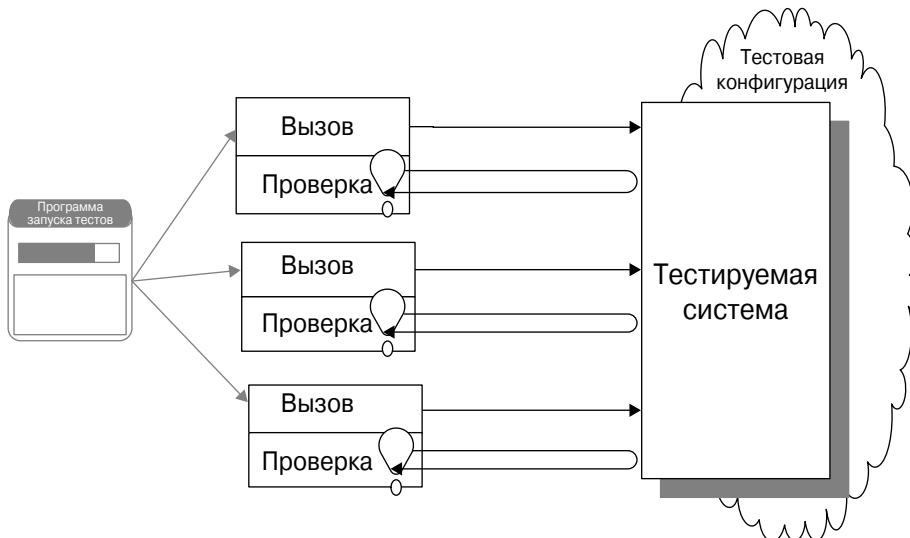
Ниже показан код абстрактного класса декоратора.

```
public class TestSetup extends TestCase {
    Test decoratedSuite;
    AbstractSetupDecorator(Test testSuiteToDecorate) {
        decoratedSuite = testSuiteToDecorate;
    }
    public void setUp() throws Exception {
        // Ответственность подкласса
    }
    public void tearDown() throws Exception {
        // Ответственность подкласса
    }
    void run() {
        setup();
        decoratedSuite.run();
        teardown();
    }
}
```

Цепочки тестов (Chained Tests)

Как создать общую тестовую конфигурацию (Shared Fixture) до запуска использующих ее тестовых методов?

Предыдущие тесты в наборе создают конфигурацию для последующих.



Общая тестовая конфигурация (Shared Fixture, с. 350) часто используется для сокращения накладных расходов на создание тестовой конфигурации. Совместное использование конфигурации требует дополнительных усилий при разработке тестов, так как приходится создавать конфигурацию и получать к ней доступ из каждого теста. Независимо от способа доступа к конфигурации она должна инициализироваться (создаваться) до первого использования.

Цепочки тестов (Chained Tests) позволяют повторно использовать тестовую конфигурацию, оставшуюся после работы предыдущего теста.

Как это работает

*Цепочки тестов (Chained Tests) используют созданные предыдущими тестами объекты. Такой подход похож на действия человека при проверке большого количества условий в пределах одного теста — создание сложной тестовой конфигурации с помощью последовательности действий и проверка результата каждого действия перед переходом к следующему. Такой же результат можно получить при использовании автоматизированных тестов, создав набор *самопроверяющихся тестов* (Self-Checking Test, с. 81), не создающих тестовую конфигурацию, а зависящих от “остатков” тестов, запущенных ранее. В отличие от шаблона *повторное использование теста для настройки тестовой конфигурации* (Reuse Test for Fixture Setup), в данном случае другие *тестовые методы* (Test Method, с. 378) из теста не вызываются. Просто делается предположение,*

что тесты были запущены и оставили после себя то, что можно использовать в качестве тестовой конфигурации.

Когда это использовать

От любви до ненависти к шаблону *цепочки тестов* (Chained Tests) один шаг. Ненавидящие этот шаблон понимают, что это просто запах *взаимодействующие тесты* (Interacting Tests; см. *Нестабильный тест*, Erratic Test, с. 267), реализованный в виде шаблона. Его сторонники предпочитают этот шаблон из-за возможности решить неприятную проблему, возникающую из-за использования *общей тестовой конфигурации* (Shared Fixture) для борьбы с *медленными тестами* (Slow Tests, с. 289). В любом случае это допустимая стратегия для рефакторинга существующих слишком длинных тестов, состоящих из последовательности зависящих друг от друга операций. Такие тесты завершают работу на первом же ложном утверждении. Преобразование подобных тестов в *цепочки тестов* (Chained Tests) происходит достаточно быстро, так как не требуется определение тестовой конфигурации для каждого теста. Это первый этап развития в сторону *независимых тестов* (Independent Tests, с. 96).

Цепочки тестов (Chained Tests) позволяют предотвратить появление “хрупких” тестов (Fragile Test, с. 277), так как они представляют собой грубую форму *инкапсуляции программного интерфейса тестируемой системы* (SUT API Encapsulation). Таким тестам не нужно взаимодействие с тестируемой системой для создания конфигурации, так как другие тесты уже воспользовались этим программным интерфейсом для ее создания. Но может возникнуть проблема “хрупкой” тестовой конфигурации (Fragile Fixture), если один из предыдущих тестов модифицирован для создания другой конфигурации. В таком случае последующие тесты, скорее всего, завершатся неудачно. Эта же проблема возникнет, если один из предыдущих тестов завершился неудачно или с ошибкой. При этом тест может оставить после себя *общую тестовую конфигурацию* (Shared Fixture) не в том состоянии, в котором ее ожидают получить последующие тесты.

Одной из ключевых проблем *цепочки тестов* (Chained Tests) является неопределенность порядка запуска тестов в пределах набора. В большинстве реализаций xUnit не дается никаких гарантий относительно порядка (исключением является пакет TestNG). Таким образом, тесты могут завершаться неудачно при установке новой версии xUnit или даже при переименовании одного из *тестовых методов* (Test Method) (если конкретная реализация xUnit сортирует *объекты теста* (Testcase Object, с. 410) по имени метода).

Еще одна проблема заключается в том, что тесты в цепочке являются *одинокими тестами* (Lonely Test), так как текущий тест зависит от предыдущих в создании конфигурации. Если тесты запускать по отдельности, они завершатся неудачно, поскольку предполагаемая тестовая конфигурация не будет создана. В результате при диагностике конкретной ошибки нельзя просто запустить один тест.

Зависимость от других тестов для настройки конфигурации всегда затрудняет понимание тестов, так как конфигурация остается невидимой для читателя — классический случай *тайныстенного гостя* (Mystery Guest; см. *Непонятный тест*, Obscure Test, с. 230). Этую проблему можно решить частично, воспользовавшись *методами поиска* (Finder Method) с подходящими именами для доступа к объектам *общей тестовой конфигурации* (Shared Fixture). Все немного проще, если *тестовые методы* (Test Method) находятся в пределах одного *класса теста* (Testcase Class, с. 401) и перечислены в том же порядке, в котором они выполняются.

Вариант: тест с настройкой тестовой конфигурации (Fixture Setup Testcase)

Если необходимо создать *общую тестовую конфигурацию* (Shared Fixture) и нельзя воспользоваться другими способами — например, “ленивой” настройкой (Lazy Setup, с. 460), *настройкой тестовой конфигурации набора* (Suite Fixture Setup, с. 465) или *декоратором настройки* (Setup Decorator, с. 471), — можно заставить тест с *настройкой тестовой конфигурации* (Fixture Setup Testcase) запускаться первым во всем наборе. Это просто сделать, если используется *перечисление тестов* (Test Enumeration, с. 425). Достаточно просто включить подходящий вызов метода `addTest` в *фабрику наборов тестов* (Test Suite Factory). Данная конструкция является вырожденным случаем *цепочки тестов* (Chained Tests), так как весь набор подключается к *тесту с настройкой тестовой конфигурации* (Fixture Setup Testcase).

Замечания по реализации

В реализации *цепочки тестов* (Chained Tests) есть два сложных момента:

- обеспечение необходимого порядка запуска тестов в наборе;
- доступ к элементам тестовой конфигурации, созданным предыдущими тестами.

Хотя в некоторых реализациях xUnit предоставляется явный механизм определения порядка запуска тестов, в большинстве реализаций на этот счет не дается никаких гарантий. Несколько экспериментов позволяют выяснить используемый порядок запуска в конкретном случае. Чаще всего оказывается, что порядок совпадает с порядком следования *тестовых методов* (Test Method) в файле с исходным кодом или с алфавитным порядком над именами тестовых методов (в таком случае простейшим решением является вставка порядкового номера в имя теста). В худшем случае всегда можно вернуться к *перечислению тестов* (Test Enumeration) для обеспечения правильного порядка добавления *объектов теста* (Testcase Object) в набор тестов.

Для обращения к объектам, созданным предыдущими тестами, необходимо воспользоваться одним из шаблонов доступа к объектам конфигурации. Если предыдущие *тестовые методы* (Test Method) находятся в том же *классе теста* (Testcase Class), достаточно сохранить ссылки на объекты в переменных класса (использование переменных экземпляра не поможет, поскольку каждый тест запускается в собственном объекте теста и не имеет доступа к переменным объектов других тестов; исключения из этого правила рассматриваются во врезке “Всегда есть исключения” на с. 411).

Если тест зависит от *тестового метода* (Test Method) из другого *класса теста* (Testcase Class), запускаемого в пределах *набора наборов* (Suite of Suites), ни одно из этих решений не поможет. Наилучшим вариантом будет использование *реестра тестовых конфигураций* (Test Fixture Registry) для хранения ссылок на используемые тестами объекты. Хорошим примером такого решения является тестовая база данных.

Очевидно, что предыдущий тест не должен удалять тестовую конфигурацию, иначе ничего не останется для повторного использования. Это требование делает *цепочки тестов* (Chained Tests) несовместимыми с *новой тестовой конфигурацией* (Fresh Fixture, с. 344).

Мотивирующий пример

Ниже приведен пример инкрементного *табличного теста* (Tabular Test), взятый из журнала Клинта Шанка (Clint Shank).

```
public class TabularTest extends TestCase {
    private Order order = new Order();
    private static final double tolerance = 0.001;
    public void testGetTotal() {
        assertEquals("initial", 0.00, order.getTotal(), tolerance);
        testAddItemAndGetTotal("first", 1, 3.00, 3.00);
        testAddItemAndGetTotal("second", 3, 5.00, 18.00);
        // И т.д.
    }
    private void testAddItemAndGetTotal(String msg,
                                        int lineItemQuantity,
                                        double lineItemPrice,
                                        double expectedTotal) {
        // Настройка
        LineItem item = new LineItem(lineItemQuantity, lineItemPrice);
        // Вызов тестируемой системы
        order.addItem(item);
        // Проверка суммы
        assertEquals(msg, expectedTotal, order.getTotal(), tolerance);
    }
}
```

Тест начинается с создания пустого заказа, сравнивает сумму с нулем и добавляет несколько пунктов, проверяя сумму после каждой операции добавления (рис. 20.3). Основной проблемой данного теста является то, что если один из подтестов завершится неудачно, остальные не будут запущены. Например, предположим, что ошибка округления сделает сумму после второго элемента неправильной. Интересно ли разработчику, совпадет ли сумма на четвертом, пятом и шестом элементах?

Замечания по рефакторингу

Данный *табличный тест* (Tabular Test) можно превратить в *цепочку тестов* (Chained Tests), разделив один *тестовый метод* (Test Method) на несколько, по одному на каждый подтест. Одним из способов преобразования является применение рефакторинга *выделение метода* (Extract Method). Это заставит воспользоваться рефакторингом *введение поля* (Introduce field) для всех локальных переменных перед применением рефакторинга *выделение метода* (Extract Method). После определения новых *тестовых методов* (Test Method) можно достаточно просто удалить старый метод и разрешить *инфраструктуре автоматизации тестов* (Test Automation Framework, с. 332) вызывать новый *тестовый метод* (Test Method) непосредственно. (Не волнуйтесь, если под рукой нет инструментов рефакторинга. Просто добавьте окончание тестового метода после каждого подтеста и введите сигнатуру нового метода перед следующим подтестом. После этого все переменные *общей тестовой конфигурации* (Shared Fixture) удаляются из первого *тестового метода* (Test Method).)

Необходимо обеспечить одинаковый порядок запуска тестов. Поскольку пакет JUnit, очевидно, сортирует *объекты теста* (Testcase Object) по имени метода, правильный порядок можно установить, включив порядковый номер в имя тестового метода.

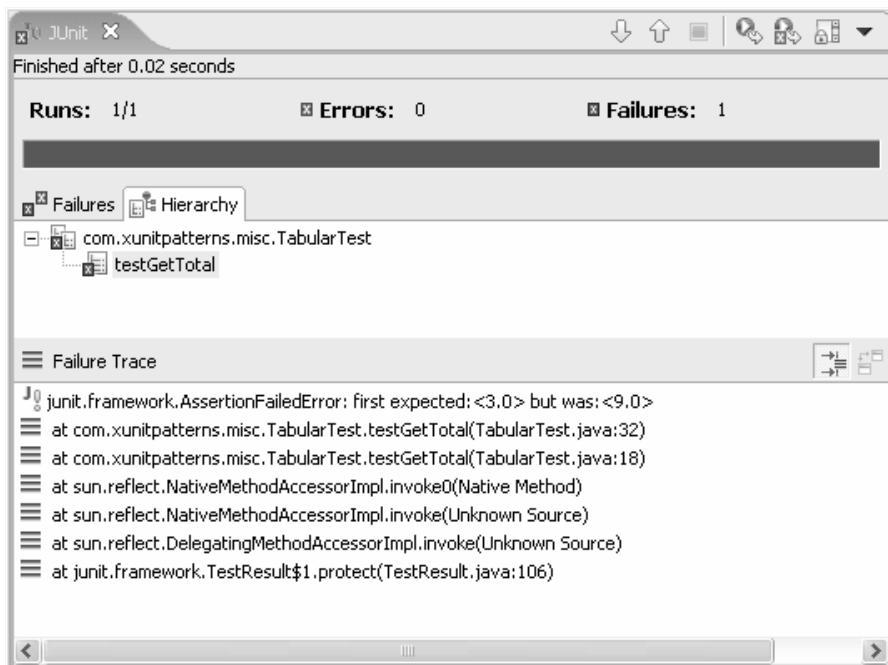


Рис. 20.3. Результаты выполнения табличного теста (Tabular Test). На нижней панели приведена информация о неудачном завершении метода табличного теста, представленного на верхней панели. Из-за неудачного завершения оставшаяся часть метода не выполняется

Наконец, необходимо преобразовать новую тестовую конфигурацию (Fresh Fixture) в общую тестовую конфигурацию (Shared Fixture). Для этого поле `order` (переменная экземпляра) необходимо заменить переменной класса (статическая переменная в языке Java), чтобы все объекты теста использовали одну и ту же переменную `order`.

Пример: цепочки тестов (Chained Tests)

Ниже показан предыдущий пример после преобразования в три отдельных теста.

```
private static Order order = new Order();
private static final double tolerance = 0.001;
public void test_01_initialTotalShouldBeZero() {
    assertEquals("initial", 0.00, order.getTotal(), tolerance);
}
public void test_02_totalAfter1stItemShouldBeOnlyItemAmount() {
    testAddItemAndGetTotal("first", 1, 3.00, 3.00);
}
public void test_03_totalAfter2ndItemShouldBeSumOfAmounts() {
    testAddItemAndGetTotal("second", 3, 5.00, 18.00);
}
private void testAddItemAndGetTotal(String msg,
                                   int lineItemQuantity,
                                   double lineItemPrice,
```

```

        double expectedTotal) {
    // Создание элемента
    LineItem item =
        new LineItem(lineItemQuantity, lineItemPrice);
    // Добавление элемента в заказ
    order.addItem(item);
    // Проверка суммы
    assertEquals(msg,expectedTotal,order.getTotal(),tolerance);
}

```

Программа запуска тестов (Test Runner, с. 405) лучше показывает, что работает, а что — нет (рис. 20.4).

К сожалению, ни один из этих тестов нельзя запустить по отдельности, так как тесты зависят друг от друга и являются *одинокими* (Lonely Test).

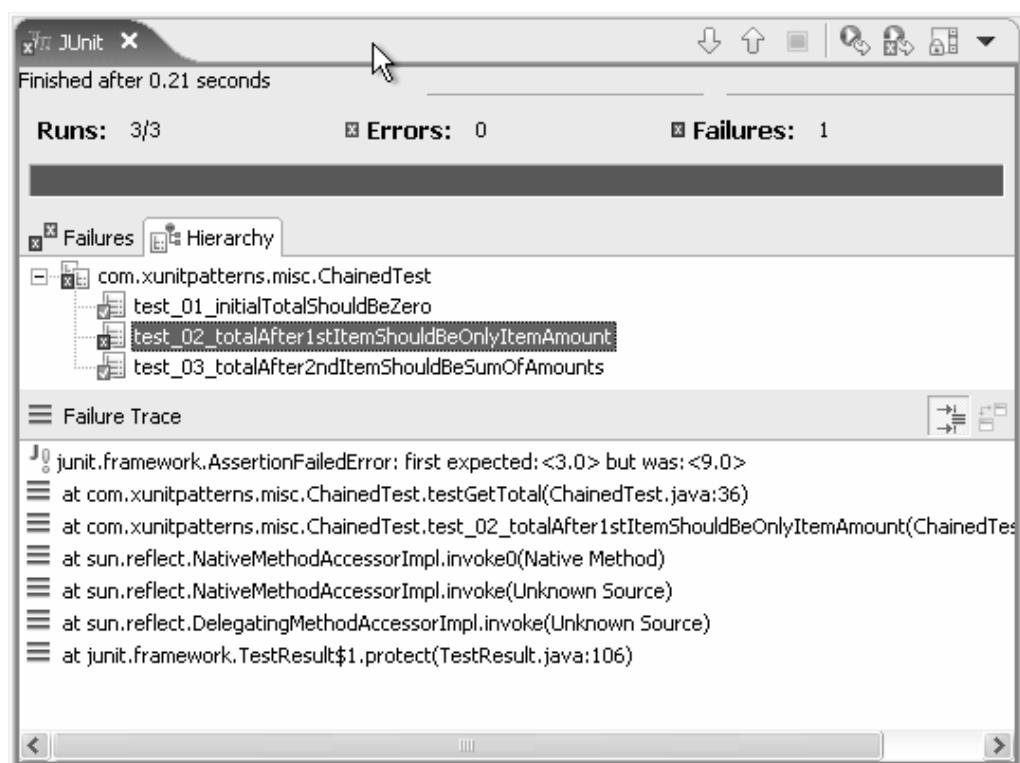


Рис. 20.4. Результат работы цепочки тестов (Chained Tests). На верхней панели показаны три тестовых метода, два из которых завершаются успешно. На нижней панели показана дополнительная информация о неудачно завершившемся teste

Глава 21

Шаблоны проверки результатов

Шаблоны в этой главе:

Стратегии проверки

Проверка состояния (State Verification)	484
Проверка поведения (Behavior Verification)	489

Стили методов с утверждениями

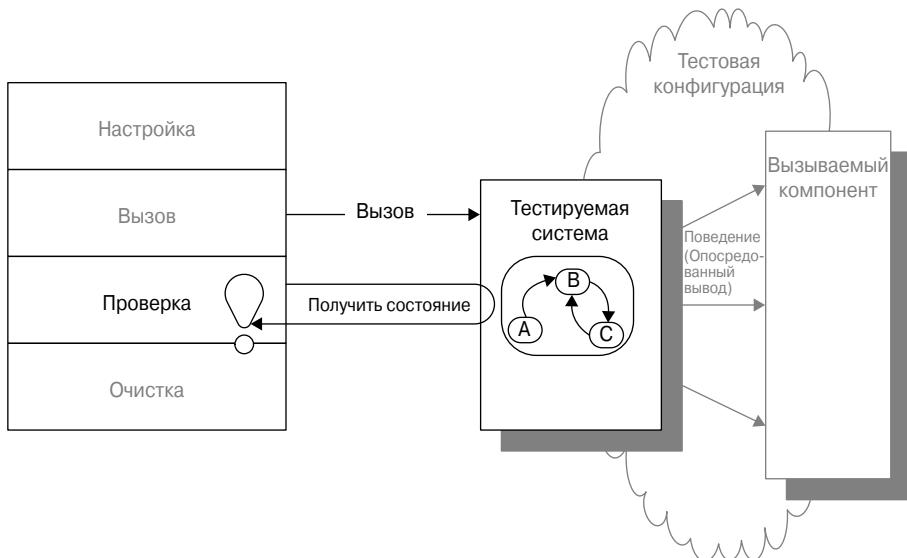
Специальное утверждение (Custom Assertion)	495
Дельта-утверждение (Delta Assertion)	505
Сторожевое утверждение (Guard Assertion).....	510
Утверждение незаконченного теста (Unfinished Test Assertion)	514

Проверка состояния (State Verification)

Также известен как:
Тестирование на основе состояния (State-Based Testing)

Как создать самопроверяющийся тест, основанный на проверке состояния?

Состояние системы после вызова тестом сравнивается с ожидаемым состоянием.



Самопроверяющийся тест (Self-Checking Test, с. 81) должен проверять наступление ожидаемого результата без ручного вмешательства разработчика. Но что подразумевается под ожидаемым результатом? Тестируемая система может иметь или не иметь “внутреннее состояние”. Если система имеет состояние, оно может измениться или не измениться после вызова системы. Разработчик теста должен определить ожидаемый результат как изменение конечного состояния или определенного процесса во время вызова тестируемой системы.

Проверка состояния (State Verification) подразумевает просмотр состояния тестируемой системы после вызова.

Как это работает

Тестируемая система вызывается через интересующие тест методы. После этого во время отдельной фазы происходит взаимодействие с тестируемой системой для получения ее текущего состояния и его сравнения с ожидаемым состоянием. Сравнение происходит с помощью *методов с утверждением* (Assertion Method, с. 390).

Обычно для доступа к состоянию тестируемой системы достаточно вызвать метод или функцию, возвращающую состояние. Это особенно справедливо при разработке на основе тестов, так как тесты требуют упрощения доступа к состоянию. Но при интеграции тестов в существующий продукт получить состояние может быть сложно. В таких случаях

придется использовать *связанный с тестом подкласс* (Test-Specific Subclass, с. 592) или другой механизм доступа к состоянию без внедрения логики теста в продукт (Test Logic in Production, с. 257).

Хороший наводящий вопрос: “Где хранится состояние тестируемой системы?” Иногда состояние хранится внутри системы, а иногда — в других компонентах, например в базе данных. В последнем случае *проверка состояния* (State Verification) заключается в доступе к состоянию внутри другого компонента (что вынуждает тест *пересекать уровни*, Layer-Crossing Test). С другой стороны, *проверка поведения* (Behavior Verification, с. 489) предполагает контроль взаимодействия тестируемой системы и других компонентов.

Когда это использовать

Проверка состояния (State Verification) может использоваться в ситуациях, когда нужно знать конечное состояние тестируемой системы, а не способ, которым система достигла этого состояния. Подобное ограниченное представление позволяет поддерживать скрытие реализации тестируемой системы.

Проверка состояния (State Verification) является вполне естественным решением при создании программного обеспечения в направлении “изнутри наружу”, т.е. сначала создается самый глубоко вложенный объект, после чего на его основе создается следующий уровень объектов. Конечно, может потребоваться использование *тестовых заглушек* (Test Stub, с. 544) для контроля за опосредованным вводом тестируемой системы. Это позволит избежать *ошибок в продукте* (Production Bugs, с. 303), связанных с не тестируанными ветвями кода. Даже в этом случае опосредованный вывод тестируемой системы не проверяется.

Если побочные эффекты вызова тестируемой системы, не отраженные в конечном состоянии, все-таки интересуют разработчиков, можно воспользоваться *проверкой поведения* (Behavior Verification) для непосредственного контроля интересующего поведения. Но при этом стоит соблюдать осторожность, так как зарегулированное программное обеспечение приведет к появлению “хрупких” тестов (Fragile Test, с. 277).

Замечания по реализации

Существует два основных стиля реализации *проверки состояния* (State Verification).

Вариант: процедурная проверка состояния (Procedural State Verification)

При *процедурной проверке состояния* (Procedural State Verification) создается последовательность вызовов *методов с утверждением* (Assertion Method), которые анализируют информацию о состоянии и сравнивают ее элементы с отдельными ожидаемыми значениями. Большинство начинающих разработчиков тестов воспринимают это решение как путь наименьшего сопротивления. Основным недостатком такого подхода является вероятность появления *непонятных тестов* (Obscure Test, с. 230), усложненных большим количеством утверждений, которые описывают ожидаемый результат. Если одна и та же последовательность утверждений должна проверяться в нескольких тестах или несколько раз в одном *тестовом методе* (Test Method, с. 378), возникает опасность *дублирования тестового кода* (Test Code Duplication, с. 254).

Вариант: спецификация ожидаемого состояния (Expected State Specification)

Также известен как:

**Ожидаемый объект
(Expected Object)**

В этом случае создается спецификация состояния тестируемой системы после вызова. Спецификация имеет форму одного или нескольких объектов с ожидаемыми значениями интересующих тест атрибутов. После этого состояние сравнивается непосредственно с этими объектами единственным вызовом *утверждения равенства* (Equality Assertion). Такой подход приводит к получению более простых и понятных тестов. *Спецификация ожидаемого состояния* (Expected State Specification) может использоваться каждый раз, когда необходимо проверить несколько атрибутов и существует возможность создать объект, аналогичный не возвращаемому тестируемой системой. Чем больше атрибутов нужно проверить и чем больше проверяющих их тестов, тем правильнее используются *спецификации ожидаемого состояния* (Expected State Specification). В большинстве крайних случаев, когда необходимо проверить большой объем данных, можно создать ожидаемую таблицу и проверить, совпадает ли она с состоянием тестируемой системы. Строковые конфигурации инфраструктуры Fit обеспечивают подобную функциональность для приемочных тестов. Утилиты, подобные DbUnit, позволяют в такой ситуации воспользоваться *манипуляцией через "черный ход"* (Back Door Manipulation, с. 359) для модульных тестов.

При создании *спецификации ожидаемого состояния* (Expected State Specification) более предпочтительным может оказаться использование *параметризованного метода создания* (Parameterized Creation Method), чтобы читатель не отвлекался на необходимые, но не важные атрибуты состояния. Чаще всего *спецификация ожидаемого состояния* (Expected State Specification) представляет собой экземпляр того же класса, который возвращается тестируемой системой. Если объект не имеет реализации равенства, подразумевающей сравнение значений атрибутов, или определение равенства в пределах теста отличается семантикой от метода *equals* интересующего объекта, использование *спецификации ожидаемого состояния* (Expected State Specification) может вызвать некоторые трудности.

В таком случае следует создать *специальное утверждение* (Custom Assertion, с. 495), в котором будет реализовано необходимое тесту условие равенства. С другой стороны, можно создать *спецификацию ожидаемого состояния* (Expected State Specification) на основе класса, уже содержащего необходимую реализацию равенства. В качестве такого класса может выступать *связанный с тестом подкласс* (Test-Specific Subclass), переопределяющий метод *equals* или простой объект передачи данных, реализующий метод *equals* (*TheRealObjectClass other*). Оба варианта более предпочтительны по сравнению с модификацией реализации метода *equals* в классе продукта (что само по себе является вариантом *засорения равенства*, Equality Pollution). Если сложно создать экземпляр класса, можно определить *поддельный объект* (Fake Object, с. 565) с необходимыми атрибутами и методом *equals*, реализующим необходимое тесту равенство. Последние несколько “фокусов” возможны за счет того, что *утверждения равенства* (Equality Assertion) обычно предлагают спецификации ожидаемого состояния сравнить себя с фактическим результатом, а не наоборот.

Спецификацию ожидаемого состояния (Expected State Specification) можно создавать на этапе проверки результатов сразу же перед использованием в *утверждении равенства* (Equality Assertion) или на этапе создания тестовой конфигурации. Второй вариант позволяет использовать атрибуты *спецификации ожидаемого состояния* (Expected State Specification) в качестве параметров тестируемой системы или в качестве основы для *вычисляемых значе-*

ний (Derived Value, с. 722) при создании других объектов для тестовой конфигурации. Это делает причинно-следственную связь между тестовой конфигурацией и *спецификацией ожидаемого состояния* (Expected State Specification) более очевидной, что, в свою очередь, позволяет использовать *тесты как документацию* (Tests as Documentation, с. 79). Это особенно полезно, когда спецификация создается за пределами области видимости читателя тестов, например при использовании *методов создания* (Creation Method).

Мотивирующий пример

В этом простом примере показан тест, вызывающий код для добавления элемента в счет. Так как тест не содержит утверждений, он не является *самопроверяющимся* (Self-Checking Test). (Естественный пример данного шаблона плохо иллюстрирует разницу между *проверкой состояния* (State Verification) и *проверкой поведения* (Behavior Verification). Обратите внимание на раздел, посвященный *проверке поведения* (Behavior Verification), в котором показан второй, более подходящий для непосредственного сравнения, пример *проверки состояния* (State Verification).)

```
public void testInvoice_addOneLineItem_quantity1() {
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
}
```

Объекты *invoice* и *product* создаются в методе *setUp* в соответствии со стратегией *неявной настройки* (Implicit Setup, с. 450).

```
public void setUp() {
    product = createAnonProduct();
    anotherProduct = createAnonProduct();
    inv = createAnonInvoice();
}
```

Замечания по рефакторингу

Первое изменение не является рефакторингом, так как приводит к изменению поведения тестов (в лучшую сторону): для начала вводятся утверждения, описывающие ожидаемый результат. Результат модификации теста является примером *процедурной проверки состояния* (Procedural State Verification), так как *тестовый метод* (Test Method) содержит последовательность вызовов встроенных *методов с утверждением* (Assertion Method).

Тестовый метод можно упростить, заставив его использовать *ожидаемый объект* (Expected Object). Сначала создается *ожидаемый объект* (Expected Object) из ожидаемого класса или подходящего *тестового двойника* (Test Double, с. 538). После этого объект инициализируется значениями, которые раньше указывались в утверждениях. Затем последовательность утверждений можно заменить *утверждением равенства* (Equality Assertion), сравнивающим полученный результат с *ожидаемым объектом* (Expected Object). Если нужно обеспечить проверку специального равенства, можно воспользоваться *специальным утверждением* (Custom Assertion).

Пример: процедурная проверка состояния (Procedural State Verification)

В данном случае в *тестовый метод* (Test Method) были добавлены утверждения, превращающие его в *самопроверяющийся тест* (Self-Checking Test). Поскольку для проверки ожидаемого результата необходимо выполнить несколько операций, получается *непонятный тест* (Obscure Test).

```
public void testInvoice_addOneLineItem_quantity1() {
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    // Проверка пункта
    LineItem actual = (LineItem) lineItems.get(0);
    assertEquals(inv, actual.getInv());
    assertEquals(product, actual.getProd());
    assertEquals(QUANTITY, actual.getQuantity());
}
```

Пример: ожидаемый объект (Expected Object)

В упрощенной версии теста вместо последовательности утверждений используются *ожидаемый объект* (Expected Object) и *утверждение равенства* (Equality Assertion).

```
public void testInvoice_addLineItem1() {
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.addItemQuantity(expItem.getProd(), expItem.getQuantity());
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem) lineItems.get(0);
    assertEquals("Item", expItem, actual);
}
```

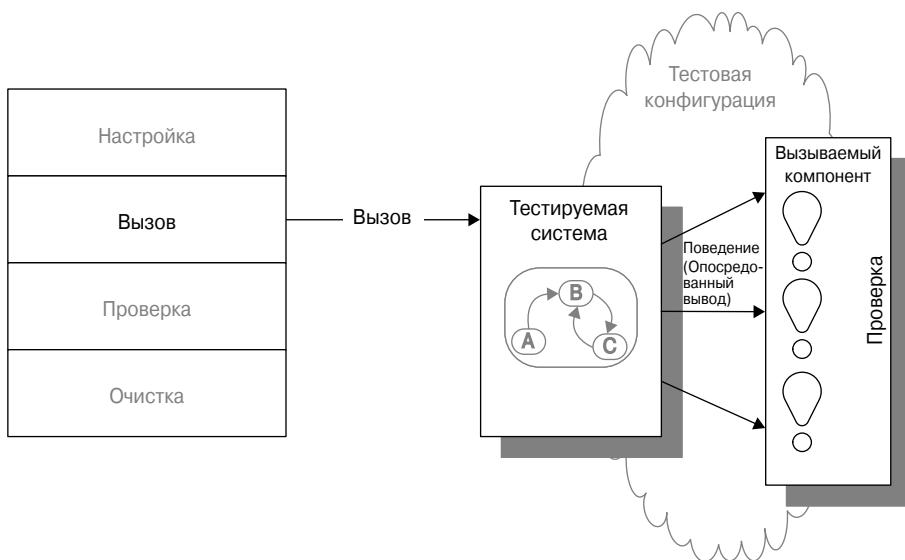
Поскольку некоторые атрибуты *ожидаемого объекта* (Expected Object) используются в качестве параметров тестируемой системы, он создается на этапе настройки тестовой конфигурации.

Проверка поведения (Behavior Verification)

Как создать самопроверяющийся тест, если не существует состояния, которое можно проверить?

Опосредованный вывод тестируемой системы перехватывается и сравнивается с ожидаемым поведением.

Также известен как:
Тестирование взаимодействия
(Interaction Testing)



Самопроверяющийся тест (Self-Checking Test, с. 81) должен проверять наступление ожидаемого результата без ручного вмешательства разработчика. Но что подразумевается под ожидаемым результатом? Тестируемая система может иметь или не иметь “внутреннее состояние”; если состояние существует, оно может меняться или не меняться в результате вызова. Кроме этого, от тестируемой системы может ожидаться вызов методов других объектов или компонентов.

Проверка поведения (Behavior Verification) подразумевает проверку опосредованного вывода тестируемой системы в процессе взаимодействия с тестом.

Как это работает

Каждый тест описывает не только взаимодействие клиентов с тестируемой системой, но и взаимодействие тестируемой системы с компонентами, от которых она зависит. Это позволяет убедиться, что система ведет себя ожидаемым образом, а не только приходит в указанное состояние.

Проверка поведения (Behavior Verification) практически всегда подразумевает замену вызываемого компонента (или взаимодействие с ним), с которым тестируемая система

связана во время выполнения. Граница между *проверкой поведения* (Behavior Verification) и *проверкой состояния* (State Verification, с. 485) может оказаться размытой, если система хранит свое состояние в вызываемом компоненте, так как оба варианта проверки потребуют теста с пересечением уровней. Основным отличием является проверка состояния внутри вызываемого компонента после завершения теста (*проверка состояния*, State Verification) или проверка обращений к методам вызываемого компонента со стороны тестируемой системы (*проверка поведения*, Behavior Verification).

Когда это использовать

Проверка поведения (Behavior Verification) является основной техникой для модульных и компонентных тестов. Проверка поведения может использоваться каждый раз, когда тестируемая система вызывает методы других объектов или компонентов. *Проверка поведения* (Behavior Verification) должна использоваться каждый раз, когда ожидаемый вывод тестируемой системы является временным и не может быть проверен в процессе простого просмотра состояния системы или вызываемого компонента после завершения теста. В результате приходится следить за опосредованным выводом непосредственно во время его генерации.

Часто *проверка поведения* (Behavior Verification) применяется при создании кода в направлении “извне во внутрь”. Такой подход — **разработка на основе потребностей** (need-driven development) — предполагает создание клиентского кода до создания вызываемого компонента. Это хороший способ определить интерфейс вызываемого компонента на основе конкретных примеров, а не на основе предположений. Главным недостатком такого подхода является использование большого количества *тестовых двойников* (Test Double, с. 538) при создании необходимых тестов. Это может привести к появлению “хрупких” тестов (Fragile Test, с. 277), так как каждый тест содержит большой объем сведений о реализации тестируемой системы. Поскольку тесты описывают поведение тестируемой системы в терминах взаимодействия с вызываемым объектом, изменение реализации системы может привести к нарушению работы большого количества тестов. Такой случай *зарегулированной программы* (Overspecified Software; см. “Хрупкий” тест, Fragile Test) может привести к росту *стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300).

На данный момент общепринятое мнение о превосходстве *проверки поведения* (Behavior Verification) над *проверкой состояния* (State Verification) еще не сложилось. *Проверка состояния* (State Verification) просто необходима в одних случаях, как *проверка поведения* (Behavior Verification) — в других. Осталось определить, должна ли *проверка поведения* (Behavior Verification) использоваться во всех случаях или чаще должна применяться *проверка состояния* (State Verification), а к *проверке поведения* (Behavior Verification) разработчики должны обращаться тогда, когда одно только состояние не показывает полной картины.

Замечания по реализации

Перед вызовом тестируемой системы через обращение к интересующим методам необходимо обеспечить способ наблюдения за ее поведением. Иногда такое наблюдение возможно благодаря механизмам, которые применяются тестируемой системой для взаимодействия с другими компонентами. Если такие механизмы недоступны, придется устанавливать *тестовый двойник* (Test Double) подходящего типа для перехвата опосредованного вывода. *Тестовый двойник* (Test Double) может использоваться

тогда, когда им можно заменить вызываемый компонент. Для этого можно воспользоваться *вставкой зависимости* (Dependency Injection, с. 684) или *поиском зависимости* (Dependency Lookup, с. 692).

Существует два фундаментально разных способа реализации *проверки поведения* (Behavior Verification). Каждый из них имеет своих сторонников. Сообщество *подставных объектов* (Mock Object, с. 558) выступает за использование “подставок” в качестве *спецификации ожидаемого поведения* (Expected Behavior Specification). Такой подход является наиболее распространенным. Несмотря на это применение *подставных объектов* (Mock Object) — не единственный способ *проверки поведения* (Behavior Verification).

Вариант: процедурная проверка поведения (Procedural Behavior Verification)

<p>При процедурной проверке поведения перехватываются вызовы методов со стороны тестируемой системы. После этого <i>тестовый метод</i> (Test Method, с. 378) получает доступ к записанным вызовам. Для сравнения записанных вызовов с ожидаемым результатом используются <i>утверждения равенства</i> (Equality Assertion; см. <i>Метод с утверждением</i>, Assertion Method, с. 390).</p>	<p>Также известен как: Ожидаемое поведение <i>(Expected Behavior)</i></p>
--	---

Самым распространенным способом перехвата опосредованного вывода тестируемой системы является установка *тестового агента* (Test Spy, с. 552) вместо вызываемого компонента на этапе создания тестовой конфигурации. На этапе проверки результатов *тестовый агент* (Test Spy) предоставляет информацию о поступивших от тестируемой системы вызовах. Для использования тестового агента (Test Spy) специальные знания о методах вызываемого компонента не нужны.

Альтернативным решением является получение этой информации у самого вызываемого компонента. Хотя такая схема не всегда возможна, в некоторых ситуациях она позволяет избежать использования *тестового двойника* (Test Double) и сократить вероятность появления *зарегулированной программы* (Overspecified Software).

Можно сократить объем кода *тестового метода* (Test Method) (и избежать дублирования *тестового кода*, Test Code Duplication, с. 254), определив *ожидаемый объект* (Expected Object) для аргументов вызовов методов или делегировав их проверку в *специальные утверждения* (Custom Assertion, с. 495).

Вариант: спецификация ожидаемого поведения (Expected Behavior Specification)

Это альтернативный способ проверки поведения. Вместо проверки опосредованного вывода постфактум с помощью последовательности утверждений *спецификация ожидаемого поведения* (Expected Behavior Specification) загружается в *подставной объект* (Mock Object), который проверяет правильность вызовов методов непосредственно после их получения.

Спецификация ожидаемого поведения (Expected Behavior Specification) может использоваться тогда, когда заранее известно, что должно происходить, и необходимо избавиться от *процедурной проверки поведения* (Procedural Behavior Verification) в *тестовом методе* (Test Method). Данный вариант шаблона позволяет получить более короткие тесты (предполагается, что используется компактное представление ожидаемого поведения) и может использоваться для неудачного завершения теста при первом отклонении фактического поведения от ожидаемого.

Одним из преимуществ использования *подставных объектов* (Mock Object) является доступность инструментов генерации *тестовых двойников* (Test Double) для большинства

реализаций xUnit. Такие инструменты значительно упрощают *спецификацию ожидаемого поведения* (Expected Behavior Specification), так как не приходится создавать *тестовый двойник* (Test Double) вручную для каждого набора тестов. Одним из недостатков использования *подставных объектов* (Mock Object) является необходимость предугадывания обращений к вызываемому объекту, включая передаваемые методам аргументы.

Мотивирующий пример

Следующий тест не является самопроверяющимся, так как не проверяет наступление ожидаемого результата. Он не содержит вызовов *методов с утверждением* (Assertion Method) и не передает ожидаемые выводы в *подставной объект* (Mock Object). Поскольку проверяется функциональность ведения журнала тестируемой системы, интересующее состояние хранится в объекте `logger`, а не в самой тестируемой системе. Автор теста не смог найти способ доступа к проверяемому состоянию.

```
public void testRemoveFlightLogging_NSC() throws Exception {
    // Настройка
    FlightDto expectedFlightDto = createARegisteredFlight();
    FlightManagementFacade facade =
        new FlightManagementFacadeImpl();
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    // Пока не найден способ проверки результата
    // Журнал содержит запись об удалении перелета
}
```

Для проверки результата разработчик, запустивший тест, должен получить доступ к базе данных и консоли журнала и сравнить фактический вывод в журнал с необходимым.

Одним из способов создания *самопроверяющегося теста* (Self-Checking Test) является добавление *спецификации ожидаемого состояния* (Expected State Specification).

```
public void testRemoveFlightLogging_ESS() throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnUnregFlight();
    FlightManagementFacadeImplTI facade =
        new FlightManagementFacadeImplTI();
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertFalse("flight still exists after being removed",
        facade.flightExists(expectedFlightDto.
            getFlightNumber()));
}
```

К сожалению, такой тест совершенно не проверяет функцию записи в журнал, принадлежащую тестируемой системе. Кроме того, на данном примере показана причина *проверки поведения* (Behavior Verification): некоторые функции тестируемой системы не влияют на ее конечное состояние и могут контролироваться только при перехвате поведения во внутренней точке наблюдения между тестируемой системой и вызываемым

компонентом или при описании поведения в терминах изменений состояния объектов, с которыми взаимодействует тестируемая система.

Замечания по рефакторингу

Внесенные в код мотивирующего примера изменения на самом деле не являются рефакторингом, так как после добавления логики проверки поведение теста изменилось. Но существует несколько вариантов рефакторинга, которые здесь стоит рассмотреть.

Для преобразования *проверки состояния* (State Verification) в *проверку поведения* (Behavior Verification) необходимо воспользоваться рефакторингом заменить зависимость *тестовым двойником* (Replace Dependency with Test Double, с. 740) для обеспечения видимости опосредованного вывода тестируемой системы с помощью *тестового агента* (Test Spy) или *подставного объекта* (Mock Object).

Для преобразования *спецификации ожидаемого поведения* (Expected Behavior Specification) в *процедурную проверку поведения* (Procedural Behavior Verification) вместо *подставного объекта* (Mock Object) устанавливается *тестовый агент* (Test Spy). После вызова тестируемой системы проверяются утверждения относительно значений, возвращаемых *тестовым агентом* (Test Spy). Полученные значения сравниваются с ожидаемыми значениями, которые передавались при создании *подставного объекта* (Mock Object), преобразованного в *тестовый агент* (Test Spy).

Для преобразования процедурной проверки поведения (Procedural Behavior Verification) в *спецификацию ожидаемого поведения* (Expected Behavior Specification) *подставной объект* (Mock Object) настраивается с использованием значений, которые сравниваются с возвращаемыми *тестовым агентом* (Test Spy) значениями. После этого вместо *тестового агента* (Test Spy) устанавливается *подставной объект* (Mock Object).

Пример: процедурная проверка поведения (Procedural Behavior Verification)

Приведенный ниже тест проверяет создание перелета, но для проверки опосредованного вывода тестируемой системы используется *процедурная проверка поведения* (Procedural Behavior Verification). Для перехвата опосредованного вывода используется *тестовый агент* (Test Spy), а перехваченные значения проверяются с помощью вызова *методов с утверждением* (Assertion Method).

```
public void testRemoveFlightLogging_recordingTestStub()
    throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnUnregFlight();
    FlightManagementFacade facade =
        new FlightManagementFacadeImpl();
    // Настройка тестового двойника
    AuditLogSpy logSpy = new AuditLogSpy();
    facade.setAuditLog(logSpy);
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertEquals("number of calls", 1,
                logSpy.getNumberOfCalls());
    assertEquals("action code",
                Helper.REMOVE_FLIGHT_ACTION_CODE,
```

```

        logSpy.getActionCode();
assertEquals("date", helper.getTodaysDateWithoutTime(),
            logSpy.getDate());
assertEquals("user", Helper.TEST_USER_NAME,
            logSpy.getUser());
assertEquals("detail",
            expectedFlightDto.getFlightNumber(),
            logSpy.getDetail());
}

```

Пример: спецификация ожидаемого поведения (Expected Behavior Specification)

В этой версии теста для определения ожидаемого поведения тестируемой системы используется инфраструктура JMock. Метод `expects` объекта `mockLog` настраивает *подставной объект* (Mock Object), описывая ожидаемое поведение (в частности, указывая ожидаемую запись журнала).

```

public void testRemoveFlight_JMock() throws Exception {
    // настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnonRegFlight();
    FlightManagementFacade facade =
        new FlightManagementFacadeImpl();
    // настройка подставного объекта
    Mock mockLog = mock(AuditLog.class);
    mockLog.expects(once()).method("logMessage")
        .with(eq(helper.getTodaysDateWithoutTime()),
              eq(Helper.TEST_USER_NAME),
              eq(Helper.REMOVE_FLIGHT_ACTION_CODE),
              eq(expectedFlightDto.getFlightNumber()));
    // Установка подставного объекта
    facade.setAuditLog((AuditLog) mockLog.proxy());
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    // Метод verify() автоматически вызывается
    // инфраструктурой JMock
}

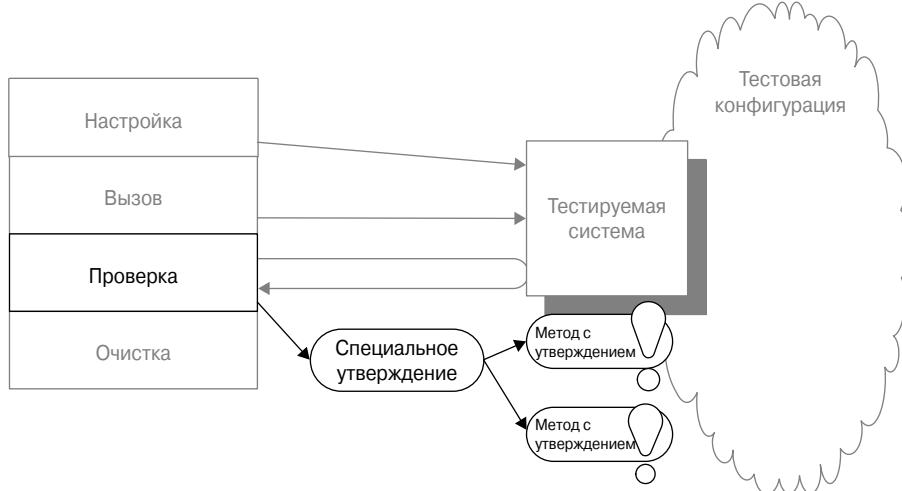
```

Специальное утверждение (Custom Assertion)

Как создать самопроверяющийся тест при наличии специфичной для теста логики равенства? Как сократить дублирование тестового кода (Test Code Duplication), если одна и та же логика используется в нескольких тестах? Как избежать появления условной логики теста (Conditional Test Logic)?

Создается специальный метод с утверждением (Assertion Method), который сравнивает только те атрибуты объектов, которые составляют специфичное для теста равенство.

Также известен как:
Заказное утверждение (Bespoke Assertion)



В большинстве реализаций xUnit предоставляется достаточно богатый набор *методов с утверждением* (Assertion Method, с. 390). Но рано или поздно оказывается, что тест проще написать при доступности утверждения, отсутствующего в стандартном наборе. Почему бы не написать такое утверждение самостоятельно?

Существует много причин создания *специальных утверждений* (Custom Assertion), но способ их создания не зависит от конкретной цели. Сложность доказательства правильности поведения системы скрывается за *методом с утверждением* (Assertion Method) с описательным именем.

Как это работает

Механизм проверки истинности чего-либо (утверждение) скрывается за описательным именем. Для этого весь общий код утверждения выделяется в *специальное утверждение* (Custom Assertion), в котором реализована логика проверки. *Специальное утверждение равенства* (Custom Equality Assertion) принимает два параметра: *ожидаемый объект* (Expected Object; см. *Проверка состояния*, State Verification, с. 484) и фактический объект.

Ключевой характеристикой *специального утверждения* (Custom Assertion) является получение всей необходимой информации для успешного или неудачного завершения теста в виде параметров. Кроме неудачного завершения теста, такие утверждения не имеют побочных эффектов.

Обычно *специальное утверждение* (Custom Assertion) создается в результате рефакторинга после идентификации общего подмножества утверждений в тестах. Для простоты создания тестов можно вызвать несуществующее *специальное утверждение* (Custom Assertion). Такой подход позволяет сосредоточиться на проверяемой логике тестируемой системы, а не на механизмах проверки.

Когда это использовать

Специальное утверждение (Custom Assertion) рекомендуется использовать, когда выполняется хотя бы одно из следующих условий.

- В нескольких тестах обнаружилась переписанная (или скопированная) одинаковая логика утверждения (см. *Дублирование тестового кода*, Test Code Duplication, с. 254).
- На этапе проверки результата используется *условная логика теста* (Conditional Test Logic, с. 243), т.е. вызовы *методов с утверждением* (Assertion Method) окружены конструкциями *if* или циклами.
- Фаза проверки результатов делает тест *непонятным* (Obscure Test, с. 230), так как вместо декларативной используется процедурная проверка.
- Поскольку утверждения не предоставляют достаточной информации, это приводит к *частой отладке* (Frequent Debugging, с. 285).

Минимизация нетестируемого кода (Minimize Untestable Code, с. 98) является основной причиной выделения логики утверждений в *специальные утверждения* (Custom Assertion). После выделения *специальных утверждений* (Custom Assertion) можно создавать *тесты специального утверждения* (Custom Assertion Test) для доказательства правильности логики проверки. Еще одним важным преимуществом использования специальных утверждений является снижение вероятности появления *непонятных тестов* (Obscure Test), так как тесты лучше *доносят намерение* (Communicate Intent, с. 95), и в результате получаются более простые в обслуживании тесты.

Если логика проверки должна взаимодействовать с тестируемой системой для определения фактического результата, вместо *специального утверждения* (Custom Assertion) можно воспользоваться *методом проверки* (Verification Method). Если в нескольких тестах отличается только значение фактического/ожидаемого объекта, а фазы настройки конфигурации и вызова тестируемой системы совпадают, стоит воспользоваться *параметризованными тестами* (Parameterized Test, с. 618). Основным преимуществом *специальных утверждений* (Custom Assertion) по сравнению с этими техниками является возможность повторного использования. Одно и то же *специальное утверждение* (Custom Assertion) может использоваться в различных условиях, так как оно не зависит от контекста (связь с окружающим миром осуществляется только через список параметров).

Чаще всего в качестве *специальных утверждений* (Custom Assertion) создаются *утверждения равенства* (Equality Assertion), но это могут быть и другие утверждения.

Вариант: специальное утверждение равенства (Custom Equality Assertion)

Специальным утверждениям равенства передаются *ожидаемый объект* (Expected Object) и фактический объект. Кроме того, в качестве параметра должно приниматься *сообщение для утверждения* (Assertion Message, с. 398), позволяющее избежать игры в *рулетку утверждений* (Assertion Roulette, с. 264). По сути это метод `equals`, реализованный в виде *внешнего метода* (Foreign Method).

Вариант: утверждение равенства атрибутов объекта (Object Attribute Equality Assertion)

Часто встречаются *специальные утверждения* (Custom Assertion), принимающие один фактический объект и несколько разных *ожидаемых объектов* (Expected Object), которые необходимо сравнить с конкретными атрибутами фактического объекта. (Набор сравниваемых атрибутов подразумевается в имени *специального утверждения* (Custom Assertion).) Ключевым отличием *специального утверждения* (Custom Assertion) от *метода проверки* (Verification Method) является взаимодействие последнего с тестируемой системой. *Утверждение равенства атрибутов объекта* (Object Attribute Equality Assertion) рассматривает только объекты, переданные в качестве параметров.

Вариант: утверждение для предметной области (Domain Assertion)

Встроенные *методы с утверждением* (Assertion Method) не зависят от предметной области. *Специальное утверждение равенства* (Custom Equality Assertion) реализует специфичное для теста равенство, но все равно сравнивает только два объекта. Еще один стиль *специальных утверждений* (Custom Assertion) является одним из компонентов языка *высокого уровня* (Higher-Level Language) для “предметной области” — *утверждение для предметной области* (Domain Assertion).

Утверждение для предметной области (Domain Assertion) представляет собой *утверждение с заявленным результатом* (Stated Outcome Assertion), указывающее на истинность утверждения в терминах предметной области. Такое утверждение позволяет поднять тесты до уровня “языка, понятного бизнесу”.

Вариант: диагностическое утверждение (Diagnostic Assertion)

Иногда *частая отладка* (Frequent Debugging) происходит каждый раз, когда неудачно завершается тест, так как утверждения показывают ложность какого-либо условия, но не показывают непосредственно проблему (т.е. утверждение показывает неравенство двух объектов, но не показывает, что именно не совпадает в этих объектах). Можно создать *специальное утверждение* (Custom Assertion) определенного типа, выглядящее как встроенное утверждение, но предоставляющее дополнительную информацию об отличиях ожидаемого и фактического значений. (Например, утверждение может показывать, какие атрибуты объектов отличаются или где не совпадают длинные строки.)

Однажды автору пришлось сравнивать строковые переменные, содержащие код XML. При каждом неудачном завершении теста приходилось открывать две строки и искать несовпадения. Наконец, на разработчиков снизошло прозрение, и было написано *специальное утверждение* (Custom Assertion), показывающее несовпадения между строками. Время, затраченное на написание утверждения, было многократно компенсировано временем работы тестов.

Вариант: метод проверки (Verification Method)

В приемочных тестах почти вся сложность проверки результата связана со взаимодействием с тестируемой системой. *Метод проверки* (Verification Method) является вариантом *специального утверждения* (Custom Assertion), которое взаимодействует с тестируемой системой, освобождая от этого вызывающий тест. Это позволяет значительно упростить тесты и приводит к более “декларативному” описанию ожидаемого результата. После создания *специального утверждения* (Custom Assertion) следующие тесты с таким же результатом создаются значительно быстрее. В некоторых случаях в *метод проверки* (Verification Method) имеет смысл интегрировать и фазу вызова тестируемой системы. Это практически *параметризованный тест* (Parameterized Test), интегрирующий всю логику теста в поддерживающий повторное использование *вспомогательный метод теста* (Test Utility Method, с. 610).

Замечания по реализации

Обычно *специальное утверждение* (Custom Assertion) реализуется в виде набора вызовов различных встроенных методов с *утверждением* (Assertion Method). В зависимости от планируемого использования может потребоваться включение стандартного шаблона *утверждения равенства* (Equality Assertion) для обеспечения правильного поведения с параметрами `null`. Поскольку *специальное утверждение* (Custom Assertion) является *методом с утверждением* (Assertion Method), оно не должно иметь побочных эффектов и не должно обращаться к тестируемой системе. (В случае необходимости следует использовать *метод проверки* (Verification Method).)

Вариант: тест специального утверждения (Custom Assertion Test)

Апологеты тестирования предлагают писать *тесты специальных утверждений* (Custom Assertion Test) для проверки *специальных утверждений* (Custom Assertion). Преимущество такого подхода очевидно: твердая уверенность в правильности тестов. В большинстве случаев в создании *теста специального утверждения* (Custom Assertion Test) нет ничего сложного, так как *метод с утверждением* (Assertion Method) принимает все свои аргументы в качестве параметров.

Специальное утверждение (Custom Assertion) может рассматриваться как тестируемая система и вызываться с различными аргументами для проверки неудачного завершения в подходящих случаях. *Утверждение с единственным результатом* (Single Outcome Assertion) требует единственного теста, так как оно не принимает никаких параметров (кроме *сообщения для утверждения*, Assertion Message). *Утверждение с заявленным результатом* (Stated Outcome Assertion) требует по одному тесту для каждого возможного (или граничного) значения. *Утверждение равенства* (Equality Assertion) нужен один тест сравнения двух эквивалентных значений, один тест сравнения объекта с самим собой и по одному тесту для каждого атрибута, неравенство которых должно привести к неудачному завершению теста. Не влияющие на равенство атрибуты должны проверяться в дополнительном teste, так как *утверждение равенства* (Equality Assertion) не должно генерировать ошибку для любого из них.

Тест специального утверждения (Custom Assertion Test) должен следовать шаблонам *простого теста успешности* (Simple Success Test) и *теста на ожидаемое исключение* (Expected Exception Test) с одним небольшим отличием: так как *метод с утверждением*

(Assertion Method) является тестируемой системой, фазы вызова и проверки результата комбинируются в одну.

Каждый тест состоит из установки *ожидаемого объекта* (Expected Object) и фактического объекта с последующим вызовом *специального утверждения* (Custom Assertion). Если объекты должны быть эквиваленты, этого достаточно. (*Инфраструктура автоматизации тестов* (Test Automation Framework, с. 332) перехватывает ложные утверждения и вынуждает тест завершиться неудачно.) Для тестов, подразумевающих ложность *специального утверждения* (Custom Assertion), можно написать *тест на ожидаемое исключение* (Expected Exception Test), но фазы вызова тестируемой системы и проверки результата будут объединены в единственный вызов *специального утверждения* (Custom Assertion).

Простым способом создания сравниваемых объектов является использование *единственного дефектного атрибута* (One Bad Attribute), т.е. создание первого объекта и глубокое копирование. Для тестов успешности необходимо модифицировать не сравниваемые атрибуты. Для тестов ложных утверждений необходимо модифицировать атрибуты, приводящие к неудачному завершению тестов.

Краткое предупреждение о возможных проблемах в некоторых реализациях xUnit: если все неудачные завершения тестов не обрабатываются в *программе запуска тестов* (Test Runner, с. 405), вызовы метода `fail` или встроенных утверждений могут добавить сообщения в журнал ошибок, даже если ошибка или исключение перехватывается в *тесте специального утверждения* (Custom Assertion Test). Единственный способ обхода этого поведения — использование “встроенной программы запуска тестов” для запуска каждого теста в отдельности и проверка, что данный конкретный тест завершился с ожидаемым сообщением об ошибке.

Мотивирующий пример

В следующем примере несколько тестовых методов содержат повторяющуюся последовательность утверждений.

```
public void testInvoice_addOneLineItem_quantity1_b() {
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    // Проверка единственного элемента
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    LineItem actual = (LineItem)lineItems.get(0);
    assertEquals(expItem.getInv(), actual.getInv());
    assertEquals(expItem.getProd(), actual.getProd());
    assertEquals(expItem.getQuantity(), actual.getQuantity());
}

public void testRemoveLineItemsForProduct_oneOfTwo() {
    // Настройка
    Invoice inv = createAnonInvoice();
    inv.addItemQuantity(product, QUANTITY);
    inv.addItemQuantity(anotherProduct, QUANTITY);
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.removeLineItemForProduct(anotherProduct);
```

```

// Проверка
List lineItems = inv.getLineItems();
assertEquals("number of items", lineItems.size(), 1);
LineItem actual = (LineItem)lineItems.get(0);
assertEquals(expItem.getInv(), actual.getInv());
assertEquals(expItem.getProd(), actual.getProd());
assertEquals(expItem.getQuantity(), actual.getQuantity());
}

//
// Добавление двух элементов
//

public void testInvoice_addTwoLineItems_sameProduct() {
    Invoice inv = createAnonInvoice();
    LineItem expItem1 = new LineItem(inv, product, QUANTITY1);
    LineItem expItem2 = new LineItem(inv, product, QUANTITY2);
    // Вызов
    inv.addItemQuantity(product, QUANTITY1);
    inv.addItemQuantity(product, QUANTITY2);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 2);
    // Проверка первого элемента
    LineItem actual = (LineItem)lineItems.get(0);
    assertEquals(expItem1.getInv(), actual.getInv());
    assertEquals(expItem1.getProd(), actual.getProd());
    assertEquals(expItem1.getQuantity(), actual.getQuantity());
    // Проверка второго элемента
    actual = (LineItem)lineItems.get(1);
    assertEquals(expItem2.getInv(), actual.getInv());
    assertEquals(expItem2.getProd(), actual.getProd());
    assertEquals(expItem2.getQuantity(), actual.getQuantity());
}

```

Обратите внимание, что первый тест заканчивается последовательностью из трех утверждений и во втором teste эта последовательность повторяется два раза, по одному для каждого элемента. Это характерный случай *дублирования тестового кода* (Test Code Duplication).

Замечания по рефакторингу

Апологеты рефакторинга заметят, что решением является применение рефакторинга *выделение метода* (Extract Method) к данным тестам. Если выделить все общие вызовы *методов с утверждением* (Assertion Method), в тестах останутся только отличающиеся элементы. Выделенные вызовы превращаются в *специальное утверждение* (Custom Assertion). Кроме того, необходимо создать *ожидаемый объект* (Expected Object), содержащий все значения, которые передавались отдельным методам с утверждением (Assertion Method). Ожидаемый объект будет передаваться в *специальное утверждение* (Custom Assertion) в качестве параметра.

Пример: специальное утверждение (Custom Assertion)

В данном teste *специальное утверждение* (Custom Assertion) используется для проверки равенства значений LineItem. По тем или иным причинам было принято решение

реализовать специфичное для теста равенство вместо использования стандартного *утверждения равенства* (Equality Assertion).

```
public void testInvoice_addOneLineItem_quantity1_() {
    Invoice inv = createAnonInvoice();
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    // Проверка одного элемента
    LineItem actual = (LineItem)lineItems.get(0);
    assertLineItemsEqual("LineItem", expItem, actual);
}
public void testAddItemQuantity_sameProduct_() {
    Invoice inv = createAnonInvoice();
    LineItem expItem1 = new LineItem(inv, product, QUANTITY1);
    LineItem expItem2 = new LineItem(inv, product, QUANTITY2);
    // Вызов
    inv.addItemQuantity(product, QUANTITY1);
    inv.addItemQuantity(product, QUANTITY2);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 2);
    // Проверка первого элемента
    LineItem actual = (LineItem)lineItems.get(0);
    assertLineItemsEqual("Item 1", expItem1, actual);
    // Проверка второго элемента
    actual = (LineItem)lineItems.get(1);
    assertLineItemsEqual("Item 2", expItem2, actual);
}
```

Тесты стали значительно меньше и доносят намерение значительно лучше. Кроме того, передается строка, указывающая на проверяемый элемент. Строковый аргумент *специального утверждения* (Custom Assertion) позволяет избежать игры в *рулетку утверждений* (Assertion Roulette) при неудачном завершении теста.

Такое упрощение теста стало возможным за счет использования следующего *специального утверждения* (Custom Assertion).

```
static void assertLineItemsEqual(
    String msg, LineItem exp, LineItem act) {
    assertEquals(msg+" Inv", exp.getInv(), act.getInv());
    assertEquals(msg+" Prod", exp.getProd(), act.getProd());
    assertEquals(msg+" Quan", exp.getQuantity(), act.getQuantity());
}
```

Данное *специальное утверждение* (Custom Assertion) сравнивает те же атрибуты объектов, что и встроенные утверждения в предыдущей версии теста. Таким образом, семантика теста не изменилась. Кроме того, имя сравниваемого атрибута добавляется к строковому параметру для получения уникального сообщения о неудачном завершении. Это позволяет быстро определить ложное утверждение, которое привело к неудачному завершению теста.

Пример: утверждение для предметной области (Domain Assertion)

В следующей версии теста утверждения были подняты на более высокий уровень для более понятного описания ожидаемого результата.

```
public void testAddOneLineItem_quantity1() {
    Invoice inv = createAnonInvoice();
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    assertInvoiceContainsOnlyThisLineItem(inv, expItem);
}
public void testRemoveLineItemsForProduct_oneOfTwo_() {
    Invoice inv = createAnonInvoice();
    inv.addItemQuantity(product, QUANTITY);
    inv.addItemQuantity(anotherProduct, QUANTITY);
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.removeLineItemForProduct(anotherProduct);
    // Проверка
    assertInvoiceContainsOnlyThisLineItem(inv, expItem);
}
```

Данное упрощение стало возможным благодаря следующему методу *утверждения для предметной области* (Domain Assertion).

```
void assertInvoiceContainsOnlyThisLineItem(
    Invoice inv,
    LineItem expItem) {
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    assertLineItemsEqual("item", expItem, actual);
}
```

В данном примере для экономии места в *утверждение для предметной области* (Domain Assertion) сообщение не передается. В реальном решении в утверждение передавался бы строковый параметр, к которому добавлялись бы сообщения отдельных утверждений. Дополнительная информация приведена в разделе о *сообщении для утверждения* (Assertion Message, с. 398).

Пример: метод проверки (Verification Method)

Если фазы вызова тестируемой системы и проверки результата в нескольких тестах практически совпадают, их можно интегрировать в повторно используемое *специальное утверждение* (Custom Assertion). Поскольку подобный подход изменяет семантику *специального утверждения* (Custom Assertion) с функции без побочных эффектов на операцию, изменяющую состояние тестируемой системы, имя такого метода обычно начинается с “verify”.

В данной версии теста перед вызовом *метода проверки* (Verification Method) тестовая конфигурация только настраивается. Метод включает в себя вызов тестируемой системы и проверку результата. Чаще всего на такую конструкцию указывают отсутствие фазы вызова тестируемой системы в вызывающем teste и присутствие вызовов методов, мо-

дифицирующих состояние объектов, передаваемых в качестве параметров *методу проверки* (Verification Method).

```
public void testAddOneLineItem_quantity2() {
    Invoice inv = createAnonInvoice();
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов и проверка
    verifyOneLineItemCanBeAdded(inv, product, QUANTITY, expItem);
}
```

Метод проверки (Verification Method) для данного примера выглядит следующим образом.

```
public void verifyOneLineItemCanBeAdded(
    Invoice inv, Product product,
    int QUANTITY, LineItem expItem) {
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    assertInvoiceContainsOnlyThisLineItem(inv, expItem);
}
```

Данный *метод проверки* (Verification Method) вызывает “чистое” *специальное утверждение* (Custom Assertion), хотя он может содержать всю логику проверки, если *специальное утверждение* (Custom Assertion) недоступно. Обратите внимание на вызов метода `addItemQuantity` параметра `inv`. В этом и состоит отличие *метода проверки* (Verification Method) от *специального утверждения* (Custom Assertion).

Пример: тест специального утверждения (Custom Assertion Test)

Это не слишком сложное *специальное утверждение* (Custom Assertion), поэтому можно обойтись и без автоматизированных тестов. Но если утверждение содержит сложную логику, стоит написать тесты, аналогичные показанным ниже.

```
public void testassertLineItemsEqual_equivalent() {
    Invoice inv = createAnonInvoice();
    LineItem item1 = new LineItem(inv, product, QUANTITY1);
    LineItem item2 = new LineItem(inv, product, QUANTITY1);
    // Вызов/проверка
    assertLineItemsEqual("This should not fail", item1, item2);
}
public void testassertLineItemsEqual_differentInvoice() {
    Invoice inv1 = createAnonInvoice();
    Invoice inv2 = createAnonInvoice();
    LineItem item1 = new LineItem(inv1, product, QUANTITY1);
    LineItem item2 = new LineItem(inv2, product, QUANTITY1);
    // Вызов/проверка
    try {
        assertLineItemsEqual("Msg", item1, item2);
    } catch (AssertionFailedError e) {
        assertEquals("e.getMessage()", "Invoice-expected: <123> but was <124>",
                    e.getMessage());
    }
}
```

```
    fail("Should have thrown exception");
}

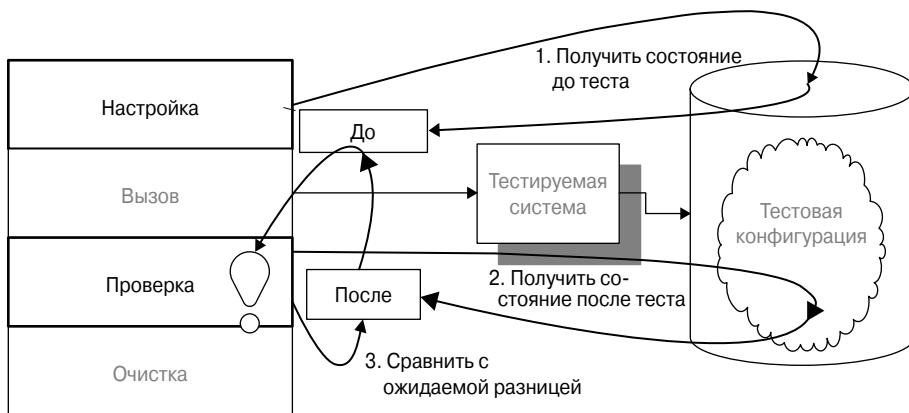
public void testassertLineItemsEqual_differentQuantity() {
    Invoice inv = createAnonInvoice();
    LineItem item1 = new LineItem(inv, product, QUANTITY1);
    LineItem item2 = new LineItem(inv, product, QUANTITY2);
    // Вызов/проверка
    try {
        assertLineItemsEqual("Msg", item1, item2);
    } catch (AssertionFailedError e) {
        pass(); // показывает, что утверждение не требуется
        return;
    }
    fail("Should have thrown exception");
}
```

В этом примере показано несколько *тестов специального утверждения* (Custom Assertion Test). Обратите внимание, что код содержит один тест “эквивалентности” и несколько тестов “отличия” (один для каждого атрибута, неравенство которого должно приводить к неудачному завершению теста). В случаях, когда утверждение должно быть ложным, использовалась вторая форма шаблона *теста на ожидаемое исключение* (Expected Exception Test), так как вызов `fail` генерирует то же исключение, что и утверждение. В одном из тестов “отличия” показан пример логики утверждения относительно сообщения исключения. (Хотя пример сокращен для экономии места, здесь достаточно информации, чтобы понять, как работает утверждение относительно сообщения.)

Дельта-утверждение (Delta Assertion)

Как создать самопроверяющийся тест, если не удается установить начальное содержимое тестовой конфигурации?

Утверждения описываются в терминах разницы между состоянием тестируемой системы до и после вызова.



При использовании *общей тестовой конфигурации* (Shared Fixture, с. 350), например тестовой базы данных, очень сложно определить утверждения, которые описывают содержимое конфигурации после вызова тестируемой системы. Это связано с тем, что другие тесты могли создать объекты в тестовой конфигурации, видимые данному утверждению, которое в результате оказалось ложным. Одно из решений — изолировать текущий тест от всех остальных тестов с помощью *схемы разбиения базы данных* (Database Partitioning Scheme). Но что если это невозможно?

Используя *дельта-утверждения* (Delta Assertion), можно меньше зависеть от начального содержимого *общей тестовой конфигурации* (Shared Fixture).

Как это работает

Перед вызовом тестируемой системы создается моментальный снимок используемых фрагментов *общей тестовой конфигурации* (Shared Fixture). После вызова тестируемой системы утверждения определяются относительно сохраненного моментального снимка. Обычно *дельта-утверждения* (Delta Assertion) проверяют правильность количества изменившихся объектов и появились ли в возвращаемой тестируемой системе коллекции ожидаемых объектов.

Когда это использовать

Дельта-утверждение (Delta Assertion) можно использовать каждый раз, когда нет полного контроля над тестовой конфигурацией и необходимо избежать появления *взаимодействующих тестов* (Interacting Tests). *Дельта-утверждения* (Delta Assertion) позво-

ляют сделать тесты менее чувствительными к изменениям в тестовой конфигурации. Кроме того, *дельта-утверждения* (Delta Assertion) в комбинации с *неявной очисткой* (Implicit Teardown, с. 533) могут использоваться для обнаружения утечек памяти или данных в тестируемом коде. Более подробно данная функция описывается ниже, во врезке “Использование дельта-утверждений для обнаружения утечки данных”.

Дельта-утверждения (Delta Assertion) хорошо работают в тестах, запускаемых *программой запуска тестов* (Test Runner) один за другим. К сожалению, они не позволяют предотвратить “войну” запуска тестов (Test Run War), так как эта проблема возникает при одновременном запуске тестов из нескольких процессов. *Дельта-утверждения* (Delta Assertion) можно использовать тогда, когда состояние тестируемой системы и конфигурации модифицируется только одним тестом. Если параллельно запущены другие тесты (не до или после, а одновременно), *дельта-утверждения* (Delta Assertion) будет недостаточно для предотвращения “войны” запуска тестов (Test Run War).

Замечания по реализации

При сохранении состояния *общей тестовой конфигурации* (Shared Fixture) или тестируемой системы необходимо убедиться, что тестируемая система не может внести изменения в моментальный снимок. Например, если моментальный снимок содержит объекты, возвращаемые системой в ответ на запрос, необходимо выполнить глубокое копирование. Простое копирование позволит получить копию только объекта Collection, а не содержащихся в коллекции объектов. Копирование только ссылок предоставит тестируемой системе возможность модифицировать предоставленные объекты. В результате ссылка на моментальный снимок, с которым будет сравниваться состояние после вызова системы, будет утрачена.

Существует несколько способов проверки правильности состояния после вызова системы. Если тест добавляет новые объекты, которые будут модифицироваться, сначала необходимо убедиться, что в коллекцию входит следующее:

- 1) правильное количество элементов;
- 2) все элементы, которые она содержала до запуска теста;
- 3) новые ожидаемые объекты (Expected Object).

Еще один подход заключается в удалении всех сохраненных элементов из результирующей коллекции и сравнение оставшихся элементов с коллекцией ожидаемых объектов. Оба подхода можно скрыть за *специальным утверждением* (Custom Assertion, с. 495) или *методом проверки* (Verification Method; см. *Специальное утверждение*).

ИСПОЛЬЗОВАНИЕ ДЕЛЬТА-УТВЕРЖДЕНИЙ ДЛЯ ОБНАРУЖЕНИЯ УТЕЧКИ ДАННЫХ

Давным-давно в одном проекте проводились эксперименты с различными способами очистки тестовой конфигурации после запуска приемочных тестов. Тесты обращались к базе данных и оставляли в ней объекты. Такое поведение приводило к различным проблемам, связанным с *неповторяемыми* (Unrepeatable Test) и *взаимодействующими тестами* (Interacting Tests). Кроме того, постоянные проблемы вызывали *медленные тесты* (Slow Tests, с. 289).

Со временем разработчикам пришла мысль наблюдать за всеми объектами, которые создавались тестами. Для этого объекты регистрировались с помощью механизма *автоматической очистки* (Automated Teardown, с. 521). Затем был придуман способ замены базы данных *поддельной базой данных* (Fake Database). Позднее был создан механизм выбора настоящей или поддельной базы данных. Работа с поддельной базой данных позволила решить множество проблем взаимодействия, хотя эти проблемы все еще повторялись при работе с настоящей базой данных — тесты оставляли после себя объекты и нужно было знать, почему. Но сначала следовало определить, какие тесты демонстрировали такое поведение.

Решение оказалось достаточно простым. В *поддельной базе данных* (Fake Database), реализованной на основе простых хэш-таблиц, был добавлен метод для подсчета количества объектов. Это значение просто сохранялось в переменной экземпляра в методе `setUp` и использовалось в качестве ожидаемого значения в *утверждении равенства* (Equality Assertion), вызываемом в методе `tearDown` для проверки правильности удаления всех объектов. (Это пример использования *дельта-утверждения* (Delta Assertion, с. 505).) После реализации данного фокуса быстро обнаружились тесты, которые страдали от утечки данных. После этого внимание было сконцентрировано на намного меньшем количестве тестов.

Даже сегодня возможность запуска одних и тех же тестов с реальной и поддельной базами данных может оказаться полезной. Точно так даже сейчас иногда тесты завершаются неудачно, когда в унаследованном методе `tearDown` появляется ложное *дельта-утверждение* (Delta Assertion). Возможно, такая же идея может применяться для перехвата утечек памяти с управлением памятью вручную (например, в C++).

Мотивирующий пример

В следующем teste система предоставляет некоторые объекты. После этого полученные объекты сравниваются с ожидаемыми.

```
public void testGetFlightsByOriginAirport_OneOutboundFlight()
    throws Exception {
    FlightDto expectedFlightDto =
        createNewFlightBetweenExistingAirports();
    // Вызов системы
    facade.createFlight(
        expectedFlightDto.getOriginAirportId(),
        expectedFlightDto.getDestinationAirportId());
    // Проверка результата
    List flightsAtOrigin = facade.getFlightsByOriginAirport(
        expectedFlightDto.getOriginAirportId());
    assertOnly1FlightInDtoList("Outbound flight at origin",
        expectedFlightDto,
        flightsAtOrigin);
}
```

К сожалению, из-за использования *общей тестовой конфигурации* (Shared Fixture) ранее запущенные тесты могли добавить собственные объекты. Такое поведение может привести к неудачному завершению текущего теста, если в наборе окажутся дополнительные, неожиданные объекты.

Замечания по рефакторингу

Для преобразования тестов с целью использования *дельта-утверждений* (Delta Assertion) сначала необходимо создать моментальный снимок данных (или коллекцию объектов), с которыми будет сравниваться результат. После этого необходимо модифицировать утверждения для учета отличий между наборами объектов/данных. Для защиты *тестовых методов* (Test Method, с. 378) от появления *условной логики теста* (Conditional Test Logic, с. 243) может потребоваться создание новых *специальных утверждений* (Custom Assertion). Хотя в качестве отправной точки можно воспользоваться и существующими утверждениями (встроенными или специальными), их придется модифицировать для учета предварительно записанных данных.

Пример: дельта-утверждение (Delta Assertion)

В данной версии теста *дельта-утверждение* (Delta Assertion) используется для проверки добавленных объектов. Тест проверяет, появился ли один новый объект и содержит ли возвращаемая тестируемой системой коллекция не только *ожидаемый объект* (Expected Object), но и все объекты, которые были в ней до запуска теста.

```
public void testCreateFlight_Delta()
    throws Exception {
    FlightDto expectedFlightDto =
        createNewFlightBetweenExistingAirports();
    // Записать начальное состояние
    List flightsBeforeCreate =
        facade.getFlightsByOriginAirport(
            expectedFlightDto.getOriginAirportId());
    // Вызвать тестируемую систему
    facade.createFlight(
        expectedFlightDto.getOriginAirportId(),
        expectedFlightDto.getDestinationAirportId());
    // Проверить результат относительно начального состояния
    List flightsAfterCreate =
        facade.getFlightsByOriginAirport(
            expectedFlightDto.getOriginAirportId());
    assertFlightIncludedInDtoList("new flight ",
        expectedFlightDto,
        flightsAfterCreate);
    assertAllFlightsIncludedInDtoList("previous flights",
        flightsBeforeCreate,
        flightsAfterCreate);
    assertEquals("Number of flights after create",
        flightsBeforeCreate.size() + 1,
        flightsAfterCreate.size());
}
```

Поскольку тестируемая система возвращает **объект передачи данных** (data transfer object), можно быть уверенными, что сохраненные перед вызовом тестируемой системы данные не могут измениться. *Специальные утверждения* (Custom Assertion) пришлось модифицировать для игнорирования объектов, существовавших до запуска теста (утверждения не настаивают, что ожидаемый объект является единственным). Новое *специальное утверждение* (Custom Assertion) проверяет наличие существовавших ранее объектов. Еще одним спосо-

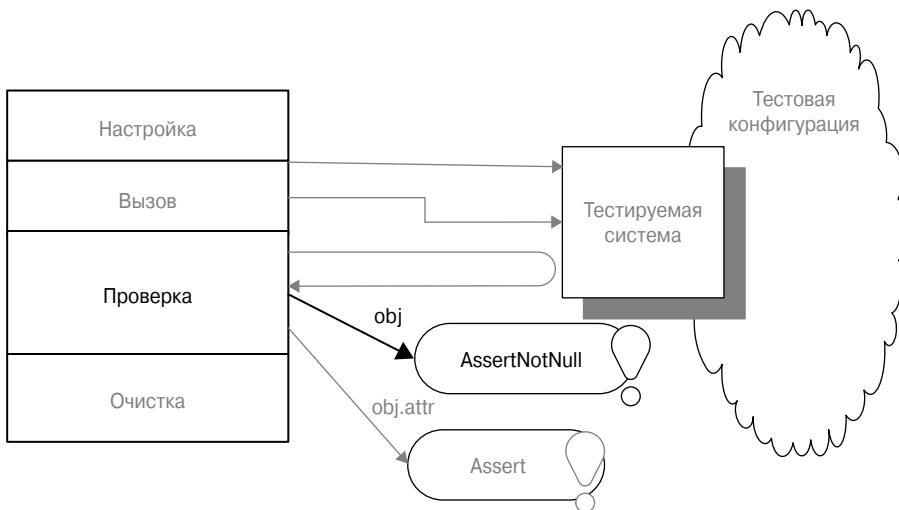
бом решения этой задачи является удаление существовавших объектов из коллекции и проверка, остались ли только *ожидаемые объекты* (Expected Object).

Реализация *специальных утверждений* (Custom Assertion) здесь не приводится, так как она содержит только сравнение объектов, не обязательное для понимания концепции *дельта-утверждений* (Delta Assertion). Апологеты тестирования не замедлят написать *тесты специального утверждения* (Custom Assertion Test) для проверки новых *специальных утверждений* (Custom Assertion).

Сторожевое утверждение (Guard Assertion)

Как избежать использования условной логики теста (Conditional Test Logic)?

Оператор `if` внутри теста заменяется утверждением, которое приводит к неудачному завершению теста, если условие не выполняется.



Некоторые виды логики проверки могут не работать, так как возвращаемая тестируемой системой информация неправильно инициализирована. Если тест сталкивается с неожиданной проблемой, он может завершиться ошибкой вместо неудачного завершения. Хотя *программа запуска тестов* (Test Runner, с. 405) попытается предоставить максимально полезную диагностическую информацию, разработчик может обеспечить лучшую диагностику, проверив конкретное условие и явно сообщив о его нарушении.

Сторожевое утверждение (Guard Assertion) является хорошим способом решения этой проблемы без внесения *условной логики в тесты* (Conditional Test Logic, с. 243).

Как это работает

Тест завершается успешно или неудачно. Для неудачного завершения тестов вызываются *методы с утверждением* (Assertion Method, с. 390), останавливающие выполнение, если утверждение оказывается ложным. Если *условная логика теста* (Conditional Test Logic) используется для обхода утверждений, которые генерируют ошибку, ее можно заменить утверждениями, приводящими к неудачному завершению теста. Кроме того, данный шаблон показывает, что ожидается истинность условия, поставленного в *сторожевом утверждении* (Guard Assertion). Ложность *сторожевого утверждения* (Guard Assertion) показывает, что не выполняется некоторое условие. При этом разработчик избавлен от необходимости угадывать результат теста, рассматривая условную логику.

Когда это использовать

Сторожевое утверждение (Guard Assertion) необходимо использовать всегда, когда нужно избежать выполнения операторов внутри *тестовых методов* (Test Method, с. 378), так как они приводят к появлению ошибки при невыполнении некоторого условия по отношению к возвращаемым тестируемой системой значениям. Данный шаблон позволяет отказаться от использования конструкции `if then else fail` вокруг чувствительных операторов. Обычно *сторожевое утверждение* (Guard Assertion) располагается между вызовом тестируемой системы и проверкой результата в *четырехфазном тесте* (Four-Phase Test, с. 387).

Вариант: утверждение о состоянии общей тестовой конфигурации (Shared Fixture State Assertion)

Если в тесте используется *общая тестовая конфигурация* (Shared Fixture, с. 350), *сторожевое утверждение* (Guard Assertion) может пригодиться в начале теста (перед вызовом тестируемой системы) для проверки соответствия общей конфигурации потребностям теста. Кроме того, читателю становится понятно, какая часть конфигурации часто используется тестом. Это упрощает использование *тестов как документации* (Tests as Documentation, с. 79).

Замечания по реализации

В качестве *сторожевого утверждения* (Guard Assertion) можно использовать *утверждения с заявленным результатом* (Stated Outcome Assertion) (например, `assertNotNil`) и *утверждения равенства* (Equality Assertion) (например, `assertEquals`). Они приводят к неудачному завершению теста и останавливают выполнение других операторов, приводящих к ошибке теста.

Мотивирующий пример

Рассмотрим следующий тест.

```
public void testWithConditionals() throws Exception {
    String expectedLastname = "smith";
    List foundPeople = PeopleFinder.
        findPeopleWithLastname(expectedLastname);
    if (foundPeople != null) {
        if (foundPeople.size() == 1) {
            Person solePerson = (Person) foundPeople.get(0);
            assertEquals(expectedLastname, solePerson.getName());
        } else {
            fail("list should have exactly one element");
        }
    } else {
        fail("list is null");
    }
}
```

В данном примере присутствует достаточное количество условных операторов, в которых можно сделать ошибку. Например, можно написать (`foundPeople == null`)

вместо (`foundPeople != null`). В С-подобных языках можно вписать = вместо ==, что вынудит тест всегда завершаться успешно!

Замечания по рефакторингу

Для преобразования этого спагетти *условной логики теста* (Conditional Test Logic) в линейную последовательность операторов можно воспользоваться рефакторингом *замена вложенных условных операторов граничным оператором* (Replace Nested Conditional with Guard Clauses). (В тестах даже одного условного оператора слишком много. Отсюда и “вложенный”!). Для обнаружения пустых указателей на объекты можно воспользоваться *утверждением с заявленным результатом* (Stated Outcome Assertion), а для проверки количества объектов в коллекции подходят *утверждения равенства* (Equality Assertion). Если все утверждения истинны, тест продолжает работу. Если одно из утверждений оказалось ложным, тест завершается неудачно перед переходом к следующему оператору.

Пример: простое сторожевое утверждение (Simple Guard Assertion)

Ниже приведена упрощенная версия теста после замены всех условных операторов утверждениями. Тест стал короче исходного варианта. Кроме того, его значительно проще читать.

```
public void testWithoutConditionals() throws Exception {
    String expectedLastname = "smith";
    List foundPeople = PeopleFinder.
        findPeopleWithLastname(expectedLastname);
    assertNotNull("found people list", foundPeople);
    assertEquals("number of people", 1, foundPeople.size());
    Person solePerson = (Person) foundPeople.get(0);
    assertEquals("last name",
        expectedLastname,
        solePerson.getName());
}
```

Теперь тест содержит линейный маршрут выполнения, что значительно повышает уверенность в его правильности!

Пример: сторожевое утверждение для общей тестовой конфигурации (Shared Fixture Guard Assertion)

Ниже приведен пример теста, зависящего от *общей тестовой конфигурации* (Shared Fixture). Если предыдущий тест (или предыдущий экземпляр данного теста) модифицирует состояние конфигурации, тестируемая система может вернуть неожиданный результат. Очень сложно определить, что причина проблемы заключается в состоянии конфигурации до запуска теста, а не в ошибке внутри тестируемой системы. Всех этих проблем можно избежать, если предположения теста описать явно с помощью *сторожевого утверждения* (Guard Assertion) на этапе поиска тестовой конфигурации.

```
public void testAddFlightsByFromAirport_OneOutboundFlight_GA()
    throws Exception {
    // Поиск тестовой конфигурации
```

```
List flights = facade.getFlightsByOriginAirport(
    ONE_OUTBOUND_FLIGHT_AIRPORT_ID );
// Сторожевое утверждение о содержимом конфигурации
assertEquals("# flights precondition", 1, flights.size());
FlightDto firstFlight = (FlightDto) flights.get(0);
// Вызов системы
BigDecimal flightNum = facade.createFlight(
    firstFlight.getOriginAirportId(),
    firstFlight.getDestAirportId());
// Проверка результата
FlightDto expFlight = (FlightDto) firstFlight.clone();
expFlight.setFlightNumber(flightNum);
List actual = facade.getFlightsByOriginAirport(
    firstFlight.getOriginAirportId());
assertExactly2FlightsInDtoList("Flights at origin",
    firstFlight,
    expFlight,
    actual);
}
```

Теперь существует способ определения ложности предположений без трудоемкой отладки! Это еще один из способов обеспечения локализации дефектов (Defect Localization, с. 78). В этом случае дефект заключается в предположениях теста о поведении предыдущих тестов.

Утверждение незаконченного теста (Unfinished Test Assertion)

Как структурировать тестовую логику, чтобы тест не остался незаконченным?

Неудачное завершение незаконченного теста обеспечивается гарантированно ложным утверждением.

```
void testSomething() {
    // Шаблон:
    // создать перелет в ... состоянии
    // вызвать ... метод
    // проверить, что перелет в ... состоянии
    fail("Unfinished Test!");
}
```

Приступая к описанию теста для конкретного фрагмента кода, полезно “очертить” тест, определив *тестовый метод* (Test Method, с. 378) в соответствующем *классе теста* (Testcase Class, с. 401), и после этого обдумать тестовые условия. Но желательно оставить напоминание, чтобы тесты не остались пустыми, если что-то отвлечет от работы. Такие тесты должны завершаться неудачно до их полной реализации.

Включение *утверждения незаконченного теста* (Unfinished Test Assertion) может стать хорошим напоминанием.

Как это работает

В каждый определяемый *тестовый метод* (Test Method) вставляется вызов `fail`. Метод `fail` представляет собой *утверждение с единственным результатом* (Single Outcome Assertion), которое всегда приводит к неудачному завершению теста. Кроме того, в вызов передается *сообщение для утверждения* (Assertion Message, с. 398) “Unfinished Test”, объясняющее причину неудачного завершения теста.

Когда это использовать

Нельзя намеренно писать тесты, которые могут случайно завершиться успешно. Неудачно завершившийся тест служит хорошим напоминанием о незаконченной работе. Приступая к созданию теста, стоит добавить к нему *утверждение незаконченного теста* (Unfinished Test Assertion) и удалить его только после полного завершения работы над тестом. Сделать это совсем не сложно, но это имеет множество преимуществ. Достаточно выработать соответствующую привычку. В некоторых средах разработки поддерживается возможность добавления *утверждения незаконченного теста* (Unfinished Test Assertion) в шаблон генерации кода *тестового метода* (Test Method).

Если тесты необходимо включить в хранилище кода до того, как код стал полностью работоспособным, не стоит удалять тесты или *утверждение незаконченного теста* (Unfinished Test Assertion) только для получения зеленого индикатора, так как это приведет к *потере теста* (Lost Test). Вместо этого тесту можно назначить атрибут `[Ignore]` (если такая возможность поддерживается используемой реализацией xUnit), переименовать тест (если используется механизм *обнаружения тестов* (Test Discovery, с. 420) на основе имен) или исключить весь *класс теста* (Testcase Class) из *набора всех тестов* (AllTests

Suite), если используется механизм *перечисления тестов* (Test Enumeration, с. 425) на уровне наборов.

Замечания по реализации

В большинстве реализаций xUnit метод `fail` уже определен. Если в применяемом пакете такой метод отсутствует, избегайте вставки `assertTrue(false)` в разные фрагменты кода. Это глупое и интуитивно непонятное решение. Вместо этого необходимо самостоятельно определить метод в виде *специального утверждения* (Custom Assertion, с. 495) и написать *тест специального утверждения* (Custom Assertion Test), который проверяет правильность его работы.

В некоторых средах разработки можно модифицировать шаблоны генерации кода. И в некоторых из них даже существует шаблон *тестового метода* (Test Method), включающий *утверждение незаконченного теста* (Unfinished Test Assertion).

Мотивирующий пример

Рассмотрим следующий *класс теста* (Testcase Class).

```
public void testPull_emptyStack() {
}
public void testPull_oneItemOnStack () {
}
public void testPull_twoItemsOnStack () {
    //To do: написать этот тест
}
public void testPull_oneItemsRemainingOnStack () {
    //To do: написать этот тест
}
```

Комментарии `// To do: ...` могут напомнить, что тест еще не завершен, если среда разработки поддерживает такую возможность. Но это напоминание не появится во время запуска тестов. После запуска этого *класса теста* (Testcase Class) появится зеленый индикатор, хотя проверяемый стек даже не реализован!

Замечания по рефакторингу

Для реализации *утверждения незаконченного теста* (Unfinished Test Assertion) достаточно добавить следующую строку в каждый тест.

```
fail("Unfinished Test!");
```

Восклицательный знак не обязательен. Возможно, лучше создать следующее *специальное утверждение* (Custom Assertion).

```
private void unfinishedTest() {
    fail("Test not implemented!");
}
```

Это позволит найти все *утверждения незаконченного теста* (Unfinished Test Assertion) с помощью функции поиска ссылок в среде разработки.

Пример: утверждение незаконченного теста (Unfinished Test Assertion)

Ниже приведены те же тесты с добавлением *утверждения незаконченного теста* (Unfinished Test Assertion).

```
public void testPull_emptyStack() {
    unfinishedTest();
}
public void testPull_oneItemOnStack () {
    unfinishedTest();
}
public void testPull_twoItemsOnStack() {
    unfinishedTest();
}
public void testPull_oneItemsRemainingOnStack () {
    unfinishedTest();
}
```

В результате получен *класс теста* (Testcase Class), который будет гарантированно завершаться неудачно, пока код не будет завершен. Неудачно завершающиеся тесты служат списком незаконченных заданий.

Пример: генерация незаконченного метода теста на основе шаблона

Примером среды разработки с поддержкой модифицируемых шаблонов генерации кода может служить Eclipse (версии 3.0). Его шаблон `testmethod` вставляет в *класс теста* (Testcase Class) следующий код.

```
public void testname() throws Exception {
    fail("ClassName::testname not implemented");
}
```

Строки `ClassName` и `testname` являются шаблонами для имен *класса теста* (Testcase Class) и *тестового метода* (Test Method). Они заполняются автоматически средой разработки. Модификация имени теста в сигнатуре приводит к автоматическому изменению имени теста в операторе `fail`. Для добавления нового тестового метода (Test Method) в класс достаточно ввести `testmethod` и нажать `<Ctrl+Пробел>`.

Глава 22

Шаблоны очистки тестовой конфигурации

Шаблоны в этой главе:

Стратегии очистки

Очистка со сборкой мусора (Garbage-Collected Teardown)	518
Автоматическая очистка (Automated Teardown)	521

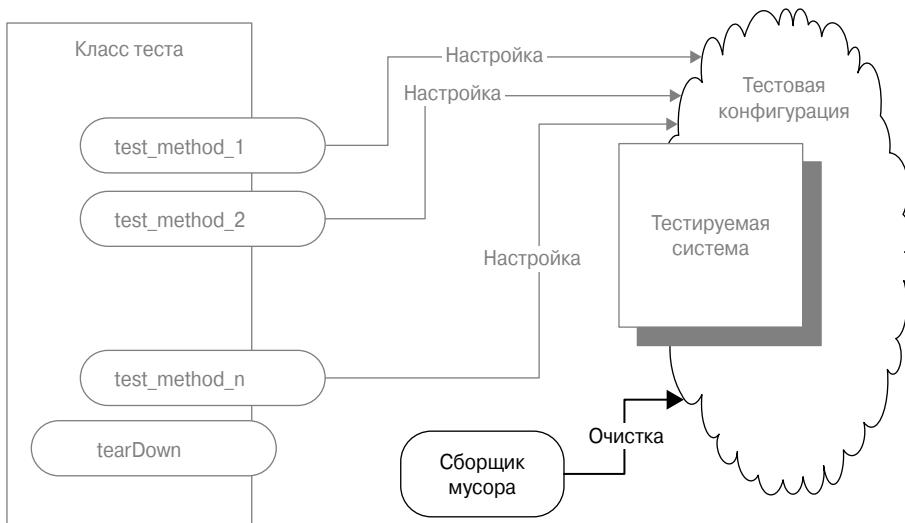
Организация кода

Встроенная очистка (In-line Teardown)	527
Неявная очистка (Implicit Teardown).....	533

Очистка со сборкой мусора (Garbage-Collected Teardown)

Как удалить тестовую конфигурацию?

Очистка оставшейся после завершения теста конфигурации выполняется механизмом сборки мусора, предоставляемым языком программирования.



Повторяемость тестов в значительной мере зависит от очистки тестовой конфигурации после завершения каждого теста перед созданием новой конфигурации для следующего теста. Такая стратегия называется *новой тестовой конфигурацией* (Fresh Fixture, с. 344). В языках с поддержкой **сборки мусора** (garbage collection) большая часть процедуры очистки выполняется автоматически, если ссылки на ресурсы хранятся в локальных переменных и переменных экземпляров.

Как это работает

Многие объекты, создаваемые во время работы тестов (включая настройку тестовой конфигурации и вызов тестируемой системы), являются временными и существуют до тех пор, пока существуют ссылки на них в создавшей их программе. Механизмы сборки мусора в современных языках используют различные алгоритмы для обнаружения “мусора”. Но самое важное — это распознавание объектов, не являющихся мусором: любой объект, достижимый из другого “живого” объекта или глобальной (т.е. статической) переменной, не попадает под действие механизма сборки мусора.

При запуске тестов *инфраструктура автоматизации тестов* (Test Automation Framework, с. 332) создает *объект теста* (Testcase Object, с. 410) для каждого *тестового метода* (Test Method, с. 378) в пределах *класса теста* (Testcase Class, с. 401) и добавляет эти объекты в *объект набора тестов* (Test Suite Object, с. 414). При каждом запуске тестов инфраструктура избавляется от существующего набора и создает новый. После удаления ста-

рого набора тестов все объекты, на которые ссылаются переменные экземпляров этих тестов, становятся кандидатами на удаление механизмом сборки мусора.

Когда это использовать

Очистка со сборкой мусора (Garbage-Collected Teardown) должна использоваться везде, где это возможно, так как она значительно упрощает процесс очистки!

Если разработка выполняется в среде без поддержки сборки мусора или используются ресурсы, не поддающиеся автоматической сборке мусора (например, файлы, сокеты, записи в базе данных), приходится удалять или освобождать эти ресурсы явно. При использовании *общей тестовой конфигурации* (Shared Fixture, с. 350) *очистка со сборкой мусора* (Garbage-Collected Teardown) недоступна, если не предпринять специальных мер для хранения ссылки на конфигурацию таким способом, чтобы она выходила из области видимости после завершения работы набора тестов.

В таких ситуациях для освобождения ресурсов можно воспользоваться *встроенной очисткой* (In-line Teardown, с. 527), *неявной очисткой* (Implicit Teardown, с. 533) или *автоматической очисткой* (Automated Teardown, с. 521).

Замечания по реализации

В некоторых реализациях xUnit и средах разработки происходит перезагрузка классов при каждом запуске набора тестов. Такое поведение может быть обозначено как функция “Reload Classes” или происходит принудительно. Необходимо соблюдать осторожность, если принято решение использовать эту функцию для *очистки со сборкой мусора* (Garbage-Collected Teardown) при хранении тестовой конфигурации в переменных класса, так как тесты могут перестать работать при смене среды разработки или при запуске тестов из командного интерпретатора (например, с помощью продукта “Cruise Contol” или из сценария компиляции).

Мотивирующий пример

В следующем teste на этапе настройки тестовой конфигурации в памяти создаются объекты, которые явно уничтожаются с помощью *встроенной очистки* (In-line Teardown). (В этом примере можно воспользоваться и *неявной очисткой* (Implicit Teardown), но это усложнит понимание происходящего.)

```
public void testCancel_proposed_UT() {
    // Настройка конфигурации
    Flight proposedFlight = createAnonymousProposedFlight();
    // Вызов тестируемой системы
    proposedFlight.cancel();
    // Проверка результата
    try{
        assertEquals(FlightState.CANCELLED,
                    proposedFlight.getStatus());
    } finally {
        // Очистка
        proposedFlight.delete();
        proposedFlight = null;
    }
}
```

Поскольку объекты не являются постоянными, код удаления `proposedFlight` не обязателен; он лишь усложняет тест.

Замечания по рефакторингу

Для использования *очистки со сборкой мусора* (Garbage-Collected Teardown) достаточно удалить лишний код очистки. Если бы для хранения ссылки на конфигурацию использовалась переменная класса, ее пришлось бы преобразовать в переменную экземпляра или локальную переменную, каждая из которых обеспечивает переход от *общей тестовой конфигурации* (Shared Fixture) к *новой тестовой конфигурации* (Fresh Fixture).

Пример: очистка со сборкой мусора (Garbage-Collected Teardown)

В переработанном тесте используется *очистка со сборкой мусора* (Garbage-Collected Teardown).

```
public void testCancel_proposed_GCT() {
    // Настройка тестовой конфигурации
    Flight proposedFlight = createAnonymousProposedFlight();
    // Вызов тестируемой системы
    proposedFlight.cancel();
    // Проверка результата
    assertEquals(FlightState.CANCELLED,
                proposedFlight.getStatus());
    // Очистка
    // Сборщик мусора удаляет proposedFlight при выходе
    // из области видимости
}
```

Обратите внимание, насколько упростился тест!

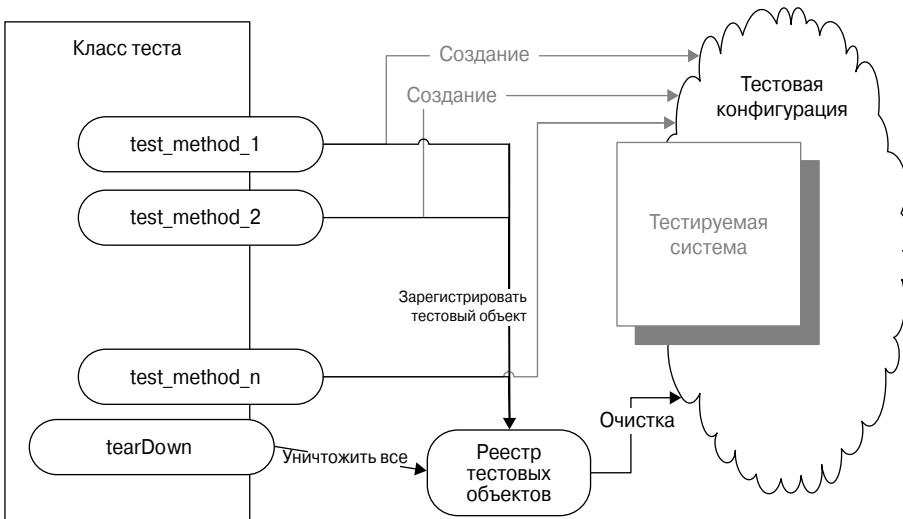
Автоматическая очистка (Automated Teardown)

Как удалить тестовую конфигурацию?

Все создаваемые тестом ресурсы отслеживаются и автоматически удаляются и/или освобождаются в процессе очистки.

Также известен как:

*Реестр тестовых объектов
(Test Object Registry)*



Повторяемость тестов в значительной мере зависит от отчистки тестовой конфигурации после завершения каждого теста перед созданием новой конфигурации для следующего теста. Такая стратегия называется *новой тестовой конфигурацией* (Fresh Fixture, с. 344). Оставшиеся объекты и записи в базе данных, а также открытые файлы и подключения могут привести как минимум к снижению быстродействия, а как максимум — к аварийному завершению работы системы или неудачному завершению теста. Некоторые из этих ресурсов можно освободить автоматически с помощью механизма сборки мусора, но часть из них приходится удалять явно.

Для создания надежного кода очистки, работающего во всех возможных ситуациях, требуется много усилий и времени. Необходимо понять, что может остаться после каждого возможного результата теста, и написать код для обработки каждого сценария. Подобная *сложная очистка* (Complex Teardown; см. *Непонятный тест*, Obscure Test, с. 230) приводит к появлению большого объема *условной логики теста* (Conditional Test Logic, с. 243) и, что хуже всего, к появлению *нетестируемого кода тестов* (Untestable Test Code; см. *Сложный в тестировании код*, Hard-to-Test Code, с. 251).

Наилучшим решением является отслеживание создаваемых объектов и их автоматическое удаление после завершения работы теста.

Как это работает

В основе решения лежит механизм регистрации каждого постоянного элемента (т.е. объекта, записи, подключения и т.д.), создаваемого во время работы теста. Для этого составляется список (или списки) зарегистрированных объектов, требующих определенных операций для их удаления. Механизм может быть не сложнее добавления объектов в коллекцию. После завершения теста коллекция перебирается и каждый объект удаляется. Все ошибки при этом должны перехватываться, чтобы проблема при удалении одного объекта не приводила к отмене удаления остальных.

Когда это использовать

Автоматическая очистка (Automated Teardown) может использоваться тогда, когда существуют постоянные ресурсы, требующие удаления для нормальной работы тестовой среды. (В приемочных тестах такая ситуация встречается чаще, чем в модульных.) Этот шаблон относится как к *неповторяемым тестам* (Unrepeatable Test), так и к *взаимодействующим тестам* (Interacting Tests), защищая среду от объектов, существующих дольше, чем выполняется тест.

Создать механизм *автоматической очистки* (Automated Teardown) совсем не сложно, а результат работы механизма позволит сэкономить много времени и усилий. Каждый такой механизм, созданный для одного проекта, можно использовать и в последующих.

Замечания по реализации

Автоматическая очистка (Automated Teardown) имеет два варианта. Базовый вариант удаляет только объекты, которые создавались на этапе генерации тестовой конфигурации. Более сложная версия удаляет все объекты, созданные тестируемой системой во время взаимодействия с тестом.

Вариант: автоматическое удаление тестовой конфигурации (Automated Fixture Teardown)

Самым простым решением является регистрация новых объектов в *методах создания* (Creation Method, с. 441). Хотя такой шаблон и не позволяет удалять объекты, создаваемые тестируемой системой, работа с конфигурацией все равно снижает вероятность возникновения ошибок.

С таким вариантом связаны две сложности:

- поиск универсального способа удаления зарегистрированных объектов;
- обеспечение запуска кода *автоматической очистки* (Automated Teardown) для каждого зарегистрированного объекта.

Учитывая, что вторую сложность обойти проще, начнем с нее. Для удаления *постоянной новой тестовой конфигурации* (Persistent Fresh Fixture) проще всего вставить вызов механизма *автоматической очистки* (Automated Teardown) в метод `tearDown` класса теста (Testcase Class, с. 401). Этот метод вызывается независимо от результата запуска теста, пока метод `setUp` завершается успешно. При удалении *общей тестовой конфигурации* (Shared Fixture, с. 350) метод `tearDown` должен вызываться только после завершения

всех *тестовых методов* (Test Method, с. 378). В таком случае можно воспользоваться шаблоном *настройки тестовой конфигурации набора* (Suite Fixture Setup, с. 465) (если он поддерживается используемой реализацией xUnit) или *декоратором настройки* (Setup Decorator, с. 471).

Вернемся к более сложной проблеме: созданию универсального механизма освобождения ресурсов. Здесь доступны два варианта. Во-первых, можно принудить все постоянные (не подверженные сборке мусора) объекты реализовать универсальный механизм очистки, вызываемый из механизма *автоматической очистки* (Automated Teardown). Во-вторых, каждый объект можно “завернуть” в другой объект, знающий, как его удалить. Вторая стратегия является примером реализации шаблона Command.

Если механизм *автоматической очистки* (Automated Teardown) реализован универсально, его можно включить в *суперкласс теста* (Testcase Superclass, с. 646), на основе которого создаются все *классы теста* (Testcase Class). В противном случае его придется разместить во *вспомогательном классе теста* (Test Helper, с. 651), видимом из всех нуждающихся в нем *классах теста* (Testcase Class). *Вспомогательный класс теста* (Test Helper), создающий объекты тестовой конфигурации и удаляющий их автоматически, иногда называется *инкубатором объектов* (Object Mother).

Поскольку это нетривиальный (и критический) фрагмент кода, механизм *автоматической очистки* (Automated Teardown) нуждается в собственных модульных тестах. Так как механизм находится за пределами *тестовых методов* (Test Method), для него можно создавать *самопроверяющиеся тесты* (Self-Checking Test, с. 81)! При достаточном уровне внимательности (или параноидальности, как считают некоторые) можно воспользоваться *дельта-утверждениями* (Delta Assertion, с. 505) для проверки того, что оставшиеся после завершения очистки объекты существовали еще до запуска теста.

Вариант: автоматическое удаление результатов (Automated Exercise Teardown)

Тесты можно сделать еще более “самоочищающимися”, удаляя объекты, созданные тестируемой системой. Для этого тестируемую систему придется проектировать с использованием наблюдаемой *фабрики объектов* (Object Factory; см. *Поиск зависимости*, Dependency Lookup, с. 692), чтобы можно было автоматически регистрировать любые объекты, создаваемые тестируемой системой во время работы. Эти объекты можно удалять во время фазы очистки конфигурации.

Мотивирующий пример

В данном примере с помощью *методов создания* (Creation Method) генерируется несколько объектов. Их придется удалить после выполнения теста. Для этого используется блок *try/finally*, обеспечивающий запуск кода *встроенной очистки* (In-line Teardown, с. 527) независимо от истинности утверждений теста.

```
public void testGetFlightsByOrigin_NoInboundFlight_SMRTD()
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("10F");
    BigDecimal inboundAirport = null;
    FlightDto expFlightDto = null;
    try {
        inboundAirport = createTestAirport("11F");
```

```
expFlightDto = createTestFlight(outboundAirport, inboundAirport);
// Вызов системы
List flightsAtDestination1 =
    facade.getFlightsByOriginAirport(inboundAirport);
// Проверка результата
assertEquals(0, flightsAtDestination1.size());
} finally {
    try {
        facade.removeFlight(expFlightDto.getFlightNumber());
    } finally {
        try {
            facade.removeAirport(inboundAirport);
        } finally {
            facade.removeAirport(outboundAirport);
        }
    }
}
```

Обратите внимание на необходимость использования вложенных конструкций `try/finally` внутри блока `finally` для защиты логики очистки от возможных ошибок.

Замечания по рефакторингу

Автоматическая очистка (Automated Teardown) внедряется в два этапа. Во-первых, необходимо добавить механизм автоматической очистки в класс теста (Testcase Class). Во-вторых, из тестов необходимо удалить весь код *встроенной очистки* (In-line Teardown).

Автоматическая очистка (Automated Teardown) может быть реализована в конкретном классе теста (Testcase Class) или может быть унаследована (или включена в виде миксина) от универсального класса. В любом случае необходимо обеспечить регистрацию новых объектов, чтобы механизм мог удалить их после завершения работы тестов. Проще всего это сделать внутри уже существующих методов создания (Creation Method). С другой стороны, можно воспользоваться рефакторингом выделение метода (Extract Method) для перемещения непосредственных вызовов конструктора в новые методы создания (Creation Method), после чего добавить в них логику регистрации.

Универсальный механизм *автоматической очистки* (Automated Teardown) должен вызываться из метода `tearDown`. Хотя это можно сделать и в собственном классе *теста* (Testcase Class), лучше вынести этот метод в *суперкласс теста* (Testcase Superclass), который наследуется всеми классами *теста* (Testcase Class). Если *суперкласс теста* (Testcase Superclass) еще не существует, его можно легко создать с помощью рефакторинга *выделение класса* (Extract Class) с последующим рефакторингом *подъем метода* (Pull up method) по отношению ко всем методам (и полям), связанным с механизмом *автоматической очистки* (Automated Teardown).

Пример: автоматическая очистка (Automated Teardown)

После рефакторинга в этом тесте не осталось ничего интересного, так как весь код очистки был удален.

```

public void testGetFlightsByOriginAirport_OneOutboundFlight()
throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("1OF");
    BigDecimal inboundAirport = createTestAirport("1IF");
    FlightDto expectedFlightDto =
        createTestFlight( outboundAirport, inboundAirport );
    // Вызов тестируемой системы
    List flightsAtOrigin =
        facade.getFlightsByOriginAirport(outboundAirport);
    // Проверка результата
    assertOnly1FlightInDtoList("Flights at origin",
        expectedFlightDto,
        flightsAtOrigin);
}

```

Вот где выполняется вся работа! *Метод создания* (Creation Method) был модифицирован для регистрации новых объектов.

```

private List allAirportIds;
private List allFlights;
protected void setUp() throws Exception {
    allAirportIds = new ArrayList();
    allFlights = new ArrayList();
}
private BigDecimal createTestAirport(String airportName)
throws FlightBookingException {
    BigDecimal newAirportId = facade.createAirport(
        airportName, " Airport" + airportName,
        "City" + airportName);
    allAirportIds.add(newAirportId);
    return newAirportId;
}

```

Ниже показана собственно логика *автоматической очистки* (Automated Teardown). В этом примере она расположена в *классе теста* (Testcase Class) и вызывается из метода *tearDown*. Чтобы не усложнять пример, логика специально предназначена для обработки объектов аэропортов и перелетов. В типичной реализации логика располагалась бы в *суперклассе теста* (Testcase Superclass), где она может использоваться всеми *классами теста* (Testcase Class), а для удаления тестовой конфигурации использовался бы универсальный механизм, не зависящий от типа объектов.

```

protected void tearDown() throws Exception {
    removeObjects(allAirportIds, "Airport");
    removeObjects(allFlights, "Flight");
}
public void removeObjects(List objectsToDelete, String type) {
    Iterator i = objectsToDelete.iterator();
    while (i.hasNext()) {
        try {
            BigDecimal id = (BigDecimal) i.next();
            if ("Airport"==type) {
                facade.removeAirport(id);
            } else {
                facade.removeFlight(id);
            }
        }
    }
}

```

```
        } catch (Exception e) {
            // Ничего не делать, если удаление не выполнено
        }
    }
}
```

При очистке общей тестовой конфигурации (Shared Fixture) метод tearDown получил бы подходящий атрибут либо аннотацию (например, @afterClass или [TestFixtureTear-Down]) или был бы перемещен в декоратор настройки (Setup Decorator).

Пример: автоматическое удаление результатов (Automated Exercise Teardown)

Следующий шаг предполагает автоматическую очистку всех объектов, созданных тестируемой системой. Для этого систему необходимо модифицировать для использования наблюдаемой *фабрики объектов* (Object Factory). В тесты придется добавить следующий код.

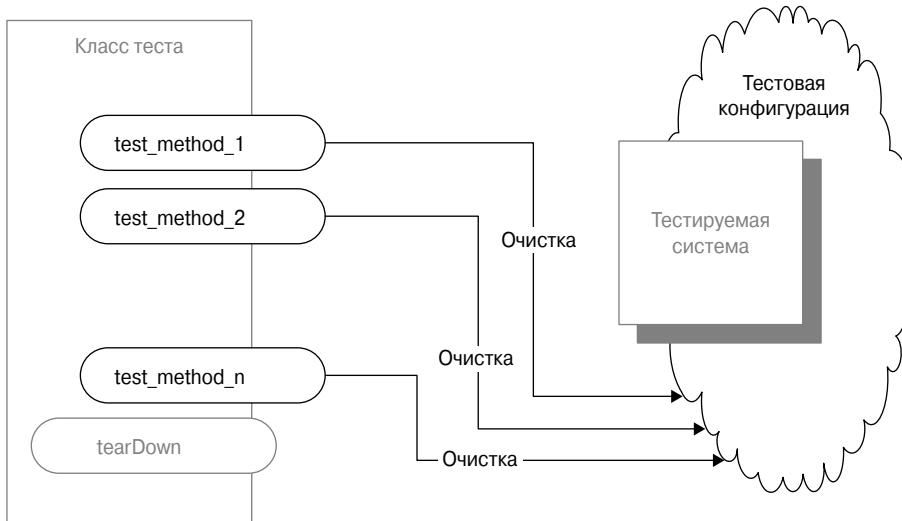
```
ResourceTracker tracker;
public void setUp() {
    tracker = new ResourceTracker();
    ObjectFactory.addObserver(tracker);
}
public void tearDown() {
    tracker.cleanup();
    ObjectFactory.removeObserver(tracker);
}
```

В последнем примере предполагается, что логика *автоматической очистки* (Automated Teardown) перемещена в метод cleanup класса ResourceTracker.

Встроенная очистка (In-line Teardown)

Как удалить тестовую конфигурацию?

Логика очистки вставляется в *тестовый метод* (Test Method) непосредственно после проверки результата.



Повторяемость тестов в значительной мере зависит от очистки тестовой конфигурации после завершения каждого теста перед созданием новой конфигурации для следующего теста. Такая стратегия называется *новой тестовой конфигурацией* (Fresh Fixture, с. 344). Оставшиеся объекты и записи в базе данных, а также открытые файлы и подключения могут привести как минимум к снижению быстродействия, а как максимум — к аварийному завершению работы системы или неудачному завершению теста. Некоторые из этих ресурсов можно освободить автоматически с помощью механизма сборки мусора, но часть из них приходится удалять явно.

Как минимум необходимо написать код *встроенной очистки* (In-line Teardown), который удаляет ресурсы, оставшиеся после выполнения теста.

Как это работает

При создании теста разработчик мысленно отслеживает все созданные тестом объекты, которые не будут удаляться автоматически. После написания кода вызова тестируемой системы и проверки результата в конец *тестового метода* (Test Method, с. 378) добавляется логика удаления всех объектов, которые не удаляются механизмом сборки мусора. При этом используются подходящие конструкции языка, обеспечивающие запуск логики очистки вне зависимости от результата работы теста.

Когда это использовать

Подходящая логика очистки должна использоваться каждый раз, когда существуют ресурсы, не освобождаемые автоматически после завершения *тестового метода* (Test Method). Если каждый тест использует различные типы объектов, можно воспользоваться *встроенной очисткой* (In-line Teardown). Необходимость удаления объектов может быть связана с появлением *неповторяемых* (Unrepeatable Test) и *медленных тестов* (Slow Tests, с. 289) в результате накопления остатков объектов после многократного запуска тестов.

В отличие от настройки тестовой конфигурации логика очистки не важна с точки зрения использования *тестов как документации* (Tests as Documentation, с. 79). Использование любой формы логики очистки может потенциально привести к *высокой стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300), и ее применения по возможности стоит избегать. Таким образом, единственным реальным преимуществом включения логики очистки в код теста является простота ее обслуживания (прямо скажем, не очень большая выгода). Практически всегда лучше использовать *автоматическую очистку* (Automated Teardown, с. 521) или *неявную очистку* (Implicit Teardown, с. 533), если тесты организованы в виде *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639) и все *классы теста* (Testcase Class, с. 401) используют одну и ту же тестовую конфигурацию.

Кроме того, *встроенная очистка* (In-line Teardown) может использоваться как промежуточный этап во время перехода к *неявной очистке* (Implicit Teardown). Сначала рассматривается возможность использования *встроенной очистки* (In-line Teardown) в каждом *тестовом методе* (Test Method), после чего общая логика выделяется в метод `tearDown`. *Встроенная очистка* (In-line Teardown) не должна использоваться, если создаваемые тестом объекты находятся под контролем автоматического механизма управления памятью. В таком случае лучше использовать *очистку со сборкой мусора* (Garbage-Collected Teardown, с. 518), так как при этом снижается количество возможных ошибок, а тесты становятся проще для понимания и обслуживания.

Замечания по реализации

Основной задачей *встроенной очистки* (In-line Teardown) является обеспечение запуска логики очистки, даже если тест завершился неудачно из-за ложного утверждения или ошибки (в тестовом методе или в тестируемой системе). Второй задачей является обеспечение защиты от дополнительных ошибок, которые может внести логика очистки.

Ключевым условием успешного завершения *встроенной очистки* (In-line Teardown) является использование языковых конструкций, обеспечивающих безусловный запуск кода очистки. В большинстве современных языков доступны конструкции обработки ошибок и/или исключений, которые пытаются запустить первый блок кода и гарантируют запуск второго блока независимо от результатов работы первого. В языке Java данная конструкция выглядит как комбинация блоков `try` и `finally`.

Вариант: пункт охраны очистки (Teardown Guard Clause)

Для защиты от неудачного завершения, вызванного попыткой очистки несуществующего ресурса, логику можно спрятать в “пункт охраны”. Это уменьшит вероятность завершения теста из-за ошибки в логике доставки.

Вариант: делегированная очистка (Delegated Teardown)

Большую часть логики очистки можно вынести из *тестового метода* (Test Method) во *вспомогательный метод теста* (Test Utility Method, с. 610). Хотя такая стратегия и позволяет сократить объем логики очистки в пределах теста, конструкции обработки ошибок останутся в тестовом методе вокруг утверждений и вызова тестируемой системы для обеспечения вызова вспомогательного метода. Практически всегда наилучшим решением является использование *неявной очистки* (Implicit Teardown).

Вариант: простейшая встроенная очистка (Naive In-line Teardown)

Так называется реализация очистки, в которой разработчик забывает добавить аналог блока `try/finally` вокруг логики теста, обеспечивающий безусловный запуск логики очистки. Такая ошибка приводит к *утечке ресурсов* (Resource Leakage), а утечка — к возникновению *нестабильных тестов* (Erratic Test).

Мотивирующий пример

В следующем teste постоянный объект (`airport`) создается как часть конфигурации. Поскольку объект хранится в базе данных, он не подвержен *очистке со сборкой мусора* (Garbage-Collected Teardown) и должен удаляться явно. Если тест не содержит логику очистки, при каждом запуске теста в базе данных создается новый объект. Если не использовать *отдельное сгенерированное значение* (Distinct Generated Value), это может привести к появлению *неповторяющегося теста* (Unrepeatable Test), так как объекты могут нарушить ограничения уникальности ключей.

```
public void testGetFlightsByOriginAirport_NoFlights_ntd()
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("10F");
    // Вызов тестируемой системы
    List flightsAtDestination1 =
        facade.getFlightsByOriginAirport(outboundAirport);
    // Проверка результата
    assertEquals(0, flightsAtDestination1.size());
}
```

Пример: простейшая встроенная очистка (Naive In-line Teardown)

В данном примере простейшей очистки после утверждения добавляется строка удаления созданного объекта `airport`.

```
public void testGetFlightsByOriginAirport_NoFlights()
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("10F");
    // Вызов тестируемой системы
    List flightsAtDestination1 =
        facade.getFlightsByOriginAirport(outboundAirport);
    // Проверка результата
    assertEquals(0, flightsAtDestination1.size());
    facade.removeAirport(outboundAirport);
}
```

К сожалению, это не решает поставленную задачу, так как логика очистки не будет работать, если в тестируемой системе возникнет ошибка или утверждение окажется ложным. Очистку тестовой конфигурации можно вставить перед утверждениями, но это не решит проблему ошибок, возникающих внутри тестируемой системы. Очевидно, требуется более универсальное решение.

Замечания по рефакторингу

Необходимо или добавить конструкцию обработки ошибок вокруг вызова тестируемой системы и утверждений, или вынести код очистки в метод `tearDown`. В любом случае следует обеспечить безусловное выполнение всего кода очистки, даже если некоторые его фрагменты завершаются неудачно. Обычно для этого нужно использовать структуру `try/finally` вокруг каждого шага процесса очистки.

Пример: встроенная очистка (In-line Teardown)

В этом примере на языке Java блок `try/finally` содержит вызов тестируемой системы и проверку результатов, обеспечивая работу кода очистки независимо от результатов вызова и проверки.

```
public void testGetFlightsByOriginAirport_NoFlights_td()
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("1OF");
    try {
        // Вызов тестируемой системы
        List flightsAtDestination1 =
            facade.getFlightsByOriginAirport(outboundAirport);
        // Проверка результата
        assertEquals(0, flightsAtDestination1.size());
    } finally {
        facade.removeAirport(outboundAirport);
    }
}
```

Теперь вызов тестируемой системы и утверждения расположены в блоке `try`, а логика очистки — в блоке `finally`. Это разделение очень важно для обеспечения работы *встроенной очистки* (In-line Teardown). Блок `catch` должен использоваться только в *тестах на ожидаемое исключение* (Expected Exception Test).

Пример: пункт охраны очистки (Teardown Guard Clause)

В данном случае в код очистки добавлен *пункт охраны очистки* (Teardown Guard Clause), чтобы предотвратить выполнение кода для несуществующего объекта `airport`.

```
public void testGetFlightsByOriginAirport_NoFlights_TDGC()
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("1OF");
    try {
        // Вызов тестируемой системы
```

```
        List flightsAtDestination1 =
            facade.getFlightsByOriginAirport(outboundAirport);
        // Проверка результата
        assertEquals(0, flightsAtDestination1.size());
    } finally {
        if (outboundAirport!=null) {
            facade.removeAirport(outboundAirport);
        }
    }
}
```

Пример: встроенная очистка нескольких ресурсов (Java)

Если в одном и том же тесте удаляется несколько ресурсов, необходимо обеспечить работу всего кода очистки независимо от ошибок в некоторых фрагментах. Эта цель достигается вложением каждого фрагмента в конструкцию гарантированного запуска кода, как в примере на языке Java, показанном ниже.

```
public void testGetFlightsByOrigin_NoInboundFlight_SMRTD() {
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("1OF");
    BigDecimal inboundAirport = null;
    FlightDto expFlightDto = null;
    try {
        inboundAirport = createTestAirport("1IF");
        expFlightDto = createTestFlight(outboundAirport, inboundAirport);
        // Вызов тестируемой системы
        List flightsAtDestination1 =
            facade.getFlightsByOriginAirport(inboundAirport);
        // Проверка результата
        assertEquals(0, flightsAtDestination1.size());
    } finally {
        try {
            facade.removeFlight(expFlightDto.getFlightNumber());
        } finally {
            try {
                facade.removeAirport(inboundAirport);
            } finally {
                facade.removeAirport(outboundAirport);
            }
        }
    }
}
```

Данный подход значительно усложняет код, если количество освобождаемых ресурсов достаточно велико. В такой ситуации ресурсы имеет смысл организовать в массив или список с последующим перебором их элементов. В результате получится наполовину реализованная *автоматическая очистка* (Automated Teardown).

Пример: делегированная очистка (Delegated Teardown)

Очистку можно делегировать и из *тестового метода* (Test Method), если не удается создать полностью универсальную стратегию очистки, подходящую для всех тестов.

```

public void testGetFlightsByOrigin_NoInboundFlight_DTD()
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("1OF");
    BigDecimal inboundAirport = null;
    FlightDto expectedFlightDto = null;
    try {
        inboundAirport = createTestAirport("1IF");
        expectedFlightDto =
            createTestFlight( outboundAirport, inboundAirport );
        // Вызов тестируемой системы
        List flightsAtDestination1 =
            facade.getFlightsByOriginAirport(inboundAirport);
        // Проверка результата
        assertEquals(0, flightsAtDestination1.size());
    } finally {
        teardownFlightAndAirports(outboundAirport,
                                   inboundAirport,
                                   expectedFlightDto);
    }
}

private void teardownFlightAndAirports(
    BigDecimal firstAirport,
    BigDecimal secondAirport,
    FlightDto flightDto)
    throws FlightBookingException {
    try {
        facade.removeFlight(flightDto.getFlightNumber());
    } finally {
        try {
            facade.removeAirport(secondAirport);
        } finally {
            facade.removeAirport(firstAirport);
        }
    }
}

```

Склонные к оптимизации разработчики заметят, что номера двух перелетов доступны в виде атрибутов объекта `flightDto`. Склонные к паранойе ответят, что из-за возможности вызова `teardownFlightAndAirports` до создания `flightDto`, на него нельзя рассчитывать для доступа к объектам `Airport`. Таким образом, объекты аэропортов приходится передавать отдельными параметрами. Необходимость таких размышлений является причиной привлекательности *автоматической очистки* (Automated Teardown) — она позволяет вообще не думать!

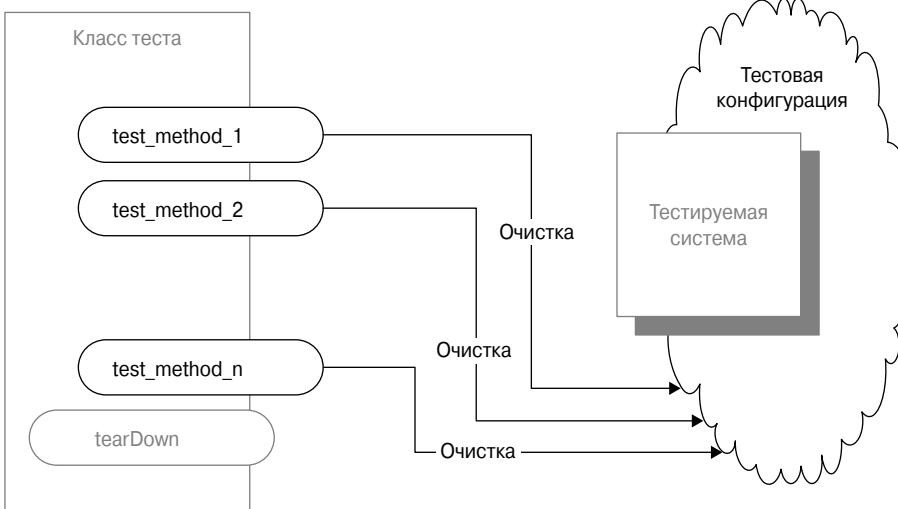
Неявная очистка (Implicit Teardown)

Как удалить тестовую конфигурацию?

Инфраструктура автоматизации тестов (Test Automation Framework) вызывает логику очистки в методе `tearDown` после завершения каждого **тестового метода** (Test Method).

Также известен как:

Очистка через “ловушки” (Hooked Teardown), Вызываемая инфраструктурой очистка (Framework-Invoked Teardown), Метод очистки (Teardown Method)



Повторяемость тестов в значительной мере зависит от очистки тестовой конфигурации после завершения каждого теста перед созданием новой конфигурации для следующего теста. Такая стратегия называется *новой тестовой конфигурацией* (Fresh Fixture, с. 344). Оставшиеся объекты и записи в базе данных, а также открытые файлы и подключения могут привести как минимум к снижению быстродействия, а как максимум — к аварийному завершению работы системы или неудачному завершению теста.

Если воспользоваться очисткой со сборкой мусора (Garbage-Collected Teardown, с. 518) не удается и в нескольких тестах требуется удалить одинаковые объекты, можно разместить логику очистки в специальном методе `tearDown`, который вызывается *инфраструктурой автоматизации тестов* (Test Automation Framework, с. 332) после завершения работы каждого *тестового метода* (Test Method, с. 378).

Как это работает

Все требующие очистки ресурсы могут быть освобождены или удалены на последней фазе *четырехфазного теста* (Four-Phase Test, с. 387), т.е. на фазе очистки тестовой конфигурации. В большинстве реализаций xUnit поддерживается концепция *неявной очистки* (Implicit Teardown), при которой метод `tearDown` каждого *объекта теста* (Testcase Object, с. 410) запускается после завершения *тестового метода* (Test Method).

Метод `tearDown` вызывается независимо от результата работы теста. Такая схема позволяет очистить конфигурацию, не заботясь о ложных утверждениях. Но имейте в виду, что во многих реализациях xUnit метод `tearDown` не вызывается, если метод `setUp` генерирует ошибку.

Когда это использовать

Неявная очистка (Implicit Teardown) может использоваться каждый раз, когда некоторым тестам требуется явное удаление или освобождение одинаковых ресурсов и ресурсы не поддерживают автоматическое удаление или освобождение. Это требование может стать следствием появления *неповторяемых* (Unrepeatable Test) или *медленных тестов* (Slow Tests, с. 289), возникающих из-за накопления мусора после многократного запуска тестов.

Если создаваемые тестами объекты представляют собой внутренние ресурсы и подчиняются автоматическим механизмам управления памятью, *очистка со сборкой мусора* (Garbage-Collected Teardown) позволяет сэкономить усилия. Если каждый тест использует собственный набор объектов, требующих удаления, наиболее подходящим решением может стать применение *встроенной очистки* (In-line Teardown, с. 527). В большинстве случаев можно избежать создания логики очистки вручную, воспользовавшись *автоматической очисткой* (Automated Teardown, с. 521).

Замечания по реализации

Логика очистки в методе `tearDown` чаще всего создается в результате рефакторинга тестов, использующих *встроенную очистку* (In-line Teardown). По ряду причин от метода `tearDown` может потребоваться “гибкость” или “приспособляемость”.

- При неудачном завершении теста или в результате ошибки *тестовый метод* (Test Method) может не создать все объекты конфигурации.
- Если *тестовые методы* (Test Method) в *классе теста* (Testcase Class, с. 401) используют неодинаковые конфигурации (т.е. *неявная настройка*, Implicit Setup, дополняется *встроенной настройкой*, In-line Setup, с. 434, или *делегированной настройкой*, Delegated Setup, с. 437), для разных тестов может потребоваться удаление разных наборов объектов.

Вариант: пункт охраны очистки (Teardown Guard Clause)

Если создано только подмножество удаляемых объектов, создания *условной логики теста* (Conditional Test Logic, с. 243) можно избежать за счет использования пункта охраны (простого оператора `if`) вокруг каждой операции удаления. Таким образом можно защититься от удаления несуществующих объектов. При такой структуре метод `tearDown` может использоваться для удаления различных вариантов тестовой конфигурации. В этом случае структура метода `tearDown` отличается от структуры метода `setUp`, который может создать только наиболее общее подмножество тестовой конфигурации для использующих ее *тестовых методов* (Test Method).

Мотивирующий пример

В следующем teste на этапе настройки конфигурации создается несколько стандартных объектов. Поскольку объекты сохраняются в базе данных, их приходится явно удалять после завершения каждого теста. Каждый тест (здесь показан только один из них) содержит одну и ту же встроенную логику для удаления объектов.

```
public void testGetFlightsByOrigin_NoInboundFlight_SMRTD()
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("1OF");
    BigDecimal inboundAirport = null;
    FlightDto expFlightDto = null;
    try {
        inboundAirport = createTestAirport("1IF");
        expFlightDto = createTestFlight(outboundAirport, inboundAirport);
        // Вызов тестируемой системы
        List flightsAtDestination1 =
            facade.getFlightsByOriginAirport(inboundAirport);
        // Проверка результата
        assertEquals(0, flightsAtDestination1.size());
    } finally {
        try {
            facade.removeFlight(expFlightDto.getFlightNumber());
        } finally {
            try {
                facade.removeAirport(inboundAirport);
            } finally {
                facade.removeAirport(outboundAirport);
            }
        }
    }
}
```

В примере показан достаточный объем *дублирования тестового кода* (Test Code Duplication, с. 254), чтобы оправдать преобразование тестов для использования *неявной очистки* (Implicit Teardown).

Замечания по рефакторингу

Сначала необходимо найти самый характерный пример кода очистки во всех тестах. После этого следует воспользоваться рефакторингом *выделение метода* (Extract Method) и присвоить полученному методу имя `tearDown`. Наконец, в оставшихся тестах необходимо удалить логику очистки.

Вокруг всех фрагментов логики, необязательных для каждого теста, необходимо разместить *пункты охраны очистки* (Teardown Guard Clause). Кроме того, каждую операцию очистки необходимо заключить в блок `try/finally` для обеспечения запуска оставшейся логики, даже если предыдущая операция завершилась неудачно.

Пример: неявная очистка (Implicit Teardown)

В этом примере показан тот же тест после выделения логики очистки в метод `tearDown`. Обратите внимание, насколько уменьшился объем теста.

```
BigDecimal outboundAirport;
BigDecimal inboundAirport;
FlightDto expFlightDto;
public void testGetFlightsByAirport_NoInboundFlights_NIT()
    throws Exception {
    // Настройка тестовой конфигурации
    outboundAirport = createTestAirport("10F");
    inboundAirport = createTestAirport("1IF");
    expFlightDto = createTestFlight(outboundAirport, inboundAirport);
    // Вызов тестируемой системы
    List flightsAtDestination1 =
        facade.getFlightsByOriginAirport(inboundAirport);
    // Проверка результата
    assertEquals(0, flightsAtDestination1.size());
}
protected void tearDown() throws Exception {
    try {
        facade.removeFlight(expFlightDto.getFlightNumber());
    } finally {
        try {
            facade.removeAirport(inboundAirport);
        } finally {
            facade.removeAirport(outboundAirport);
        }
    }
}
```

Обратите внимание, что вокруг вызова тестируемой системы и утверждений нет блока `try/finally`. Такая структура показывает, что это не *тест на ожидаемое исключение* (Expected Exception Test). Кроме того, не требуется *пункт охраны очистки* (Teardown Guard Clause) перед каждой операцией, так как блок `try/finally` обеспечивает запуск оставшейся логики. Содержащие конфигурацию локальные переменные были преобразованы в переменные экземпляра, чтобы метод `tearDown` мог получить доступ к существующей тестовой конфигурации.

Глава 23

Шаблоны тестовых двойников

Шаблоны в этой главе:

Тестовый двойник (Test Double) 538

Использование тестового двойника

Тестовая заглушка (Test Stub) 544

Тестовый агент (Test Spy) 552

Подставной объект (Mock Object) 558

Поддельный объект (Fake Object) 565

Создание тестового двойника

Настраиваемый тестовый двойник (Configurable Test Double) 571

Фиксированный тестовый двойник (Hard-Coded Test Double) 581

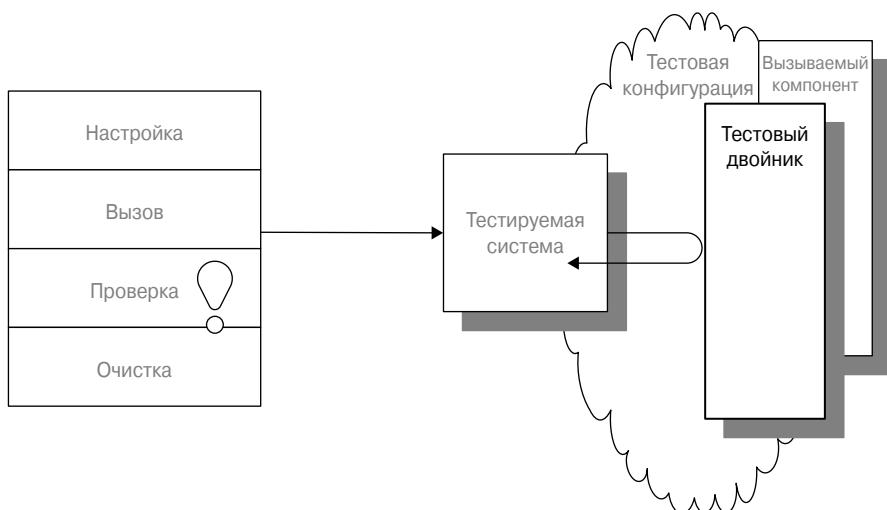
Связанный с тестом подкласс (Test-Specific Subclass) 591

Тестовый двойник (Test Double)

Также известен как:
Самозванец (*Imposter*)

Как обеспечить независимую проверку логики,
если код, от которого она зависит, недоступен?
Как избежать появления медленных тестов?

Компонент, от которого зависит тестируемая система, заменяется предназначенным для тестиования эквивалентом.



Иногда систему сложно тестировать, так как она зависит от других компонентов, которые невозможно использовать в тестовой среде. Такая ситуация может возникать из-за недоступности компонентов, из-за недоступности необходимых тесту возвращаемых значений или из-за нежелательных побочных эффектов обращения к таким компонентам. В других случаях стратегия тестирования может потребовать большей видимости или контроля над внутренним поведением тестируемой системы.

Если в создаваемых тестах невозможно использовать (или принято решение не использовать) настоящий вызываемый компонент, применяется *тестовый двойник* (Test Double). *Тестовый двойник* (Test Double) необязательно должен вести себя, как настоящий компонент. От него требуется только предоставление такого же программного интерфейса, чтобы тестируемая система не отличала его от настоящего!

Как это работает

Если создатели фильма хотят снять что-то потенциально рискованное, они нанимают каскадера, заменяющего актера в опасных сценах. Двойник имеет специальную подготовку, необязательно является актером, но знает, как падать с большой высоты или выполнять необходимые в конкретной сцене трюки. Степень сходства двойника и актера

определяется природой снимаемой сцены. Обычно все можно организовать так, чтобы вместо актера можно было снять того, кто немного его напоминает.

Для тестирования настоящий вызываемый компонент (не саму тестируемую систему!) можно заменить собственным эквивалентом “каскадера” — *тестовым двойником* (Test Double). На этапе создания тестовой конфигурации настоящий компонент заменяется *тестовым двойником* (Test Double). В зависимости от типа теста поведение двойника может быть фиксированным или настраиваться на этапе создания конфигурации. Во время взаимодействия с *тестовым двойником* (Test Double) система не знает, что вызываемый компонент ненастоящий, и это позволяет невозможный тест сделать возможным.

Независимо от используемого варианта *тестового двойника* (Test Double) не стоит забывать, что реализовать весь интерфейс тестируемой системы не нужно. Вместо этого представляется только та функциональность, которая необходима в пределах конкретного теста. Можно даже создавать разные *тестовые двойники* (Test Double) для разных тестов, использующих один и тот же вызываемый компонент.

Когда это использовать

Тестовый двойник (Test Double) подходящего типа может потребоваться в следующих ситуациях.

- При наличии *нетестированного требования* (Untested Requirement) из-за отсутствия в тестируемой системе или в вызываемом компоненте точки наблюдения для опосредованного вывода, который приходится проверять с помощью *проверки поведения* (Behavior Verification, с. 489).
- При наличии *нетестированного кода* (Untested Code) из-за отсутствия точки управления, позволяющей бы вызываемому компоненту обеспечить тестируемую систему необходимым опосредованным вводом.
- При наличии *медленных тестов* (Slow Tests, с. 289) и необходимости ускорить их работу (для повышения частоты запуска).

В каждом из перечисленных сценариев можно использовать тот или иной тип *тестового двойника* (Test Double). Конечно, при использовании двойников следует соблюдать осторожность, так как тестируемая система проверяется не в той конфигурации, в которой она будет работать при промышленной эксплуатации. По этой причине должен существовать как минимум один тест, проверяющий работоспособность тестируемой системы без двойника. Кроме того, нельзя подменять двойником проверяемую часть тестируемой системы, так как это может привести к появлению тестов, проверяющих совсем другое программное обеспечение! Чрезмерное использование *тестовых двойников* (Test Double) может привести к появлению “хрупких” тестов (Fragile Test, с. 277) в результате *зарегулированности программы* (Overspecified Software).

Существует несколько основных типов *тестовых двойников* (Test Double). Все они перечислены на рис. 23.1. Варианты реализации каждого шаблона рассматриваются в соответствующих разделах.

Эти варианты классифицируются по способам и причинам применения *тестовых двойников* (Test Double). Варианты создания двойников рассматриваются в разделе, посвященном реализации.



Рис. 23.1. Типы тестовых двойников (Test Double). Объект-заглушка (Dummy Object) является альтернативой шаблонам значений. Тестовая заглушка (Test Stub) используется для проверки опосредованного ввода. Тестовый агент (Test Spy) и подставной объект (Mock Object) используются для проверки опосредованного вывода. Поддельный объект (Fake Object) позволяет обеспечить альтернативную реализацию

Вариант: тестовая заглушка (Test Stub)

Тестовая заглушка (Test Stub, с. 544) используется для замены настоящего компонента, от которого зависит тестируемая система, чтобы обеспечить тест контрольной точкой для опосредованного ввода тестируемой системы. Это позволяет тесту переключить тестируемую систему на ветвь кода, не выполняемую в обычной ситуации. Дальнейшая классификация тестовых заглушек (Test Stub) производится на основе типа опосредованного ввода, который вставляется в тестируемую систему. Генератор ответов (Responder) вставляет допустимые значения, а диверсант (Saboteur) вставляет ошибки или исключения.

Иногда термин “тестовая заглушка” применяется для обозначения временной реализации объекта или процедуры. Во избежание путаницы в этой книге такой вариант называется *временной тестовой заглушкой* (Temporary Test Stub).

Вариант: тестовый агент (Test Spy)

Более сложную версию *тестовой заглушки* (Test Stub) — *тестовый агент* (Test Spy), — можно использовать в качестве точки наблюдения для опосредованного вывода тестируемой системы. Как и в случае *тестовой заглушки* (Test Stub), *тестовый агент* (Test Spy) может передавать в тестируемую систему значения в ответ на вызовы методов. Но он перехватывает опосредованный вывод и сохраняет его для последующей проверки тестом. Таким образом, во многих отношениях *тестовый агент* (Test Spy) — это “просто” *тестовая заглушка* (Test Stub) с функцией записи. Хотя *тестовый агент* (Test Spy) используется для тех же целей, что и *подставной объект* (Mock Object, с. 558), тесты с его использованием намного больше похожи на тесты с использованием заглушки.

Вариант: подставной объект (Mock Object)

Подставной объект (Mock Object) может использоваться в качестве точки наблюдения для проверки опосредованного вывода тестируемой системы во время ее работы. Обычно *подставной объект* (Mock Object) содержит функциональность *тестовой заглушки* (Test Stub), так как он должен возвращать значения в ответ на вызовы, но основное внимание при его реализации уделяется проверке опосредованного вывода. Таким образом, *подставной объект* (Mock Object) — это значительно больше, чем *тестовая заглушка* (Test Stub) с дополнительными утверждениями: он используется совершенно иначе.

Вариант: поддельный объект (Fake Object)

Поддельный объект (Fake Object, с. 565) используется для замены функциональности настоящего вызываемого компонента по причинам, отличным от проверки опосредованного ввода и вывода. Обычно *поддельный объект* (Fake Object) содержит реализацию той же функциональности, что и настоящий вызываемый компонент, но основанную на значительно более простых средствах. Хотя *поддельный объект* (Fake Object) обычно создается специально для тестирования, тест не использует его в качестве контрольной точки и точки наблюдения.

Чаще всего *поддельный объект* (Fake Object) используется, когда вызываемый компонент еще недоступен, работает слишком медленно или не может применяться в тестовой среде из-за нежелательных побочных эффектов. Во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351 рассматривается сокрытие доступа к базе данных за интерфейсом уровня сохранения объектов с последующей заменой базы данных хранящимися в памяти хэш-таблицами. В результате такой модификации тесты стали работать в 50 раз быстрее. Способы преобразования тестируемой системы для упрощения тестирования рассматриваются в главах 6 и 11.

Вариант: объект-заглушка (Dummy Object)

Некоторые сигнатуры методов тестируемой системы требуют передачи объектов в качестве параметров. Если ни тесту, ни тестируемой системе эти объекты не нужны, можно передать *объект-заглушку* (Dummy Object), который может быть представлен ссылкой со значением `null`, экземпляром класса `Object` или экземпляром *псевдообъекта* (Pseudo-Object; см. *Фиксированный тестовый двойник*, Hard-Coded Test Double, с. 581). В этом отношении *объект-заглушка* (Dummy Object) не является *тестовым двойником* (Test Double), а представляет собой альтернативу *точного значения* (Literal Value, с. 718), *вычисляемого значения* (Derived Value, с. 722) и *сгенерированного значения* (Generated Value, с. 726).

Вариант: процедурная тестовая заглушка (Procedural Test Stub)

Тестовый двойник (Test Double), реализованный на процедурном языке программирования, часто называется тестовой заглушкой, но здесь такая реализация называется *процедурной тестовой заглушкой* (Procedural Test Stub), чтобы отличать ее от современного варианта двойника — *тестовой заглушки* (Test Stub). Обычно *процедурная тестовая заглушка* (Procedural Test Stub) используется для тестирования и отладки в то время, пока остальной код еще не готов. Очень редко такие объекты “загружаются” во время выполнения, но иногда в коде присутствует условие относительно флага “отладки”. Это один из вариантов *логики теста в продукте* (Test Logic in Production, с. 257).

Замечания по реализации

При создании *тестовых двойников* (Test Double) необходимо учитывать несколько условий (рис. 23.2).

- Должен ли двойник быть связан с конкретным тестом или должен повторно использоваться большим количеством тестов?
- Нужно ли создавать код для *тестового двойника* (Test Double) или он генерируется “на лету”?

- Как тестируемая система узнает, какой *тестовый двойник* (Test Double) необходимо использовать (установка)?

Здесь рассматриваются первое и второе условия. Генерация *тестовых двойников* (Test Double) рассматривается в разделе о *настраиваемых тестовых двойниках* (Configurable Test Double).

Поскольку способы создания *тестовых двойников* (Test Double) не зависят от их поведения (т.е. они позволяют создавать как *тестовые заглушки*, Test Stub, так и *подставные объекты*, Mock Object), описания способов создания *фиксированных тестовых двойников* (Hard-Coded Test Double) и *настраиваемых тестовых двойников* (Configurable Test Double) вынесены в отдельные шаблоны.

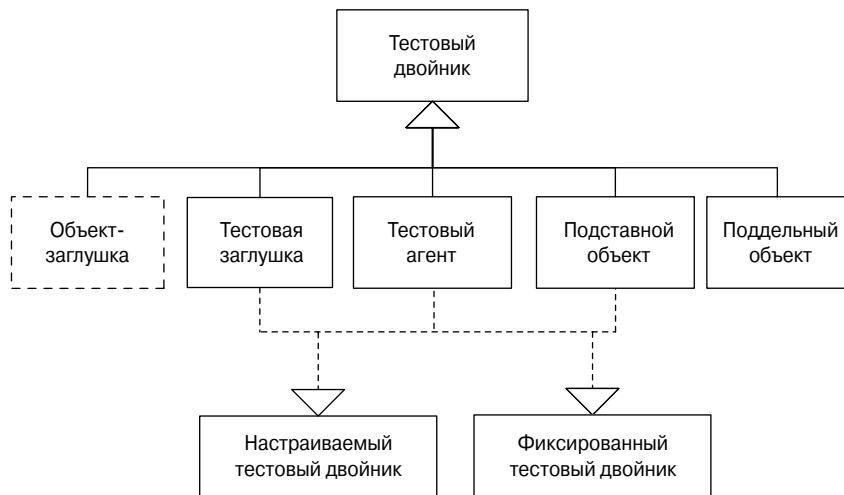


Рис. 23.2. Типы тестовых двойников (Test Double) и варианты реализации. Только тестовая заглушка (Test Stub), тестовый агент (Test Spy) и подставной объект (Mock Object) могут создаваться разработчиком или настраиваться тестом. Объект-заглушка (Dummy Object) не имеет реализации, а поддельный объект (Fake Object) устанавливается, но не контролируется тестом

Вариант: ненастраиваемый тестовый двойник (Unconfigurable Test Double)

Ни *объект-заглушка* (Dummy Object), ни *поддельный объект* (Fake Object) не требуют настройки (каждый по своим причинам). Объект-заглушка не используется получателем, поэтому не требует “настоящей” реализации. С другой стороны, *поддельный объект* (Fake Object) требует “настоящей” реализации, но она намного проще реализации замещаемого объекта. Таким образом, ни тест, ни разработчик не должны настраивать “консервированные” ответы или ожидания; *тестовый двойник* (Test Double) просто устанавливается и тестируемая система использует его, как настоящий компонент.

Вариант: фиксированный тестовый двойник (Hard-Coded Test Double)

Если планируется использовать конкретный *тестовый двойник* (Test Double) в единственном teste, часто проще запрограммировать *тестовый двойник* (Test Double) на возврат конкретных значений (в случае *тестовой заглушки*, Test Stub) или на ожидание кон-

крайних вызовов методов (в случае *подставного объекта*, Mock Object). *Фиксированные тестовые двойники* (Hard-Coded Test Double) обычно создаются самими разработчиками. Они существуют в нескольких формах, включая *тестовый шунт* (Self Shunt), в котором *класс теста* (Testcase Class) выступает в роли *тестового двойника* (Test Double), *анонимный внутренний тестовый двойник* (Anonymous Inner Test Double), в котором языковые возможности используются для создания *тестового двойника* (Test Double) внутри *тестового метода* (Test Method, с. 378), и *тестовый двойник* (Test Double), реализованный в виде отдельного *класса тестового двойника* (Test Double Class). Каждый из вариантов более подробно рассматривается в разделе о *фиксированных тестовых двойниках* (Hard-Coded Test Double).

Вариант: настраиваемый тестовый двойник (Configurable Test Double)

Если необходимо использовать одну и ту же реализацию *тестового двойника* (Test Double) в нескольких тестах, стоит обратить внимание на *настраиваемый тестовый двойник* (Configurable Test Double). Хотя разработчик может создать его вручную, во многих реализациях xUnit предоставляются средства для генерации *настраиваемых тестовых двойников* (Configurable Test Double).

Установка тестового двойника

Перед вызовом тестируемую систему необходимо вынудить использовать *тестовый двойник* (Test Double) вместо оригинального объекта. Для установки *тестового двойника* (Test Double) на этапе настройки тестовой конфигурации можно воспользоваться любым шаблоном *заменяемой зависимости* (substitutable dependency). *Настраиваемый тестовый двойник* (Configurable Test Double) должен быть настроен еще до вызова тестируемой системы. Обычно такая настройка происходит и до установки.

Пример: тестовый двойник (Test Double)

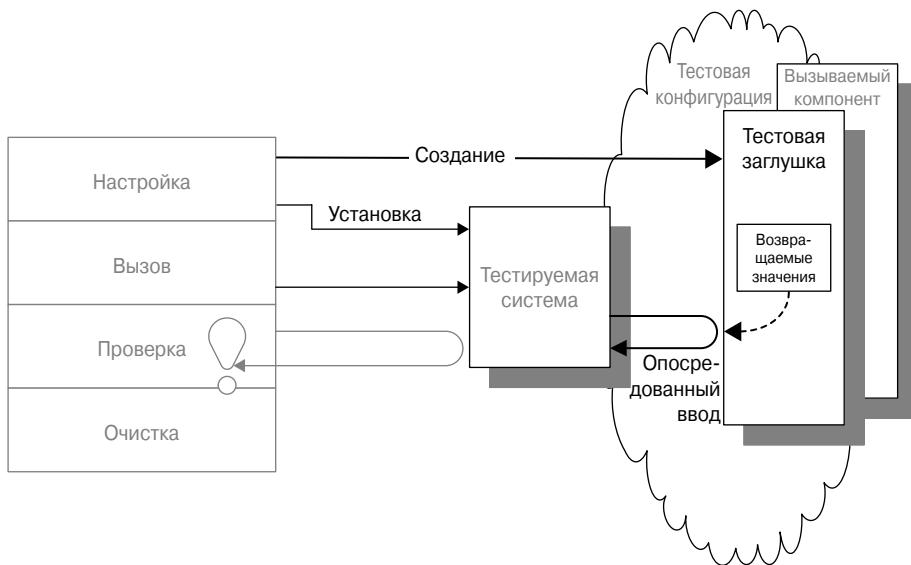
Поскольку существует множество причин использования *тестовых двойников* (Test Double), сложно привести единственный пример, характеризующий причины использования каждого варианта. Поэтому обратите внимание на примеры в разделах, посвященных конкретным шаблонам.

Тестовая заглушка (Test Stub)

Также известен как:
Заглушка (Stub)

Как независимо проверить логику, зависящую от опосредованного ввода со стороны других компонентов?

Настоящий объект заменяется объектом, созданным специально для теста. Новый объект передает интересующий тест опосредованный ввод в тестируемую систему.



В большинстве ситуаций среда (или контекст), в которой работает тестируемая система, значительно влияет на ее поведение. Для получения адекватного контроля над опосредованным вводом тестируемой системы, возможно, придется заменить некоторую часть контекста контролируемой сущностью, а именно — на *тестовую заглушки* (Test Stub).

Как это работает

Сначала определяется предназначенная для теста реализация интерфейса, от которого зависит тестируемая система. Эта реализация настраивается отвечать на вызовы системы значениями (или исключениями), которые приведут к выполнению *нетестированного кода* (Untested Code) внутри тестируемой системы. Перед вызовом системы устанавливается *тестовая заглушки* (Test Stub), которая используется вместо настоящей реализации. При вызове со стороны тестируемой системы во время работы теста *тестовая заглушки* (Test Stub) возвращает определенные ранее значения. После этого тест может обычным образом проверить ожидаемый результат.

Когда это использовать

Ключевым критерием использования *тестовых заглушек* (Test Stub) является наличие *нетестированного кода* (Untested Code), вызванного невозможностью контролировать опосредованный ввод тестируемой системы. *Тестовая заглушка* (Test Stub) может использоваться в качестве контрольной точки, позволяющей управлять поведением тестируемой системы с помощью различного опосредованного ввода, когда не требуется проверка опосредованного вывода. Кроме того, *тестовая заглушка* (Test Stub) может использоваться для перехода через определенную точку внутри тестируемой системы, в которой вызывается недоступное в тестовой среде программное обеспечение.

Если необходима точка наблюдения, позволяющая проверять опосредованный вывод тестируемой системы, рассмотрите использование *подставного объекта* (Mock Object, с. 558) или *тестового агента* (Test Spy, с. 552). Конечно, должен существовать способ установки *тестового двойника* (Test Double, с. 538) в тестируемую систему, иначе вообще нет смысла говорить о применении двойников.

Вариант: генератор ответов (Responder)

Тестовая заглушка (Test Stub), которая используется для вставки допустимого опосредованного ввода в тестируемую систему для нормального продолжения работы, называется *генератором ответов* (Responder). *Генераторы ответов* (Responder) обычно используются при тестировании “счастливого маршрута”, когда настоящий компонент невозможно контролировать, он еще недоступен или его нельзя использовать в среде разработки. Тесты с использованием генератора ответов всегда являются *простыми тестами успешности* (Simple Success Test).

Вариант: диверсант (Saboteur)

Тестовая заглушка (Test Stub), которая используется для вставки недопустимого опосредованного ввода в тестируемую систему, называется *диверсантом* (Saboteur). Ее назначение — нарушить работу тестируемой системы, чтобы проверить ее поведение в подобной ситуации. “Диверсия” может заключаться в возврате неожиданных значений или объектов, а может принимать форму генерации исключения или ошибки времени выполнения. Тесты на основе диверсанта могут быть *простыми тестами успешности* (Simple Success Test) или *тестами на ожидаемое исключение* (Expected Exception Test) в зависимости от ожидаемого поведения тестируемой системы в ответ на опосредованный ввод.

Вариант: временная тестовая заглушка (Temporary Test Stub)

Временная тестовая заглушка используется вместо еще недоступного вызываемого компонента. Обычно такой вариант *тестовой заглушки* (Test Stub) состоит из пустой оболочки настоящего класса с фиксированными возвращаемыми значениями. После появления вызываемого компонента он заменяет временную тестовую заглушку. Такое решение обычно используется при разработке на основе тестов в направлении “извне во внутрь”. После добавления кода такие оболочки эволюционируют в настоящие классы. При разработке на основе потребностей чаще используются *подставные объекты* (Mock Object), так как необходимо убедиться в вызове правильных методов временной заглушки. Кроме того, *подставные объекты* (Mock Object) продолжают использоваться даже после того, как настоящий вызываемый компонент становится доступным.

Вариант: процедурная тестовая заглушка (Procedural Test Stub)

Это *тестовая заглушка* (Test Stub), написанная на процедурном языке программирования. Ее достаточно сложно создавать на процедурных языках, не поддерживающих использование процедурных переменных (или указателей на функцию). В большинстве случаев приходится вставлять конструкцию `if testing then` в код продукта (вариант *логики теста в продукте*, Test Logic in Production, с. 257).

Вариант: обрезание цепочки сущностей (Entity Chain Snipping)

Обрезание цепочки сущностей (Entity Chain Snipping) является специальным случаем *генератора ответов* (Responder), который используется для замены сложной сети объектов одной *тестовой заглушкой* (Test Stub). Включив заглушку, можно значительно ускорить настройку конфигурации (особенно, если объекты должны сохраняться в базе данных); тесты при этом значительно упрощаются.

Замечания по реализации

Тестовые заглушки (Test Stub) необходимо использовать осторожно, так как тестируемая система проверяется не в той конфигурации, в которой она обычно работает. Должен существовать как минимум один тест, проверяющий работу системы без *тестовой заглушки* (Test Stub). Среди начинающих разработчиков распространена одна ошибка: они пытаются заменить заглушками фрагменты тестируемой системы, которые не проверяются в конкретном teste. Очень важно понимать, что играет роль тестируемой системы, а что выступает в роли тестовой конфигурации. Кроме того, обратите внимание, что чрезмерное использование *тестовых заглушки* (Test Stub) может привести к появлению *зарегулированных программ* (Overspecified Software).

Тестовая заглушка (Test Stub) может создаваться различными способами в зависимости от поставленных требований и доступного инструментария.

Вариант: фиксированная тестовая заглушка (Hard-Coded Test Stub)

В этом случае ответы заглушки зафиксированы в ее программной логике. Подобные *тестовые заглушки* (Test Stub) создаются специально для конкретного теста или для небольшого количества тестов. Дополнительная информация о таком варианте приводится в разделе о *фиксированных тестовых двойниках* (Hard-Coded Test Double, с. 581).

Вариант: настраиваемая тестовая заглушка (Configurable Test Stub)

Чтобы избежать создания фиксированной тестовой заглушки для каждого теста, можно воспользоваться *настраиваемой тестовой заглушкой* (Configurable Test Stub). Тест настраивает заглушку на этапе создания конфигурации. Во многих реализациях xUnit предлагаются инструменты для генерации *настраиваемых тестовых двойников* (Configurable Test Double, с. 571), включая *настраиваемые тестовые заглушки* (Configurable Test Stub).

Мотивирующий пример

Следующий тест проверяет базовую функциональность компонента, форматирующего HTML-строку с текущим временем. К сожалению, он зависит от реальных системных часов, поэтому редко завершается успешно!

```
public void testDisplayCurrentTime_AtMidnight() {
    // Настройка тестовой конфигурации
    TimeDisplay sut = new TimeDisplay();
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка непосредственного вывода
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals( expectedTimeString, result);
}
```

Можно решить эту проблему, вынудив тест рассчитывать ожидаемый результат в зависимости от текущего времени.

```
public void testDisplayCurrentTime_whenever() {
    // Настройка тестовой конфигурации
    TimeDisplay sut = new TimeDisplay();
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    Calendar time = new DefaultTimeProvider().getTime();
    StringBuffer expectedTime = new StringBuffer();
    expectedTime.append("<span class=\"tinyBoldText\">");
    if ((time.get(Calendar.HOUR_OF_DAY) == 0)
        && (time.get(Calendar.MINUTE) <= 1)) {
        expectedTime.append(" Midnight");
    } else if ((time.get(Calendar.HOUR_OF_DAY) == 12)
        && (time.get(Calendar.MINUTE) == 0)) { // полдень
        expectedTime.append("N3oon");
    } else {
        SimpleDateFormat fr = new SimpleDateFormat("h:mm a");
        expectedTime.append(fr.format(time.getTime()));
    }
    expectedTime.append("</span>");
    assertEquals(expectedTime, result);
}
```

Полученный *гибкий тест* (Flexible Test) приводит к появлению двух проблем. Во-первых, некоторые условия теста никогда не проверяются. (Хотите оставаться на рабочем месте до полуночи, чтобы проверить работоспособность программного обеспечения в полночь?) Во-вторых, тест дублирует логику тестируемой системы для расчета ожидаемого результата. Как доказать правильность логики?

Замечания по рефакторингу

Правильная проверка опосредованного ввода возможна при получении контроля над временем. Для этого воспользуйтесь рефакторингом *заменить зависимость тестовым двойником* (Replace Dependency with Test Double, с. 740) для замены настоящих системных часов (которые представлены здесь объектом `TimeProvider`) виртуальными часами, которые можно реализовать в виде *тестовой заглушки* (Test Stub), настраиваемой тестом. При настройке конфигурации тест указывает время, которое будет выступать в роли опосредованного ввода тестируемой системы.

Пример: генератор ответов (Responder) в виде фиксированной тестовой заглушки

Следующий тест проверяет одно из тестовых условий “счастливого маршрута”. Генератор ответов (Responder) используется для управления опосредованным вводом тестируемой системы. Ожидаемый результат можно безопасно зафиксировать, так как он основан на времени, вставляемом в тестируемую систему в виде опосредованного ввода.

```
public void testDisplayCurrentTime_AtMidnight()
    throws Exception {
    // Настройка тестовой конфигурации
    //      Настройка тестового двойника
    TimeProviderTestStub tpStub = new TimeProviderTestStub();
    tpStub.setHours(0);
    tpStub.setMinutes(0);
    // Создание тестируемой системы
    TimeDisplay sut = new TimeDisplay();
    // Установка тестового двойника
    sut.setTimeProvider(tpStub);
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}
```

В данном teste используется следующая реализация фиксированной *тестовой заглушки* (Test Stub).

```
private Calendar myTime = new GregorianCalendar();
/**
 * Полный конструктор для объекта TimeProviderTestStub
 * Параметр 'hours' описывает часы с использованием 24-часовых суток
 *      (например, 10 -- 10 AM, 12 -- полдень, 22 -- 10 PM, 0 -- полночь)
 * Параметр 'minutes' описывает минуты текущего часа
 *      (например, 0 -- час ровно, 1 -- 1 минута после наступления текущего часа)
 */
public TimeProviderTestStub(int hours, int minutes) {
    setTime(hours, minutes);
}
public void setTime(int hours, int minutes) {
    setHours(hours);
    setMinutes(minutes);
}
// Конфигурационный интерфейс
public void setHours(int hours) {
    // 0 -- полночь; 12 -- полдень
    myTime.set(Calendar.HOUR_OF_DAY, hours);
}
public void setMinutes(int minutes) {
    myTime.set(Calendar.MINUTE, minutes);
}
// Интерфейс, используемый тестируемой системой
public Calendar getTime() {
    // @return -- последнее установленное значение времени
    return myTime;
}
```

Пример: динамически создаваемый генератор ответов (Responder)

Ниже представлен же тест, записанный с помощью *настраиваемого тестового двойника* (Configurable Test Double), который создан на основе инфраструктуры JMock.

```
public void testDisplayCurrentTime_AtMidnight_JM()
    throws Exception {
    // Настройка тестовой конфигурации
    TimeDisplay sut = new TimeDisplay();
    // Настройка тестового двойника
    Mock tpStub = mock(TimeProvider.class);
    Calendar midnight = makeTime(0,0);
    tpStub.stubs().method("getTime").
        withNoArguments().
        will(returnValue(midnight));
    // Установка тестового двойника
    sut.setTimeProvider((TimeProvider) tpStub);
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}
```

В этом примере не показана реализация *тестовой заглушки* (Test Stub), так как в JMock заглушка реализуется на основе механизма интроспекции. Таким образом, необходимо написать вспомогательный *метод теста* (Test Utility Method, с. 610), который называется `makeTime` и содержит логику для создания возвращаемого объекта `Calendar`. В фиксированной тестовой заглушки эта логика находилась в методе `getTime`.

Пример: диверсант (Saboteur) в виде анонимного внутреннего класса (Anonymous Inner Class)

В следующем teste *диверсант* (Saboteur) используется для вставки некорректного опосредованного ввода в тестируемую систему для проверки ее поведения в подобной ситуации.

```
public void testDisplayCurrentTime_exception()
    throws Exception {
    // Настройка тестовой конфигурации
    // Определение и создание тестовой заглушки
    TimeProvider testStub = new TimeProvider()
    { // Анонимная внутренняя тестовая заглушки
        public Calendar getTime() throws TimeProviderEx {
            throw new TimeProviderEx("Sample");
        }
    };
    // Создание тестируемой системы
    TimeDisplay sut = new TimeDisplay();
    sut.setTimeProvider(testStub);
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка непосредственного вывода
```

```

String expectedTimeString =
    "<span class=\"error\">Invalid Time</span>";
assertEquals("Exception", expectedTimeString, result);
}

```

В данном случае для генерации исключения используется *внутренний тестовый двойник* (Inner Test Double). Ожидается, что тестируемая система сможет нормально обработать это исключение. Одной из интересных особенностей этого теста является использование шаблона *простого теста успешности* (Simple Success Test) вместо *теста на ожидаемое исключение* (Expected Exception Test), хотя в качестве опосредованного ввода вставляется исключение. Причиной такого выбора является предполагаемый перехват исключения внутри тестируемой системы с соответствующим изменением форматирования строки. Никакого исключения в такой ситуации система генерировать не должна.

Пример: обрезание цепочки сущностей (Entity Chain Snipping)

В этом примере проверяется объект `Invoice`, но для создания этого объекта требуется объект `Customer`. Объекту `Customer` требуется объект `Address`, которому, в свою очередь, требуется объект `City`. Таким образом, только для создания тестируемой системы создается множество дополнительных объектов. Предположим, что поведение объекта `Invoice` зависит от атрибута объекта `Customer`, который рассчитывается на основе объекта `Address` через вызов метода `get_zone` объекта `Customer`.

```

public void testInvoice_addLineItem_noECS() {
    final int QUANTITY = 1;
    Product product = new Product(getUniqueNumberAsString(),
        getUniqueNumber());
    State state = new State("West Dakota", "WD");
    City city = new City("Centreville", state);
    Address address = new Address("123 Blake St.", city, "12345");
    Customer customer= new Customer(getUniqueNumberAsString(),
        getUniqueNumberAsString(),
        address);
    Invoice inv = new Invoice(customer);
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    assertLineItemsEqual("",expItem, actual);
}

```

В этом teste необходимо проверить только логику объекта `Invoice`, которая зависит от атрибута `zone`, а не способ расчета значения этого атрибута на основе адреса объекта `Customer`. (Существует отдельный модульный тест для объекта `Customer`, который проверяет правильность расчета атрибута `zone`.) Код настройки адреса, города и другой информации просто отвлекает читателя.

Ниже приведен тот же тест, но с использованием *тестовой заглушки* (Test Stub) вместо объекта `Customer`. Обратите внимание, насколько упростилась настройка конфигурации из-за *обрезания цепочки сущностей* (Entity Chain Snipping)!

```

public void testInvoice_addLineItem_ECS() {
    final int QUANTITY = 1;
    Product product = new Product(getUniqueNumberAsString(),
                                   getUniqueNumber());
    Mock customerStub = mock(ICustomer.class);
    customerStub.stubs().method("getZone").will(returnValue(ZONE_3));
    Invoice inv = new Invoice((ICustomer)customerStub.proxy());
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    assertLineItemsEqual("", expItem, actual);
}

```

Для создания заглушки `customerStub` вместо объекта `Customer` использовалась инфраструктура JMock. При вызове метода `getZone` заглушка возвращает значение `ZONE_3`. Это все, что необходимо для проверки поведения объекта `Invoice`. При этом удалось избежать создания дополнительных объектов, отвлекающих внимание читателя тестов. Кроме того, из данного теста становится предельно ясно, что поведение объекта `Invoice` зависит исключительно от возвращаемого значения метода `get_zone` и ни от каких атрибутов объектов `Customer` и `Address` не зависит.

Источники дополнительной информации

Практически во всех книгах по тестированию с использованием инфраструктуры xUnit рассматривается использование *тестовых заглушки* (Test Stub), поэтому соответствующие источники здесь не упоминаются. Но при чтении других книг не забывайте, что термин *тестовая заглушка* (Test Stub) часто используется для описания *подставных объектов* (Mock Object). В приложении Б приводится подробное сравнение терминологии, применяемой в различных книгах и статьях.

Свен Гортс рассматривает различные способы использования *тестовых заглушки* (Test Stub) [UTwHCM]. Многие из предложенных им названий использованы в данной книге. Некоторые из них были модифицированы в соответствии с требованиями языка шаблонов. Паоло Перротта написал шаблон, описывающий общий пример *генератора ответов* (Responder), который называется Virtual Clock. Он использует *тестовую заглушку* (Test Stub) в качестве декоратора [GOF] для настоящих системных часов. Заглушка позволяет “замораживать” и запускать время повторно. Конечно, для большинства тестов можно воспользоваться *фиксированной тестовой заглушкой* (Hard-Coded Test Stub) или *настраиваемой тестовой заглушкой* (Configurable Test Stub).

Тестовый агент (Test Spy)

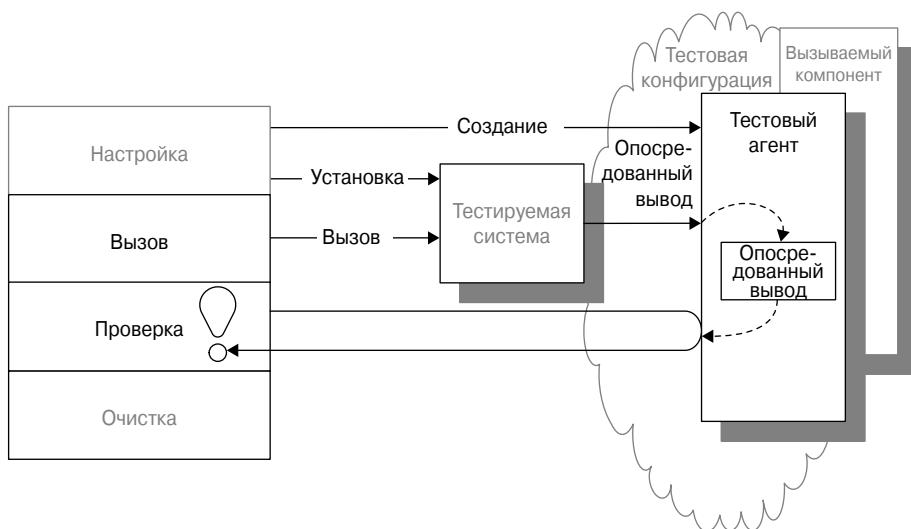
Также известен как:

Агент (*Spy*), Записывающая тестовая заглушка (*Recording Test Stub*)

Как реализовать проверку поведения (*Behavior Verification*)? Как обеспечить независимую проверку логики с опосредованным выводом в сторону других программных компонентов?

Для перехвата вызовов опосредованного вывода используется **тестовый двойник** (*Test Double*).

Позднее тест проверяет правильность перехваченного вывода.



В большинстве ситуаций среда (или контекст), в которой работает тестируемая система, значительно влияет на ее поведение. Для получения адекватной видимости опосредованного вывода тестируемой системы, возможно, придется заменить часть контекста тем, что позволит перехватывать вывод тестируемой системы.

Применение **тестового агента** (*Test Spy*) является простым и интуитивно понятным способом реализации **проверки поведения** (*Behavior Verification*, с. 489) через точку наблюдения, которая делает видимым опосредованный вывод тестируемой системы для последующей проверки.

Как это работает

Перед вызовом тестируемой системы **тестовый агент** (*Test Spy*) устанавливается вместо вызываемого компонента, который используется тестируемой системой. **Тестовый агент** (*Test Spy*) проектируется для работы в качестве точки наблюдения. Он записывает вызовы методов, которые тестируемая система совершает во время взаимодействия с тестом. На этапе проверки результатов тест сравнивает фактические значения, переданные агенту тестируемой системой, с ожидаемыми значениями.

Когда это использовать

Ключевым критерием использования *тестового агента* (Test Spy) является наличие *нетестированного требования* (Untested Requirement) из-за невозможности наблюдения за побочными эффектами вызова методов тестируемой системы. *Тестовый агент* (Test Spy) является естественным решением по расширению тестов для перехвата подобного опосредованного вывода, так как *методы с утверждением* (Assertion Method, с. 553) вызываются тестом по окончании взаимодействия с тестируемой системой, как и в “нормальных” тестах. *Тестовый агент* (Test Spy) просто выступает в роли точки наблюдения, предоставляющей *тестовому методу* (Test Method, с. 378) доступ к значениям, записанным во время работы тестируемой системы.

Тестовый агент (Test Spy) имеет смысл использовать в следующих ситуациях.

- При проверке опосредованного вывода тестируемой системы, когда невозможно предугадать значения всех атрибутов взаимодействия с тестируемой системой.
- Необходимо обеспечить видимость утверждений в пределах теста, а выражение ожиданий в терминах *подставного объекта* (Mock Object, с. 558) не кажется достаточно очевидным.
- Тест требует специального равенства (поэтому нельзя использовать стандартное определение равенства, реализованное в тестируемой системе) и используются инструменты генерации *подставных объектов* (Mock Object), но нет контроля над вызываемыми *методами с утверждением* (Assertion Method).
- Информацию о ложном утверждении невозможно эффективно передать в *программу запуска тестов* (Test Runner, с. 405). Такая ситуация может возникать при запуске тестируемой системы внутри контейнера, перехватывающего все исключения и затрудняющего передачу результатов, а также при работе системы не в том процессе или потоке, в котором находится вызвавший ее тест. (Оба варианта требуют рефакторинга, позволяющего непосредственно проверять логику тестируемой системы, но это тема для другой главы.)
- Необходимо получить доступ ко всем исходящим вызовам со стороны тестируемой системы до проверки основанных на них утверждений.

Если ни одно из этих условий не выполняется, следует рассмотреть использование *подставного объекта* (Mock Object). Для того чтобы избавиться от *нетестированного кода* (Untested Code) с помощью контроля над опосредованным вводом тестируемой системы достаточно простой *тестовой заглушки* (Test Stub, с. 544).

В отличие от *подставных объектов* (Mock Object) *тестовый агент* (Test Spy) не приводит к неудачному завершению теста при первом отклонении от ожидаемого поведения. Таким образом, тесты смогут включить в *сообщение для утверждения* (Assertion Message, с. 398) более подробную диагностическую информацию, собираемую после того момента, когда *подставной объект* (Mock Object) мог бы привести к неудачному завершению теста. Но на момент неудачного завершения теста для вызова *методов с утверждением* (Assertion Method) доступна только информация, находящаяся внутри *тестового метода* (Test Method). Если требуется информация, доступная только во время взаимодействия с тестируемой системой, ее необходимо явно перехватывать с помощью *тестового агента* (Test Spy) или переходить к использованию *подставных объектов* (Mock Object).

Конечно, использовать *тестовые двойники* (Test Double, с. 538) невозможно, если в тестируемой системе не предусмотрен подходящий механизм замены зависимости.

Замечания по реализации

Сам по себе *тестовый агент* (Test Spy) может быть реализован в виде *фиксированного тестового двойника* (Hard-Coded Test Double, с. 581) или *настраиваемого тестового двойника* (Configurable Test Double, с. 571). Более подробная информация о реализации этих вариантов приводится в соответствующих разделах, поэтому здесь представлено только краткое описание. Для установки *тестового агента* (Test Spy) до вызова тестируемой системы может использоваться любой механизм замены зависимости.

Ключевой особенностью *тестового агента* (Test Spy) является вызов утверждений из *тестового метода* (Test Method). Таким образом, до вызова утверждений тест должен получить опосредованный вывод, перехваченный *тестовым агентом* (Test Spy). Для получения данных может использоваться один из нескольких способов.

Вариант: интерфейс извлечения (Retrieval Interface)

Тестовый агент (Test Spy) можно определить как отдельный класс с *интерфейсом извлечения* (Retrieval Interface), который предоставляет доступ к записанной информации. *Тестовый метод* (Test Method) устанавливает *тестовый агент* (Test Spy) вместо нормального вызываемого компонента на этапе создания тестовой конфигурации. После взаимодействия с тестируемой системой записанный опосредованный вывод передается в *тестовый метод* (Test Method) через *интерфейс извлечения* (Retrieval Interface) и используется в качестве аргументов при вызове *методов с утверждением* (Assertion Method).

Вариант: тестовый шунт (Self Shunt)

Также известен как: *Петля (Loopback)*

Тестовый агент (Test Spy) и *класс теста* (Testcase Class, с. 401) можно собрать в один объект — *тестовый шунт* (Self Shunt). *Тестовый метод* (Test Method) устанавливает себя, *объект теста* (Testcase Object, с. 410), в качестве вызываемого компонента тестируемой системы. При любых обращениях к вызываемому компоненту тестируемая система вызывает методы *объекта теста* (Testcase Object), сохраняющие фактические значения в переменные экземпляра, доступные *тестовому методу* (Test Method). Внутри *тестового агента* (Test Spy) могут вызываться методы утверждений. В таком случае *тестовый шунт* (Self Shunt) больше похож на *подставной объект* (Mock Object), а не на *тестовый агент* (Test Spy). В статически типизированных языках *класс теста* (Testcase Class) должен реализовать исходящий интерфейс (точку наблюдения), от которого зависит тестируемая система, чтобы класс теста был совместим по типу с переменными, в которых хранится вызываемый компонент.

Вариант: внутренний тестовый двойник (Inner Test Double)

Популярным способом реализации *тестового агента* (Test Spy) в виде *фиксированного тестового двойника* (Hard-Coded Test Double) является использование **анонимного внутреннего класса** (anonymous inner class) или **блочной конструкции** (block closure) внутри *тестового метода* (Test Method). Класс или блок сохраняет фактические значения в локальные переменные или переменные экземпляра, доступные *тестовому методу* (Test Method). По сути, это еще один способ реализации *тестового шунта* (Self Shunt).

Вариант: реестр опосредованного вывода (Indirect Output Registry)

Еще одним решением является сохранение фактических параметров в хорошо известном месте, где к ним может получить доступ *тестовый метод* (Test Method). Например, *тестовый агент* (Test Spy) может сохранять значения в файле или в объекте Registry.

Мотивирующий пример

Приведенный ниже тест проверяет базовую функциональность удаления объекта перелета, но не проверяет опосредованный вывод тестируемой системы, а именно — добавление записи в журнал при каждом удалении объекта перелета с указанием даты и времени, а также имени пользователя, оформившего запрос.

```
public void testRemoveFlight() throws Exception {
    // Настройка
    FlightDto expectedFlightDto = createARegisteredFlight();
    FlightManagementFacade facade = new FlightManagementFacadeImpl();
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertFalse("flight should not exist after being removed",
        facade.flightExists(expectedFlightDto.
            getFlightNumber()));
}
```

Замечания по рефакторингу

Для добавления проверки опосредованного вывода в существующий тест можно воспользоваться рефакторингом *заменить зависимость тестовым двойником* (Replace Dependency with Test Double, с. 740). При этом в логику настройки конфигурации необходимо добавить код создания *тестового агента* (Test Spy), его настройки на подходящие возвращаемые значения и его установки. В конце теста добавляются утверждения, сравнивающие ожидаемые имена методов и аргументы опосредованного вывода с фактическими значениями из *тестового агента* (Test Spy), полученные через *интерфейс извлечения* (Retrieval Interface).

Пример: тестовый агент (Test Spy)

В улучшенной версии теста в качестве *тестового агента* (Test Spy) используется объект logSpy. Оператор facade.setAuditLog(logSpy) устанавливает *тестовый агент* (Test Spy) с помощью шаблона *вставки метода установки* (Setter Injection). Методы getDate, getActionCode и другие принадлежат *интерфейсу извлечения* (Retrieval Interface) и предназначены для доступа к фактическим аргументам.

```
public void testRemoveFlightLogging_recordingTestStub()
    throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnUnregFlight();
    FlightManagementFacade facade = new FlightManagementFacadeImpl();
    // Настройка тестового двойника
    AuditLogSpy logSpy = new AuditLogSpy();
    facade.setAuditLog(logSpy);
```

```

// Вызов
facade.removeFlight(expectedFlightDto.getFlightNumber());
// Проверка
assertFalse("flight still exists after being removed",
            facade.flightExists(expectedFlightDto.
                                  getFlightNumber()));
assertEquals("number of calls", 1,
            logSpy.getNumberofCalls());
assertEquals("action code",
            Helper.REMOVE_FLIGHT_ACTION_CODE,
            logSpy.getActionCode());
assertEquals("date", helper.getTodaysDateWithoutTime(),
            logSpy.getDate());
assertEquals("user", Helper.TEST_USER_NAME,
            logSpy.getUser());
assertEquals("detail",
            expectedFlightDto.getFlightNumber(),
            logSpy.getDetail());
}

```

Этот тест зависит от следующего определения *тестового агента* (Test Spy).

```

public class AuditLogSpy implements AuditLog {
    // Поля, в которые записывается информация о фактическом
    // использовании
    private Date date;
    private String user;
    private String actionCode;
    private Object detail;
    private int numberofCalls = 0;
    // Записывающая реализация настоящего интерфейса AuditLog
    public void logMessage(Date date,
                           String user,
                           String actionCode,
                           Object detail) {
        this.date = date;
        this.user = user;
        this.actionCode = actionCode;
        this.detail = detail;
        numberofCalls++;
    }
    // Интерфейс извлечения
    public int getNumberofCalls() {
        return numberofCalls;
    }
    public Date getDate() {
        return date;
    }
    public String getUser() {
        return user;
    }
    public String getActionCode() {
        return actionCode;
    }
    public Object getDetail() {
        return detail;
    }
}

```

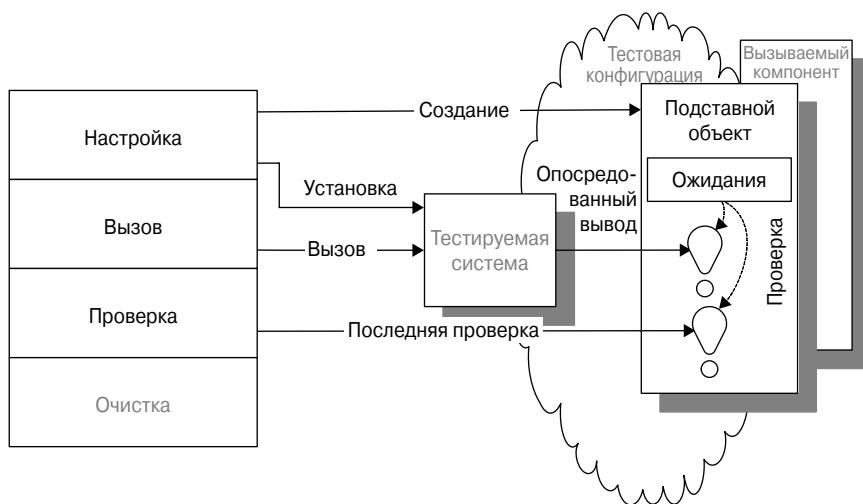
Конечно, *интерфейс извлечения* (Retrieval Interface) может быть реализован через открытые поля агента, что позволит отказаться от методов доступа. Другие варианты реализации рассматриваются в разделе о *фиксированных тестовых двойниках* (Hard-Coded Test Double).

Подставной объект (Mock Object)

Как реализовать проверку поведения (Behavior Verification) для опосредованного вывода тестируемой системы?

Как независимо проверять логику, которая зависит от опосредованного ввода других программных компонентов?

Объект, от которого зависит тестируемая система, заменяется специально созданным для теста объектом, проверяющим правильность своего использования со стороны тестируемой системы.



В одних случаях среда (или контекст), в которой работает тестируемая система, значительно влияет на ее поведение. В других случаях приходится заглядывать “внутрь” тестируемой системы, чтобы подтвердить наступление ожидаемого поведения¹.

Подставной объект (Mock Object) — это мощный инструмент для *проверки поведения* (Behavior Verification, с. 489) без *дублирования тестового кода* (Test Code Duplication, с. 254) между похожими тестами. Этот способ работает за счет делегирования проверки опосредованного вывода в *тестовый двойник* (Test Double, с. 538).

Как это работает

Сначала определяется *подставной объект* (Mock Object), в котором реализован тот же интерфейс, что и у объекта, от которого зависит тестируемая система. После этого во время тестирования *подставной объект* (Mock Object) настраивается с помощью значений, которые должны передаваться в тестируемую систему, а также вызовов методов

¹ Технически тестируемой системой является любое тестируемое программное обеспечение без компонентов, от которых оно зависит. Таким образом, слово “внутрь” употребляется неправильно. Лучше рассматривать получающий опосредованный вывод вызываемый компонент как элемент, находящийся “за” тестируемой системой и являющийся частью тестовой конфигурации.

(с ожидаемыми аргументами), ожидаемых от тестируемой системы. Перед вызовом системы *подставной объект* (Mock Object) устанавливается таким образом, чтобы система использовала его вместо настоящей реализации. В ответ на вызов со стороны тестируемой системы *подставной объект* (Mock Object) сравнивает фактически полученные аргументы с ожидаемыми с помощью *утверждений равенства* (Equality Assertion) и приводит к неудачному завершению теста, если они не совпадают. Сам тест никаких утверждений не содержит!

Когда это использовать

Подставной объект (Mock Object) может использоваться в качестве точки наблюдения для проверки поведения (Behavior Verification), когда необходимо избежать появления *не-тестированного требования* (Untested Requirement), вызванного невозможностью наблюдать побочные эффекты методов тестируемой системы. Данный шаблон обычно используется во время эндосякопического тестирования или при разработке на основе потребностей. *Подставной объект* (Mock Object) при проверке состояния (State Verification, с. 484) не нужен, так как в подобной ситуации можно воспользоваться *тестовой заглушкой* (Test Stub, с. 544) или *поддельным объектом* (Fake Object, с. 565). Некоторые разработчики нашли новые применения для средств создания *подставных объектов* (Mock Object), но в большинстве случаев они являются примерами использования *тестовых заглушки* (Test Stub), а не *подставных объектов* (Mock Object).

Для использования *подставного объекта* (Mock Object) должна иметься возможность предугадывать значения большинства или всех аргументов методов еще до вызова тестируемой системы. *Подставной объект* (Mock Object) не должен использоваться, если информацию о ложном утверждении нельзя донести до *программы запуска тестов* (Test Runner). Такая ситуация может возникать, когда тестируемая система работает внутри контейнера, перехватывающего и обрабатывающего все исключения. В таком случае лучше воспользоваться *тестовым агентом* (Test Spy, с. 552).

Подставной объект (Mock Object) (особенно созданный с помощью специализированного набора инструментов) использует методы `equals` различных сравниваемых объектов. Если равенство внутри теста отличается от интерпретации равенства в тестируемой системе, воспользоваться *подставным объектом* (Mock Object) не удастся или придется добавлять метод `equals` туда, где он не нужен. Этот запах называется *засорением равенства* (Equality Pollution). В некоторых реализациях *подставных объектов* (Mock Object) данной проблемы удается избежать за счет указания “компаратора”, используемого в *утверждениях равенства* (Equality Assertion).

Подставной объект (Mock Object) может быть “строгим” или “ослабленным” (или, как его иногда называют, “хорошим”). “Строгий” *подставной объект* (Mock Object) приводит к неудачному завершению теста, если вызовы поступают не в том порядке, который был указан при программировании *подставного объекта* (Mock Object). “Ослабленный” *подставной объект* (Mock Object) допускает поступление вызовов в другом порядке.

Замечания по реализации

Тесты, написанные с использованием *подставных объектов* (Mock Object), отличаются от традиционных тестов, так как ожидаемое поведение описывается до вызова тестируемой системы. В результате начинающим разработчикам сложнее писать и понимать

такие тесты. Одного этого фактора достаточно, чтобы предпочтение отдавалось тестам на основе *тестовых агентов* (Test Spy).

При использовании *подставного объекта* (Mock Object) стандартный *четырехфазный тест* (Four-Phase Test, с. 387) несколько модифицируется. В частности, фаза настройки тестовой конфигурации делится на три операции, а фаза проверки результата исчезает (кроме возможного вызова метода “последней проверки” в конце теста).

Настройка конфигурации:

- тест создает *подставной объект* (Mock Object);
- тест настраивает *подставной объект* (Mock Object); в случае *фиксированных тестовых двойников* (Hard-Coded Test Double, с. 581) эта операция пропускается;
- тест устанавливает *подставной объект* (Mock Object) в тестируемую систему.

Вызов тестируемой системы:

- тестируемая система вызывает *подставной объект* (Mock Object), который проверяет утверждения.

Проверка результатов:

- тест вызывает метод “последней проверки”.

Очистка тестовой конфигурации:

- не затрагивается.

Рассмотрим отличия более подробно.

Создание

Во время настройки тестовой конфигурации *четырехфазного теста* (Four-Phase Test) необходимо создать *подставной объект* (Mock Object), который будет использоваться вместо реального компонента. В зависимости от доступных в языке программирования инструментов можно создать класс вручную, воспользоваться генератором или использовать динамически генерированный *подставной объект* (Mock Object).

Настройка ожидаемых значений

Поскольку инструменты создания *подставных объектов* (Mock Object), доступные во многих реализациях xUnit, обычно создают *настраиваемые подставные объекты* (Configurable Mock Object, с. 571), необходимо настроить ожидаемые вызовы методов с параметрами, а также значения, возвращаемые каждой функцией. (Некоторые инфраструктуры работы с *подставными объектами* (Mock Object) позволяют отключить проверку вызовов методов или их параметров.) Обычно такая настройка происходит до установки *тестового двойника* (Test Double).

Данная операция не нужна при использовании *фиксированного тестового двойника* (Hard-Coded Test Double), например *внутреннего тестового двойника* (Inner Test Double).

Установка

Конечно, для использования *подставного объекта* (Mock Object) должен существовать способ установки *тестовых двойников* (Test Double) в тестируемую систему. Можно использовать любой поддерживающий шаблон замены зависимости. При разработке на ос-

нове тестов распространен подход на основе *вставки зависимости* (Dependency Injection, с. 684). Более приверженные традициям разработчики предпочитают использовать *поиск зависимости* (Dependency Lookup, с. 692).

Использование

При вызове методов *подставной объект* (Mock Object) сравнивает полученные вызовы (и их аргументы) с ожидаемыми. Если вызов оказался неожиданным или аргументы — некорректными, утверждение приводит к немедленному неудачному завершению теста. Если вызов правильный, но поступил с нарушением порядка, строгий *подставной объект* (Mock Object) немедленно завершает тест. С другой стороны, ослабленный *подставной объект* (Mock Object) отмечает получение вызова и продолжает работу. Пропущенные вызовы обнаруживаются методом последней проверки.

Если метод имеет исходящие параметры или возвращаемые значения, *подставной объект* (Mock Object) должен вернуть или модифицировать то, что позволит системе продолжить выполнение тестового сценария. Такое поведение может быть фиксированым или настраиваться одновременно с ожидаемыми вызовами. В этом поведение совпадает с *тестовыми заглушками* (Test Stub), но обычно возвращаются значения для счастливого маршрута.

Последняя проверка

Проверка результатов, в основном, происходит внутри *подставного объекта* (Mock Object) во время вызова со стороны тестируемой системы. *Подставной объект* (Mock Object) приводит к неудачному завершению теста, если методы вызываются с неправильными аргументами или в неожиданном порядке. Но что происходит, если вызовы никогда не получаются *подставным объектом* (Mock Object)? *Подставной объект* (Mock Object) может не обнаружить, что тест уже завершился и пора проверять невыполненные ожидания. Таким образом, возникает необходимость вызывать метод последней проверки. Некоторые наборы инструментов для работы с *подставными объектами* (Mock Object) содержат способ автоматического вызова этого метода через метод `tearDown` (обычно для этого класс теста должен наследовать специальный класс `MockObjectTestCase`). В большинстве случаев ответственность за вызов метода последней проверки возлагается на разработчика.

Мотивирующий пример

В следующем teste проверяется базовая функциональность создания объекта перелета, но не проверяется опосредованный вывод тестируемой системы, а именно — тестируемая система должна записывать каждую операцию над объектом перелета, время и дату создания, а также имя пользователя, инициировавшего операцию.

```
public void testRemoveFlight() throws Exception {
    // Настройка
    FlightDto expectedFlightDto = createARegisteredFlight();
    FlightManagementFacade facade = new FlightManagementFacadeImpl();
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
```

```

        assertFalse("flight should not exist after being removed",
                    facade.flightExists( expectedFlightDto.
                                         getFlightNumber() ) );
    }
}

```

Замечания по рефакторингу

Для добавления проверки опосредованного вывода в существующие тесты можно воспользоваться рефакторингом *заменить зависимость тестовым двойником* (Replace Dependency with Test Double, с. 740). Для этого в фазу настройки тестовой конфигурации необходимо добавить код создания *подставного объекта* (Mock Object), его настройки для ожидаемых вызовов методов, аргументов и возвращаемых значений, а также установки с помощью любого подходящего механизма замены зависимости. В конце теста добавляется вызов метода последней проверки (если это необходимо инфраструктуре подставных объектов).

Пример: созданный вручную подставной объект (Mock Object)

В улучшенной версии теста в качестве *подставного объекта* (Mock Object) выступает mockLog. Метод setExpectedLogMessage используется для указания ожидаемой записи в журнале. Вызов facade.setAuditLog(mockLog) устанавливает *подставной объект* (Mock Object) с помощью шаблона *вставки метода установки* (Setter Injection). Наконец, метод verify() проверяет, вызывался ли метод logMessage().

```

public void testRemoveFlight_Mock() throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnonRegFlight();
    // Настройка подставного объекта
    ConfigurableMockAuditLog mockLog =
        new ConfigurableMockAuditLog();
    mockLog.setExpectedLogMessage(
        helper.getTodaysDateWithoutTime(),
        Helper.TEST_USER_NAME,
        Helper.REMOVE_FLIGHT_ACTION_CODE,
        expectedFlightDto.getFlightNumber());
    mockLog.setExpectedNumberCalls(1);
    // Установка подставного объекта
    FlightManagementFacade facade = new FlightManagementFacadeImpl();
    facade.setAuditLog(mockLog);
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertFalse("flight still exists after being removed",
                facade.flightExists( expectedFlightDto.
                                     getFlightNumber() ) );
    mockLog.verify();
}

```

Данное решение основано на следующем созданном вручную *подставном объекте* (Mock Object). Для экономии места здесь показан только метод logMessage.

```

public void logMessage(Date actualDate,
                      String actualUser,

```

```

        String actualActionCode,
        Object actualDetail) {
actualNumberCalls++;
Assert.assertEquals("date", expectedDate, actualDate);
Assert.assertEquals("user", expectedUser, actualUser);
Assert.assertEquals("action code",
                    expectedActionCode,
                    actualActionCode);
Assert.assertEquals("detail", expectedDetail, actualDetail);
}

```

Методы с утверждением (Assertion Method) вызываются как статические методы. В JUnit такое решение является обязательным, поскольку класс *подставного объекта* (Mock Object) не является подклассом TestCase, а значит, не наследует утверждения от класса Assert. В других реализациях xUnit могут предоставляться другие механизмы доступа к *методам с утверждением* (Assertion Method). Например, в NUnit предоставляются только статические методы класса Assert, поэтому даже *тестовые методы* (Test Method, с. 378) вынуждены, таким образом, обращаться к *методам с утверждением* (Assertion Method). В Test::Unit, реализации xUnit для языка Ruby, методы предоставляются через **миксины** (mixin). В результате они могут вызываться как обычно.

Пример: динамически генерируемый подставной объект (Mock Object)

В последнем примере использовался созданный вручную *подставной объект* (Mock Object). Но в большинстве реализаций xUnit предоставляются инфраструктуры динамической генерации таких объектов. Ниже показан тот же тест, переписанный с использованием инфраструктуры JMock.

```

public void testRemoveFlight_JMock() throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnonRegFlight();
    FlightManagementFacade facade = new FlightManagementFacadeImpl(
        // Настройка подставного объекта
        Mock mockLog = mock(AuditLog.class);
        mockLog.expects(once()).method("logMessage")
            .with(eq(helper.getTodaysDateWithoutTime()),
                  eq(Helper.TEST_USER_NAME),
                  eq(Helper.REMOVE_FLIGHT_ACTION_CODE),
                  eq(expectedFlightDto.getFlightNumber()));
    // Установка подставного объекта
    facade.setAuditLog((AuditLog) mockLog.proxy());
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertFalse("flight still exists after being removed",
               facade.flightExists(expectedFlightDto.
                                    getFlightNumber()));
    // Метод verify() автоматически вызывается инфраструктурой JMock
}

```

Обратите внимание на “прозрачный” *конфигурационный интерфейс* (Configuration Interface), предоставленный инфраструктурой JMock для простого описания ожидаемых вызовов методов. Кроме того, JMock позволяет описывать компаратор, который должен

применяться утверждениями. В этом случае вызов `eq` приводит к вызову принятого по умолчанию метода `equals`.

Источники дополнительной информации

Практически в каждой книге по автоматизированному тестированию с помощью xUnit существует раздел, посвященный *подставным объектам* (Mock Object), поэтому такие ресурсы здесь не перечислены. Читая другие книги не забывайте, что термин *подставной объект* (Mock Object) используется для описания *тестовых заглушек* (Test Stub), а иногда и *поддельных объектов* (Fake Object). Более подробно терминология из различных книг и статей сравнивается в приложении Б.

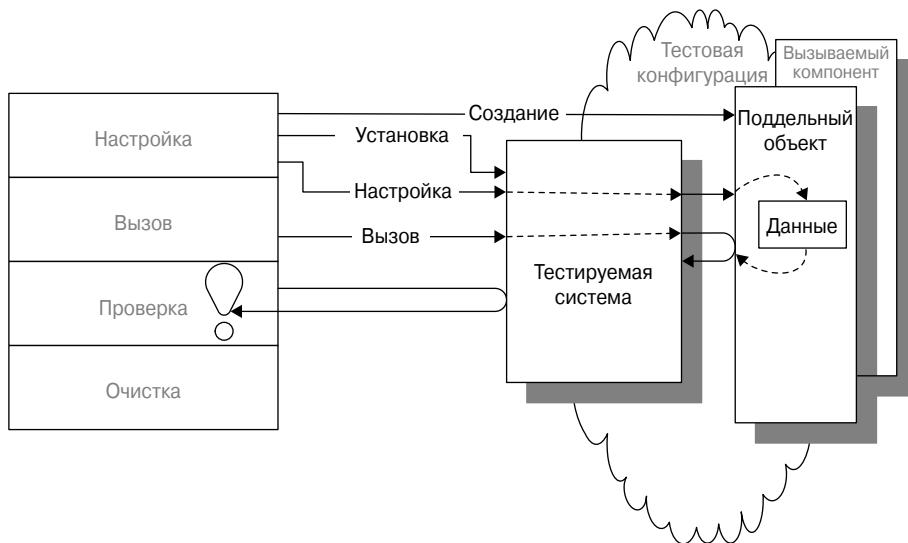
Поддельный объект (Fake Object)

Как независимо проверить логику, если вызываемый компонент использовать нельзя?

Компонент, от которого зависит тестируемая система, заменяется более простой реализацией.

Также известен как:

Кукла (Dummy)



Тестируемая система часто зависит от других компонентов или систем. Хотя взаимодействие с такими компонентами может быть обязательным, побочные эффекты в исполнении настоящего вызываемого компонента могут быть нежелательны или вообще недопустимы.

Поддельный объект (Fake Object) представляет собой значительно более простую реализацию функциональности, предоставляемой вызываемым компонентом, но без нежелательных побочных эффектов.

Как это работает

Создается или обнаруживается очень простая реализация той же функциональности, которая предоставляется компонентами, от которых зависит тестируемая система. После этого системе дается команда использовать новую реализацию вместо настоящего вызываемого компонента. Такая реализация может не иметь “особенностей” настоящего вызываемого компонента, например может не поддаваться масштабированию, но должна предоставлять эквивалентные службы в ответ на запросы тестируемой системы. Тестируемая система не должна подозревать, что используется не настоящий вызываемый компонент.

Поддельный объект (Fake Object) является одним из вариантов *тестового двойника* (Test Double, с. 538) и во многом напоминает *тестовую заглушку* (Test Stub, с. 544), вклю-

чая необходимость вставки в тестируемую систему через механизм заменяемой зависимости. В то время как *тестовая заглушка* (Test Stub) выступает в роли точки управления для вставки опосредованного ввода, *поддельный объект* (Fake Object) эту задачу не решает: он просто обеспечивает возможность взаимодействия. Такое взаимодействие (между тестируемой системой и *поддельным объектом*, Fake Object) происходит достаточно часто и переданные в качестве аргументов методов значения часто возвращаются в качестве результатов при вызове других методов. Это поведение отличает *поддельный объект* (Fake Object) от *тестовых заглушек* (Test Stub) и *подставных объектов* (Mock Object, с. 558), в которых ответы на запросы зафиксированы в коде или настраиваются тестом.

Хотя обычно тест не настраивает *поддельный объект* (Fake Object), сложная настройка тестовой конфигурации, предполагающая инициализацию состояния вызываемого компонента, в случае *поддельного объекта* (Fake Object) может выполняться непосредственно с помощью *манипуляций через “черный ход”* (Back Door Manipulation, с. 359). Такие механизмы, как *загрузчик данных* (Data Loader) и *настройка через “черный ход”* (Back Door Setup), можно использовать без опасений получить *зарегулированную программу* (Over-specified Software), так как привязка осуществляется только к интерфейсу между *поддельным объектом* (Fake Object) и тестируемой системой. Интерфейс настройки *поддельного объекта* (Fake Object) интересен только в пределах теста.

Когда это использовать

Поддельный объект (Fake Object) желательно использовать в ситуациях, когда тестируемая система зависит от недоступных, замедляющих или затрудняющих тестиирование компонентов, а тесты зависят от более сложных последовательностей поведения, чем можно реализовать в пределах *тестовой заглушки* (Test Stub) или *подставного объекта* (Mock Object). Использование *поддельного объекта* (Fake Object) оправдано, если быстрее создать более простую реализацию, чем создать и запрограммировать подходящий *подставной объект* (Mock Object).

С помощью *поддельного объекта* (Fake Object) можно избежать появления *зарегулированной программы* (Over-specified Software), так как ожидаемые вызовы не фиксируются в пределах теста. Тестируемая система может изменять количество и последовательность обращений к вызываемому компоненту, и это не приведет к неудачному завершению теста.

Если необходимо управлять опосредованным вводом или проверять опосредованный вывод тестируемой системы, следует использовать *подставной объект* (Mock Object) или *тестовую заглушки* (Test Stub).

Ниже рассматриваются конкретные ситуации, в которых настоящий компонент заменяется *поддельным объектом* (Fake Object).

Вариант: поддельная база данных (Fake Database)

При использовании шаблона *поддельная база данных* (Fake Database) настоящая база данных или уровень хранения заменяется *поддельным объектом* (Fake Object) с эквивалентной функциональностью, но со значительно лучшими характеристиками быстродействия. Автору часто приходилось заменять базу данных набором хранящихся в памяти хэш-таблиц, которые служили быстрым механизмом извлечения объектов, “сохраненных” ранее во время работы теста.

Вариант: база данных в памяти (In-Memory Database)

Еще одним примером *поддельного объекта* (*Fake Object*) может служить небольшая не использующая диск база данных, установленная вместо полноценной базы данных, хранящей данные на жестком диске. Подобная *база данных в памяти* (*In-Memory Database*) значительно ускоряет работу тестов (как минимум на порядок), но обеспечивает большую функциональность, чем *поддельная база данных* (*Fake Database*).

Вариант: поддельная Web-служба (Fake Web Service)

При тестировании программного обеспечения, зависящего от других компонентов, доступных через Web-службы, можно создать фиксированную или основанную на данных реализацию, которая используется вместо настоящей Web-службы и позволяет ускорить работу тестов. При этом можно отказаться от создания экземпляра настоящей Web-службы в тестовой среде.

Вариант: поддельный уровень обслуживания (Fake Service Layer)

При тестировании пользовательских интерфейсов можно избежать *чувствительности к данным* (*Data Sensitivity*) и *чувствительности к поведению* (*Behavior Sensitivity*), заменив компонент с реализацией **служебного уровня** (*service layer*) *поддельным объектом* (*Fake Object*), возвращающим записанные или основанные на данных результаты. Такой подход позволяет уделить основное внимание тестированию пользовательского интерфейса, не обращая внимание на изменение возвращаемых данных с течением времени.

Замечания по реализации

Использование *поддельного объекта* (*Fake Object*) предусматривает две операции:

- создание *поддельного объекта* (*Fake Object*);
- установку *поддельного объекта* (*Fake Object*).

Создание поддельного объекта (Fake Object)

Большинство *поддельных объектов* (*Fake Object*) создается вручную. Часто *поддельный объект* (*Fake Object*) используется для замены настоящей реализации, страдающей от проблем задержек при обмене сообщениями или обмене данными с диском, более легкой реализацией, умеющейся в памяти. Учитывая наличие богатых библиотек в большинстве языков программирования, обычно достаточно легко создать поддельную реализацию, удовлетворяющую потребности тестируемой системы (как минимум в пределах конкретного теста).

Достаточно популярна стратегия, при которой создание *поддельного объекта* (*Fake Object*) начинается с поддержки конкретного набора тестов, в которых тестируемой системе требуется доступ к подмножеству функций вызываемого компонента. При положительном результате можно попытаться расширить *поддельный объект* (*Fake Object*) для поддержки дополнительных тестов. Со временем оказывается, что все тесты могут работать с использованием *поддельного объекта* (*Fake Object*). (Замена базы данных хеш-таблицами с 50-кратным приростом быстродействия описывается во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351.)

Установка поддельного объекта (Fake Object)

Конечно, для использования преимуществ *поддельного объекта* (Fake Object) должен существовать способ его установки в тестируемую систему. Можно использовать любой поддерживаемый тестируемой системой шаблон замены зависимости. При разработке на основе тестов часто применяется шаблон *вставки зависимости* (Dependency Injection, с. 684). Приверженные традиционным подходам разработчики могут предпочтеть *поиск зависимости* (Dependency Lookup, с. 692). Последний способ больше подходит для установки *поддельной базы данных* (Fake Database), позволяющей ускорить работу приемочных тестов. *Вставка зависимости* (Dependency Injection) с такими тестами работает не очень хорошо.

Мотивирующий пример

В приведенном ниже примере тестируемая система должна записывать и считывать записи из базы данных. Тест создает конфигурацию в базе данных (несколько записей), тестируемая система инициирует еще несколько операций чтения и записи в базу данных, после чего тест удаляет записи из базы данных (несколько операций удаления). Все операции требуют времени (несколько секунд на тест). Секунды складываются в минуты, и очень быстро разработчики отказываются от частого запуска тестов. Ниже показан один из таких тестов.

```
public void testReadWrite() throws Exception{
    // Настройка
    FlightMngtFacade facade = new FlightMngtFacadeImpl();
    BigDecimal yyC = facade.createAirport("YYC", "Calgary", "Calgary");
    BigDecimal lax = facade.createAirport("LAX", "LAX Intl", "LA");
    facade.createFlight(yyC, lax);
    // Вызов
    List flights = facade.getFlightsByOriginAirport(yyC);
    // Проверка
    assertEquals("# of flights", 1, flights.size());
    Flight flight = (Flight) flights.get(0);
    assertEquals("origin",
                yyC, flight.getOrigin().getCode());
}
```

Тест вызывает метод `createAirport` **фасада служб** (service facade), который, кроме всего прочего, вызывает уровень доступа к данным. Ниже показана реализация нескольких вызываемых методов.

```
public BigDecimal createAirport(String airportCode,
                                String name,
                                String nearbyCity)
throws FlightBookingException{
    TransactionManager.beginTransaction();
    Airport airport = dataAccess.
        createAirport(airportCode, name, nearbyCity);
    logMessage("Wrong Action Code", airport.getCode()); //bug
    TransactionManager.commitTransaction();
    return airport.getId();
}
public List getFlightsByOriginAirport(
    BigDecimal originAirportId)
```

```

throws FlightBookingException {
if (originAirportId == null)
    throw new InvalidArgumentException(
        "Origin Airport Id has not been provided",
        "originAirportId", null);
Airport origin = dataAccess.getAirportByPrimaryKey(originAirportId);
List flights = dataAccess.getFlightsByOriginAirport(origin);
return flights;
}

```

Вызовы `dataAccess.createAirport`, `dataAccess.createFlight` и `TransactionManager.commitTransaction` больше всего замедляют работу теста. Вызовы `dataAccess.getAirportByPrimaryKey` и `dataAccess.getFlightsByOriginAirport` влияют на снижение скорости меньше.

Замечания по рефакторингу

Этапы добавления *поддельного объекта* (*Fake Object*) очень напоминают процесс вставки *подставного объекта* (*Mock Object*). Если объект еще не существует, воспользуйтесь рефакторингом заменить зависимость тестовым двойником (*Replace Dependency with Test Double*, с. 740) для обеспечения механизма замены вызываемого компонента *поддельным объектом* (*Fake Object*); обычно для хранения ссылки на объект используется поле (атрибут). В статически типизированных языках для добавления поддельной реализации, возможно, придется прибегнуть к рефакторингу *выделение метода* (*Extract Method*). После этого полученный интерфейс используется как тип переменной, хранящей ссылку на заменяемую зависимость.

Одним из отличий является отсутствие необходимости настраивать *поддельный объект* (*Fake Object*) с помощью ожидаемых вызовов и возвращаемых значений; тестовая конфигурация настраивается как обычно.

Пример: поддельная база данных (*Fake Database*)

В данном примере создан *поддельный объект* (*Fake Object*), заменяющий собой базу данных, т.е. *поддельная база данных* (*Fake Database*) реализована на основе хэш-таблиц, полностью хранящихся в памяти. Тест практически не меняется, но скорость его выполнения значительно возрастает.

```

public void testReadWrite_inMemory() throws Exception{
    // Настройка
    FlightMgmtFacadeImpl facade = new FlightMgmtFacadeImpl();
    facade.setDao(new InMemoryDatabase());
    BigDecimal yyc = facade.createAirport("YYC", "Calgary", "Calgary");
    BigDecimal lax = facade.createAirport("LAX", "LAX Intl", "LA");
    facade.createFlight(yyc, lax);
    // Вызов
    List flights = facade.getFlightsByOriginAirport(yyc);
    // Проверка
    assertEquals("# of flights", 1, flights.size());
    Flight flight = (Flight) flights.get(0);
    assertEquals("origin",
                yyc, flight.getOrigin().getCode());
}

```

Ниже представлена реализация *поддельной базы данных* (Fake Database).

```
public class InMemoryDatabase implements FlightDao{
    private List airports = new Vector();
    public Airport createAirport(String airportCode,
                                 String name, String nearbyCity)
        throws DataException, InvalidArgumentException {
        assertParamtersAreValid(airportCode, name, nearbyCity);
        assertAirportDoesntExist(airportCode);
        Airport result = new Airport(getNextAirportId(),
                                      airportCode, name, createCity(nearbyCity));
        airports.add(result);
        return result;
    }
    public Airport getAirportByPrimaryKey(BigDecimal airportId)
        throws DataException, InvalidArgumentException {
        assertAirportNotNull(airportId);
        Airport result = null;
        Iterator i = airports.iterator();
        while (i.hasNext()) {
            Airport airport = (Airport) i.next();
            if (airport.getId().equals(airportId)) {
                return airport;
            }
        }
        throw new DataException("Airport not found:"+airportId);
    }
}
```

Осталось создать реализацию метода, устанавливающего *поддельную базу данных* (Fake Database) в фасад, и позволить разработчикам запускать все тесты после каждой модификации кода.

```
public void setDao(FlightDao) {
    dataAccess = dao;
}
```

Источники дополнительной информации

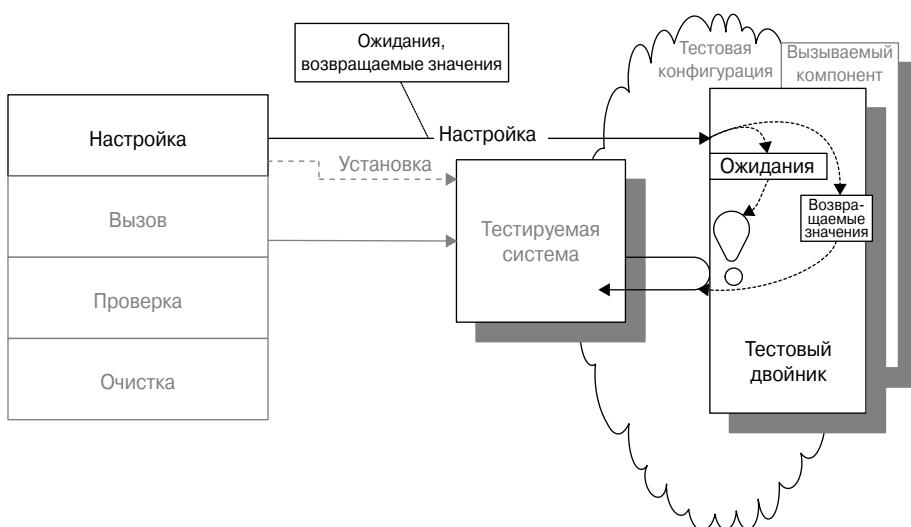
Во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351 замена базы данных хэш-таблицами с 50-кратным приростом быстродействия описывалась более подробно. В приложении Б приводится полное сравнение терминологии, используемой в книгах и статьях по этой теме.

Настраиваемый тестовый двойник (Configurable Test Double)

Как сообщить тестовому двойнику (Test Double), каких вызовов ожидать и какие значения возвращать?

Создается пригодный для повторного использования тестовый двойник (Test Double), получающий возвращаемые и проверяемые значения на этапе настройки тестовой конфигурации.

Также известен как:
Настраиваемый подставной объект (Configurable Mock Object), Настраиваемый тестовый агент (Configurable Test Spy), Настраиваемая тестовая заглушка (Configurable Test Stub)



В некоторых тестах уникальные значения должны вставляться в тестируемую систему в качестве опосредованного ввода или проверяться в качестве опосредованного вывода. При таком подходе обычно следует использовать *тестовые двойники* (Test Double, с. 538) в качестве проводников между тестируемой системой и тестом. В то же время *тестовому двойнику* (Test Double) необходимо как-то сообщить возвращаемые или проверяемые значения.

Настраиваемый тестовый двойник (Configurable Test Double) позволяет сократить *дублирование тестового кода* (Test Code Duplication, с. 254) за счет повторного использования *тестового двойника* (Test Double) в нескольких тестах. Ключом к повторному использованию является настройка проверяемых или возвращаемых значений на этапе выполнения.

Как это работает

Тестовый двойник (Test Double) создается с переменными экземпляра, в которых хранятся возвращаемые или ожидаемые значения аргументов вызываемых методов. Тест инициализирует эти переменные на этапе создания тестовой конфигурации, вызывая соответствующие методы интерфейса *тестового двойника* (Test Double). При вызове методов *тестового двойника* (Test Double) со стороны тестируемой системы двойник исполь-

зует содержимое подходящей переменной в качестве возвращаемого значения или параметра утверждений.

Когда это использовать

Настраиваемый тестовый двойник (Configurable Test Double) может использоваться каждый раз, когда необходимо похожее, но немного отличающееся поведение в нескольких зависящих от *тестовых двойников* (Test Double) тестах и необходимо избежать *дублирования тестового кода* (Test Code Duplication) и *непонятных тестов* (Obscure Test, с. 230) (для этого используемые *тестовым двойником* (Test Double) значения должны быть видны при чтении теста). Если тестовый двойник будет применяться в единственном экземпляре, следует воспользоваться *фиксированным тестовым двойником* (Hard-Coded Test Double, с. 581), так как дополнительная сложность при создании *настраиваемого тестового двойника* (Configurable Test Double) может не окупиться в дальнейшем.

Замечания по реализации

Тестовый двойник (Test Double) называется настраиваемым, поскольку он предоставляет тестам возможность настройки с указанием возвращаемых значений и/или ожидаемых аргументов методов. *Настраиваемая тестовая заглушка* (Configurable Test Stub, с. 571) и *настраиваемый тестовый агент* (Configurable Test Spy, с. 571) нуждаются в способе настройки ответов на вызовы методов. *Настраиваемый подставной объект* (Configurable Mock Object, с. 571) нуждается в способе настройки ожидаемого поведения (какие методы и с какими аргументами должны вызываться).

Существует много способов создания *настраиваемых тестовых двойников* (Configurable Test Double). Выбор конкретной реализации основан на двух относительно независимых решениях:

- 1) как будет настраиваться двойник;
- 2) как он будет закодирован.

Имеется два распространенных способа настройки тестового двойника. Наиболее популярный подход предполагает предоставление *конфигурационного интерфейса* (Configuration Interface), который используется тестом для настройки возвращаемых в качестве опосредованного ввода значений или указания ожидаемых значений опосредованного вывода. С другой стороны, *настраиваемый тестовый двойник* (Configurable Test Double) может иметь два режима. *Режим настройки* (Configuration Mode) используется во время создания тестовой конфигурации для указания опосредованного ввода или ожидаемого опосредованного вывода через вызовы методов *настраиваемого тестового двойника* (Configurable Test Double) с ожидаемыми аргументами. Перед установкой *настраиваемый тестовый двойник* (Configurable Test Double) переводится в нормальный режим (“режим воспроизведения”).

Очевидным способом создания *настраиваемого тестового двойника* (Configurable Test Double) является написание соответствующего кода вручную. Но в некоторых случаях уже имеются инструменты для генерации настраиваемых двойников. Существует два типа подобных генераторов: генераторы кода и инструменты, создающие объекты на этапе выполнения. Было разработано несколько поколений подобных инструментов, и некоторые из

них были перенесены на другие языки программирования. Список доступных генераторов для используемого языка программирования можно найти на сайте <http://xprogramming.com>. Если поиск не дал результата, можно написать *тестовый двойник* (Test Double) самостоятельно, хотя это и потребует дополнительных усилий.

Вариант: конфигурационный интерфейс (Configuration Interface)

Конфигурационный интерфейс (Configuration Interface) состоит из набора методов, предоставляемых *настраиваемым тестовым двойником* (Configurable Test Double) специально для установки возвращаемых и ожидаемых значений. Тест просто вызывает эти методы на этапе создания тестовой конфигурации. Тестируемая система использует “другие” методы *настраиваемого тестового двойника* (Configurable Test Double), составляющие “нормальный” интерфейс, и не подозревает о существовании *конфигурационного интерфейса* (Configuration Interface).

Существует два типа конфигурационного интерфейса. Ранние версии средств разработки, например MockMaker, генерировали отдельный метод для каждого настраиваемого значения. Набор этих методов установки и составлял *конфигурационный интерфейс* (Configuration Interface). Более поздние решения, например JMock, предоставляют универсальный интерфейс, используемый для создания *спецификации ожидаемого поведения* (Expected Behavior Specification), которая интерпретируется *настраиваемым тестовым двойником* (Configurable Test Double) на этапе выполнения. Хорошо продуманный интерфейс может значительно упростить чтение и понимание кода теста.

Вариант: режим настройки (Configuration Mode)

Можно избежать определения дополнительного набора методов для настройки *тестового двойника* (Test Double) и воспользоваться *режимом настройки* (Configuration Mode), в котором тест “обучает” *настраиваемый тестовый двойник* (Configurable Test Double) ожидаемому поведению. Такой способ настройки двойника может ввести в заблуждение: почему *тестовый метод* (Test Method, с. 378) вызывает методы другого объекта раньше методов тестируемой системы? Осознание, что этот подход является вариантом “записи и воспроизведения”, значительно упрощает понимание данной концепции.

Основным преимуществом использования *режима настройки* (Configuration Mode) является возможность не создавать отдельный набор методов для настройки двойника за счет повторного использования методов,ываемых тестируемой системой. (Разработчику приходится обеспечивать установку возвращаемых значений методов, поэтому требуется как минимум один дополнительный метод.) С другой стороны, каждый вызываемый тестируемой системой метод имеет две ветви выполнения: одну — для *режима настройки* (Configuration Mode), вторую — для режима использования.

Вариант: созданный вручную тестовый двойник (Hand-Built Test Double)

Созданный вручную тестовый двойник (Hand-Built Test Double) определяется разработчиком для одного или нескольких конкретных тестов. *Фиксированный тестовый двойник* (Hard-Coded Test Double) по определению является *созданным вручную тестовым двойником* (Hand-Built Test Double), а *настраиваемый тестовый двойник* (Configurable Test Double) может быть созданным вручную или сгенерированным. В данной книге *созданные вручную тестовые двойники* (Hand-Built Test Double) используются в

большом количестве примеров, так как происходящее значительно проще понять, рассматривая простой код. Это и есть основное преимущество созданных вручную тестовых двойников. Некоторые считают его настолько важным, что используют исключительно созданные вручную двойники. К такому варианту можно прибегнуть и при недоступности сторонних инструментов или невозможности их использования в соответствии с принятой политикой группы разработки продукта или компании.

Вариант: статически генерированный тестовый двойник (Statically Generated Test Double)

В ранних версиях средств разработки для создания *статически генерированных тестовых двойников* (Statically Generated Test Double) использовались генераторы кода. После этого код компилировался и связывался с написанным вручную кодом теста. Обычно код находится в хранилище исходного кода. Конечно, при каждом изменении интерфейса класса приходится повторно генерировать код *статически генерированного тестового двойника* (Statically Generated Test Double). Желательно включить эту операцию в сценарий автоматической компиляции, чтобы она гарантированно происходила при модификации интерфейса класса.

Создание экземпляра *статически генерированного тестового двойника* (Statically Generated Test Double) не отличается от создания экземпляра *созданного вручную тестового двойника* (Hand-Built Test Double), т.е. имя генерированного класса используется для создания *настраиваемого тестового двойника* (Configurable Test Double).

Во время рефакторинга возникает интересная проблема. Предположим, интерфейс заменяемого класса меняется за счет добавления аргумента в один из методов. Нужно ли выполнять рефакторинг генерированного кода? Или *статически генерированный тестовый двойник* (Statically Generated Test Double) должен быть повторно генерирован после рефакторинга заменяемого им кода? При наличии современных инструментов рефакторинга проще за одну операцию модифицировать генерированный код и использующий его тест, но такая стратегия может оставить двойник без логики проверки аргумента или переменных для нового параметра. Таким образом, стоит повторно генерировать *статически генерированный тестовый двойник* (Statically Generated Test Double) после завершения рефакторинга, чтобы обеспечить его правильную работу и гарантировать возможность его воссоздания с помощью генератора кода.

Вариант: динамически генерируемый тестовый двойник (Dynamically Generated Test Double)

Новые средства разработки от сторонних разработчиков позволяют генерировать *настраиваемый тестовый двойник* (Configurable Test Double) на этапе выполнения с помощью механизма интроспекции используемого языка программирования. Утилита просматривает класс или интерфейс и создает объект, способный понять все вызовы его методов. Такие *настраиваемые тестовые двойники* (Configurable Test Double) могут интерпретировать спецификацию поведения на этапе выполнения или сразу генерировать выполняемый код. В любом случае не существует генерируемого исходного кода, который приходится обслуживать и генерировать повторно. Недостатком этого подхода является отсутствие кода, на который можно посмотреть.

Большинство современных инструментов генерируют *подставные объекты* (Mock Object), так как они используются чаще всего. Эти объекты можно использовать и в качестве *тестовых заглушек* (Test Stub), поскольку они позволяют устанавливать возвращаемые значения конкретных методов. Если проверять вызовы методов или передаваемых аргументов не нужно, в большинстве инструментов поддерживается возможность перечисления “не интересующих” тест аргументов. Учитывая, что большинство инструментов генерируют *подставные объекты* (Mock Object), обычно они не предоставляют *интерфейс извлечения* (Retrieval Interface; см. *Тестовый агент*, Test Spy).

Мотивирующий пример

Ниже показан тест, использующий *фиксированный тестовый двойник* (Hard-Coded Test Double) для управления временем.

```
public void testDisplayCurrentTime_AtMidnight_HCM()
    throws Exception {
    // Настройка тестовой конфигурации
    // Создание экземпляра фиксированной тестовой заглушки
    TimeProvider testStub = new MidnightTimeProvider();
    // Создание экземпляра тестируемой системы
    TimeDisplay sut = new TimeDisplay();
    // Вставка заглушки в тестируемую систему
    sut.setTimeProvider(testStub);
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка непосредственного вывода
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}
```

Тест сложно понять, не посмотрев на реализацию *фиксированного тестового двойника* (Hard-Coded Test Double). Если определение не находится поблизости, легко представить, как оно способствует появлению *тайного гостя* (Mystery Guest).

```
class MidnightTimeProvider implements TimeProvider {
    public Calendar getTime() {
        Calendar myTime = new GregorianCalendar();
        myTime.set(Calendar.HOUR_OF_DAY, 0);
        myTime.set(Calendar.MINUTE, 0);
        return myTime;
    }
}
```

Проблему *непонятного теста* (Obscure Test) можно решить с помощью *тестового шунта* (Self Shunt), сделав *фиксированный тестовый двойник* (Hard-Coded Test Double) видимым внутри теста.

```
public class SelfShuntExample extends TestCase
implements TimeProvider {
    public void testDisplayCurrentTime_AtMidnight() throws Exception {
        // Настройка тестовой конфигурации
        TimeDisplay sut = new TimeDisplay();
        // Настройка подставного объекта
        sut.setTimeProvider(this); // установка тестового шунта
    }
}
```

```

// Вызов тестируемой системы
String result = sut.getCurrentTimeAsHtmlFragment();
// Проверка непосредственного результата
String expectedTimeString =
    "<span class=\"tinyBoldText\">Midnight</span>";
assertEquals("Midnight", expectedTimeString, result);
}
public Calendar getTime() {
    Calendar myTime = new GregorianCalendar();
    myTime.set(Calendar.MINUTE, 0);
    myTime.set(Calendar.HOUR_OF_DAY, 0);
    return myTime;
}
}

```

К сожалению, поведение *тестового двойника* (Test Double) придется встраивать в каждый класс теста (Testcase Class, с. 401), которому оно требуется, что приведет к *дублированию тестового кода* (Test Code Duplication).

Замечания по рефакторингу

Рефакторинг теста, использующего *фиксированный тестовый двойник* (Hard-Coded Test Double), в направлении использования стороннего настраиваемого тестового двойника (Configurable Test Double) выполняется относительно просто. Достаточно следовать инструкциям, предоставленным вместе со средством разработки, для создания *настраиваемого тестового двойника* (Configurable Test Double). После этого двойник настраивается с использованием тех же значений, что и *фиксированный тестовый двойник* (Hard-Coded Test Double). Кроме того, часть фиксированной логики двойника придется вынести в *тестовый метод* (Test Method) с последующей передачей двойнику на этапе настройки.

Преобразовать сам *фиксированный тестовый двойник* (Hard-Coded Test Double) в *настраиваемый тестовый двойник* (Configurable Test Double) сложнее, но не намного, если описывается только простое поведение. (Для более сложного поведения имеет смысл рассмотреть существующие средства разработки и их последующий перенос в используемую среду.) Сначала необходимо обеспечить способ установки ожидаемых или возвращаемых значений. Лучше всего начать с модификации теста в соответствии с выбранным механизмом взаимодействия с *настраиваемым тестовым двойником* (Configurable Test Double). После создания экземпляра на этапе настройки тестовой конфигурации подходящие значения передаются в *настраиваемый тестовый двойник* (Configurable Test Double) через *конфигурационный интерфейс* (Configuration Interface) или с использованием *режима настройки* (Configuration Mode). После определения способа взаимодействия с *настраиваемым тестовым двойником* (Configurable Test Double) можно воспользоваться рефакторингом *введение поля* (Introduce field) для создания переменных экземпляра, содержащих фиксированные ранее значения.

Пример: конфигурационный интерфейс (Configuration Interface) на основе методов установки

В следующем примере показано, как тест использует созданный вручную *конфигурационный интерфейс* (Configuration Interface) с помощью *вставки метода установки* (Setter Injection).

```

public void testDisplayCurrentTime_AtMidnight()
    throws Exception {
    // Настройка тестовой конфигурации
    // Настройка тестового двойника
    TimeProviderTestStub tpStub = new TimeProviderTestStub();
    tpStub.setHours(0);
    tpStub.setMinutes(0);
    // Создание экземпляра тестируемой системы
    TimeDisplay sut = new TimeDisplay();
    // Установка тестового двойника
    sut.setTimeProvider(tpStub);
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}

```

Настраиваемый тестовый двойник (Configurable Test Double) реализован следующим образом.

```

class TimeProviderTestStub implements TimeProvider {
    // Конфигурационный интерфейс (Configuration Interface)
    public void setHours(int hours) {
        // 0 -- полночь; 12 -- полдень
        myTime.set(Calendar.HOUR_OF_DAY, hours);
    }
    public void setMinutes(int minutes) {
        myTime.set(Calendar.MINUTE, minutes);
    }
    // Используемый тестируемой системой интерфейс
    public Calendar getTime() {
        // @return последнее установленное время
        return myTime;
    }
}

```

Пример: конфигурационный интерфейс (Configuration Interface) на основе конструктора выражений

Рассмотрим отличие интерфейса, определенного в предыдущем примере, от интерфейса, предоставленного инфраструктурой JMock. Инфраструктура JMock генерирует *подставные объекты* (Mock Object) динамически и предоставляет очевидный и универсальный интерфейс для их настройки. Ниже показан тот же тест, но преобразованный для использования инфраструктуры JMock.

```

public void testDisplayCurrentTime_AtMidnight_JM()
    throws Exception {
    // Настройка тестовой конфигурации
    TimeDisplay sut = new TimeDisplay();
    // Настройка тестового двойника
    Mock tpStub = mock(TimeProvider.class);
    Calendar midnight = makeTime(0, 0);
    tpStub.stubs().method("getTime").

```

```

        withNoArguments() .
        will(returnValue(midnight));
    // Установка тестового двойника
    sut.setTimeProvider((TimeProvider) tpStub);
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}

```

В данном случае логика создания возвращаемого времени перенесена в *класс теста* (Testcase Class), так как сделать это в универсальной инфраструктуре подставных объектов совершенно невозможно. Для создания возвращаемого времени в данном случае используется *вспомогательный метод теста* (Test Utility Method, с. 610). В следующем примере показан настраиваемый *подставной объект* (Mock Object) и несколько ожидаемых параметров.

```

public void testRemoveFlight_JMock() throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnonRegFlight();
    FlightManagementFacade facade = new FlightManagementFacadeImpl();
    // Настройка подставного объекта
    Mock mockLog = mock(AuditLog.class);
    mockLog.expects(once()).method("logMessage")
        .with(eq(helper.getTodaysDateWithoutTime()),
              eq(Helper.TEST_USER_NAME),
              eq(Helper.REMOVE_FLIGHT_ACTION_CODE),
              eq(expectedFlightDto.getFlightNumber()));
    // Установка подставного объекта
    facade.setAuditLog(AuditLog mockLog.proxy());
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertFalse("flight still exists after being removed",
               facade.flightExists(expectedFlightDto.
                    getFlightNumber()));
    // Метод verify() вызывается автоматически инфраструктурой JMock
}

```

Спецификация ожидаемого поведения (Expected Behavior Specification) создается с помощью вызова таких методов, как expects, once и method, которые описывают, как *настраиваемый тестовый двойник* (Configurable Test Double) должен использоваться и что он должен возвращать. Инфраструктура JMock поддерживает описание значительно более сложного поведения (например, нескольких вызовов одного и того же метода с различными аргументами и возвращаемыми значениями), чем созданный вручную *настраиваемый тестовый двойник* (Configurable Test Double).

Пример: режим настройки (Configuration Mode)

В следующем примере тест модифицирован для использования *подставного объекта* (Mock Object) с *режимом настройки* (Configuration Mode).

```

public void testRemoveFlight_ModalMock() throws Exception {
    // Настройка тестовой конфигурации
    FlightDto expectedFlightDto = createAnonRegFlight();
    // Настройка подставного объекта (в режиме настройки)
    ModalMockAuditLog mockLog = new ModalMockAuditLog();
    mockLog.logMessage(Helper.getTodaysDateWithoutTime(),
        Helper.TEST_USER_NAME,
        Helper.REMOVE_FLIGHT_ACTION_CODE,
        expectedFlightDto.getFlightNumber());
    mockLog.enterPlaybackMode();
    // Установка подставного объекта
    FlightManagementFacade facade = new FlightManagementFacadeImpl();
    facade.setAuditLog(mockLog);
    // Вызов
    facade.removeFlight(expectedFlightDto.getFlightNumber());
    // Проверка
    assertFalse("flight still exists after being removed",
        facade.flightExists(expectedFlightDto.
            getFlightNumber()));
    mockLog.verify();
}

```

В данном случае тест вызывает методы *настраиваемого тестового двойника* (Configurable Test Double) во время создания тестовой конфигурации. Если заранее неизвестно, что тест использует подставной объект, структура теста может показаться непонятной. Наиболее очевидным указателем на суть теста является вызов метода `enterPlaybackMode`, который вынуждает *настраиваемый тестовый двойник* (Configurable Test Double) прекратить сохранение ожидаемых значений и использовать полученные параметры в утверждениях.

Ниже показан используемый тестом *настраиваемый тестовый двойник* (Configurable Test Double).

```

private int mode = record;
public void enterPlaybackMode() {
    mode = playback;
}
public void logMessage(Date date,
    String user,
    String action,
    Object detail) {
    if (mode == record) {
        Assert.assertEquals("Only supports 1 expected call",
            0, expectedNumberCalls);
        expectedNumberCalls = 1;
        expectedDate = date;
        expectedUser = user;
        expectedCode = action;
        expectedDetail = detail;
    } else {
        Assert.assertEquals("Date", expectedDate, date);
        Assert.assertEquals("User", expectedUser, user);
        Assert.assertEquals("Action", expectedCode, action);
        Assert.assertEquals("Detail", expectedDetail, detail);
    }
}

```

Оператор `if` проверяет текущий режим (запись или воспроизведение). Поскольку этот простой созданный вручную *настраиваемый тестовый двойник* (Configurable Test Double) поддерживает сохранение только одного значения, *сторожевое утверждение* (Guard Assertion, с. 510) приводит к неудачному завершению теста при попытке записи больше чем одного вызова метода. Остаток блока `then` сохраняет параметры в переменные, которые используются в качестве ожидаемых значений в *утверждениях равенства* (Equality Assertion) из раздела `else`.

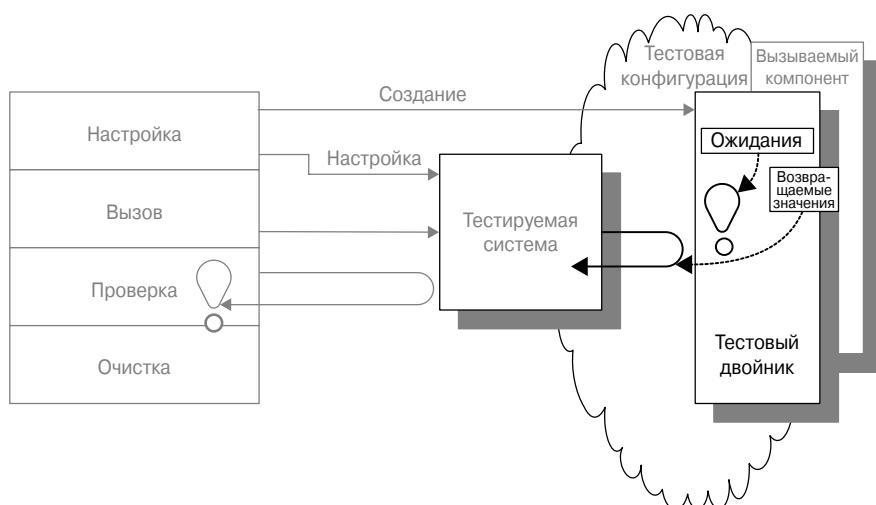
Фиксированный тестовый двойник (Hard-Coded Test Double)

Как сообщить тестовому двойнику (Test Double) возвращаемые или ожидаемые значения?

Тестовый двойник (Test Double) создается вручную с записью возвращаемых и/или ожидаемых значений непосредственно в его коде.

Также известен как:

Фиксированный подставной объект (Hard-Coded Mock Object), Фиксированная тестовая заглушка (Hard-Coded Test Stub), Фиксированный тестовый агент (Hard-Coded Test Spy)



Существует множество причин использования *тестовых двойников* (Test Double, с. 538) во время разработки *полностью автоматизированных тестов* (Fully Automated Tests, с. 81). Поведение *тестового двойника* (Test Double) может отличаться от одного теста к другому. Существует множество способов определения этого поведения.

Если *тестовый двойник* (Test Double) слишком прост или предназначен для использования в единственном teste, наилучшим решением является запись необходимого поведения непосредственно в коде *тестового двойника* (Test Double).

Как это работает

Разработчик тестов записывает поведение *тестового двойника* (Test Double) в его коде. Например, если *тестовый двойник* (Test Double) должен возвращать значение в ответ на вызов метода, это значение записывается в операторе `return`. Если необходимо убедиться, что некоторый параметр имеет конкретное значение, в метод вставляется утверждение с фиксированным ожидаемым значением.

Когда это использовать

Обычно *фиксированный тестовый двойник* (Hard-Coded Test Double) используется для реализации очень простого или очень специализированного поведения *тестового двойника* (Test Double). В качестве *фиксированного тестового двойника* (Hard-Coded Test

`Double`) может выступать *тестовая заглушка* (Test Stub, с. 544), *тестовый агент* (Test Spy, с. 552) или *подставной объект* (Mock Object, с. 558). Выбор зависит от содержимого методов, вызываемых тестируемой системой.

Поскольку *фиксированный тестовый двойник* (Hard-Coded Test Double) пишется разработчиком, его создание может потребовать больших усилий, чем использование *настраиваемых тестовых двойников* (Configurable Test Double, с. 571) на основе средств разработки от сторонних разработчиков. Кроме того, создание двойника вручную приведет к увеличению объема поддерживаемого и перерабатываемого кода при изменениях в тестируемой системе. Если в разных тестах требуется разное поведение *тестового двойника* (Test Double), а использование *фиксированных тестовых двойников* (Hard-Coded Test Double) приводит к большому объему *дублирования тестового кода* (Test Code Duplication, с. 254), рассмотрите использование *настраиваемого тестового двойника* (Configurable Test Double).

Замечания по реализации

Фиксированный тестовый двойник (Hard-Coded Test Double) по своей природе является *созданным вручную тестовым двойником* (Hand-Built Test Double), так как нет никакого смысла генерировать *фиксированный тестовый двойник* (Hard-Coded Test Double) автоматически. Его можно реализовать в виде отдельного класса, но чаще всего он используется в языках программирования, поддерживающих блоки или внутренние классы. Все эти возможности языков позволяют обойти накладные расходы создания файлов и/или классов, связанные с созданием тестового двойника вручную. Кроме того, поведение двойника описывается недалеко от точки его использования. В некоторых языках это может усложнить чтение тестов, особенно при использовании анонимных внутренних классов, с определением которых связаны значительные синтаксические накладные расходы. В языках с непосредственной поддержкой блоков знакомые с подходящими идиомами разработчики могут использовать *фиксированные тестовые двойники* (Hard-Coded Test Double) для упрощения чтения теста.

Существует несколько способов реализации *фиксированного тестового двойника* (Hard-Coded Test Double). Каждый из них обладает своими преимуществами и недостатками.

Вариант: класс тестового двойника (Test Double Class)

Фиксированный тестовый двойник (Hard-Coded Test Double) можно реализовать в классе, отдельном от тестируемой системы или *класса теста* (Testcase Class). Это позволяет повторно использовать *фиксированный тестовый двойник* (Hard-Coded Test Double) в нескольких *классах теста* (Testcase Class), но может привести к появлению *непонятных тестов* (Obscure Test), так как важный опосредованный ввод-вывод тестируемой системы выносится за пределы теста (и, возможно, за пределы досягаемости читателя теста). В зависимости от реализации *класса тестового двойника* (Test Double Class) результатом может стать увеличение объема кода, который необходимо поддерживать.

Одним из способов сохранения совместимости по типам с компонентом является реализация *класса тестового двойника* (Test Double Class) как подкласса заменяемого компонента. После создания подкласса достаточно переопределить методы, поведение которых необходимо изменить.

Вариант: подкласс как тестовый двойник (Test Double Subclass)

Фиксированный тестовый двойник (Hard-Coded Test Double) можно реализовать и в виде подкласса настоящего вызываемого компонента, переопределяя поведение методов, которые должны вызываться тестируемой системой. К сожалению, такой подход имеет непредсказуемые последствия, если тестируемая система вызовет не переопределенные методы. Кроме того, код теста тесно привязывается к реализации вызываемого компонента, что может привести к появлению зарегулированной программы (Overspecified Software). Использование *подкласса как тестового двойника* (Test Double Subclass) может быть оправдано в очень специфических ситуациях (например, когда другие варианты недоступны), но, в целом, использовать такую стратегию не рекомендуется.

Вариант: тестовый шунт (Self Shunt)

Можно реализовать вызываемые тестируемой системой методы в *классе теста* (Testcase Class) и установить *объект теста* (Testcase Object, с. 410) в качестве *тестового двойника* (Test Double). Такое решение называется *тестовым шунтом* (Self Shunt).

Также известен как:
Петля (Loopback), Класс теста как тестовый двойник (Testcase Class as Test Double)

В качестве *тестового шунта* (Self Shunt) могут использоваться *тестовая заглушка* (Test Stub), *тестовый агент* (Test Spy) и *подставной объект* (Mock Object). Выбор зависит от поведения вызываемых тестируемой системой методов. В каждом случае методу потребуется доступ к переменным экземпляра *класса теста* (Testcase Class) для получения информации о необходимых действиях или ожидаемых значениях. В статически типизированных языках *класс теста* (Testcase Class) должен реализовывать интерфейс, от которого зависит тестируемая система.

Обычно *тестовый шунт* (Self Shunt) используется для создания *фиксированного тестового двойника* (Hard-Coded Test Double), предназначенного для конкретного *класса теста* (Testcase Class). Если только один *тестовый метод* (Test Method, с. 378) нуждается в использовании *фиксированного тестового двойника* (Hard-Coded Test Double), *внутренний тестовый двойник* (Inner Test Double) при наличии адекватной поддержки со стороны языка программирования может значительно упростить понимание теста.

Вариант: внутренний тестовый двойник (Inner Test Double)

Популярным способом реализации *фиксированного тестового двойника* (Hard-Coded Test Double) является его запись в виде анонимного внутреннего класса или блочной конструкции внутри *тестового метода* (Test Method). Такая стратегия предоставляет *тестовому двойнику* (Test Double) доступ к переменным и константам экземпляра *класса теста* (Testcase Class) и даже к локальным переменным *тестового метода* (Test Method), что позволяет отказаться от настройки *тестового двойника* (Test Double).

Хотя название этого варианта связано с конструкцией в языке Java, во многих языках программирования существует эквивалентный механизм определения кода для последующего запуска, основанный на блоках или блочных конструкциях.

Обычно *внутренний тестовый двойник* (Inner Test Double) используется при создании относительно простых *фиксированных тестовых двойников* (Hard-Coded Test Double), которые применяются в пределах единственного *тестового метода* (Test Method). Многие считают *внутренний тестовый двойник* (Inner Test Double) более интуитивно понятным,

чем *тестовый шунт* (Self Shunt), так как происходящее очевидно из кода *тестового метода* (Test Method). Но у читателей, незнакомых с синтаксисом анонимных внутренних классов или блоков, могут возникнуть сложности с пониманием.

Вариант: псевдообъект (Pseudo-Object)

Одной из сложностей при создании *фиксированного тестового двойника* (Hard-Coded Test Double) является необходимость реализации всех методов интерфейса, которые тестируемая система может вызвать. В таких статически типизированных языках, как Java и C#, необходимо реализовать все методы интерфейса, подразумеваемого классом или типом, который связан со способом доступа к вызываемому объекту. Часто это “вынуждает” наследовать настоящий вызываемый компонент, чтобы избежать создания пустых реализаций этих методов.

Одним из способом сокращения затрат на разработку является предоставление принятого по умолчанию класса, в котором реализованы все методы интерфейса и генерируется уникальная ошибка. После этого *фиксированный тестовый двойник* (Hard-Coded Test Double) можно реализовать через наследование данного конкретного класса с последующим переопределением того метода, который должен вызываться тестируемой системой. Если тестируемая система вызовет другие методы, *псевдообъект* (Pseudo-Object) вернет ошибку и приведет к неудачному завершению теста.

Мотивирующий пример

В следующем teste проверяется базовая функциональность компонента, форматирующего строку HTML с текущим временем. К сожалению, компонент зависит от настоящих системных часов и тест редко завершается успешно!

```
public void testDisplayCurrentTime_AtMidnight() {
    // Настройка тестовой конфигурации
    TimeDisplay sut = new TimeDisplay();
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка непосредственного вывода
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals( expectedTimeString, result );
}
```

Замечания по рефакторингу

Наиболее распространенным решением для отказа от настоящего компонента является использование *фиксированного тестового двойника* (Hard-Coded Test Double)². Для осуществления этого перехода необходимо самостоятельно создать *тестовый двойник* (Test Double) и установить его из *тестового метода* (Test Method). Кроме того, если тестируемая система не предоставляет механизм установки, возможно, придется добавить

² Движение в направлении от *настраиваемого тестового двойника* (Configurable Test Double) в сторону *фиксированного тестового двойника* (Hard-Coded Test Double) происходит достаточно редко, так как после каждой модификации *тестовый двойник* (Test Double) должен становиться более (а не менее) пригодным для повторного использования.

метод установки *тестового двойника* (Test Double) с помощью одного из шаблонов *вставки зависимости* (Dependency Injection, с. 684). Соответствующий процесс рассматривается в описании рефакторинга заменить зависимость тестовым двойником (Replace Dependency with Test Double, с. 740).

Пример: класс тестового двойника (Test Double Class)

Ниже показан тот же тест, модифицированный для использования класса *фиксированного тестового двойника* (Hard-Coded Test Double), поддерживающего управление временем.

```
public void testDisplayCurrentTime_AtMidnight_HCM()
    throws Exception {
    // Настройка тестовой конфигурации
    // Создание экземпляра фиксированной тестовой заглушки
    TimeProvider testStub = new MidnightTimeProvider();
    // Создание экземпляра тестируемой системы
    TimeDisplay sut = new TimeDisplay();
    // Вставка заглушки в тестовую систему
    sut.setTimeProvider(testStub);
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка непосредственного вывода
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}
```

Этот тест сложно понять, не увидев реализацию *фиксированного тестового двойника* (Hard-Coded Test Double). Если реализация недоступна, такой подход очень быстро приведет к появлению *непонятных тестов* (Obscure Test).

```
class MidnightTimeProvider implements TimeProvider {
    public Calendar getTime() {
        Calendar myTime = new GregorianCalendar();
        myTime.set(Calendar.HOUR_OF_DAY, 0);
        myTime.set(Calendar.MINUTE, 0);
        return myTime;
    }
}
```

В зависимости от используемого языка программирования *класс тестового двойника* (Test Double Class) можно определить в нескольких местах, например в теле *класса теста* (Testcase Class) (как внутренний класс), в виде отдельного класса в том же файле или отдельного класса — в другом файле. Конечно, чем дальше от *тестового метода* (Test Method) хранится *класс тестового двойника* (Test Double Class), тем больше он превращается в *таинственного гостя* (Mystery Guest).

Пример: тестовый шунт (Self Shunt) (петля)

Ниже показан тест, в котором для управления временем используется *тестовый шунт* (Self Shunt).

```

public class SelfShuntExample extends TestCase
implements TimeProvider {
    public void testDisplayCurrentTime_AtMidnight() throws Exception {
        // Настройка тестовой конфигурации
        TimeDisplay sut = new TimeDisplay();
        // Настройка подставного объекта
        sut.setTimeProvider(this); // Установка тестового шунта
        // Вызов тестируемой системы
        String result = sut.getCurrentTimeAsHtmlFragment();
        // Проверка непосредственного вывода
        String expectedTimeString =
            "<span class=\"tinyBoldText\">Midnight</span>";
        assertEquals("Midnight", expectedTimeString, result);
    }
    public Calendar getTime() {
        Calendar myTime = new GregorianCalendar();
        myTime.set(Calendar.MINUTE, 0);
        myTime.set(Calendar.HOUR_OF_DAY, 0);
        return myTime;
    }
}

```

Обратите внимание, что *тестовый метод* (Test Method), который устанавливает *фиксированный тестовый двойник* (Hard-Coded Test Double), и реализация метода `getTime`, вызываемого тестируемой системой, принадлежат одному классу. Для установки *фиксированного тестового двойника* (Hard-Coded Test Double) использован шаблон *вставки метода установки* (Setter Injection). Поскольку этот пример написан на статически типизированном языке, пришлось добавить строку `implements TimeProvider` в объявление *класса теста* (Testcase Class). В противном случае оператор `sut.setTimeProvider(this)` не был бы принят компилятором. В динамически типизированных языках этот шаг не требуется.

Пример: внутренний тестовый двойник (Inner Test Double) в виде подкласса

Ниже показан тест на основе инфраструктуры JUnit, в котором *внутренний тестовый двойник* (Inner Test Double) в виде подкласса реализован с помощью синтаксиса **анонимного внутреннего класса** (anonymous inner class).

```

public void testDisplayCurrentTime_AtMidnight_AIM() throws Exception {
    // Настройка тестовой конфигурации
    // Определение и создание экземпляра тестовой заглушки
    TimeProvider testStub = new TimeProvider() {
        // Анонимная внутренняя заглушка
        public Calendar getTime() {
            Calendar myTime = new GregorianCalendar();
            myTime.set(Calendar.MINUTE, 0);
            myTime.set(Calendar.HOUR_OF_DAY, 0);
            return myTime;
        }
    };
    // Создание экземпляра тестируемой системы
    TimeDisplay sut = new TimeDisplay();
    // Вставка заглушки в тестируемую систему
    sut.setTimeProvider(testStub);
}

```

```
// Вызов тестируемой системы
String result = sut.getCurrentTimeAsHtmlFragment();
// Проверка непосредственного вывода
String expectedTimeString =
    "<span class=\"tinyBoldText\">Midnight</span>";
assertEquals("Midnight", expectedTimeString, result);
}
```

В данном случае в вызове new для определения *фиксированного тестового двойника* (Hard-Coded Test Double) используется имя настоящего вызываемого класса (TimeProvider). Заключение определения метода gettime в фигурные скобки после имени класса приводит к созданию анонимного внутреннего тестового двойника внутри *тестового метода* (Test Method).

Пример: внутренний тестовый двойник (Inner Test Double), наследующий псевдокласс

Предположим, что одна реализация метода заменена другой и первую реализацию необходимо оставить для обратной совместимости. Но создаваемые тесты не должны ее вызывать. В таком случае можно воспользоваться следующим определением *псевдообъекта* (Pseudo-Object).

```
/**
 * Базовый класс для фиксированных тестовых заглушек
 *
 * и подставных объектов
 */
public class PseudoTimeProvider implements ComplexTimeProvider {
    public Calendar getTime() throws TimeProviderEx {
        throw new PseudoClassException();
    }
    public Calendar getTimeDifference(Calendar baseTime,
                                     Calendar otherTime)
        throws TimeProviderEx {
        throw new PseudoClassException();
    }
    public Calendar getTime(String timeZone) throws TimeProviderEx {
        throw new PseudoClassException();
    }
}
```

После этого можно написать тест, который подтверждает, что прежняя версия метода getTime не вызывается через наследование и переопределение новой версии метода (которая должна вызываться тестируемой системой).

```
public void testDisplayCurrentTime_AtMidnight_PS() throws Exception {
    // Настройка тестовой конфигурации
    // Определение и создание экземпляра тестовой загрузки
    TimeProvider testStub = new PseudoTimeProvider()
    { // Анонимная внутренняя заглушка
        public Calendar getTime(String timeZone) {
            Calendar myTime = new GregorianCalendar();
            myTime.set(Calendar.MINUTE, 0);
```

```

        myTime.set(Calendar.HOUR_OF_DAY, 0);
        return myTime;
    }
};

// Создание экземпляра тестируемой системы
TimeDisplay sut = new TimeDisplay();
// Вставка заглушки в тестируемую систему
sut.setTimeProvider(testStub);
// Вызов тестируемой системы
String result = sut.getCurrentTimeAsHtmlFragment();
// Проверка непосредственного вывода
String expectedTimeString =
    "<span class=\"tinyBoldText\">Midnight</span>";
assertEquals("Midnight", expectedTimeString, result);
}

```

При вызове любого другого метода вызываются методы базового класса и генерируется исключение. Таким образом, если при запуске теста вызывается один из непреопределенных методов, вывод будет выглядеть следующим образом.

```

com..PseudoClassEx: Unexpected call to unsupported method.
at com..PseudoTimeProvider.getTime(PseudoTimeProvider.java:22)
at com..TimeDisplay.getCurrentTimeAsHtmlFragment(TimeDisplay.java:64)
at com..TimeDisplayTestSolution.
    testDisplayCurrentTime_AtMidnight_PS(
        TimeDisplayTestSolution.java:247)

```

ЧТО В ИМЕНИ ШАБЛОНА?

ВАЖНОСТЬ ОСМЫСЛЕННЫХ ИМЕН

Имена важны, так как они являются ключевой частью общения. Имена — это ярлыки, которые крепятся к концепциям. Хорошие имена способствуют донесению концепций. Это справедливо, если общение происходит со знающими имена людьми, но это особенно важно при общении с не знающими. Рассмотрим следующий пример.

Давным-давно, еще в начале работы с шаблонами, автор участвовал в конференции Pattern Languages of Programs (PLoP) (<http://www.hillside.net/conferences/plop>). На ней широкоизвестный автор Джим Копlien разрабатывал язык для описания организационных шаблонов. Один из шаблонов назывался “Гора Буффало” (Buffalo Mountain). Еще один назывался “Архитектор тоже реализует” (Architect Also Implements). Эти два имени шаблона находились на противоположных концах спектра возможных имен.

Смысль шаблона “Архитектор тоже реализует” можно определить из самого названия, даже если не читать подробное описание. Это имя как подразумевает шаблон, так и несет собственный смысл.

С другой стороны, “Гора Буффало” никак не доносит скрытый за названием смысл. До сих пор вспоминается история этого имени, но никак не удается вспомнить смысл самого шаблона. Имя основано на графике с визуализацией относящихся к шаблону данных. Один из рецензентов обратил внимание, что график напоминает профиль горы Буффало. В результате шаблон получил запоминающееся, но ничего не означающее имя.

Рассмотрим название шаблона *тестовый шунт* (Self Shunt). Это пример не очень понятного имени, так как термин “шунт” используется только в некоторых специализированных областях. Майкл Фезерс привел историю этого имени в своем описании шаблона.

Но если это описание не прочитать, имя останется всего лишь именем. Более описательное имя выглядело бы как “Класс теста как тестовый двойник” или “Петля”, но и последний вариант страдает от некоторой неоднозначности, так как не ясно, что на что замыкается. Название *тестовый шунт* (Self Shunt) прижилось только из-за своей распространенности.

ДРУГИЕ ЗАМЕЧАНИЯ ПО ПОВОДУ ИМЕНОВАНИЯ

Кто-то может спросить, почему для некоторых шаблонов в книге предлагаются альтернативные имена. Одна из причин была рассмотрена выше. Еще одной причиной является необходимость стройной “системы имен” в больших коллекциях шаблонов (например, в этой книге).

Проиллюстрируем вторую причину на примере. Часто рекомендуется использовать метод `setUp` для создания тестовой конфигурации. Такой подход выносит логику настройки конфигурации из отдельных *тестовых методов* (Test Method, с. 378) в единственное место, из которого логика может использоваться повторно. Обычно такой шаблон называется *общим методом настройки* (Shared Setup Method), но в этом **языке шаблонов** (pattern language) он называется *неявной настройкой* (Implicit Setup, с. 449). Почему?

Это связано с именами других шаблонов в пределах языка. С одной стороны, *общий метод настройки* (Shared Setup Method) можно легко спутать с существующим шаблоном *общая тестовая конфигурация* (Shared Fixture, с. 350). (Первый шаблон описывает совместное использование кода, а второй — совместное использование объектов конфигурации во время выполнения.) С другой стороны, две основные альтернативы *неявной настройки* (Implicit Setup) называются *встроенная настройка* (In-line Setup, с. 432) и *делегированная настройка* (Delegated Setup, с. 437). Согласитесь, что набор “*встроенная настройка* (In-line Setup), *делегированная настройка* (Delegated Setup) и *неявная настройка* (Implicit Setup)” выглядит лучше, чем “*встроенная настройка* (In-line Setup), *делегированная настройка* (Delegated Setup) и *общий метод настройки* (Shared Setup Method)”. Связь между именами шаблонов еще более очевидна, если учесть все основные альтернативные шаблоны при создании системы имен.

ЗАЧЕМ СТАНДАРТИЗИРОВАТЬ ШАБЛОНЫ ТЕСТИРОВАНИЯ

В последнем акте этой мыльной оперы рассматривается важность стандартизации имен шаблонов автоматизации тестирования, особенно *тестовых заглушек* (Test Stub, с. 544) и *подставных объектов* (Mock Object, с. 558). Ключевой проблемой является краткость донесения мысли.

Если кто-то предлагает разработчику “Вставь сюда подставку” (“Put a mock in it”) (с иронией!), что это за совет? В зависимости от значения “подставки” (“mock”) может предлагаться управление опосредованным вводом тестируемой системы с помощью *тестовой заглушки* (Test Stub) или замена базы данных *поддельной базой данных* (Fake Database) для ускорения работы тестов в 50 раз. (Да, в 50! Дополнительная информация о этом приводится во врезке “Быстрые тесты без общей тестовой конфигурации” на с. 351.) А возможно, разработчику предлагают проверить правильность вызова методов со стороны тестируемой системы через установку “энергичного” *подставного объекта* (Mock Object), настроенного на *ожидаемое поведение* (Expected Behavior). Если каждый использует термин “подставка” (“mock”) для обозначения *подставного объекта* (Mock Object) — не больше и не меньше, — то совет будет иметь вполне определенное значение. Но сейчас совет очень расплывчат, так как любой *тестовый двойник* (Test Double, с. 538) может называться подставным объектом (несмотря на возражения авторов исходной статьи по *подставным объектам*, Mock Object).

Источники дополнительной информации

Если хотите узнать, что означает “Гора Буффало”, ознакомьтесь с ее описанием на странице:

<http://www1.bell-labs.com/user/cope/Patterns/Process/section29.html>

Описание шаблона “Архитектор тоже реализует” доступно по адресу:

<http://www1.bell-labs.com/user/cope/Patterns/Process/section16.html>

Что интересно, Алистер Коуберн написал подобное сравнение имен шаблонов в статье на своем сайте (<http://alistair.cockburn.us>). Для сравнения он выбрал те же самые имена шаблонов. Совпадение или шаблон?

Эта схема позволяет не только неудачно завершить тест, но и значительно упрощает определение вызванного метода. Дополнительным преимуществом является допустимость вызова неожиданных методов без дополнительных трудозатрат.

Источники дополнительной информации

Примеры *тестового шунта* (Self Shunt) приводятся во многих “справочниках” по разработке на основе тестов ([TDD-APG], [TDD-BE], [UTwJ], [PUT] и [JuPG]). Исходное описание сделано Майклом Фезерсом и доступно по адресу:

<http://www.objectmentor.com/resources/articles/SelfShunPtrn.pdf>

Описание исходного шаблона “Шунт” (“Shunt”) доступно по адресу:

<http://http://c2.com/cgi/wiki?ShuntPattern>

Там же приводится список альтернативных имен, включая “Lookback”. Дополнительная информация о выборе осмысленных имен шаблонов приведена выше, во врезке “Что в имени шаблона?”.

Шаблон *псевдообъект* (Pseudo-Object) рассматривается в статье “Pseudo-Classes: Very Simple and Lightweight Mock Object-like Classes for Unit-Testing”, доступной по адресу:

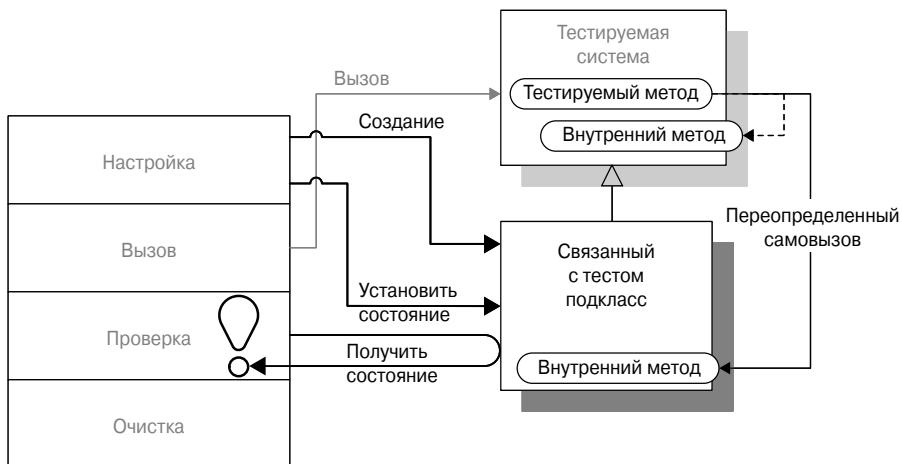
<http://www.devx.com/Java/Article/22599/1954?pf=true>

Связанный с тестом подкласс (Test-Specific Subclass)

Как обеспечить тестирование кода, если необходимо получить доступ к закрытому состоянию тестируемой системы?

В подкласс тестируемой системы добавляются методы, предоставляющие доступ к необходимому для теста состоянию или поведению.

Также известен как:
Связанное с тестом расширение (Test-Specific Extension)



Если изначально тестируемая система не проектировалась с учетом тестирования, может оказаться, что тест не может получить доступ к состоянию, которое необходимо инициализировать или проверить во время соответствующей фазы теста.

Связанный с тестом подкласс (Test-Specific Subclass) является простым, но очень мощным способом открыть тестируемую систему без модификации ее кода.

Как это работает

Для решения этой задачи определяется подкласс тестируемой системы и добавляются методы, модифицирующие поведение системы в достаточной мере для тестирования через дополнительные точки управления и наблюдения. Обычно модификация поведения предполагает добавление методов для просмотра и установки переменных экземпляра, а также для установки системы в конкретное состояние без перемещения по всему жизненному циклу объекта.

Поскольку *связанный с тестом подкласс* (Test-Specific Subclass) хранится там же, где и использующие его тесты, представление других компонентов приложения о тестируемой системе не меняется.

Когда это использовать

Связанный с тестом подкласс (Test-Specific Subclass) имеет смысл использовать каждый раз, когда необходимо модифицировать систему для упрощения тестирования, но внесение этих изменений в код приведет к появлению *логики теста в продукте* (Test Logic in Production, с. 257). Хотя *связанный с тестом подкласс* (Test-Specific Subclass) имеет несколько применений, во всех них подкласс облегчает тестирование, предоставляя доступ к содержимому тестируемой системы. Но *связанный с тестом подкласс* (Test-Specific Subclass) является обоюдоострым мечом: нарушение инкапсуляции еще теснее связывает тесты с реализацией, что в итоге может привести к появлению “хрупких” тестов (Fragile Test, с. 277).

Вариант: открывающий состояние подкласс (State-Exposing Subclass)

При *проверке состояния* (State Verification, с. 484) можно создать подкласс тестируемой системы (или одного из ее компонентов) и получить доступ к внутреннему состоянию, проверяемому с помощью *методов с утверждением* (Assertion Method, с. 390). Обычно при этом добавляются методы доступа к переменным экземпляра. Кроме того, тесту можно позволить устанавливать состояние, чтобы избежать появления *непонятных тестов* (Obscure Test), вызванных усложненной логикой настройки.

Вариант: раскрывающий поведение подкласс (Behavior-Exposing Subclass)

Для того чтобы изолированно тестировать отдельные шаги сложного алгоритма, можно создать подкласс тестируемой системы для доступа к закрытым методом, реализующим самовызовы. Поскольку в большинстве языков расширение видимости методов не допускается, в *связанном с тестом подклассе* (Test-Specific Subclass) приходится использовать другие имена и вызывать методы суперкласса.

Вариант: модифицирующий поведение подкласс (Behavior-Modifying Subclass)

Если тестируемая система имеет нежелательное во время выполнения теста поведение, его можно переопределить методом с пустым телом. Такой способ лучше всего работает в случае тестируемых систем, использующих самовызовы (или шаблоны методов) для делегирования этапов алгоритма собственным методам или методам подкласса.

Вариант: подкласс как тестовый двойник (Test Double Subclass)

Также известен как:	Для обеспечения совместимости по типу между <i>тестовым двойником</i> (Test Double, с. 538) и заменяемым вызываемым компонентом <i>тестовый двойник</i> (Test Double) можно сделать подклассом заменяемого компонента. Это единственный возможный способ создания <i>тестового двойника</i> (Test Double), который будет приниматься компилятором при статической типизации переменных конкретными классами, т.е. когда в качестве типа переменной указывается конкретный класс, а не абстрактный класс или интерфейс. (В динамически типизированных языках, например в Ruby, Python, Perl и JavaScript, эта конструкция не требуется.) После этого нужно переопределить методы, поведение которых необходимо изменить, а также добавить методы
<i>Тестовый двойник в виде подкласса</i> (Subclassed Test Double)	

для преобразования *тестового двойника* (Test Double) в *настраиваемый тестовый двойник* (Configurable Test Double, с. 571), если это необходимо.

В отличие от *модифицирующего поведение подкласса* (Behavior-Modifying Subclass) *подкласс как тестовый двойник* (Test Double Subclass) не просто “подстраивает” поведение тестируемой системы (или ее части), а полностью заменяет его.

Вариант: замененный единственный экземпляр класса (Substituted Singleton)

Это специальный случай *подкласса как тестового двойника* (Test Double Subclass). Он используется при замене вызываемого компонента *тестовым двойником* (Test Double), когда тестируемая система не поддерживает механизмы *вставки зависимости* (Dependency Injection, с. 684) и *поиска зависимости* (Dependency Lookup, с. 692).

Также известен как:
Единственный экземпляр класса в виде подкласса (Subclassed Singleton), *Заменяемый единственный экземпляр класса (Substitutable Singleton)*

Замечания по реализации

При использовании *связанного с тестом подкласса* (Test-Specific Subclass) возникает ряд проблем.

- **Гранулярность функций.** Любое поведение, которое необходимо переопределить или сделать видимым, должно находиться в собственном методе. Достигается за счет использования небольших методов и самовызовов.
- **Видимость функций.** Подклассы должны получать доступ к атрибутам и поведению класса тестируемой системы. В основном, эта проблема характерна для таких статически типизированных языков, как Java, C# и C++. Динамически типизированные языки обычно не настаивают на видимости.

Как и в случае с *тестовыми двойниками* (Test Double), нельзя заменять поведение, которое проверяется данным тестом.

В языках с поддержкой расширения классов без подклассов (например, в Smalltalk, Ruby, JavaScript и в других динамически типизированных языках) *связанный с тестом подкласс* (Test-Specific Subclass) может быть реализован в виде расширения класса в пакете теста. Но не стоит забывать, что если расширения попадут в промышленную эксплуатацию, это можно будет расценивать как *логику теста в продукте* (Test Logic in Production).

Видимость функций

В языках с принудительной областью видимости переменных и методов, возможно, придется изменить видимость переменных для обеспечения доступа со стороны подкласса. Хотя такое изменение влияет на код тестируемой системы, оно считается значительно менее опасным, чем простое изменение видимости на `public` (что предоставляет доступ любому коду приложения) или добавление методов тестов непосредственно в тестируемую систему.

Например, в языке Java видимость переменных экземпляра можно изменить с `private` на `protected`, что обеспечит доступ *связанному с тестом подклассу* (Test-Specific Subclass). Точно так можно изменить и видимость методов.

Гранулярность функций

Длинные методы сложно тестировать, так как часто они имеют множество зависимостей. С другой стороны, короткие методы тестировать значительно проще, так как они решают единственную задачу. Самовызовы позволяют сократить размер методов. Части алгоритма делегируются другим методам, реализованным в том же классе. Такая стратегия позволяет тестировать методы независимо. Кроме того, можно подтвердить правильность последовательности вызова, переопределив их в *подклассе как тестовом двойнике* (Test Double Subclass).

Самовызов является частью хорошего объектно-ориентированного дизайна, так как он позволяет сохранить небольшой размер и ограниченную ответственность методов. Добавление самовызыва может потребоваться в ситуациях, когда некоторая часть алгоритма в длинном методе зависит от компонентов, вызывать которые нежелательно (например, от базы данных). Данная вероятность исключительно высока, если тестируемая система создается на основе архитектуры **сценариев транзакций** (transaction script) [PEAA]. Самовызовы можно просто интегрировать с помощью рефакторинга *выделение метода* (Extract Method), который поддерживается большинством современных средств разработки.

Мотивирующий пример

Тест в следующем примере является неопределенным, так как зависит от времени. Тестируемая система представляет собой объект, форматирующий строку времени для вывода на Web-странице. Объект получает время от класса с единственным экземпляром TimeProvider, который извлекает значение времени из объекта календаря, полученного из контейнера.

```
public void testDisplayCurrentTime_AtMidnight() throws Exception {
    // Настройка тестируемой системы
    TimeDisplay theTimeDisplay = new TimeDisplay();
    // Вызов тестируемой системы
    String actualTimeString =
        theTimeDisplay.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight",
        expectedTimeString,
        actualTimeString);
}

public void testDisplayCurrentTime_AtOneMinuteAfterMidnight()
    throws Exception {
    // Настройка тестируемой системы
    TimeDisplay actualTimeDisplay = new TimeDisplay();
    // Вызов тестируемой системы
    String actualTimeString =
        actualTimeDisplay.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    String expectedTimeString =
        "<span class=\"tinyBoldText\">12:01 AM</span>";
```

```

        assertEquals("12:01 AM",
                     expectedTimeString,
                     actualTimeString);
    }
}

```

Эти тесты редко завершаются успешно и никогда не завершаются успешно одновременно! Код внутри тестируемой системы выглядит следующим образом.

```

public String getCurrentTimeAsHtmlFragment() {
    Calendar timeProvider;
    try {
        timeProvider = getTime();
    } catch (Exception e) {
        return e.getMessage();
    }
    // И т.д.
}
protected Calendar getTime() {
    return TimeProvider.getInstance().getTime();
}

```

Ниже показан код класса с единственным экземпляром.

```

public class TimeProvider {
    protected static TimeProvider soleInstance = null;
    protected TimeProvider() {};
    public static TimeProvider getInstance() {
        if (soleInstance==null) soleInstance = new TimeProvider();
        return soleInstance;
    }
    public Calendar getTime() {
        return Calendar.getInstance();
    }
}

```

Замечания по рефакторингу

Природа используемого рефакторинга для вставки *связанного с тестом подкласса* (Test-Specific Subclass) зависит от причин его установки. При использовании *связанного с тестом подкласса* (Test-Specific Subclass) с целью открытия “закрытых фрагментов” тестируемой системы или переопределения нежелательных компонентов поведения он просто определяется как подкласс тестируемой системы и его экземпляр создается на этапе настройки тестовой конфигурации.

Но, применяя *связанный с тестом подкласс* (Test-Specific Subclass) для замены вызываемого компонента, необходимо использовать рефакторинг *заменить зависимость тестовым двойником* (Replace Dependency with Test Double, с. 740), чтобы тестируемая система использовала подкласс вместо вызываемого компонента.

В любом случае в зависимости от потребностей тестов в *связанном с тестом подклассе* (Test-Specific Subclass) с помощью доступных языковых конструкций (подклассов или миксинов) переопределяются или добавляются новые методы.

Пример: модифицирующий поведение подкласс (Behavior-Modifying Subclass) на основе тестовой заглушки (Test Stub)

Поскольку тестируемая система использует самовызов метода `getTime` для получения значения времени от объекта `TimeProvider`, для управления временем можно воспользоваться *тестовым двойником в виде подкласса* (Subclassed Test Double)³. На основе этой идеи можно создавать новые тесты (здесь показан только один).

```
public void testDisplayCurrentTime_AtMidnight() {
    // Настройка тестовой конфигурации
    TimeDisplayTestStubSubclass tss = new TimeDisplayTestStubSubclass();
    TimeDisplay sut = tss;
    // Настройка тестового двойника (Test Double)
    tss.setHours(0);
    tss.setMinutes(0);
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals(expectedTimeString, result);
}
```

Обратите внимание на использование *связанного с тестом подкласса* (Test-Specific Subclass) для объявления переменной, хранящей указатель на тестируемую систему. Такой подход позволяет обеспечить видимость *конфигурационного интерфейса* (Configuration Interface) в пределах теста⁴. Для документирования процесса экземпляр *связанного с тестом подкласса* (Test-Specific Subclass) хранится в переменной `sut`. Это безопасное преобразование, так как он является подклассом класса тестируемой системы. Подобный способ позволяет избежать появления *тайного гостя* (Mystery Guest), который вызван фиксированием важного опосредованного ввода тестируемой системы внутри *тестовой заглушки* (Test Stub, с. 544).

С помощью приведенного ниже примера использования можно реализовать *связанный с тестом подкласс* (Test-Specific Subclass).

```
public class TimeDisplayTestStubSubclass extends TimeDisplay {
    private int hours;
    private int minutes;
    // Переопределенный метод
    protected Calendar getTime() {
        Calendar myTime = new GregorianCalendar();
        myTime.set(Calendar.HOUR_OF_DAY, this.hours);
        myTime.set(Calendar.MINUTE, this.minutes);
        return myTime;
    }
    /*
```

³ Это стало возможным потому, что метод `getTime` был определен, как `protected`. Для закрытого (`private`) метода такой фокус оказался бы невозможным.

⁴ Можно было бы воспользоваться *фиксированным тестовым двойником* (Hard-Coded Test Double, с. 581), но тогда пришлось бы использовать свой *связанный с тестом подкласс* (Test-Specific Subclass) для каждого теста. Каждый подкласс просто содержал бы фиксированные возвращаемые значения метода `getTime`.

```

    * Конфигурационный интерфейс
 */
public void setHours(int hours) {
    this.hours = hours;
}
public void setMinutes(int minutes) {
    this.minutes = minutes;
}
}

```

Это не интегральное исчисление — достаточно реализовать методы, используемые тестом.

Пример: модифицирующий поведение подкласс (Behavior-Modifying Subclass) на основе подменяемого единственного экземпляра класса

Предположим, что метод `getTime` был объявлен как `private5`, `static`, `final` или `sealed6` и т.д. Такое объявление не позволит переопределить поведение метода в *связанном с тестом подклассе* (Test-Specific Subclass). Как исправить *неопределенный тест* (Nondeterministic Test)?

В данном случае для предоставления времени используется единственный экземпляр класса, поэтому самым простым решением является его замена *подклассом как тестовым двойником* (Test Double Subclass). Это возможно, пока подкласс может получить доступ к переменной `soleInstance`. Рефакторинг *введение локального расширения* (Introduce local extension) (особенно его вариант на основе подкласса) позволяет создать *связанный с тестом подкласс* (Test-Specific Subclass). Создание теста до кода помогает понять структуру реализуемого интерфейса.

```

public void testDisplayCurrentTime_AtMidnight() {
    TimeDisplay sut = new TimeDisplay();
    // Установка единственного экземпляра класса
    TimeProviderTestSingleton timeProvideSingleton =
        TimeProviderTestSingleton.overrideSoleInstance();
    timeProvideSingleton.setTime(0,0);
    // Вызов тестируемой системы
    String actualTimeString = sut.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals(expectedTimeString, actualTimeString);
}

```

Получив тест, использующий заменяемый единственный экземпляр класса, можно переходить к реализации, создав подкласс и определив используемые тестом методы.

```

public class TimeProviderTestSingleton extends TimeProvider {
    private Calendar myTime = new GregorianCalendar();
    private TimeProviderTestSingleton() {};
    // Интерфейс установки
    static TimeProviderTestSingleton overrideSoleInstance() {

```

⁵ Объявленный как `private` метод не виден для подклассов и не может быть переопределен.

⁶ Этот вариант запрещает подклассу переопределять поведение метода.

```

// Можно сохранить настоящий экземпляр,
// но здесь это не делается!
soleInstance = new TimeProviderTestSingleton();
return (TimeProviderTestSingleton) soleInstance;
}
// Используемый тестом конфигурационный интерфейс
public void setTime(int hours, int minutes) {
    myTime.set(Calendar.HOUR_OF_DAY, hours);
    myTime.set(Calendar.MINUTE, minutes);
}
// Нормальный интерфейс, используемый клиентом
public Calendar getTime() {
    return myTime;
}
}

```

В данном случае *тестовый двойник* (Test Double) представляет собой подкласс настоящего компонента и переопределяет метод экземпляра, вызываемый клиентом.

Пример: раскрывающий поведение подкласс (Behavior-Exposing Subclass)

Предположим, необходимо непосредственно проверить метод `getTime`. Так как `getTime` является защищенным методом и находится не в том пакете, в котором хранится класс `TimeDisplay`, тест не сможет вызвать этот метод. Можно попытаться сделать тест подклассом класса `TimeDisplay` или переместить тест в тот же пакет, что и `TimeDisplay`. К сожалению, оба варианта имеют отрицательные стороны и не всегда возможны.

Более общим решением является доступ к поведению через *раскрывающий поведение подкласс* (Behavior-Exposing Subclass). Для этого можно определить *связанный с тестом подкласс* (Test-Specific Subclass) и добавить метод с квалификатором `public`, вызывающий проверяемый метод.

```

public class TimeDisplayBehaviorExposingTss extends TimeDisplay {
    public Calendar callgetTime() {
        return super.getTime();
    }
}

```

После этого можно написать тест, использующий *раскрывающий поведение подкласс* (Behavior-Exposing Subclass).

```

public void testgetTime_default() {
    // Создание тестируемой системы
    TimeDisplayBehaviorExposingTss tsSut =
        new TimeDisplayBehaviorExposingTss();
    // Вызов тестируемой системы хотелось бы
    // сделать таким:
    //     Calendar time = sut.getTime();
    // Но приходится делать
    Calendar time = tsSut.callgetTime();
    // Проверка результата
    assertEquals( defaultTime, time );
}

```

Пример: определение характерного для теста равенства на основе модифицирующего поведение подкласса (Behavior-Modifying Subclass)

Ниже показан пример простого теста, который завершается неудачно, так как передаваемый в вызов `assertEquals` объект не содержит реализации предназначенного для теста равенства. Иначе говоря, принятый по умолчанию метод `equals` возвращает значение `false`, хотя тест считает оба объекта равными.

```
protected void setUp() throws Exception {
    oneOutboundFlight = findOneOutboundFlightDto();
}
public void testGetFlights_OneFlight() throws Exception {
    // Вызов тестируемой системы
    List flights = facade.getFlightsByOriginAirport(
        oneOutboundFlight.getOriginAirportId());
    // Проверка результата
    assertEquals("Flights at origin - number of flights: ",
        1,
        flights.size());
    FlightDto actualFlightDto = (FlightDto)flights.get(0);
    assertEquals("Flight DTOs at origin",
        oneOutboundFlight,
        actualFlightDto);
}
```

Одним из решений является создание *специального утверждения* (Custom Assertion, с. 495). Еще одно решение предполагает использование *связанного с тестом подкласса* (Test-Specific Subclass) для добавления подходящего определения равенства, предназначенного специально для теста. Для создания *связанного с тестом подкласса* (Test-Specific Subclass) в виде *ожидаемого объекта* (Expected Object) можно немного модифицировать код настройки тестовой конфигурации.

```
private FlightDtoTss oneOutboundFlight;
private FlightDtoTss findOneOutboundFlightDto() {
    FlightDto realDto = helper.findOneOutboundFlightDto();
    return new FlightDtoTss(realDto) ;
}
```

Наконец, *связанный с тестом подкласс* (Test-Specific Subclass) можно реализовать через копирование и сравнение только тех полей, которые относятся к рассматриваемому в teste равенству.

```
public class FlightDtoTss extends FlightDto {
    public FlightDtoTss(FlightDto realDto) {
        this.destAirportId = realDto.getDestinationAirportId();
        this.equipmentType = realDto.getEquipmentType();
        this.flightNumber = realDto.getFlightNumber();
        this.originAirportId = realDto.getOriginAirportId();
    }
    public boolean equals(Object obj) {
        FlightDto otherDto = (FlightDto) obj;
        if (otherDto == null) return false;
        if (otherDto.getDestAirportId() != this.destAirportId)
```

```

        return false;
    if (otherDto.getOriginAirportId() != this.originAirportId)
        return false;
    if (otherDto.getFlightNumber() != this.flightNumber)
        return false;
    if (otherDto.getEquipmentType() != this.equipmentType)
        return false;
    return true;
}
}

```

В этом случае поля настоящего объекта были скопированы в экземпляр *связанного с тестом подкласса* (Test-Specific Subclass), но с тем же успехом можно было просто “завернуть” в подкласс настоящий объект. Существуют и другие способы создания подкласса. Их количество ограничено только воображением разработчика.

Кроме того, в данном примере предполагается доступность подходящей реализации метода `toString` в базовом классе. Этот метод выводит значения сравниваемых полей в ответ на вызов из метода `assertEquals`, когда метод `equals` возвращает значение `false`. В противном случае определить, почему объекты считаются неравными, невозможно.

Пример: открывающий состояние подкласс (State-Exposing Subclass)

Предположим, что существует следующий тест, которому необходим объект `Flight` в конкретном состоянии.

```

protected void setUp() throws Exception {
    super.setUp();
    scheduledFlight = createScheduledFlight();
}
Flight createScheduledFlight() throws InvalidRequestException{
    Flight newFlight = new Flight();
    newFlight.schedule();
    return newFlight;
}
public void testDeschedule_shouldEndUpInUnscheduleState()
            throws Exception {
    scheduledFlight.deschedule();
    assertTrue("isUnsched", scheduledFlight.isUnscheduled());
}

```

Чтобы создать тестовую конфигурацию для данного теста, потребуется вызвать метод `schedule` объекта `Flight`.

```

public class Flight{
    protected FlightState currentState = new UnscheduledState();
    /**
     * Переводит объект Flight из состояния <code>unscheduled</code>
     * в состояние <code>scheduled</code>.
     * @throws InvalidRequestException при запросе некорректного
     *         перехода
     */
    public void schedule() throws InvalidRequestException{
        currentState.schedule();
    }
}

```

В классе `Flight` используется шаблон `State`, а обработка метода `schedule` делегируется любому объекту `State`, на который указывает переменная `currentState`. Если метод `schedule` не работает с принятым по умолчанию содержимым переменной `currentState`, тест завершается неудачно на этапе создания конфигурации. Этой проблемы можно избежать, воспользовавшись *открывающим состоянием подклассом* (`State-Exposing Subclass`), в котором доступен метод для непосредственного перехода в нужное состояние, а значит, получения независимого теста.

```
public class FlightTss extends Flight {  
    public void becomeScheduled() {  
        currentState = new ScheduledState();  
    }  
}
```

Добавив новый метод `becomeScheduled` в *связанный с тестом подкласс* (`Test-Specific Subclass`), можно защититься от случайного переопределения поведения тестируемой системы. После этого достаточно создать экземпляр подкласса, модифицировав *метод создания* (`Creation Method`, с. 441).

```
Flight createScheduledFlight() throws InvalidRequestException{  
    FlightTss newFlight = new FlightTss();  
    newFlight.becomeScheduled();  
    return newFlight;  
}
```

Обратите внимание, что объявлен возврат объекта класса `Flight`, хотя на самом деле возвращается экземпляр *связанного с тестом подкласса* (`Test-Specific Subclass`) с дополнительным методом.

Глава 24

Шаблоны организации тестов

Шаблоны в этой главе:

Именованный набор тестов (Named Test Suite) 604

Повторное использование кода тестов

Вспомогательный метод теста (Test Utility Method) 610

Параметризованный тест (Parameterized Test) 618

Структура класса теста

Класс теста для каждого класса (Testcase Class per Class) 627

Класс теста для каждой функции (Testcase Class per Feature) 633

Класс теста для каждой тестовой конфигурации

(Testcase Class per Fixture) 639

Расположение вспомогательных методов

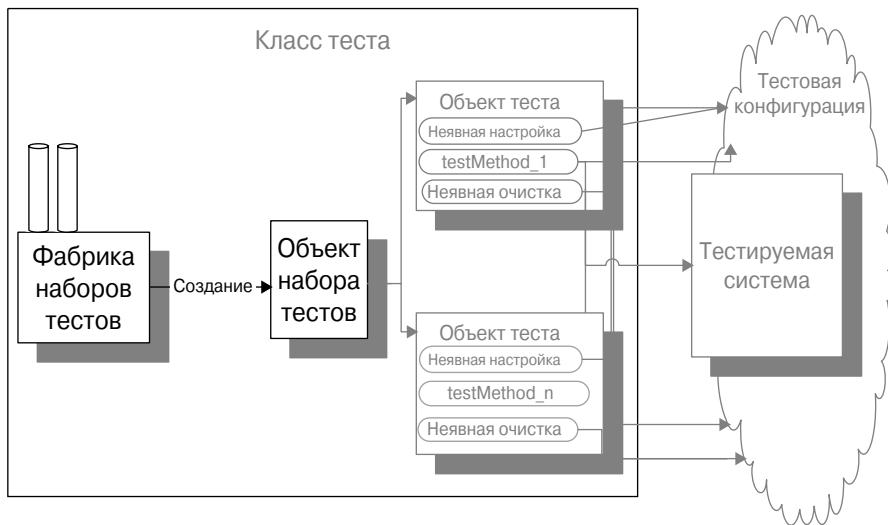
Суперкласс теста (Testcase Superclass) 646

Вспомогательный класс теста (Test Helper) 651

Именованный набор тестов (Named Test Suite)

Как запускать тесты произвольными группами?

Необходимо определить тестовый набор с подходящим именем, в котором содержится интересующее разработчика множество тестов, запускаемых одной группой.



Большое количество тестов необходимо систематизировать. Набор тестов позволяет группировать тесты с одной и той же функциональностью. Хотя необходимо иметь возможность запускать все тесты приложения или компонента, должен существовать механизм запуска только интересующих тестов, относящихся к подмножеству функциональности или компонентов системы.

Именованный набор тестов (Named Test Suite) позволяет запускать предварительно определенный набор тестов.

Как это работает

Для каждой группы связанных тестов можно определить специальную *фабрику наборов тестов* (Test Suite Factory) с описательным именем. Метод фабрики может использовать любой из способов создания наборов тестов для возврата *объекта набора тестов* (Test Suite Object, с. 414), содержащего интересующие *объекты теста* (Testcase Object, с. 410).

Когда это использовать

Хотя в большинстве случаев с помощью одной команды можно запустить все тесты, иногда приходится запускать только подмножество тестов. Наиболее распространенной причиной такого подхода является экономия времени. Если применяемая реализация xUnit не поддерживает *выбор тестов* (Test Selection), интересующие тесты разбросаны по разным контекстам и некоторые из них запускать нельзя, следует воспользоваться *набором с подмножеством* (Subset Suite).

Вариант: набор всех тестов (AllTests Suite)

Часто необходимо запустить все доступные тесты. В небольших системах существует стандартный способ запуска *набора всех тестов* (AllTests Suite) после получения кода из хранилища (что гарантирует известную отправную точку для начала работы) и перед каждым включением кода в хранилище (чтобы проверить работоспособность кода). Обычно *набор всех тестов* (AllTests Suite) создается для каждого пакета или пространства имен программного обеспечения, что позволяет запустить подмножество тестов после каждой модификации кода в цикле “красный, зеленый, рефакторинг”.

Вариант: набор с подмножеством (Subset Suite)

Часто разработчики не желают запускать тесты из-за их *медленного завершения* (Slow Tests, с. 289). Тесты для компонентов, использующих базу данных, работают значительно медленнее тестов, работающих в памяти. Определив один *именованный набор тестов* (Named Test Suite) для тестов с базой данных и другой *именованный набор тестов* (Named Test Suite) для тестов, работающих в памяти, можно избежать использования базы данных, запустив соответствующий *набор с подмножеством* (Subset Suite).

Еще одной причиной отказа от запуска тестов является недоступность используемого ими контекста. Например, если на рабочей станции разработчика отсутствует Web-сервер или если развертывание программного обеспечения на Web-сервере отнимает слишком много времени, не имеет смысла запускать тесты компонентов, основанных на Web-сервере (их запуск потребует больше времени и тесты все равно завершат работу неудачно).

Вариант: набор из одного теста (Single Test Suite)

Вырожденной формой *набора с подмножеством* (Subset Suite) является *набор из одного теста* (Single Test Suite), в пределах которого создается единственный *объект теста* (Testcase Object), позволяющий запустить единственный *тестовый метод* (Test Method, с. 378). Этот вариант может оказаться исключительно полезным при недоступности *обозревателя набора тестов* (Test Tree Explorer), а также в ситуации, когда *тестовый метод* (Test Method) для своего запуска требует один из вариантов *декоратора настройки* (Setup Decorator, с. 471). Только по этой причине некоторые разработчики держат открытый *класс теста* (Testcase Class) “Мои тесты” в своей рабочей среде.

Замечания по реализации

Концепция запуска именованных наборов тестов не зависит от процедуры их создания. Например, для явного создания наборов тестов можно использовать механизм *перечисления тестов* (Test Enumeration), а для поиска всех тестов в определенном каталоге — механизм *обнаружения тестов* (Test Discovery, с. 420). Кроме того, в пределах набора тестов можно воспользоваться механизмом *выбора тестов* (Test Selection, с. 429) для динамического создания меньшего набора. В одних реализациях xUnit нужно определить *набор всех тестов* (AllTests Suite) для каждого пакета или подсистемы вручную. В других, например в NUnit, *объект набора тестов* (Test Suite Object) создается автоматически для каждого набора тестов.

При использовании *перечисления тестов* (Test Enumeration) и наличии *именованных наборов тестов* (Named Test Suite) для различных подмножеств тестов лучше определить *набор всех тестов* (AllTests Suite) в терминах этих наборов. При реализации *набора*

всех тестов (AllTests Suite) как набора наборов (Suite of Suites) класса *теста* (Testcase Class) придется добавить только в один именованный набор тестов (Named Test Suite); после этого коллекция тестов добавляется в набор *всех тестов* (AllTests Suite) в локальном контексте, а также в именованный набор тестов (Named Test Suite) и в контексты более высокого уровня.

Замечания по рефакторингу

Этапы рефакторинга существующего кода в именованный набор тестов (Named Test Suite) зависят от используемого варианта именованных наборов. По этой причине здесь не приводится мотивирующий пример и сразу рассматриваются примеры именованных наборов тестов (Named Test Suite).

Пример: набор всех тестов (AllTests Suite)

Набор всех тестов (AllTests Suite) позволяет запускать все тесты для различных подмножеств функциональности в соответствии с выбором разработчика. Для каждого компонента или контекста (например, пакета Java) определяется специальный набор тестов (и соответствующая фабрика наборов тестов, Test Suite Factory), который называется AllTests. В метод suite фабрики наборов тестов (Test Suite Factory) добавляются все тесты текущего контекста и все именованные наборы тестов (Named Test Suite) во вложенных контекстах (например, из вложенных пакетов Java). Таким образом, при запуске именованного набора тестов (Named Test Suite) верхнего уровня запускаются все именованные наборы тестов (Named Test Suite) вложенных контекстов.

Ниже приведен пример кода для запуска всех тестов, подходящий для большинства реализаций xUnit.

```
public class AllTests {
    public static Test suite() {
        TestSuite suite = new TestSuite("Test for allJUnitTests");
        //JUnit-BEGIN$
        suite.addTestSuite(
            com.clrstream.camug.example.test.InvoiceTest.class);
        suite.addTest(com.clrstream.ex7.test.AllTests.suite());
        suite.addTest(com.clrstream.ex8.test.AllTests.suite());
        suite.addTestSuite(
            com.xunitpatterns.guardassertion.Example.class);
        //JUnit-END$
        return suite;
    }
}
```

В данном случае пришлось использовать несколько методов, так как добавляются другие именованные наборы тестов (Named Test Suite) и объекты набора тестов (Test Suite Object), представляющие единственный класс теста (Testcase Class). В пакете JUnit для этого используются разные методы. В других реализациях xUnit могут использоваться методы с одинаковыми сигнатурами.

Еще одной особенностью данного примера, о которой стоит упомянуть, являются комментарии JUnit-start и JUnit-end. Среда разработки (в данном случае — Eclipse) автоматически генерирует список команд между двумя комментариями, что является полуавтоматической формой обнаружения тестов (Test Discovery).

Пример: набор специального назначения (Special-Purpose Suite)

Предположим, существует три основных пакета (A, B и C), содержащих бизнес-логику. В каждый пакет входят классы, работающие с объектами в памяти и в базе данных. Для каждого из пакетов необходимо создать соответствующие пакеты тестов. Некоторые тесты в каждом пакете потребуют доступа к базе данных, а остальные смогут работать, полностью умешаясь в памяти.

Для всей системы и для каждого пакета (A, B и C) нужна возможность запуска следующих наборов тестов:

- все тесты;
- все тесты, использующие базу данных;
- все тесты, работающие в памяти.

В результате получается 12 именованных наборов тестов (по три именованных набора для каждого из четырех контекстов).

В каждом из трех пакетов (A, B и C) необходимо определить следующие именованные наборы тестов (Named Test Suite):

- AllDbTests (путем добавления всех *классов теста*, Testcase Class, содержащих тесты баз данных);
- AllInMemoryTests (путем добавления всех *классов теста*, Testcase Class, полностью работающих в памяти);
- AllTests (путем комбинирования наборов AllDbTests и AllInMemoryTests).

После этого на верхнем уровне контекста тестирования определяются *именованные наборы тестов* (Named Test Suite) с такими же именами:

- AllDbTests (путем комбинирования всех классов тестов из наборов AllDbTests для пакетов A, B и C);
- AllInMemoryTests (путем комбинирования всех классов тестов из наборов AllInMemoryTests для пакетов A, B и C);
- AllTests (путем комбинирования всех классов тестов из наборов AllTests для пакетов A, B и C; получается нормальный набор *всех тестов*, AllTests Suite).

Чтобы включить некоторые тесты из единственного *класса теста* (Testcase Class) в *именованные наборы тестов* (Named Test Suite), класс необходимо разделить на несколько классов для включения в каждый контекст (например, тесты с использованием базы данных и работающие в памяти).

Пример: набор из одного теста (Single Test Suite)

В некоторых ситуациях, особенно при использовании отладчика, запуск всех тестов в пределах класса может быть нежелательным. Одним из способов запуска только подмножества тестов является использование *обозревателя набора тестов* (Test Tree Explorer). Если эта возможность недоступна, можно скрыть за комментарием все ненужные тесты, скопировать класс теста и удалить ненужные тесты или изменить имена или атрибуты тестов, которые используются алгоритмом *обнаружения тестов* (Test Discovery).

```

public class LostTests extends TestCase {
    public LostTests(String name) {
        super(name);
    }
    public void xtestOne() throws Exception {
        fail("test not implemented");
    }
    /*
    public void testTwo() throws Exception {
        fail("test not implemented");
    }
    */
    public void testSeventeen() throws Exception {
        assertTrue(true);
    }
}

```

Если не восстановить внесенные модификации после решения возникшей проблемы, каждый из перечисленных подходов потенциально может привести к появлению *потерянных тестов* (Lost Test). *Набор из одного теста* (Single Test Suite) позволяет запускать конкретный тест (или тесты) без модификации соответствующих классов теста (Testcase Class). При таком решении учитывается тот факт, что в большинстве реализаций xUnit требуется конструктор *класса теста* (Testcase Class) с одним аргументом. Аргумент содержит имя метода, который будет вызываться экземпляром класса с помощью механизма интrosпекции. Конструктор с одним аргументом вызывается по одному разу для каждого *тестового метода* (Test Method) в классе. Полученный *объект теста* (Testcase Object) добавляется в *объект набора тестов* (Test Suite Object). Это пример шаблона *подключаемое поведение* (Pluggable behavior).

Единственный тест можно запускать с помощью реализации класса *фабрики наборов тестов* (Test Suite Factory) с единственным методом suite, который создает экземпляр интересующего *класса теста* (Testcase Class), вызывая конструктор с одним аргументом и передавая ему имя запускаемого *тестового метода* (Test Method). Получив от метода suite *объект набора тестов* (Test Suite Object), содержащий только один *объект теста* (Testcase Object), можно запускать единственный тест без модификации соответствующего *класса теста* (Testcase Class).

```

public class MyTest extends TestCase {
    public static Test suite() {
        return new LostTests("testSeventeen");
    }
}

```

Желательно всегда иметь под рукой класс *набора из одного теста* (Single Test Suite) и просто включать в него интересующий тест за счет модификации оператора импорта и метода suite. Часто приходится поддерживать несколько классов *наборов из одного теста* (Single Test Suite) для быстрого переключения между тестами. Такой подход значительно проще, чем поиск вручную в *обозревателе набора тестов* (Test Tree Explorer) и запуск конкретного теста. (Хотя у других разработчиков все может быть наоборот!)

Пример: контрольный набор тестов (Smoke Test Suite)

Для создания **контрольного набора тестов** (smoke test suite) идея *набора специального назначения* (Special-Purpose Suite) комбинируется со способом получения *набора из одного теста* (Single Test Suite). Такая стратегия предполагает выбор характерных тестов из каждой области системы и их включение в один *объект набора тестов* (Test Suite Object).

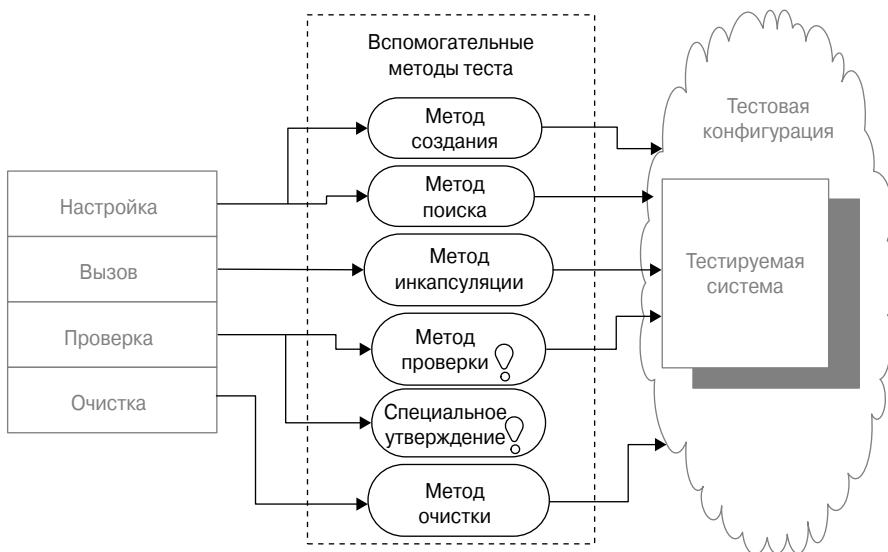
```
public class SmokeTestSuite extends TestCase {  
    public static Test suite() {  
        TestSuite mySuite = new TestSuite("Smoke Tests");  
        mySuite.addTest(new LostTests("testSeventeen"));  
        mySuite.addTest(new SampleTests("testOne"));  
        mySuite.addTest(new FlightManagementFacadeTest(  
            "testGetFlightsByOriginAirports_TwoOutboundFlights" ));  
        // здесь можно добавить дополнительные тесты  
        return mySuite;  
    }  
}
```

Такая схема не обеспечивает полноценное тестирование, но позволяет быстро обнаружить нарушения в основной функциональности.

Вспомогательный метод теста (Test Utility Method)

Как сократить дублирование тестового кода (Test Code Duplication)?

Повторно используемая логика теста скрывается внутри вспомогательного метода с подходящим именем.



При создании тестов время от времени возникает необходимость повторения одной и той же логики в большом количестве тестов. Изначально при создании новых тестов с той же логикой происходит “клонирование и модификация”. Но рано или поздно приходит понимание, что подобное *дублирование тестового кода* (Test Code Duplication, с. 254) становится источником проблем. На этом этапе необходимо рассмотреть вариант создания *вспомогательных методов теста* (Test Utility Method).

Как это работает

Подпрограмма и функция были одними из первых механизмов повторного использования логики в нескольких местах программы. *Вспомогательный метод теста* (Test Utility Method) реализует тот же принцип в объектно-ориентированном коде. Любая логика, которая используется в нескольких тестах, выносится во *вспомогательный метод теста* (Test Utility Method). После этого метод можно вызывать из разных тестов и даже несколько раз из одного теста. Конечно, меняющаяся от вызова к вызову информация передается во вспомогательный метод в виде аргументов.

Когда это использовать

Вспомогательные методы теста (Test Utility Method) должны использоваться каждый раз, когда тестовая логика встречается в нескольких тестах и ее необходимо использовать

повторно. Кроме того, *вспомогательные методы теста* (Test Utility Method) позволяют гарантировать правильность работы логики. Чтобы быть в этом уверенными, следует создать *самопроверяющиеся тесты* (Self-Checking Test) для повторно используемой тестовой логики. Поскольку *тестовые методы* (Test Method) тестируемому не поддаются, лучше вынести требующую проверки логику во *вспомогательные методы теста* (Test Utility Method) и уже после этого выполнить проверку.

Основным недостатком использования шаблона *вспомогательных методов теста* (Test Utility Method) является создание дополнительного программного интерфейса, который приходится изучать разработчикам. Этот недостаток можно значительно уменьшить за счет использования описательных имен вспомогательных методов и рефакторинга для их определения.

Существует множество типов *вспомогательных методов теста* (Test Utility Method). Ниже рассматриваются наиболее популярные варианты. Некоторые из них настолько важны, что для них отведен отдельный раздел книги.

Вариант: метод создания (Creation Method)

Методы создания (Creation Method, с. 441) используются для создания готовых объектов, входящих в тестовую конфигурацию. В них скрываются сложность процедуры создания объектов и их взаимосвязи с другими объектами. *Методы создания* (Creation Method) настолько важны, что считаются отдельным шаблоном.

Вариант: подключаемый метод (Attachment Method)

Подключаемый метод (Attachment Method) представляет собой специальный вариант *метода создания* (Creation Method) и предназначается для модификации уже существующих объектов, входящих в тестовую конфигурацию.

Вариант: метод поиска (Finder Method)

Любую логику извлечения объектов из *общей тестовой конфигурации* (Shared Fixture, с. 350) можно скрыть внутри функции, которая возвращает необходимый объект. Такой функции присваивается описательное имя, чтобы каждый читатель мог понять тип используемой тестовой конфигурации.

Методы поиска (Finder Method) должны использоваться каждый раз при поиске объектов в существующей *общей тестовой конфигурации* (Shared Fixture), соответствующих определенным критериям. Это позволит избежать появления “хрупкой” *тестовой конфигурации* (Fragile Fixture) и высокой стоимости обслуживания тестов (High Test Maintenance Cost, с. 300). *Методы поиска* (Finder Method) могут использоваться как при чистой стратегии на основе *общей тестовой конфигурации* (Shared Fixture), так и при гибридной стратегии на основе *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture). Кроме того, *методы поиска* (Finder Method) позволяют предотвратить появление *непонятных тестов* (Obscure Test, с. 230) за счет сокрытия механизма поиска необходимых объектов, позволяя читателям уделить основное внимание причинам использования конкретного объекта и его взаимосвязи с ожидаемым результатом, который описан в утверждениях. Это делает тесты более пригодными к использованию в качестве документации.

Хотя большинство *методов поиска* (Finder Method) возвращает единственную ссылку на объект, этот объект может оказаться корнем целого дерева объектов (т.е. счет может ссылаться на покупателя и различные адреса, а также содержать список товаров). В некоторых ситуациях может потребоваться определение *метода поиска* (Finder Method), который возвращает коллекцию (Array или Hash) объектов, но такое применение *методов поиска* (Finder Method) не столь распространено. Кроме того, *методы поиска* (Finder Method) могут обновлять параметры для передачи дополнительных объектов вызвавшему тесту, хотя такой подход не способствует интуитивному пониманию (в сравнении с использованием функции). Инициализация переменных экземпляра не рекомендуется как способ возврата объектов, поскольку подобный подход вносит путаницу и не позволяет позднее вынести *метод поиска* (Finder Method) во *вспомогательный класс теста* (Test Helper, с. 651).

Метод поиска (Finder Method) находит объекты в *общей тестовой конфигурации* (Shared Fixture) несколькими способами: через использование прямых ссылок (переменные экземпляра или класса, инициализированные на этапе создания тестовой конфигурации), через поиск объектов по известному ключу или по соответствуанию заданному критерию. Преимуществом использования прямых ссылок или известных ключей является возврат одного и того же объекта при каждом вызове теста. Основным недостатком является вероятность модификации объекта другим тестом, после чего объект перестает соответствовать критериям, подразумеваемым именем метода поиска. Поиск по критерию помогает избежать этой проблемы, хотя полученные тесты будут работать медленнее и окажутся менее определенными, если при каждом запуске будут использоваться разные объекты. В любом случае при каждом изменении *общей тестовой конфигурации* (Shared Fixture) код приходится модифицировать в меньшем количестве мест (по сравнению с непосредственным обращением к объектам из *тестовых методов*, Test Method).

Вариант: метод инкапсуляции тестируемой системы (SUT Encapsulation Method)

Также известен как:

Инкапсуляция программного интерфейса тестируемой системы (SUT API Encapsulation)

Еще одной причиной использования *вспомогательных методов теста* (Test Utility Method) является инкапсуляция ненужного знания о программном интерфейсе тестируемой системы. Что значит “ненужного”? Любой метод тестируемой системы, который вызывается, но не проверяется в конкретном teste, добавляет связность между тестом и системой. *Методы создания* (Creation Method) и *специальные утверждения* (Custom Assertion, с. 495) являются достаточно распространенными примерами *методов инкапсуляции тестируемой системы* (SUT Encapsulation Method), чтобы для них были выделены отдельные разделы. В данном разделе рассматриваются менее распространенные применения таких методов. Например, если вызываемый (или используемый для проверки результата) метод имеет сложную сигнатуру, увеличивается объем работы для написания и обслуживания кода теста, а также усложняется его понимание. Этой проблемы можно избежать, если скрыть вызовы в *методе инкапсуляции тестируемой системы* (SUT Encapsulation Method) с описательным именем и простой сигнатурой.

Вариант: специальное утверждение (Custom Assertion)

Специальное утверждение (Custom Assertion) используется для определения пред назначенного для теста равенства, пригодного для повторного использования в нескольких тестах. Такое утверждение скрывает сложность сравнения ожидаемого и фактического

результатов. Обычно *специальное утверждение* (Custom Assertion) не имеет побочных эффектов и не взаимодействует с тестируемой системой для получения результата; эта задача должна быть решена вызывающей стороной.

Вариант: метод проверки (Verification Method)

Методы проверки (Verification Method) используются для проверки возврата ожидаемого результата. Они скрывают от теста сложность проверки результата. В отличие от *специального утверждения* (Custom Assertion) *методы проверки* (Verification Method) взаимодействуют с тестируемой системой.

Вариант: параметризованный тест (Parameterized Test)

Наиболее полным вариантом *вспомогательных методов теста* (Test Utility Method) является *параметризованный тест* (Parameterized Test, с. 618). По своей сути он является полноценным тестом, пригодным для повторного использования в различных ситуациях. В него передаются изменяющиеся от теста к тесту данные, а *параметризованный тест* (Parameterized Test) выполняет все фазы *четырехфазного теста* (Four-Phase Test, с. 387).

Вариант: метод очистки (Cleanup Method)

Методы очистки (Cleanup Method)¹ используются на этапе удаления тестовой конфигурации для удаления всех ресурсов, сохранившихся после завершения работы теста. Более подробное описание и примеры применения методов очистки приводятся в разделе, посвященном *автоматической очистке* (Automated Teardown, с. 521).

Замечания по реализации

Основным доводом против использования *вспомогательных методов теста* (Test Utility Method) является вынесение части логики за пределы теста, что может усложнить его понимание. Одним из способов обхода этой проблемы является присвоение вспомогательным методам описательных имен. На самом деле правильно выбранные имена могут сделать тесты еще более простыми для понимания, так как появление *непонятных тестов* (Obscure Test) предотвращается за счет определения языка *высокого уровня* (Higher-Level Language, с. 95). Кроме того, *вспомогательные методы теста* (Test Utility Method) должны оставаться небольшими и независимыми. Для этого все параметры должны передаваться явно в виде аргументов (а не через переменные экземпляра). Все объекты должны передаваться в тест в виде возвращаемого значения или обновленного параметра.

Для формирования описательных имен сами тесты должны вынуждать разработчика создавать *вспомогательные методы теста* (Test Utility Method), а не разработчик выдумывать методы, которые могут понадобиться позднее. Такой подход в направлении “извне вовнутрь” позволяет избежать “заимствования завтраших проблем” и помогает построить минимальное решение.

В написании пригодных к использованию *вспомогательных методов теста* (Test Utility Method) нет ничего сложного. Более сложной проблемой является их размещение. Ес-

¹ Кое-где может использоваться название “teardown method”, но оно пересекается с названием метода, применяемым в шаблоне *неявная очистка* (Implicit Teardown, с. 533).

ли *вспомогательный метод теста* (Test Utility Method) нужен только в пределах одного класса теста (Testcase Class, с. 401), его можно сохранить в этом классе. Если *вспомогательный метод теста* (Test Utility Method) используется несколькими классами, решение несколько усложняется. Ключевая проблема связана с видимостью типов. Клиентские классы должны видеть *вспомогательный метод теста* (Test Utility Method), а вспомогательный метод должен видеть все типы и классы, от которых он зависит. Если список таких зависимостей невелик или все классы и/или типы видны из одного места, *вспомогательный метод теста* (Test Utility Method) можно разместить в общем *суперклассе теста* (Testcase Superclass, с. 646), определенном в пределах конкретного проекта или в пределах целой компании. Если вспомогательный метод зависит от типов и/или классов, которые не видны из одного места, видимого всем клиентам, его придется разместить во *вспомогательном классе теста* (Test Helper), в соответствующем пакете или подсистеме. В больших системах со значительным количеством объектов предметной области для каждой группы (пакета) связанных объектов предметной области создается по одному *вспомогательному классу теста* (Test Helper).

Вариант: тест вспомогательного метода теста (Test Utility Test)

Одним из основных преимуществ использования *вспомогательных методов теста* (Test Utility Method) является возможность их проверки нормальными тестами. Конкретная природа таких тестов зависит от проверяемого метода, но в качестве примера можно привести *тест специального утверждения* (Custom Assertion Test).

Мотивирующий пример

В следующем примере показан тест, который написало бы большинство начинающих разработчиков.

```
public void testAddItemQuantity_severalQuantity_v1() {
    Address billingAddress = null;
    Address shippingAddress = null;
    Customer customer = null;
    Product product = null;
    Invoice invoice = null;
    try {
        // настройка тестовой конфигурации
        billingAddress = new Address("1222 1st St SW",
            "Calgary", "Alberta",
            "T2N 2V2", "Canada");
        shippingAddress = new Address("1333 1st St SW",
            "Calgary", "Alberta",
            "T2N 2V2", "Canada");
        customer = new Customer(99, "John", "Doe",
            new BigDecimal("30"),
            billingAddress,
            shippingAddress);
        product = new Product(88, "SomeWidget",
            new BigDecimal("19.99"));
        invoice = new Invoice(customer);
        // Вызов тестируемой системы
        invoice.addItemQuantity(product, 5);
        // Проверка результата
    }
```

```

List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
    LineItem actItem = (LineItem) lineItems.get(0);
    assertEquals("inv", invoice, actItem.getInv());
    assertEquals("prod", product, actItem.getProd());
    assertEquals("quant", 5, actItem.getQuantity());
    assertEquals("discount",
        new BigDecimal("30"),
        actItem.getPercentDiscount());
    assertEquals("unit price",
        new BigDecimal("19.99"),
        actItem.getUnitPrice());
    assertEquals("extended",
        new BigDecimal("69.96"),
        actItem.getExtendedPrice());
} else {
    assertTrue("Invoice should have 1 item", false);
}
} finally {
// Очистка
deleteObject(invoice);
deleteObject(product);
deleteObject(customer);
deleteObject(billingAddress);
deleteObject(shippingAddress);
}
}

```

Этот тест сложно понять, так как он содержит множество запахов кода, включая *непонятный тест* (Obscure Test) и *фиксированные данные теста* (Hard-Coded Test Data).

Замечания по рефакторингу

Часто *вспомогательные методы теста* (Test Utility Method) создаются за счет извлечения повторно используемой логики из существующих тестов. Для этих целей можно воспользоваться рефакторингом *выделение метода* (Extract Method). Полученный вспомогательный метод можно оставить в том же классе теста, а можно вынести в суперкласс с помощью рефакторинга *подъем метода* (Pull up method) или перенести в другой класс с помощью рефакторинга *перемещение метода* (Move method).

Пример: вспомогательный метод теста (Test Utility Method)

Ниже показан переработанный вариант теста. Обратите внимание, насколько он упростился. Это только один пример использования *вспомогательных методов теста* (Test Utility Method).

```

public void testAddItemQuantity_severalQuantity_v13() {
    final int QUANTITY = 5;
    final BigDecimal CUSTOMER_DISCOUNT = new BigDecimal("30");
    // Настройка тестовой конфигурации
    Customer customer =
        findActiveCustomerWithDiscount(CUSTOMER_DISCOUNT);
    Product product = findCurrentProductWith3DigitPrice( );
    Invoice invoice = createInvoice(customer);
}

```

```

// Вызов тестируемой системы
invoice.addItemQuantity(product, QUANTITY);
// Проверка результата
final BigDecimal BASE_PRICE = product.getUnitPrice().
    multiply(new BigDecimal(QUANTITY));
final BigDecimal EXTENDED_PRICE =
    BASE_PRICE.subtract(BASE_PRICE.multiply(
        CUSTOMER_DISCOUNT.movePointLeft(2)));
LineItem expected =
    createLineItem(QUANTITY, CUSTOMER_DISCOUNT,
        EXTENDED_PRICE, product, invoice);
assertContainsExactlyOneLineItem(invoice, expected);
}

```

Рассмотрим внесенные изменения пошагово. Сначала код создания объектов Customer и Product заменен вызовами *методов поиска* (Finder Method), извлекающими эти объекты из *недомодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture). Код был изменен, так как модификация этих объектов не планируется.

```

protected Customer findActiveCustomerWithDiscount(
    BigDecimal percentDiscount) {
    return CustomerHome.findCustomerById(
        ACTIVE_CUSTOMER_WITH_30PC_DISCOUNT_ID);
}

```

После этого был написан *метод создания* (Creation Method) для объекта Invoice, в который добавляются объекты LineItem.

```

protected Invoice createInvoice(Customer customer) {
    Invoice newInvoice = new Invoice(customer);
    registerTestObject(newInvoice);
    return newInvoice;
}
List testObjects;
protected void registerTestObject(Object testObject) {
    testObjects.add(testObject);
}

```

Для того чтобы избежать *встроенной очистки* (In-line Teardown, с. 528), каждый созданный объект регистрируется в механизме *автоматической очистки* (Automated Tear-down), вызываемом из метода tearDown.

```

private void deleteTestObjects() {
    Iterator i = testObjects.iterator();
    while (i.hasNext()) {
        try {
            deleteObject(i.next());
        } catch (RuntimeException e) {
            // Ничего не делать; достаточно обеспечить
            // переход к следующему объекту в списке
        }
    }
}
public void tearDown() {
    deleteTestObjects();
}

```

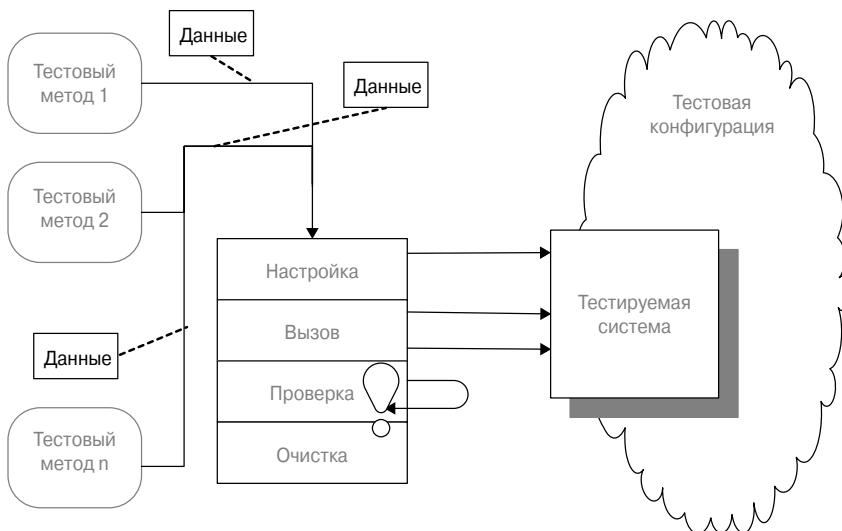
Наконец, было сформировано *специальное утверждение* (Custom Assertion), проверяющее правильность добавленного в объект Invoice объекта LineItem.

```
void assertContainsExactlyOneLineItem( Invoice invoice,
                                         LineItem expected) {
    List lineItems = invoice.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actItem = (LineItem)lineItems.get(0);
    assertLineItemsEqual("",expected, actItem);
}
```

Параметризованный тест (Parameterized Test)

Как сократить дублирование тестового кода (Test Code Duplication), когда одна и та же логика присутствует во многих тестах?

Необходимая для создания тестовой конфигурации и проверки результатов информация передается во вспомогательный метод, в котором реализован полный жизненный цикл теста.



Процесс тестирования может состоять из повторяющихся действий не только из-за повторного запуска одного и тот же теста, но и из-за незначительных отличий между разными тестами. Например, одна и та же логика теста должна запускаться с незначительно отличающимися входными параметрами, чтобы можно было убедиться, что результат тоже отличается незначительно. Все тесты такой группы будут состоять из одинаковых операций. Хотя создание большого количества тестов помогает обеспечить полное покрытие кода, с точки зрения обслуживания это не очень хорошее решение, так как модификация в алгоритме одного из тестов потребует модификации всех остальных тестов в группе.

Параметризованный тест (Parameterized Test) обеспечивает возможность повторного использования логики теста в нескольких тестовых методах (Test Method, с. 378).

Как это работает

Конечно, решение предполагает выделение общей логики во вспомогательный метод. Если выделенная логика содержит все четыре фазы *четырехфазного теста* (Four-Phase Test, с. 387), т.е. настройку тестовой конфигурации, вызов тестируемой системы, проверку результата и очистку конфигурации, полученный вспомогательный метод можно на-

звать *параметризованным тестом* (Parameterized Test). Такой тест обеспечивает максимальное покрытие при минимальном объеме обслуживаемого кода. Кроме того, при необходимости можно очень просто добавить новые тесты.

При наличии подходящего вспомогательного метода предполагающий выполнение последовательности сложных операций тест можно заменить одной строкой кода. Обнаруженные совпадения в нескольких тестах можно выделить во *вспомогательный метод теста* (Test Utility Method, с. 610), принимающий в качестве аргументов информацию, отличающую тесты один от другого. В результате *тестовый метод* (Test Method) передает в *параметризованный тест* (Parameterized Test) в качестве параметров информацию, отличающую тесты друг от друга.

Когда это использовать

Параметризованный тест (Parameterized Test) может использоваться каждый раз, когда *дублирование тестового кода* (Test Code Duplication, с. 254) приводит к реализации одного алгоритма в нескольких тестах, отличающихся только данными. Отличающиеся данные превращаются в аргументы, передаваемые в *параметризованный тест* (Parameterized Test), а логика скрывается внутри вспомогательного метода. Кроме того, *параметризованный тест* (Parameterized Test) помогает избежать появления *непонятных тестов* (Obscure Test, с. 230). Сокращение повторений логики делает *класс теста* (Testcase Class, с. 401) значительно более коротким. Кроме того, *параметризованный тест* (Parameterized Test) является хорошим промежуточным этапом при создании *управляемых данными тестов* (Data-Driven Test, с. 322). Имя *параметризованного теста* (Parameterized Test) отображается на глагол или “слово операции” в управляемом данными teste (Data-Driven Test), а параметры являются атрибутами.

Если выделенный вспомогательный метод не выполняет настройку тестовой конфигурации, он еще называется *методом проверки* (Verification Method). Если при этом он и не взаимодействует с тестируемой системой, он называется *специальным утверждением* (Custom Assertion).

Замечания по реализации

Параметризованный тест (Parameterized Test) должен иметь описательное имя, позволяющее читателю теста сразу определить его назначение. Имя должно показывать, что метод реализует полный жизненный цикл теста. В соответствии с одним из соглашений имени теста начинается или заканчивается на “test”. Наличие параметров подсказывает, что тест является параметризованным. В большинстве реализаций xUnit, поддерживающих механизм *обнаружения тестов* (Test Discovery, с. 420), *объекты теста* (Testcase Object, с. 410) создаются только для методов, имена которых содержат “test”, а сигнатура не содержит параметров. Такое ограничение не препятствует созданию *параметризованных тестов* (Parameterized Test) с именами, содержащими “test”. Как минимум в одной реализации xUnit (MbUnit) *параметризованные тесты* (Parameterized Test) реализованы на уровне *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332). Расширения существуют и для других реализаций. В числе первых можно назвать DDSteps для инфраструктуры JUnit.

Апологеты тестирования обязательно предложат написать *самопроверяющийся тест* (Self-Checking Test, с. 81) для проверки *параметризованного теста* (Parameterized Test).

Преимущества такого подхода очевидны (включая повышенную уверенность в собственных тестах) и в большинстве случаев не требуют значительных усилий. Хотя это и сложнее, чем писать модульные тесты для *специальных утверждений* (Custom Assertion), поскольку в процессе работы происходит взаимодействие с тестируемой системой. Скорее всего, придется заменить тестируемую систему *тестовым двойником* (Test Double), что позволит проверять правильность ее вызова и управлять возвращаемыми значениями².

Вариант: табличный тест (Tabular Test)

Также известен как: *Строковый тест* (Row Test) Несколько рецензентов этой книги написали о регулярно используемом варианте *параметризованного теста* (Parameterized Test) — о табличном teste (Tabular Test).

По сути этот вариант совпадает с *параметризованным тестом* (Parameterized Test), но все множество значений хранится в единственном *тестовом методе* (Test Method). К сожалению, такой подход превращает тест в “энергичный” (Eager Test), так как проверяется несколько тестовых условий. Это не проблема, если все тесты завершаются успешно, но если одна из “строк” таблицы значений завершается неудачно, *локализация дефектов* (Defect Localization, с. 78) оказывается недостаточно конкретной.

Существует еще одна потенциальная проблема: “строки тестов” могут зависеть одна от другой (случайно или намеренно), так как они выполняются в пределах одного *объекта теста* (Testcase Object). Пример такого поведения можно наблюдать в *инкрементном табличном teste* (Incremental Tabular Test).

Несмотря на потенциальные проблемы *табличные teste* (Tabular Test) могут обеспечить очень эффективное тестирование. Как минимум в одном варианте xUnit табличные teste реализованы на уровне инфраструктуры: в MbUnit предоставляется атрибут [RowTest], обозначающий *параметризованные teste* (Parameterized Test), а еще один атрибут [Row(x, y, . . .)] позволяет указать передаваемые параметры. Вероятно, эта возможность будет перенесена в другие реализации xUnit. (Это намек!)

Вариант: инкрементный табличный тест (Incremental Tabular Test)

Инкрементный табличный тест (Incremental Tabular Test) представляет собой вариант табличного teste, в котором в качестве конфигурации намеренно используется результат работы предыдущих строк teste. Такой подход представляет собой намеренно созданные *взаимодействующие teste* (Interacting Tests), получившие отдельное название — *цепочки teste* (Chained Tests, с. 477); но в данном случае teste расположены в пределах одного *тестового метода* (Test Method). Шаги в пределах *тестового метода* (Test Method) напоминают шаги “DoFixture” в инфраструктуре Fit, но без информации по каждому неудачно завершившемуся шагу³.

² В данном случае терминология становится очень запутанной, так как заменять тестируемую систему *тестовым двойником* (Test Double) нельзя. Строго говоря, заменяется объект, который данным тестом рассматривается как тестируемая система. Поскольку фактически проверяется поведение *параметризованного теста* (Parameterized Test), обычно играющий роль тестируемой системы объект превращается в вызываемый компонент. Голова идет кругом только от попытки это описать; к счастью, все не настолько сложно и структура станет понятной при попытке ее реализовать.)

³ Это происходит из-за завершения *тестового метода* (Test Method) после первого ложного утверждения в большинстве реализаций xUnit.

Вариант: тест на основе цикла (Loop-Driven Test)

Если тестируемую систему необходимо проверить со всеми значениями из определенного списка или диапазона, *параметризованный тест* (Parameterized Test) вызывается из цикла, перебирающего допустимые значения. Вложив один цикл в другой, можно проверить поведение системы на комбинациях входных значений. Основным требованием к такому тестированию является перечисление ожидаемых результатов для каждого входного значения (или их комбинации) или использование вычисляемых значений (Calculated Value) без внесения логики продукта внутрь теста (Production Logic in Test). *Тест на основе цикла* (Loop-Driven Test) страдает от большинства проблем, характерных для *табличных тестов* (Tabular Test), так как несколько тестов скрываются внутри одного *тестового метода* (Test Method), а значит, и внутри одного *объекта теста* (Testcase Object).

Мотивирующий пример

В следующем примере показаны тесты на основе пакета `runit` (Ruby Unit), проверяющие инфраструктуру публикации Web-сайтов, написанную на языке Ruby в процессе работы над этой книгой. Все *простые тесты успешности* (Simple Success Test) для тэгов перекрестных ссылок проходят через одну и ту же последовательность: определение входного кода XML, определение ожидаемого кода HTML, замена выходного файла, настройка обработчика кода XML, извлечение полученного кода HTML и его сравнение с ожидаемым кодом HTML.

```
def test_extref
  # Настройка
  sourceXml = "<extref id='abc' />"
  expectedHtml = "<a href='abc.html'>abc</a>"
  mockFile = MockFile.new
  @handler = setupHandler(sourceXml, mockFile)
  # Вызов
  @handler.printBodyContents
  # Проверка
  assert_equals_html(expectedHtml, mockFile.output,
    "extref: html output")
end
def testTestterm_normal
  sourceXml = "<testterm id='abc' />"
  expectedHtml = "<a href='abc.html'>abc</a>"
  mockFile = MockFile.new
  @handler = setupHandler(sourceXml, mockFile)
  @handler.printBodyContents
  assert_equals_html(expectedHtml, mockFile.output,
    "testterm: html output")
end
def testTestterm_plural
  sourceXml = "<testterms id='abc' />"
  expectedHtml = "<a href='abc.html'>abcs</a>"
  mockFile = MockFile.new
  @handler = setupHandler(sourceXml, mockFile)
  @handler.printBodyContents
  assert_equals_html(expectedHtml, mockFile.output,
    "testterms: html output")
end
```

Хотя большая часть общей логики уже выделена в метод `setupHandler`, некоторая доля *дублирования тестового кода* (*Test Code Duplication*) осталась. В этом случае оставалось как минимум 20 тестов с одинаковыми структурами (и еще больше тестов ожидали своей очереди), поэтому было принято решение значительно упростить создание таких тестов.

Замечания по рефакторингу

Переработка в направлении *параметризованного теста* (*Parameterized Test*) напоминает движение в сторону *специальных утверждений* (*Custom Assertion*). Основным отличием является включение обращений к тестируемой системе в рефакторинг *выделение метода* (*Extract Method*). Поскольку тесты практически идентичны, после определения тестовой конфигурации и ожидаемых результатов оставшийся код можно выделить в *параметризованный тест* (*Parameterized Test*).

Пример: параметризованный тест (*Parameterized Test*)

В следующих тестах логика сокращена до двух шагов: инициализации двух переменных и вызова вспомогательного метода, в котором выполняются все необходимые операции. Вспомогательный метод и является *параметризованным тестом* (*Parameterized Test*).

```
def test_extref
  sourceXml = "<extref id='abc' />"
  expectedHtml = "<a href='abc.html'>abc</a>"
  generateAndVerifyHtml(sourceXml, expectedHtml, "<extref>")
end
def test_testterm_normal
  sourceXml = "<testterm id='abc' />"
  expectedHtml = "<a href='abc.html'>abc</a>"
  generateAndVerifyHtml(sourceXml, expectedHtml, "<testterm>")
end
def test_testterm_plural
  sourceXml = "<testterms id='abc' />"
  expectedHtml = "<a href='abc.html'>abcs</a>"
  generateAndVerifyHtml(sourceXml, expectedHtml, "<plural>")
end
```

Сократить этот тест можно, определив следующий *параметризованный тест* (*Parameterized Test*).

```
def generateAndVerifyHtml(sourceXml, expectedHtml,
                           message, &block)
  mockFile = MockFile.new
  sourceXml.delete!("\t")
  @handler = setupHandler(sourceXml, mockFile)
  block.call unless block == nil
  @handler.printBodyContents
  actual_html = mockFile.output
  assert_equal_html(expectedHtml,
                    actual_html,
                    message + "html output")
  actual_html
end
```

Основное отличие этого *параметризованного теста* (Parameterized Test) от *метода проверки* (Verification Method) — наличие первых трех фаз *четырехфазного теста* (Four-Phase Test) (начиная с настройки и заканчивая проверкой), в то время как *метод проверки* (Verification Method) содержит только фазы вызова тестируемой системы и проверки результата. Обратите внимание, что тесту не требуется фаза очистки конфигурации, так как используется *очистка со сборкой мусора* (Garbage-Collected Teardown, с. 518).

Пример: независимый табличный тест (Independent Tabular Test)

Ниже показан тот же тест, но реализованный в виде *независимого табличного теста* (Independent Tabular Test).

```
def test_a_href_Generation
  row("extref", "abc", "abc.html", "abc")
  row("testterm", 'abc', "abc.html", "abc")
  row("testterms", 'abc', "abc.html", "abcs")
end
def row(tag, id, expected_href_id, expected_a_contents)
  sourceXml = "<" + tag + " id='" + id + "'>"
  expectedHtml = "<a href='" + expected_href_id + "'>
    " + expected_a_contents + "</a>"
  msg = "<" + tag + "> "
  generateAndVerifyHtml(sourceXml, expectedHtml, msg)
end
```

Неправда ли, красивое и компактное представление различных тестовых условий? В данном случае рефакторинг *встраивание временной переменной* (In-line temp) применялся к локальным переменным sourceXml и expectedHtml в списке аргументов метода generateAndVerifyHtml и различные *тестовые методы* (Test Method) были объединены в один. Большая часть работы заключалась в форматировании таблицы для ее приведения в соответствие с шириной страницы в книге (что в реальной жизни делать никогда не придется). Это ограничение продиктовало необходимость обрезания текста в каждой строке и изменение кода HTML и ожидаемого кода XML в методе row. Имя row было выбрано для единства с примером для инфраструктуры MbUnit, показанным далее в этом разделе, но с тем же успехом метод мог называться test_element.

К сожалению, с точки зрения *программы запуска тестов* (Test Runner, с. 406) в отличие от предыдущих примеров это один тест. Поскольку все тесты находятся в пределах одного *тестового метода* (Test Method), неудачное завершение любой из строк, кроме последней, приведет к потере информации. В данном примере не приходится думать о *взаимодействующих тестах* (Interacting Tests), так как метод generateAndVerify создает новую конфигурацию при каждом вызове. Но в реальной ситуации нельзя забывать о такой возможности развития событий.

Пример: инкрементный табличный тест (Incremental Tabular Test)

Поскольку *табличный тест* (Tabular Test) определен в пределах одного *тестового метода* (Test Method), он запускается в пределах единственного *объекта теста* (Testcase Object). Такая структура позволяет создавать последовательность действий. Ниже представлен пример, который Клинт Шанк приводит в своем журнале.

```

public class TabularTest extends TestCase {
    private Order order = new Order();
    private static final double tolerance = 0.001;
    public void testGetTotal() {
        assertEquals("initial", 0.00, order.getTotal(), tolerance);
        testAddItemAndGetTotal("first", 1, 3.00, 3.00);
        testAddItemAndGetTotal("second", 3, 5.00, 18.00);
        // И т.д.
    }
    private void testAddItemAndGetTotal(String msg,
                                       int lineItemQuantity,
                                       double lineItemPrice,
                                       double expectedTotal) {
        // Настройка
        LineItem item = new LineItem(lineItemQuantity,
                                      lineItemPrice);
        // Вызов тестируемой системы
        order.addItem(item);
        // Проверка суммы
        assertEquals(msg, expectedTotal, order.getTotal(), tolerance);
    }
}

```

Обратите внимание, что каждая строка *инкрементного табличного теста* (Incremental Tabular Test) основана на действиях предыдущей строки.

Пример: табличный тест (Tabular Test) с поддержкой со стороны инфраструктуры (MbUnit)

Ниже приведен пример из документации к инфраструктуре MbUnit, в котором показано использование атрибута [RowTest] для обозначения *параметризованных тестов* (Parameterized Test) и атрибута [Row(x, y, . . .)] для перечисления передаваемых в него параметров.

```

[RowTest()]
[Row(1, 2, 3)]
[Row(2, 3, 5)]
[Row(3, 4, 8)]
[Row(4, 5, 9)]
public void tAdd(Int32 x, Int32 y, Int32 expectedSum)
{
    Int32 Sum;
    Sum = this.Subject.Add(x,y);
    Assert.AreEqual(expectedSum, Sum);
}

```

Кроме “синтаксического сахара” атрибутов [Row(x, y, . . .)], код удивительно напоминает предыдущий пример. Но код не страдает от потери *локализации дефектов* (Defect Localization), поскольку каждая строка рассматривается как отдельный тест. Для преобразования предыдущего примера в такой формат потребуется только функция поиска и замены текстового редактора.

Пример: тест на основе цикла (Loop-Driven Test) (перечисленные значения)

В следующем примере тестируемая система вызывается в цикле с разными входными параметрами.

```
public void testMultipleValueSets() {
    // Настройка тестовой конфигурации
    Calculator sut = new Calculator();
    TestValues[] testValues = {
        new TestValues(1,2,3),
        new TestValues(2,3,5),
        new TestValues(3,4,8), // особый случай!
        new TestValues(4,5,9)
    };
    for (int i = 0; i < testValues.length; i++) {
        TestValues values = testValues[i];
        // Вызов тестируемой системы
        int actual = sut.calculate(values.a, values.b);
        // Проверка результата
        assertEquals(message(i), values.expectedSum, actual);
    }
}
private String message(int i) {
    return "Row " + String.valueOf(i);
}
```

В данном случае перечисляются ожидаемые результаты для каждого набора входных значений. Такая стратегия позволяет избежать попадания логики продукта *внутрь теста* (Production Logic in Test).

Пример: тест на основе цикла (Loop-Driven Test) с использованием вычисляемых значений (Calculated Values)

Следующий пример выглядит немного сложнее.

```
public void testCombinationsOfInputValues() {
    // Настройка тестовой конфигурации
    Calculator sut = new Calculator();
    int expected; // Будет определено внутри циклов
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            // Вызов тестируемой системы
            int actual = sut.calculate(i, j);
            // Проверка результата
            if (i==3 & j==4) // особый случай
                expected = 8;
            else
                expected = i+j;
            assertEquals(message(i,j), expected, actual);
        }
    }
}
private String message(int i, int j) {
    return "Cell( " + String.valueOf(i) + ", "
           + String.valueOf(j) + " )";
}
```

К сожалению, этот пример страдает от *логики продукта внутри теста* (Production Logic in Test), так как приходится обрабатывать специальный случай.

Источники дополнительной информации

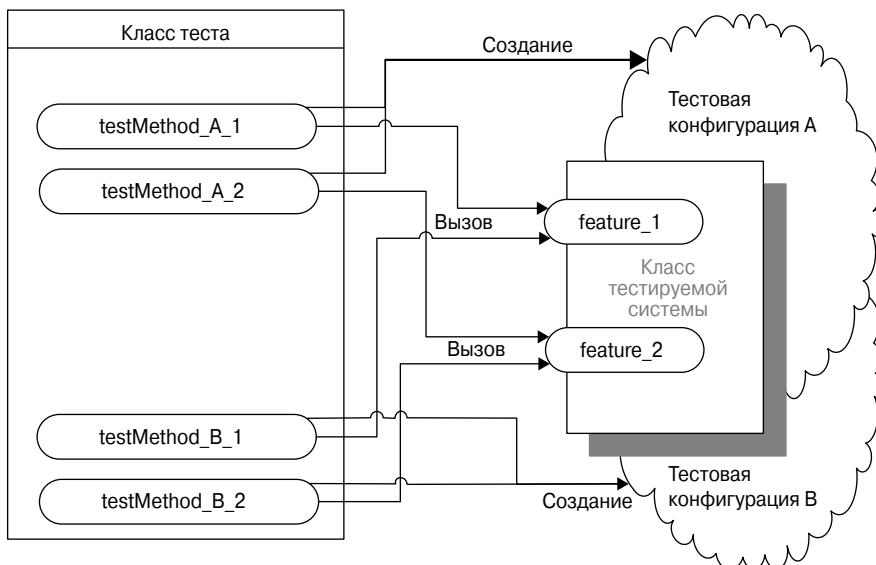
Более подробно атрибуты [RowTest] и [Row()] описываются в документации к инфраструктуре MbUnit. На сайте www.ddsteps.org приводится описание расширений DDSteps для инфраструктуры JUnit. Хотя имя расширения и указывает, что оно предназначено для создания управляемых данными тестов (Data-Driven Test), приведенные примеры содержат параметризованные тесты (Parameterized Test). Дополнительные аргументы в пользу применения табличных тестов (Tabular Test) можно найти в журнале Клинта Шанка по адресу:

<http://clintshank.javadevelopersjournal.com/tabulartests.htm>

Класс теста для каждого класса (Testcase Class per Class)

Как организовать тестовые методы (Test Method) в классы теста (Testcase Class)?

Все тестовые методы (Test Method) для одного класса тестируемой системы размещаются в одном классе теста (Testcase Class).



С ростом количества *тестовых методов* (Test Method, с. 378) приходится принимать решение об их размещении в *классах теста* (Testcase Class, с. 401). Выбор стратегии организации тестов значительно влияет на общее восприятие написанных тестов. Кроме того, такой выбор оказывает влияние на доступные стратегии настройки тестовой конфигурации.

Создание *класса теста для каждого класса* (Testcase Class per Class) является простым способом для начала организации тестов.

Как это работает

Отдельный *класс теста* (Testcase Class) создается для каждого класса, который должен проверяться тестами. Каждый *класс теста* (Testcase Class) выступает в роли хранилища всех *тестовых методов* (Test Method), проверяющих поведение тестируемого класса.

Когда это использовать

Использование организации *класса теста для каждого класса* (Testcase Class per Class) является хорошей отправной точкой при небольшом количестве *тестовых методов* (Test Method) или в начале создания тестов для проверяемой системы. С ростом количества тестов приходит понимание требований к тестовой конфигурации, что может вызвать

необходимость разделения *класса теста* (Testcase Class) на несколько классов. Результатом разделения может стать *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639), если существует небольшое количество часто используемых отправных точек, или *класс теста для каждой функции* (Testcase Class per Feature, с. 633), если проверяется несколько отдельных функций. Как любит говорить Кент Бек: “Пусть код диктует дальнейшие действия!”

Замечания по реализации

Для выбора имени *класса теста* (Testcase Class) не нужно прилагать больших усилий: просто возьмите имя класса тестируемой системы и добавьте в его начало или в конец слово “Test”. Имена методов должны отражать как минимум начальное состояние (тестовую конфигурацию) и вызываемый метод (функцию), а также описывать передаваемые в тестируемую систему параметры. После учета этих требований “места” на ожидаемый результат в имени метода не остается, поэтому читателю придется заглядывать в тело *тестового метода* (Test Method) и определять ожидаемый результат самостоятельно.

При использовании такой организации тестов создание тестовой конфигурации является основной проблемой во время реализации. Рано или поздно у разных *тестовых методов* (Test Method) возникнут конфликтующие требования к конфигурации, что затруднит использование *неявной настройки* (Implicit Setup, с. 449) и вынудит применять *встроенную настройку* (In-line Setup, с. 434) или *делегированную настройку* (Delegated Setup, с. 437). Еще одной проблемой является обеспечение видимости природы тестовой конфигурации в пределах *тестовых методов* (Test Method) без создания *непонятных тестов* (Obscure Test, с. 230). Кроме случаев очень простой *встроенной настройки* (In-line Setup), *делегированная настройка* (Delegated Setup) на основе *методов создания* (Creation Method, с. 441) позволяет получить вполне понятные читателю тесты.

Пример: класс теста для каждого класса (Testcase Class per Class)

В данном примере организация *класса теста для каждого класса* (Testcase Class per Class) используется для структурирования *тестовых методов* (Test Method) для класса Flight, имеющего три состояния (Unscheduled, Scheduled и AwaitingApproval) и четыре метода (schedule, requestApproval, deSchedule и approve). Поскольку класс имеет состояния, необходим как минимум один тест для каждого метода в каждом состоянии.

```
public class FlightStateTest extends TestCase {
    public void testRequestApproval_FromScheduledState() throws Exception {
        Flight flight = FlightTestHelper.getAnonymousFlightInScheduledState();
        try {
            flight.requestApproval();
            fail("not allowed in scheduled state");
        } catch (InvalidRequestException e) {
            assertEquals("InvalidRequestException.getRequest()", "requestApproval",
                        e.getRequest());
            assertTrue("isScheduled()", flight.isScheduled());
        }
    }
    public void testRequestApproval_FromUnscheduledState()
        throws Exception {
```

```
Flight flight = FlightTestHelper.  
    getAnonymousFlightInUnscheduledState();  
flight.requestApproval();  
assertTrue("isAwaitingApproval()",  
         flight.isAwaitingApproval());  
}  
public void testRequestApproval_FromAwaitingApprovalState()  
throws Exception {  
    Flight flight = FlightTestHelper.  
        getAnonymousFlightInAwaitingApprovalState();  
    try {  
        flight.requestApproval();  
        fail("not allowed in awaitingApproval state");  
    } catch (InvalidRequestException e) {  
        assertEquals("InvalidRequestException.getRequest()",  
                    "requestApproval",  
                    e.getRequest());  
        assertTrue("isAwaitingApproval()",  
                  flight.isAwaitingApproval());  
    }  
}  
public void testSchedule_FromUnscheduledState()  
throws Exception {  
    Flight flight = FlightTestHelper.  
        getAnonymousFlightInUnscheduledState();  
    flight.schedule();  
    assertTrue("isScheduled()", flight.isScheduled());  
}  
public void testSchedule_FromScheduledState()  
throws Exception {  
    Flight flight = FlightTestHelper.  
        getAnonymousFlightInScheduledState();  
    try {  
        flight.schedule();  
        fail("not allowed in scheduled state");  
    } catch (InvalidRequestException e) {  
        assertEquals("InvalidRequestException.getRequest()",  
                    "schedule",  
                    e.getRequest());  
        assertTrue("isScheduled()", flight.isScheduled());  
    }  
}  
public void testSchedule_FromAwaitingApprovalState()  
throws Exception {  
    Flight flight = FlightTestHelper.  
        getAnonymousFlightInAwaitingApprovalState();  
    try {  
        flight.schedule();  
        fail("not allowed in scheduled state");  
    } catch (InvalidRequestException e) {  
        assertEquals("InvalidRequestException.getRequest()",  
                    "schedule",  
                    e.getRequest());  
        assertTrue("isAwaitingApproval()",  
                  flight.isAwaitingApproval());  
    }  
}  
public void testDeschedule_FromScheduledState()
```

```

throws Exception {
Flight flight = FlightTestHelper.
    getAnonymousFlightInScheduledState();
flight.deschedule();
assertTrue("isUnscheduled()", flight.isUnscheduled());
}
public void testDeschedule_FromUnscheduledState()
throws Exception {
Flight flight = FlightTestHelper.
    getAnonymousFlightInUnscheduledState();
try {
    flight.deschedule();
    fail("not allowed in unscheduled state");
} catch (InvalidRequestException e) {
    assertEquals("InvalidRequestException.getRequest()",
        "deschedule",
        e.getRequest());
    assertTrue("isUnscheduled()", flight.isUnscheduled());
}
}
public void testDeschedule_FromAwaitingApprovalState()
throws Exception {
Flight flight = FlightTestHelper.
    getAnonymousFlightInAwaitingApprovalState();
try {
    flight.deschedule();
    fail("not allowed in awaitingApproval state");
} catch (InvalidRequestException e) {
    assertEquals("InvalidRequestException.getRequest()",
        "deschedule",
        e.getRequest());
    assertTrue("isAwaitingApproval()", flight.isAwaitingApproval());
}
}
public void testApprove_FromScheduledState()
throws Exception {
Flight flight = FlightTestHelper.
    getAnonymousFlightInScheduledState();
try {
    flight.approve("Fred");
    fail("not allowed in scheduled state");
} catch (InvalidRequestException e) {
    assertEquals("InvalidRequestException.getRequest()",
        "approve",
        e.getRequest());
    assertTrue("isScheduled()", flight.isScheduled());
}
}
public void testApprove_FromUnscheduledState()
throws Exception {
Flight flight = FlightTestHelper.
    getAnonymousFlightInUnscheduledState();
try {
    flight.approve("Fred");
    fail("not allowed in unscheduled state");
} catch (InvalidRequestException e) {
    assertEquals("InvalidRequestException.getRequest()",

```

```
        "approve",
        e.getRequest());
    assertTrue("isUnscheduled()", flight.isUnscheduled());
}
}

public void testApprove_FromAwaitingApprovalState()
throws Exception {
    Flight flight = FlightTestHelper.
        getAnonymousFlightInAwaitingApprovalState();
    flight.approve("Fred");
    assertTrue("isScheduled()", flight.isScheduled());
}

public void testApprove_NullArgument() throws Exception {
    Flight flight = FlightTestHelper.
        getAnonymousFlightInAwaitingApprovalState();
    try {
        flight.approve(null);
        fail("Failed to catch no approver");
    } catch (InvalidArgumentException e) {
        assertEquals("e.getArgumentName()", "approverName", e.getArgumentName());
        assertNull("e.getArgumentValue()", e.getArgumentValue());
        assertTrue("isAwaitingApproval()", flight.isAwaitingApproval());
    }
}

public void testApprove_InvalidApprover() throws Exception {
    Flight flight = FlightTestHelper.
        getAnonymousFlightInAwaitingApprovalState();
    try {
        flight.approve("John");
        fail("Failed to validate approver");
    } catch (InvalidArgumentException e) {
        assertEquals("e.getArgumentName()", "approverName",
                    e.getArgumentName());
        assertEquals("e.getArgumentValue()", "John",
                    e.getArgumentValue());
        assertTrue("isAwaitingApproval()", flight.isAwaitingApproval());
    }
}
```

В этом примере для создания *новой тестовой конфигурации* (Fresh Fixture, с. 344) используется *делегированная настройка* (Delegated Setup). За счет этого достигается декларативный стиль создания конфигурации. Но все равно класс оказался достаточно большим и отслеживание *тестовых методов* (Test Method) требует заметных усилий. Даже анализ с помощью средств среды разработки не проясняет ситуацию. Можно видеть проверяемые условия теста, но, не заглянув внутрь *тестового метода* (Test Method), нельзя определить ожидаемый результат (рис. 24.1).

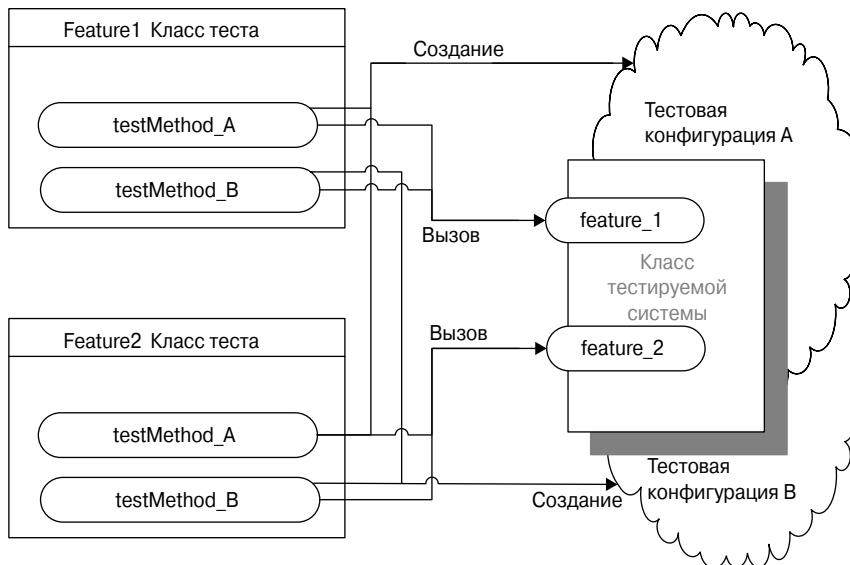


Рис. 24.1. Пример организации класса теста для каждого класса (Testcase Class per Class) в окне Package Explorer среды разработки Eclipse. Обратите внимание, что начальное состояние и проверяемое событие отражены в именах тестовых методов (Test Method)

Класс теста для каждой функции (Testcase Class per Feature)

Как организовать тестовые методы (Test Method) в классы теста (Testcase Class)?

Тестовые методы (Test Method) группируются в классы теста (Testcase Class) в соответствии с проверяемой функцией тестируемой системы.



С ростом количества *тестовых методов* (Test Method, с. 378) приходится принимать решение об их размещении в *классах теста* (Testcase Class, с. 401). Выбор стратегии организации тестов значительно влияет на общее восприятие написанных тестов. Кроме того, такой выбор оказывает влияние на доступные стратегии настройки тестовой конфигурации.

Использование организации *класса теста для каждой функции* (Testcase Class per Feature) обеспечивает системный способ разбиения *класса теста* (Testcase Class) на несколько классов меньшего размера без модификации *тестовых методов* (Test Method).

Как это работает

Тестовые методы (Test Method) группируются в *классы теста* (Testcase Class) в соответствии с проверяемой функцией. Такая организационная схема позволяет получить *классы теста* (Testcase Class) меньшего размера и с одного взгляда оценить все тестовые условия для конкретной функции проверяемого класса.

Когда это использовать

Организация *класса теста для каждой функции* (Testcase Class per Feature) может использоваться при большом количестве *тестовых методов* (Test Method), когда спецификация каждой функции тестируемой системы должна быть более очевидной. К сожалению, при этом от-

дельные *тестовые методы* (Test Method) не становится проще ни с точки зрения программирования, ни с точки зрения их понимания. В этом отношении определенными преимуществами обладает только организация *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639). Использование данной организации не оправдано, если каждая функция тестируемой системы требует одного-двух тестов. В таком случае можно продолжать использование организации *класс теста для каждого класса* (Testcase Class per Class, с. 627).

Обратите внимание, что большое количество функций, возложенных на один класс, является “запахом”, показывающим превышение данным классом осмысленного объема ответственности. Обычно организация *класс теста для каждого класса* (Testcase Class per Class) используется при создании приемочных тестов для методов фасада служб.

Вариант: класс теста для каждого метода (Testcase Class per Method)

Если класс содержит методы, принимающие большое количество разных параметров, для таких методов будет создано много тестов. Все эти *тестовые методы* (Test Method) можно сгруппировать в один класс теста, что соответствует организации *класс теста для каждого метода* (Testcase Class per Method).

Вариант: класс теста для каждой функции (Testcase Class per Feature)

Хотя обычно под “функцией” класса подразумевается единственная операция, в этом качестве может выступать и набор методов, работающих с одной и той же переменной экземпляра. Например, методы *set* и *get* объекта Java Bean можно рассматривать как одну (тривиальную) “функцию” класса. Точно так **объект доступа к данным** (data access object) может предоставлять методы для чтения и записи объектов. Тестировать эти методы изолированно достаточно сложно, поэтому запись и чтение объектов одного типа можно рассматривать как функцию.

Вариант: класс теста для каждого пользовательского сценария (Testcase Class per User Story)

При инкрементной разработке (как в методологии экстремального программирования) новый *тестовый метод* (Test Method) для каждого сценария можно добавлять в отдельный *класс теста* (Testcase Class). Такая практика позволяет избежать конфликтов при включении кода в хранилище, когда разные люди работают над разными историями, затрагивающими один и тот же класс тестируемой системы. Шаблон *класс теста для каждого пользовательского сценария* (Testcase Class per User Story) может превратиться (а может и не превратиться) в *класс теста для каждой функции* (Testcase Class per Feature) или *класс теста для каждого метода* (Testcase Class per Method) в зависимости от разбиения пользовательских сценариев.

Замечания по реализации

Поскольку *класс теста* (Testcase Class) описывает требования к отдельной функции тестируемой системы, имеет смысл присвоить ему имя в соответствии с проверяемой функцией. Точно так имя каждого *тестового метода* (Test Method) может быть основано на проверяемом тестовом условии. Такая структура имен позволяет с одного взгляда оценить все тестовые условия, проверяемые *тестовыми методами* (Test Method) *класса теста* (Testcase Class).

Одним из следствий применения такой организации является большее количество *классов теста* (Testcase Class) для одного класса продукта. Поскольку необходимо запускать все тесты этого класса, *классы теста* (Testcase Class) должны размещаться в одной папке с подпапками, пакете или пространстве имен. При использовании механизма *перечисления тестов* (Test Enumeration, с. 425) для агрегации всех *классов теста* (Testcase Class) в единственный набор тестов можно воспользоваться *набором всех тестов* (AllTests Suite).

Мотивирующий пример

В приведенном ниже примере шаблон *класс теста для каждого класса* (Testcase Class per Class) используется для структурирования *тестовых методов* (Test Method) для класса Flight, имеющего три состояния (Unscheduled, Scheduled и AwaitingApproval) и четыре метода (schedule, requestApproval, deSchedule и approve). Поскольку класс имеет состояния, необходим как минимум один тест для каждого метода в каждом состоянии. (Для экономии места многие методы указаны без тела. Полный листинг приведен в предыдущем разделе.)

```
public class FlightStateTest extends TestCase {
    public void testRequestApproval_FromScheduledState()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInScheduledState();
        try {
            flight.requestApproval();
            fail("not allowed in scheduled state");
        } catch (InvalidRequestException e) {
            assertEquals("InvalidRequestException.getRequest()",
                "requestApproval",
                e.getRequest());
            assertTrue("isScheduled()", flight.isScheduled());
        }
    }
    public void testRequestApproval_FromUnscheduledState()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInUnscheduledState();
        flight.requestApproval();
        assertTrue("isAwaitingApproval()", flight.isAwaitingApproval());
    }
    public void testRequestApproval_FromAwaitingApprovalState()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInAwaitingApprovalState();
        try {
            flight.requestApproval();
            fail("not allowed in awaitingApproval state");
        } catch (InvalidRequestException e) {
            assertEquals("InvalidRequestException.getRequest()",
                "requestApproval",
                e.getRequest());
            assertTrue("isAwaitingApproval()", flight.isAwaitingApproval());
        }
    }
}
```

```

public void testSchedule_FromUnscheduledState()
    throws Exception {
    Flight flight = FlightTestHelper.
        getAnonymousFlightInUnscheduledState();
    flight.schedule();
    assertTrue("isScheduled()", flight.isScheduled());
}
public void testSchedule_FromScheduledState()
    throws Exception {
    // Тела остальных методов пропущены
    // для экономии места
}
}

```

В этом примере используется *делегированная настройка* (Delegated Setup, с. 437) *новой тестовой конфигурации* (Fresh Fixture, с. 344), что позволяет добиться более декларативного стиля создания тестовой конфигурации. Несмотря на это класс имеет значительные размеры и следить за *тестовыми методами* (Test Method) становится достаточно сложно. Поскольку *тестовые методы* (Test Method) в этом *классе теста* (Testcase Class) обращаются к четырем разным методам, это хороший пример теста, который можно преобразовать в организацию *класса теста для каждой функции* (Testcase Class per Feature).

Замечания по рефакторингу

Для сокращения размера каждого *класса теста* (Testcase Class) и получения описательных имен *тестовых методов* (Test Method) их можно организовать в виде шаблона *класс теста для каждой функции* (Testcase Class per Feature). Сначала необходимо определить количество создаваемых классов и список *тестовых методов* (Test Method), которые будут принадлежать каждому из них. Если одни *классы теста* (Testcase Class) окажутся меньше, чем другие, проще начинать работу именно с них. После этого с помощью рефакторинга *выделение класса* (Extract Class) [Ref] создается один из новых *классов теста* (Testcase Class) и ему назначается имя, указывающее на проверяемую функцию. После этого с помощью рефакторинга *перемещение метода* (Move Method) (или простого “вырезания и вставки”) каждый *тестовый метод* (Test Method) переносится из старого класса в новый вместе со всеми используемыми переменными экземпляра.

Этот процесс повторяется до тех пор, пока в исходном классе не останется проверка одной функции. После этого исходный класс переименовывается в соответствии с проверяемой функцией. На этом этапе каждый *класс теста* (Testcase Class) должен без проблем компилироваться и запускаться, но это еще не конец. Для получения всех преимуществ от организации *класса теста для каждой функции* (Testcase Class per Feature) осталось выполнить последнюю операцию. К каждому *тестовому методу* (Test Method) необходимо применить рефакторинг *переименование метода* (Rename Method) [Ref], чтобы имя метода указывало на проверяемое условие. В процессе рефакторинга из имен можно удалить указание на проверяемую функцию, так как эта информация отражена в имени *класса теста* (Testcase Class). В результате в имени метода остается “место” для описания начального состояния (тестовой конфигурации) и ожидаемого результата. Если для каждой функции существует несколько тестов с различными аргументами методов, стоит найти способ включения и этих элементов тестового условия в имя метода.

Еще один способ такого рефакторинга предполагает копирование исходного *класса теста* (Testcase Class) с последующим переименованием (как показано выше). После

этого необходимо просто удалить *тестовые методы* (Test Method), не соответствующие проверяемой функции в конкретном классе. Постарайтесь не удалить все копии *тестовых методов* (Test Method), а также не оставить одинаковые методы в нескольких *классах теста* (Testcase Class). Для обхода таких потенциальных ошибок можно создать одну копию *класса теста* (Testcase Class) для каждой функции с последующим переименованием, как показано выше. Затем необходимо просто удалить *тестовые методы* (Test Method), не соответствующие проверяемой функции в конкретном классе. После завершения создания классов исходный *класс теста* (Testcase Class) можно удалить.

Пример: класс теста для каждой функции (Testcase Class per Feature)

В этом примере предыдущий набор тестов преобразован в соответствии с организацией *класс теста для каждой функции* (Testcase Class per Feature).

```
public class TestScheduleFlight extends TestCase {
    public void testUnscheduled_shouldEndUpInScheduled()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInUnscheduledState();
        flight.schedule();
        assertTrue("isScheduled()", flight.isScheduled());
    }
    public void testScheduledState_shouldThrowInvalidRequestEx()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInScheduledState();
        try {
            flight.schedule();
            fail("not allowed in scheduled state");
        } catch (InvalidRequestException e) {
            assertEquals("InvalidRequestException.getRequest()",
                "schedule",
                e.getRequest());
            assertTrue("isScheduled()", flight.isScheduled());
        }
    }
    public void testAwaitingApproval_shouldThrowInvalidRequestEx()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInAwaitingApprovalState();
        try {
            flight.schedule();
            fail("not allowed in scheduled state");
        } catch (InvalidRequestException e) {
            assertEquals("InvalidRequestException.getRequest()",
                "schedule",
                e.getRequest());
            assertTrue("isAwaitingApproval()",
                flight.isAwaitingApproval());
        }
    }
}
```

В тестовых методах изменились только имена. Поскольку в именах описываются предварительные условия (тестовая конфигурация), проверяемая функция и ожидаемый результат, список тестов в среде разработки позволяет оценить общую картину (рис. 24.2) и использовать *тесты как документацию* (Tests as Documentation, с. 79).

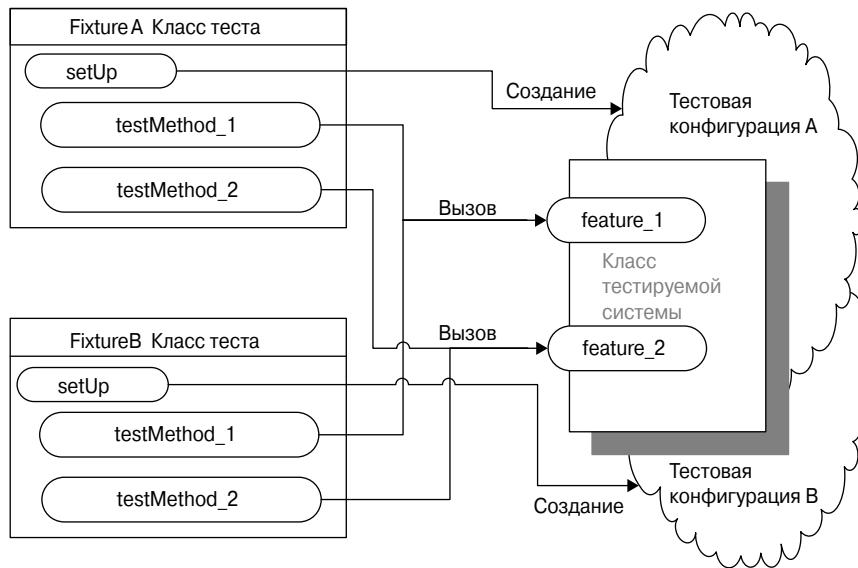


Рис. 24.2. Пример организации класса теста для каждой функции (Testcase Class per Feature) в окне Package Explorer среды разработки Eclipse. Обратите внимание, что в именах методов отсутствует имя проверяемого метода, что оставляет место для описания начального и ожидаемого конечного состояния

Класс теста для каждой тестовой конфигурации (Testcase Class per Fixture)

Как организовать тестовые методы (Test Method) в классы теста (Testcase Class)?

**Тестовые методы (Test Method) группируются в классы теста (Testcase Class)
в соответствии с используемой тестовой конфигурацией.**



С ростом количества *тестовых методов* (Test Method, с. 378) приходится принимать решение об их размещении в *классах теста* (Testcase Class, с. 401). Выбор стратегии организации тестов значительно влияет на общее восприятие написанных тестов. Кроме того, такой выбор оказывает влияние на доступные стратегии настройки тестовой конфигурации.

Организация *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture) позволяет воспользоваться механизмом *неявной настройки* (Implicit Setup, с. 449), который предоставляется *инфраструктурой автоматизации тестов* (Test Automation Framework, с. 332).

Как это работает

Тестовые методы (Test Method) группируются в *классы теста* (Testcase Class) в зависимости от используемой в качестве отправной точки тестовой конфигурации. Такая организация позволяет воспользоваться *неявной настройкой* (Implicit Setup), при которой вся логика создания тестовой конфигурации переносится в метод `setUp` класса теста. В результате основное внимание в каждом тестовом методе уделяется вызову тестируемой системы и проверке полученного результата.

Когда это использовать

Организацию *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture) можно использовать при наличии группы *тестовых методов* (Test Method), применяющих одну и ту же тестовую конфигурацию, когда необходимо сделать тестовые методы как можно более простыми. Если каждому тесту требуется уникальная конфигурация, использовать организацию *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture) не имеет смысла, так как в результате получится большое количество классов, содержащих один тест. В таком случае лучше использовать организацию *класса теста для каждой функции* (Testcase Class per Feature, с. 633) или *класса теста для каждого класса* (Testcase Class per Class, с. 627).

Одним из преимуществ такой организации тестов является простота, с которой можно оценить полноту тестирования операций для каждого начального состояния. В каждом *классе теста* (Testcase Class) должно присутствовать одинаковое количество методов, что легко заметить в окне “обозревателя методов” среды разработки. Данная особенность делает организацию *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture) особенно полезной для поиска *отсутствующих модульных тестов* (Missing Unit Test) задолго до передачи продукта в промышленную эксплуатацию.

Организация *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture) является ключевым элементом *разработки на основе поведения* (behavior-driven development). В результате получаются очень короткие тестовые методы, содержащие по одному утверждению на тест. В комбинации с соглашением об именовании методов, учитывающим ожидаемый результат теста, такой шаблон позволяет использовать *тесты как документацию* (Tests as Documentation, с. 79).

Замечания по реализации

Поскольку тестовая конфигурация создается в методе `setUp`, который вызывается *инфраструктурой автоматизации тестов* (Test Automation Framework, с. 332), для хранения ссылки на тестовую конфигурацию должна использоваться переменная экземпляра. Переменную класса использовать нельзя, так как это приведет к появлению *общей тестовой конфигурации* (Shared Fixture, с. 350) и *неустойчивых тестов* (Erratic Test, с. 267), которые часто сопровождают такой тип конфигурации. (Во врезке “Всегда есть исключения” на с. 411 перечислены реализации xUnit, в которых при использовании переменных экземпляра не гарантируется независимость тестов.)

Класс теста (Testcase Class) соответствует одной тестовой конфигурации, поэтому имеет смысл отразить данный факт в его имени. Точно так каждому методу можно присвоить имя в соответствии с вызываемым методом тестируемой системы, характеристиками передаваемых аргументов и ожидаемым результатом вызова метода.

Одним из побочных эффектов использования организации *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture) является большое количество *классов теста* (Testcase Class). Может потребоваться способ группирования разных классов тестов, проверяющих один класс тестируемой системы. Один из способов сделать это — создать вложенную папку, пакет или пространство имен для хранения этих классов теста. При использовании механизма *перечисления тестов* (Test Enumeration, с. 425) может потребоваться создание *набора всех тестов* (AllTests Suite), в котором будут собраны все классы теста, организованные в виде *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture).

Еще одним побочным эффектом является распределение тестов одной функции системы по нескольким *классам теста* (Testcase Class). Такое распределение может рассматриваться положительно, если функции тесно связаны друг с другом, так как это отражает их взаимосвязь. С другой стороны, если функции не связаны, такое распределение может ввести в заблуждение. В таком случае может потребоваться рефакторинг в направлении *класса теста для каждой функции* (Testcase Class per Feature) или применение рефакторинга *выделение класса* (Extract Class) по отношению к тестируемой системе, если такой симптом указывает на слишком большую степень ответственности проверяемого класса.

Мотивирующий пример

В следующем примере используется организация *тестовых методов* (Test Method) в виде *класса теста для каждого класса* (Testcase Class per Class). Проверяется класс Flight, имеющий три состояния (Unscheduled, Scheduled и AwaitingApproval) и четыре метода (schedule, requestApproval, deSchedule и approve). Поскольку класс имеет состояние, требуется как минимум один тестовый метод для каждого метода в каждом состоянии. (Для экономии места тела методов не приводятся. Полный листинг приведен в разделе, посвященном организации *класса теста для каждого класса* (Testcase Class per Class).)

```
public class FlightStateTest extends TestCase {
    public void testRequestApproval_FromScheduledState()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInScheduledState();
        try {
            flight.requestApproval();
            fail("not allowed in scheduled state");
        } catch (InvalidRequestException e) {
            assertEquals("InvalidRequestException.getRequest()",
                "requestApproval",
                e.getRequest());
            assertTrue("isScheduled()", flight.isScheduled());
        }
    }
    public void testRequestApproval_FromUnscheduledState()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInUnscheduledState();
        flight.requestApproval();
        assertTrue("isAwaitingApproval()",
            flight.isAwaitingApproval());
    }
    public void testRequestApproval_FromAwaitingApprovalState()
        throws Exception {
        Flight flight = FlightTestHelper.
            getAnonymousFlightInAwaitingApprovalState();
        try {
            flight.requestApproval();
            fail("not allowed in awaitingApproval state");
        } catch (InvalidRequestException e) {
            assertEquals("InvalidRequestException.getRequest()",

```

```

        "requestApproval",
        e.getRequest());
    assertTrue("isAwaitingApproval()", 
        flight.isAwaitingApproval());
}
}

public void testSchedule_FromUnscheduledState()
throws Exception {
    Flight flight = FlightTestHelper.
        getAnonymousFlightInUnscheduledState();
    flight.schedule();
    assertTrue("isScheduled()", flight.isScheduled());
}

public void testSchedule_FromScheduledState()
throws Exception {
    // Для экономии места тела некоторых методов
    // не приводятся
}
}
}

```

В этом примере используется *делегированная настройка* (Delegated Setup, с. 437) *новой тестовой конфигурации* (Fresh Fixture, с. 344), что позволяет добиться более декларативного стиля создания тестовой конфигурации. Несмотря на это класс имеет значительные размеры и следить за *тестовыми методами* (Test Method) становится достаточно сложно. Поскольку *тестовые методы* (Test Method) этого *класса теста* (Testcase Class) используют три разные тестовые конфигурации (по одной для каждого состояния, в котором может находиться объект перелета), это хороший пример рефакторинга в направлении *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture).

Замечания по рефакторингу

Чтобы сократить *дублирование тестового кода* (Test Code Duplication, с. 254) в логике настройки тестовой конфигурации и упростить понимание *тестовых методов* (Test Method), можно выполнить преобразование класса для организации в виде *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture). Сначала необходимо определить количество создаваемых классов и распределение существующих *тестовых методов* (Test Method) по новым классам. Если некоторые *классы теста* (Testcase Class) оказываются меньше остальных, лучше начинать именно с них. После этого с помощью рефакторинга *выделение класса* (Extract Class) создается один из *классов теста* (Testcase Class), в имени которого описывается используемая тестовая конфигурация. Затем по отношению к принадлежащим этому классу *тестовым методам* (Test Method) выполняется рефакторинг *перемещение метода* (Move method) с перемещением всех используемых переменных экземпляра.

Процесс повторяется до тех пор, пока в исходном классе не останется одна тестовая конфигурация. После этого оставшийся класс переименовывается в соответствии с используемой конфигурацией. На этом этапе каждый *класс теста* (Testcase Class) должен компилироваться и запускаться, но это еще не все. Для получения всех преимуществ от использования шаблона *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture) осталось выполнить две операции. Сначала из всех *тестовых методов* (Test Method) необходимо выделить всю общую логику создания тестовой

конфигурации в метод `setUp`, что позволит перейти к механизму *неявной настройки* (Implicit Setup). Такая настройка вполне допустима, поскольку каждый *тестовый метод* (Test Method) в пределах класса использует одну тестовую конфигурацию. После этого необходимо воспользоваться рефакторингом *переименование метода* (Rename Method), чтобы имя *тестового метода* (Test Method) отражало проверяемую функциональность. Из имени метода можно убрать упоминание о начальном состоянии, так как эта информация отражена в имени класса. В результате в имени метода остается достаточно “места” для описания действия (вызываемого метода и природы его аргументов), а также ожидаемого результата.

Существует и другой способ получения такой организации. Для получения этого шаблона можно создать по одной (соответствующим образом переименованной) копии *класса теста* (Testcase Class) для каждой тестовой конфигурации, удалив ненужные *тестовые методы* (Test Method) из каждой копии, а после завершения работы над копиями удалив исходный *класс теста* (Testcase Class).

Пример: класс теста для каждой тестовой конфигурации (Testcase Class per Fixture)

В данном примере показанный ранее набор тестов преобразован для использования организации *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture). (Для экономии места показан только один *класс теста* (Testcase Class). Остальные выглядят практически так же.)

```
public class TestScheduledFlight extends TestCase {
    Flight scheduledFlight;
    protected void setUp() throws Exception {
        super.setUp();
        scheduledFlight = createScheduledFlight();
    }
    Flight createScheduledFlight() throws InvalidRequestException{
        Flight newFlight = new Flight();
        newFlight.schedule();
        return newFlight;
    }
    public void testDeschedule_shouldEndUpInUnscheduleState()
        throws Exception {
        scheduledFlight.deschedule();
        assertTrue("isUnsched", scheduledFlight.isUnscheduled());
    }
    public void testRequestApproval_shouldThrowInvalidRequestEx() {
        try {
            scheduledFlight.requestApproval();
            fail("not allowed in scheduled state");
        } catch (InvalidRequestException e) {
            assertEquals("InvalidRequestException.getRequest ()",
                        "requestApproval", e.getRequest());
            assertTrue("isScheduled()", scheduledFlight.isScheduled());
        }
    }
    public void testSchedule_shouldThrowInvalidRequestEx() {
        try {
```

```
        scheduledFlight.schedule();
        fail("not allowed in scheduled state");
    } catch (InvalidRequestException e) {
        assertEquals("InvalidRequestException.getRequest()", "schedule",
                    e.getRequest());
        assertTrue("isScheduled()", scheduledFlight.isScheduled());
    }
}

public void testApprove_shouldThrowInvalidRequestEx()
throws Exception {
try {
    scheduledFlight.approve("Fred");
    fail("not allowed in scheduled state");
} catch (InvalidRequestException e) {
    assertEquals("InvalidRequestException.getRequest()", "approve",
                e.getRequest());
    assertTrue("isScheduled()", scheduledFlight.isScheduled());
}
}
```

Обратите внимание, насколько упростился каждый *тестовый метод* (Test Method)! Поскольку каждому методу присвоено описательное имя, можно использовать *тесты как документацию* (Tests as Documentation). Рассматривая список методов в “обозревателе” среди разработки, можно легко определить начальное состояние (тестовую конфигурацию), действие (вызываемый метод) и ожидаемый результат (возвращаемый или выраженный в виде состояния после завершения теста), даже не открывая тело метода (рис. 24.3).

“Общий план” показывает, что, если объект Flight находится в состоянии awaitingApproval, тесты проверяют только аргументы метода approve. Обнаружив такую закономерность, можно попытаться определить, является ли это недостатком тестов или частью спецификации (например, результат вызова метода approve для некоторых состояний объекта Flight не определен).



Рис. 24.3. Тесты, организованные в виде класса теста для каждой тестовой конфигурации (Testcase Class per Fixture) и показанные в окне Package Explorer среды разработки Eclipse.
Обратите внимание, что в имени метода не указывается начальное состояние, что оставляет достаточно места для перечисления вызываемого метода и ожидаемого конечного состояния

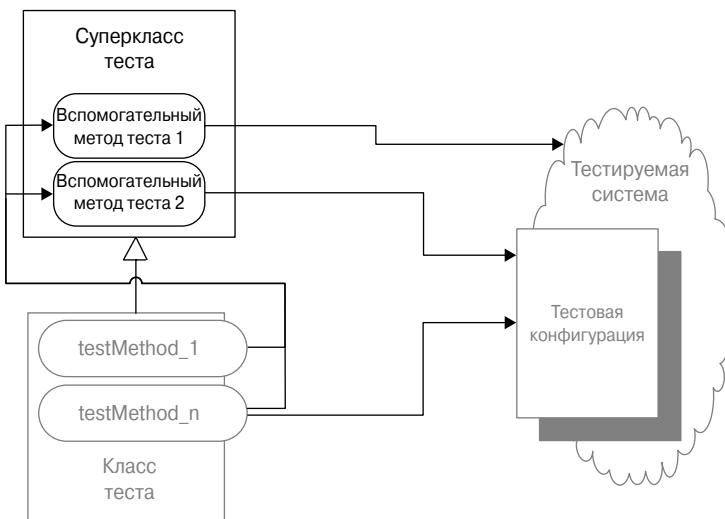
Суперкласс теста (Testcase Superclass)

Также известен как:

Абстрактный тест (Abstract Testcase), Абстрактная тестовая конфигурация (Abstract Test Fixture), Базовый класс теста (Testcase Baseclass)

Где разместить код вспомогательных методов теста (Test Utility Method)?

Повторно используемая тестовая логика наследуется от абстрактного суперкласса теста (Testcase Superclass).



При создании тестов рано или поздно возникает необходимость повторения одной и той же логики в большом количестве тестов. Изначально может возникнуть соблазн “скопировать и модифицировать” код для новых тестов. Но рано или поздно для хранения этой логики будут использоваться *вспомогательные методы теста* (Test Utility Method, с. 610). Но где их разместить?

Одним из вариантов размещения *вспомогательных методов теста* (Test Utility Method) является *суперкласс теста* (Testcase Superclass).

Как это работает

Для хранения повторно используемых *вспомогательных методов теста* (Test Utility Method) объявляется абстрактный суперкласс, который будет наследоваться несколькими *классами теста* (Testcase Class, с. 401). При этом методы должны быть видимы для подклассов (например, в языке Java для этого используется квалификатор `protected`). После этого полученный абстрактный класс используется в качестве суперкласса (базового класса) для всех тестов, в которых его логика должна использоваться повторно. В результате логика будет доступна через вызовы методов, как будто они определены в самом *классе теста* (Testcase Class).

Когда это использовать

Суперкласс теста (Testcase Superclass) может использоваться в ситуациях, когда необходимо повторно применять *вспомогательные методы теста* (Test Utility Method) в нескольких классах теста (Testcase Class) и можно найти или определить суперкласс теста, от которого будут наследоваться все тесты, требующие доступа к этой логике.

Данный шаблон предполагает, что в используемом языке программирования поддерживается наследование, наследование не используется для другой (конфликтующей) цели и *вспомогательный метод теста* (Test Utility Method) не требует доступа к конкретным типам, которые не видны из *суперкласса теста* (Testcase Superclass).

Выбор между *суперклассом теста* (Testcase Superclass) и *вспомогательным классом теста* (Test Helper, с. 651) сводится к проблеме видимости типов. Клиентский класс должен видеть *вспомогательный метод теста* (Test Utility Method), а *вспомогательный метод теста* (Test Utility Method) должен видеть типы и классы, от которых он зависит. Если список зависимостей невелик или все зависимости видны из одного места, *вспомогательный метод теста* (Test Utility Method) можно разместить в общем *суперклассе теста* (Testcase Superclass), который определяется для всей компании или для конкретного проекта. Если *вспомогательный метод теста* (Test Utility Method) зависит от типов и/или классов, которые не видны из одной точки, к которой могут получить доступ все клиенты, его придется разместить во *вспомогательном классе теста* (Test Helper) соответствующего пакета тестов или подсистемы.

Вариант: вспомогательный миксин (Test Helper Mixin)

В языках с поддержкой миксинов *вспомогательный миксин* (Test Helper Mixin) обеспечивает преимущества обоих вариантов. Как и при использовании *вспомогательного класса теста* (Test Helper), можно выбирать включаемые *вспомогательные миксины* (Test Helper Mixin), не ограничиваясь иерархией наследования с единственным родителем. Как и при использовании *вспомогательного объекта* (Test Helper Object), можно хранить состояние теста в миксине, не создавая экземпляр и не делегируя эту задачу в отдельный объект. Как и при использовании *суперкласса теста* (Testcase Superclass), доступ ко всем атрибутам и методам можно получить через указатель `self`.

Замечания по реализации

В реализациях xUnit, в которых *класс теста* (Testcase Class) должен наследовать *суперкласс теста* (Testcase Superclass), предоставленный инфраструктурой автоматизации тестов (Test Automation Framework, с. 332), этот класс определяется как суперкласс нашего *суперкласса теста* (Testcase Superclass). В реализациях, использующих атрибуты методов или аннотации для идентификации *тестовых методов* (Test Method, с. 378), можно наследовать любой подходящий класс.

Методы *суперкласса теста* (Testcase Superclass) могут быть реализованы в виде методов класса или в виде методов экземпляра. Для *вспомогательных методов теста* (Test Utility Method) без состояния вполне допустимо использовать методы класса. Если методы класса использовать по какой-либо причине не удается, их можно реализовать в виде методов экземпляра. В любом случае из-за наследования методов доступ к ним осуществляется так же, как к методам, определенным в самом *классе теста* (Testcase Class). Если

язык поддерживает управление видимостью методов, необходимо обеспечить достаточную видимость для подклассов (например, воспользоваться квалификатором `protected` в языке Java).

Мотивирующий пример

В следующем примере показан *вспомогательный метод теста* (Test Utility Method), определенный в *классе теста* (Testcase Class).

```
public class TestRefactoringExample extends TestCase {
    public void testAddOneLineItem_quantity1() {
        Invoice inv = createAnonInvoice();
        LineItem expItem = new LineItem(inv, product, QUANTITY);
        // Вызов
        inv.addItemQuantity(product, QUANTITY);
        // Проверка
        assertInvoiceContainsOnlyThisLineItem(inv, expItem);
    }
    void assertInvoiceContainsOnlyThisLineItem(
            Invoice inv,
            LineItem expItem) {
        List lineItems = inv.getLineItems();
        assertEquals("number of items", lineItems.size(), 1);
        LineItem actual = (LineItem)lineItems.get(0);
        assertLineItemsEqual("", expItem, actual);
    }
}
```

Данный *вспомогательный метод теста* (Test Utility Method) недоступен для повторного использования за пределами данного класса или в его подклассах.

Замечания по рефакторингу

Вспомогательный метод теста (Test Utility Method) можно сделать более пригодным для повторного использования, перенеся его в *суперкласс теста* (Testcase Superclass) с помощью рефакторинга *подъем метода* (Pull Up Method). Поскольку метод наследуется в *классе теста* (Testcase Class), к нему можно получить доступ как к локально определенному. Если *вспомогательный метод теста* (Test Utility Method) получает доступ к переменным экземпляра, необходимо воспользоваться рефакторингом *подъем поля* (Pull Up Field) для переноса этих переменных в область видимости *вспомогательного метода теста* (Test Utility Method). Если *тестовым методам* (Test Method) также необходим доступ к этим полям, в языках с ограничением видимости поле придется делать видимым для подклассов (например, с помощью квалификаторов `default` и `protected` в языке Java).

Пример: суперкласс теста (Testcase Superclass)

Поскольку метод наследуется в *классе теста* (Testcase Class), к нему возможен доступ как к определенному локально. Таким образом, его использование не меняется.

```

public class TestRefactoringExample extends OurTestCase {
    public void testAddItemQuantity_severalQuantity_v12() {
        // Настройка тестовой конфигурации
        Customer cust = createACustomer(new BigDecimal("30"));
        Product prod = createAProduct(new BigDecimal("19.99"));
        Invoice invoice = createInvoice(cust);
        // Вызов тестируемой системы
        invoice.addItemQuantity(prod, 5);
        // Проверка результата
        LineItem expected = new LineItem(invoice, prod, 5,
            new BigDecimal("30"), new BigDecimal("69.96"));
        assertContainsExactlyOneLineItem(invoice, expected);
    }
}

```

Единственным отличием являются класс, в котором определен метод, и параметры его видимости.

```

public class OurTestCase extends TestCase {
    void assertContainsExactlyOneLineItem(Invoice invoice,
                                         LineItem expected) {
        List lineItems = invoice.getLineItems();
        assertEquals("number of items", lineItems.size(), 1);
        LineItem actItem = (LineItem)lineItems.get(0);
        assertLineItemsEqual("",expected, actItem);
    }
}

```

Пример: вспомогательный миксин (Test Helper Mixin)

Ниже показаны тесты, написанные на языке Ruby с использованием инфраструктуры Test::Unit.

```

def test_extref
    # настройка
    sourceXml = "<extref id='abc' />"
    expectedHtml = "<a href='abc.html'>abc</a>"
    mockFile = MockFile.new
    @handler = setupHandler(sourceXml, mockFile)
    # Вызов
    @handler.printBodyContents
    # Проверка
    assert_equals_html(expectedHtml, mockFile.output,
                      "extref: html output")
end

def testTestterm_normal
    sourceXml = "<testterm id='abc' />"
    expectedHtml = "<a href='abc.html'>abc</a>"
    mockFile = MockFile.new
    @handler = setupHandler(sourceXml, mockFile)
    @handler.printBodyContents
    assert_equals_html(expectedHtml, mockFile.output,
                      "testterm: html output")
end
def testTestterm_plural

```

```

sourceXml = "<testterms id='abc' />"
expectedHtml = "<a href='abc.html'>abcs</a>"
mockFile = MockFile.new
@handler = setupHandler(sourceXml, mockFile)
@handler.printBodyContents
assert_equals_html(expectedHtml, mockFile.output,
                  "testterms: html output")
end

```

В этих тестах присутствует достаточно большая доля *дублирования тестового кода* (Test Code Duplication, с. 254). Для решения этой проблемы можно воспользоваться рефакторингом *выделение метода* (Extract Method) для создания *вспомогательного метода теста* (Test Utility Method). После этого для облегчения повторного использования вспомогательный метод переносится во *вспомогательный миксин* (Test Helper Mixin). Поскольку функциональность миксина считается частью *класса теста* (Testcase Class), к ней можно получать доступ так же, как к определенным локально. Таким образом, использование метода не изменилось.

```

class CrossrefHandlerTest < Test::Unit::TestCase
  include HandlerTest
  def test_extref
    sourceXml = "<extref id='abc' />"
    expectedHtml = "<a href='abc.html'>abc</a>"
    generateAndVerifyHtml(sourceXml, expectedHtml, "<extref>")
  end

```

Единственным отличием являются точка определения метода и его видимость. В частности, в языке Ruby определение миксинов должно находиться в блоке `module`, а не в блоке `class`.

```

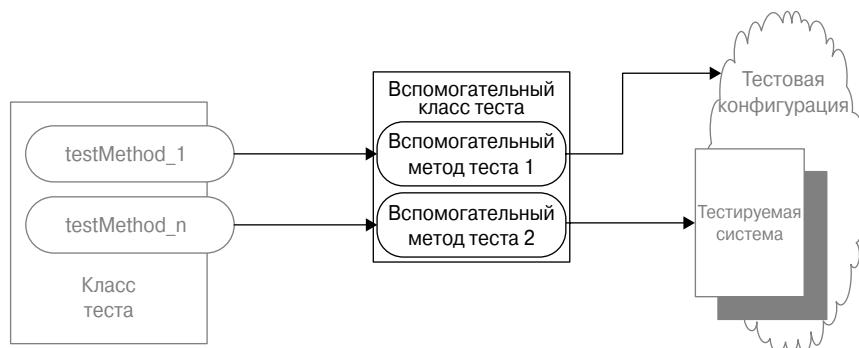
module HandlerTest
  def generateAndVerifyHtml(sourceXml, expectedHtml,
                            message, &block)
    mockFile = MockFile.new
    sourceXml.delete!("\t")
    @handler = setupHandler(sourceXml, mockFile )
    block.call unless block == nil
    @handler.printBodyContents
    actual_html = mockFile.output
    assert_equal_html(expectedHtml,
                      actual_html,
                      message + "html output")
    actual_html
  end

```

Вспомогательный класс теста (Test Helper)

Где разместить код вспомогательных методов теста (Test Utility Method)?

Следует определить вспомогательный класс, в котором расположены все вспомогательные методы теста (Test Utility Method), необходимые для повторного использования в нескольких тестах.



При создании тестов рано или поздно возникает необходимость повторения одной и той же логики в большом количестве тестов. Изначально может возникнуть соблазн “скопировать и модифицировать” код для новых тестов. Но рано или поздно для хранения этой логики будут использоваться *вспомогательные методы теста* (Test Utility Method, с. 610). Но где их разместить?

Одним из вариантов размещения повторно используемой логики теста является *вспомогательный класс теста* (Test Helper).

Как это работает

Определяется отдельный класс, в котором хранятся *вспомогательные методы теста* (Test Utility Method), доступные нескольким *классам теста* (Testcase Class, с. 401). В каждом teste, которому необходимо использовать вспомогательную логику, доступ осуществляется через статические вызовы методов или через специально созданный экземпляр.

Когда это использовать

Вспомогательный класс теста (Test Helper) можно применять для совместного использования логики или переменных в нескольких *классах теста* (Testcase Class), когда невозможно (или принято решение о невозможности) найти или определить *суперкласс теста* (Testcase Superclass, с. 646), который будут наследовать все тесты, требующие доступа к данной логике. Такое решение оправдано в нескольких ситуациях: возможно, используемый язык программирования не поддерживает наследование (например, Visual Basic 5 или 6), возможно, наследование уже используется по конфликтующей причине или *вспомогательный метод теста* (Test Utility Method) должен получать доступ к конкретным типам, которые не видны в *суперклассе теста* (Testcase Superclass).

Выбор между *вспомогательным классом теста* (Test Helper) и *суперклассом теста* (Testcase Superclass) сводится к видимости типов. Клиентский класс должен видеть *вспомогательный метод теста* (Test Utility Method), а *вспомогательный метод теста* (Test Utility Method) должен видеть типы и классы, от которых он зависит. Если список зависимостей невелик или все зависимости видны из одного места, *вспомогательный метод теста* (Test Utility Method) можно разместить в общем *суперклассе теста* (Testcase Superclass), который определяется для всей компании или для конкретного проекта. Если *вспомогательный метод теста* (Test Utility Method) зависит от типов и/или классов, которые не видны из одной точки, к которой могут получить доступ все клиенты, его придется разместить во *вспомогательном классе теста* (Test Helper) соответствующего пакета тестов или подсистемы. В больших системах с большим количеством групп объектов предметной области распространена практика создания одного *вспомогательного класса теста* (Test Helper) для каждой группы (пакета) связанных объектов предметной области.

Вариант: реестр тестовых конфигураций (Test Fixture Registry)

Реестр представляет собой хорошо известный объект, к которому можно получить доступ из любой точки программы. Реестр можно использовать для хранения и получения объектов для различных фрагментов программы или тестов. (Объекты реестра часто путают с единственными объектами классов (Singleton), которые также широкоизвестны, но имеют только один экземпляр. Объектов реестра может быть сколько угодно.) *Реестр тестовых конфигураций* (Test Fixture Registry) позволяет тестам одновременно получать доступ к одной и той же тестовой конфигурации. В зависимости от реализации *вспомогательного класса теста* (Test Helper) может потребоваться предоставление разных экземпляров *реестра тестовых конфигураций* (Test Fixture Registry) каждой программе запуска тестов (Test Runner, с. 405), что позволит предотвратить “войну” запуска тестов (Test Run War). Распространенным примером *реестра тестовых конфигураций* (Test Fixture Registry) является “песочница” с базой данных (Database Sandbox, с. 658).

Обычно *реестр тестовых конфигураций* (Test Fixture Registry) используется совместно с *декоратором настройки* (Setup Decorator, с. 471) или механизмом “ленивой” настройки (Lazy Setup, с. 460). При использовании *настройки тестовой конфигурации набора* (Suite Fixture Setup, с. 465) эта возможность не требуется, так как только тесты в пределах одного *класса теста* (Testcase Class) совместно используют тестовую конфигурацию. В таком случае для хранения тестовой конфигурации достаточно использовать переменную класса.

Вариант: инкубатор объектов (Object Mother)

Шаблон *инкубатор объектов* (Object Mother) представляет собой агрегат из нескольких шаблонов, каждый из которых делает небольшой, но заметный вклад в упрощение управления тестовой конфигурацией. *Инкубатор объектов* (Object Mother) состоит из одного или нескольких *вспомогательных классов теста* (Test Helper), представляющих *методы создания* (Creation Method, с. 441) и *подключаемые методы* (Attachment Method), которые в дальнейшем используются тестами для создания готовых объектов тестовой конфигурации. *Инкубаторы объектов* (Object Mother) часто предоставляют несколько *методов создания* (Creation Method), позволяющих генерировать экземпляры одного и того же класса, методы которого предоставляют возможность получать тестируемые объекты в разных начальных состояниях (*Достигающий указанного состояния метод*, Named State Reaching Method). Кроме того, *инкубатор объектов*

(Object Mother) может автоматически удалять создаваемые объекты, что является примером *автоматической очистки* (Automated Teardown).

Поскольку однозначного определения шаблона *инкубатор объектов* (Object Mother) не существует, при описании его возможностей желательно рассматривать конкретные составляющие шаблоны (например, *автоматическую очистку*, Automated Teardown).

Замечания по реализации

Методы *вспомогательного класса теста* (Test Helper) можно реализовать в виде методов класса или методов экземпляра. Выбор зависит от степени изоляции тестов.

Вариант: вспомогательный класс теста (Test Helper Class)

Если ни один из *вспомогательных методов теста* (Test Utility Method) не имеет внутреннего состояния, простейшим решением является реализация функциональности вспомогательного класса в виде методов класса, а использующие эти методы тесты будут получать к ним доступ с помощью соответствующей конструкции языка для доступа к элементам класса, например, `ClassName.methodName`. Если необходимо хранить ссылки на объекты тестовой конфигурации, их также можно разместить в переменных класса, но при этом необходимо соблюдать осторожность, чтобы не создать *общую тестовую конфигурацию* (Shared Fixture, с. 350) (если, конечно, такая конфигурация не создается намеренно). В таком случае результатом будет *реестр тестовых конфигураций* (Test Fixture Registry).

Вариант: вспомогательный объект (Test Helper Object)

Если по какой-либо причине использовать методы класса нельзя, можно попытаться воспользоваться методами экземпляра. В таком случае клиентский тест должен будет создать экземпляр *вспомогательного класса теста* (Test Helper) и сохранить его в переменной экземпляра. Эта переменная будет использоваться для доступа к методам полученного объекта. Такой шаблон вполне допустим, если *вспомогательный класс теста* (Test Helper) содержит ссылки на тестовую конфигурацию или объекты тестируемой системы и необходимо защититься от появления *общей тестовой конфигурации* (Shared Fixture). Этот шаблон полезен в ситуации, когда *вспомогательный класс теста* (Test Helper) содержит ожидания для *набора подставных объектов* (Mock Object, с. 558), так как шаблон позволяет проверить правильность чередования вызовов между *подставными объектами* (Mock Object).

Мотивирующий пример

В следующем примере показан *вспомогательный метод теста* (Test Utility Method), расположенный внутри *класса теста* (Testcase Class).

```
public class TestUtilityExample extends TestCase {
    public void testAddOneLineItem_quantity1() {
        Invoice inv = createAnonInvoice();
        LineItem expItem = new LineItem(inv, product, QUANTITY);
        // Вызов
        inv.addItemQuantity(product, QUANTITY);
```

```

    // Проверка
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    assertEquals("", expItem, actual);
}
}

```

Вспомогательный метод теста (Test Utility Method) невозможно использовать повторно за пределами конкретного класса.

Замечания по рефакторингу

Этот *вспомогательный метод теста* (Test Utility Method) можно сделать более доступным для повторного использования, переместив его во *вспомогательный класс теста* (Test Helper). Подобное преобразование обычно сводится к применению рефакторинга *перемещение метода* (Move Method). Одна возможная проблема связана с использованием переменных экземпляра для передачи аргументов или возврата данных из *вспомогательного метода теста* (Test Utility Method). Такие “глобальные данные” необходимо преобразовать в явные аргументы и возвращаемые значения еще до применения рефакторинга *перемещение метода* (Move Method).

Пример: вспомогательный класс теста (Test Helper) с методами класса

В модифицированной версии предыдущего теста *вспомогательный метод теста* (Test Utility Method) преобразован в метод *вспомогательного класса теста* (Test Helper). Это значит, что доступ к методу возможен и без создания экземпляра.

```

public class TestUtilityExample extends TestCase {
    public void testAddOneLineItem_quantity1_staticHelper() {
        Invoice inv = createAnonInvoice();
        LineItem expItem = new LineItem(inv, product, QUANTITY);
        // вызов
        inv.addItemQuantity(product, QUANTITY);
        // проверка
        TestHelper.assertContainsExactlyOneLineItem(inv, expItem);
    }
}

```

Пример: вспомогательный класс теста (Test Helper) с методами экземпляра

В этом примере *вспомогательный метод теста* (Test Utility Method) перенесен во *вспомогательный класс теста* (Test Helper) в виде метода экземпляра. Обратите внимание, что доступ к методу осуществляется через ссылку на объект (переменную, в которой хранится экземпляр *вспомогательного класса теста*, Test Helper).

```
public class TestUtilityExample extends TestCase {  
    public void testAddOneLineItem_quantity1_instanceHelper() {  
        Invoice inv = createAnonInvoice();  
        LineItem expItem = new LineItem(inv, product, QUANTITY);  
        // Вызов  
        inv.addItemQuantity(product, QUANTITY);  
        // Проверка  
        TestHelper helper = new TestHelper();  
        helper.assertInvContainsExactlyOneLineItem(inv, expItem);  
    }  
}
```


Глава 25

Шаблоны баз данных

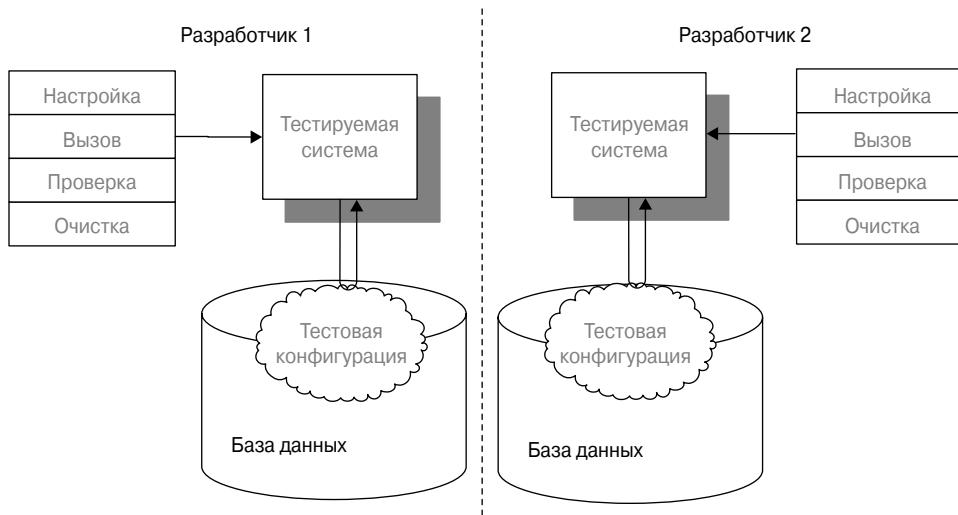
Шаблоны в этой главе:

“Песочница” с базой данных (Database Sandbox)	658
Тест хранимой процедуры (Stored Procedure Test).....	662
Очистка усечением таблиц (Table Truncation Teardown)	668
Очистка откатом транзакции (Transaction Rollback Teardown)	675

“Песочница” с базой данных (Database Sandbox)

Как разрабатывать и тестировать программное обеспечение, зависящее от базы данных?

Каждому разработчику и тестеру предоставляется отдельная база данных.



Во многих приложениях база данных используется для хранения постоянного состояния приложения. Некоторые тесты такого приложения обязательно потребуют доступа к базе данных. К сожалению, база данных является основной причиной появления *нестабильных тестов* (Erratic Test, с. 267), так как данные сохраняются между запусками тестов. Тесты необходимо предохранять от взаимодействия, чтобы предотвратить пересечение тестовых конфигураций. Это особенно сложно, когда среда разработки содержит только одну тестовую базу данных и все тесты всех разработчиков работают с одной и той же базой данных.

“Песочница” с базой данных (Database Sandbox) является одним из способов защиты тестов от взаимодействия через случайный доступ к одним и тем же записям.

Как это работает

Каждому пользователю для работы предоставается отдельная, внутренне непротиворечивая “песочница”. В ней содержится пользовательская копия всего кода, а также, что еще важнее, собственная копия базы данных. При такой организации рабочей среды каждый пользователь может как угодно модифицировать базы данных для запуска приложения с тестами, не задумываясь о взаимодействии своих тестов с тестами других пользователей.

Когда это использовать

“Песочница” с базой данных (Database Sandbox) должна использоваться при создании или модификации приложения, большая часть функциональности которого зависит от базы данных. Важность такой организации работы значительно повышается, если применяется *общая тестовая конфигурация* (Shared Fixture, с. 350). Используя “песочницу” с базой данных (Database Sandbox), можно избежать начала “войн” запуска тестов (Test Run War) между разными пользователями базы данных. В зависимости от выбранной реализации разные пользователи могут получить право модифицировать структуру базы данных. “Песочница” с базой данных (Database Sandbox) не предотвращает появление *не-повторяемых тестов* (Unrepeatable Test) или *взаимодействующих тестов* (Interacting Tests), так как защищает друг от друга разных пользователей (и их тесты); тесты одного пользователя все еще могут получать общий доступ к тестовой конфигурации.

Замечания по реализации

Приложение должно поддерживать настройку, чтобы замена тестовой базы данных не требовала модификации кода. Обычно для этого информация о конфигурации базы данных читается из файла, модифицированная копия которого находится в рабочей среде каждого пользователя.

“Песочницу” с базой данных (Database Sandbox) можно реализовать несколькими способами. Основной выбор происходит между предоставлением каждому пользователю отдельного экземпляра базы данных и эмуляцией отдельного экземпляра. Обычно предоставление отдельной базы данных является предпочтительным вариантом. Но такая схема не всегда оказывается возможной, особенно если система лицензирования делает ее недопустимой с финансовой точки зрения.

Вариант: выделенная “песочница” с базой данных (Dedicated Database Sandbox)

Каждому разработчику или тестовому пользователю предоставляется отдельный экземпляр базы данных. Обычно для этого в тестовой среде каждого пользователя устанавливается легкая реализация системы управления базами данных. В качестве примеров такой технологии можно привести MySQL и Personal Oracle. Экземпляр базы данных может храниться на пользовательском компьютере, совместно используемом сервере или на выделенном “виртуальном сервере”, который работает на совместно используемой аппаратной платформе сервера.

Выделенная “песочница” с базой данных (Dedicated Database Sandbox) является предпочтительным решением, так как предоставляет большую гибкость и позволяет разработчикам модифицировать схему базы данных, загружать собственные тестовые данные и выполнять другие операции.

Вариант: схема базы данных на программу запуска тестов (DB-Schema per Test-Runner)

При такой реализации каждый разработчик и пользователь работает с отдельным экземпляром базы данных, который обеспечивается встроенными механизмами поддержки нескольких схем.

Одним из преимуществ такого шаблона относительно выделенной “песочницы” является возможность совместного использования *немодифицируемой общей тестовой кон-*

фигурации (Immutable Shared Fixture), определенной в общей схеме, с одновременным размещением модифицируемой тестовой конфигурации в отдельной схеме каждого пользователя. Обратите внимание, что данный шаблон не допускает модификации структуры базы данных отдельными пользователями (или допускает, но не в той степени, в которой это возможно при использовании *выделенной “песочницы” с базой данных*, Dedicated Database Sandbox). Кроме того, и разработчикам, и пользователям приходится применять одну и ту же структуру базы данных. Это может вызвать процедурные проблемы при развертывании новой структуры базы данных в процессе разработки.

Вариант: схема разбиения базы данных (Database Partitioning Scheme)

Каждому разработчику и пользователю предоставляется отдельный набор данных в пределах единственной *“песочницы” с базой данных* (Database Sandbox). Каждый пользователь может модифицировать любые данные, кроме данных, принадлежащих другим пользователям.

При таком подходе сокращаются накладные расходы на администрирование базы данных, но затраты на управление данными возрастают. Поскольку разработчикам запрещено модифицировать схему базы данных, *схема разбиения базы данных* (Database Partitioning Scheme) не подходит для эволюционной разработки. С другой стороны, она позволяет защититься от появления *взаимодействующих тестов* (Interacting Tests) между разными запусками одной *программы запуска тестов* (Test Runner), т.е. каждый тест получает уникальный ключ, например *CustomerNumber*, который используется для всех данных. В результате другие тесты будут использовать разные данные. Такой шаблон можно комбинировать с другими вариантами *“песочницы” с базой данных* (Database Sandbox), чтобы предотвратить появление *взаимодействующих тестов* (Interacting Tests) при использовании *общей тестовой конфигурации* (Shared Fixture). Обратите внимание, что этот шаблон не защищает от *неповторяемых тестов* (Unrepeatable Test), если не используется *отдельное сгенерированное значение* (Distinct Generated Value).

Мотивирующий пример

В следующем teste в качестве аргументов конструктора объекта *Product* используются *точные значения* (Literal Value), которые сохраняются в базе данных, используемой совместно несколькими разработчиками. Имя в объекте *Product* должно быть уникальным.

```
public void testProductPrice_HCV() {
    // Настстройка
    Product product =
        new Product( 88,                      // ID
                     "Widget",                // Имя
                     new BigDecimal("19.99")); // Цена
    // Вызов тестируемой системы
    // ...
}
```

К сожалению, при запуске этого теста с совместно используемой базой данных можно начать *“войну” запуска тестов* (Test Run War) независимо от тщательности очистки объекта *Product* после завершения работы каждого теста. Это связано с созданием одного и того же объекта *Product* одним и тем же тестом при одновременном запуске с помощью другого экземпляра *программы запуска тестов* (Test Runner).

Замечания по рефакторингу

При создании выделенной “песочницы” с базой данных (Dedicated Database Sandbox) для каждого разработчика и пользователя модификация кода тестов не нужна. Таким образом, для независимой работы при запуске из разных программ запуска тестов (Test Runner, с. 405) от тестов ничего не требуется. Но тестируемой системе необходимо минимальное изменение, позволяющее подключаться к разным экземплярам базы данных в зависимости от конфигурационных данных. Реализация этого изменения зависит от применяемой технологии и в данной книге не рассматривается.

Чтобы преобразовать тест для использования схемы разбиения базы данных (Database Partitioning Scheme), достаточно заменить точные значения (Literal Value) вызовами соответствующих методов `getUnique`, которым в качестве параметра передается идентификатор текущего экземпляра программы запуска тестов (Test Runner).

Пример: схема разбиения базы данных (Database Partitioning Scheme)

Ниже показан тот же тест, но применяющий схему разбиения базы данных (Database Partitioning Scheme) для использования разных наборов продуктов в каждом teste. В метод `getUniqueString` в качестве параметра передается строка, основанная на MAC-адресе сетевого интерфейса компьютера.

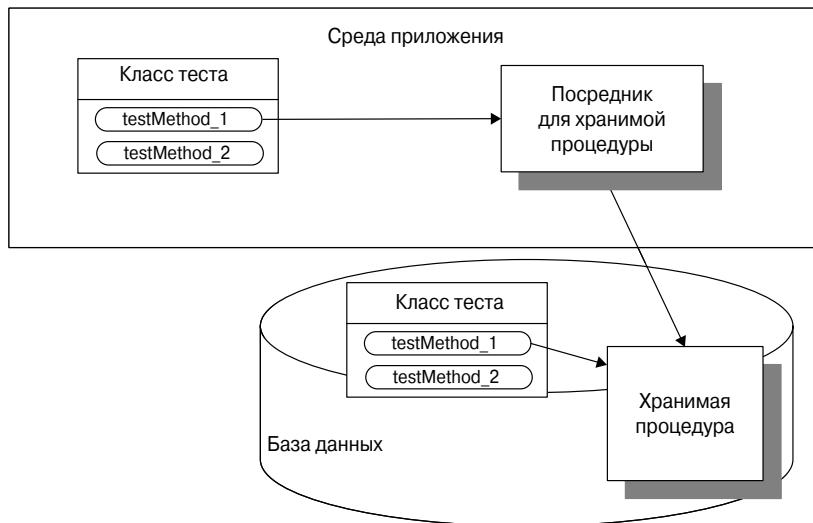
```
public void testProductPrice_DPS() {
    // Настройка
    Product product =
        new Product(getUniqueInt(), // ID
                    getUniqueString(getMacAddress()), // Имя
                    new BigDecimal("19.99")); // Цена
    // Вызов тестируемой системы
    // ...
}
static int counter = 0;
int getUniqueInt() {
    counter++;
    return counter;
}
BigDecimal getUniqueBigDecimal() {
    return new BigDecimal(getUniqueInt());
}
String getUniqueString(String baseName) {
    return baseName.concat(String.valueOf( getUniqueInt() ));
}
```

Теперь этот тест можно запускать на разных компьютерах с помощью общего экземпляра базы данных и не бояться начала “войны” запуска тестов (Test Run War).

Тест хранимой процедуры (Stored Procedure Test)

Как независимо проверять логику при наличии хранимых процедур?

Для каждой хранимой процедуры создается полностью автоматизированный тест (Fully Automated Tests).



Во многих приложениях, в которых база данных используется для хранения постоянного состояния приложения, хранимые процедуры и триггеры используются для увеличения быстродействия и обработки данных при обновлении.

Тест хранимой процедуры (Stored Procedure Test) обеспечивает автоматизированное тестирование кода, хранящегося в базе данных.

Как это работает

Модульные тесты для хранимых процедур создаются независимо от клиентского программного обеспечения. Тесты могут пересекать или не пересекать уровни в зависимости от природы проверяемых хранимых процедур.

Когда это использовать

Тесты хранимых процедур (Stored Procedure Test) необходимо создавать для всей нетривиальной логики, реализованной в виде хранимых процедур. Такой шаблон позволяет проверять их правильность независимо от клиентского приложения. Этот подход еще более важен, если хранимые процедуры используются более чем одним приложением или создаются другой группой разработчиков. *Тесты хранимых процедур* (Stored Procedure Test) важны в ситуациях, когда нельзя обеспечить их адекватную проверку посредством обращения к интерфейсам приложения (что является *опосредованным тестированием* (Indirect Testing) — одной из форм *непонятного теста*, Obscure Test, с. 230). Кроме того,

с помощью *тестов хранимых процедур* (Stored Procedure Test) можно перечислить все условия, при которых будет вызываться хранимая процедура, и то, что должно происходить в таких условиях. Сам факт обдумывания этих условий, скорее всего, приведет к совершенствованию дизайна, а это достаточно распространенный результат разработки с первоочередным написанием тестов.

Замечания по реализации

Существует два фундаментально отличающихся способа реализации *тестов хранимых процедур* (Stored Procedure Test).

1. Тесты можно разрабатывать на том же языке программирования, на котором написаны сами хранимые процедуры, с последующим запуском внутри базы данных.
2. Тесты можно разрабатывать на языке программирования, на котором написано основное приложение, а доступ к хранимым процедурам получать через шаблон *удаленный прокси* (Remote Proxy).

Можно даже создавать тесты обоими способами сразу. Например, разработчики хранимых процедур могут создавать модульные тесты на языке программирования базы данных, а разработчики приложения могут готовить приемочные тесты на языке программирования приложения и запускать их в процессе компиляции приложения.

В любом случае необходимо решить, как тест будет настраивать конфигурацию (состояние базы данных “до”) и проверять ожидаемый результат (состояние базы данных “после” и все ожидаемые операции, например каскадное удаление). Тест может непосредственно взаимодействовать с базой данных для вставки и/или проверки данных (вариант *манipуляции через “черный ход”*, Back Door Manipulation, с. 359) либо использовать другую хранимую процедуру (тест через открытый интерфейс).

Вариант: хранящийся в базе данных тест хранимой процедуры (In-Database Stored Procedure Test)

Одним из преимуществ использования инфраструктуры xUnit для автоматизированного тестирования является возможность создания тестов на том же языке, на котором написан проверяемый код. Это значительно упрощает изучение способов тестирования и избавляет от необходимости осваивать новый язык программирования, его отладчик и т.д. Развив эту идею до логического завершения, можно предположить, что имеет смысл создавать тесты хранимых процедур на том же языке, на котором написаны процедуры. Очевидно, что эти тесты будут запускаться внутри базы данных. К сожалению, это требование может усложнить их запуск в процессе интеграции [SCM].

Использование такого варианта шаблона *тест хранимой процедуры* (Stored Procedure Test) оправдано при наличии такого опыта написания кода на языке хранимых процедур, который больше, чем опыт работы в среде разработки приложения, и нет требования запускать тесты из одного места. Например, такой подход может быть оправдан в группе разработки или обслуживания данных, создающей хранимые процедуры для других команд. Еще одним условием использования этого варианта является хранение процедур отдельно от логики приложения. В этом случае тесты хранимых процедур можно будет хранить в том же хранилище, где находится тестируемая система (т.е. хранимые процедуры).

Хранящиеся в базе данных тесты хранимой процедуры (In-Database Stored Procedure Test) допускают более тщательное тестирование (и разработку на основе тестов) хранимых процедур, так как возможен более глубокий доступ к реализации. Конечно, такое нарушение инкапсуляции может привести к появлению *зарегулированной программы* (Overspecified Software). Если в клиентском коде используется уровень доступа к данным, для этого уровня все равно придется создавать модульные тесты на языке программирования приложения. Это позволит проверить правильность обработки ошибок (например, невозможность подключения).

В некоторых базах данных поддерживается несколько языков программирования. В таком случае для тестов можно выбрать более удобный язык, а хранимые процедуры создавать на традиционном языке. Например, в базах данных Oracle поддерживаются языки PL/SQL и Java, поэтому тесты JUnit могут использоваться для тестирования хранимых процедур, написанных на языке программирования PL/SQL. Точно так в базах данных Microsoft SQL Server поддерживается язык C#, поэтому тесты на базе инфраструктуры NUnit, написанные на языке C#, можно использовать для проверки процедур, написанных на языке Transact-SQL.

Вариант: удаленный тест хранимой процедуры (Remoted Stored Procedure Test)

Удаленный тест хранимой процедуры (Remoted Stored Procedure Test) позволяет писать тесты на том же языке программирования, на котором написаны тесты клиентского приложения. Доступ к хранимой процедуре осуществляется через *удаленный прокси* (Remote Proxy), который скрывает механизм взаимодействия с этой процедурой. Такой прокси-объект может быть создан на основе шаблонов *фасад служб* (Service Facade) или Command (например, JDBC CallableStatement в языке Java).

Удаленные тесты хранимой процедуры (Remoted Stored Procedure Test) по сути являются тестами компонентов, так как рассматривают хранимую процедуру как “черный ящик”. Поскольку эти тесты работают за пределами базы данных и обычно не существует простых способов вставки или проверки данных, скорее всего, они работают через открытый интерфейс (вызывая другие хранимые процедуры для создания тестовой конфигурации, проверки результата и выполнения других необходимых операций). В некоторых реализациях xUnit предоставляются расширения, специально предназначенные для такого тестирования (например, DbUnit — для языка Java или NDBUnit — для языков платформы .NET).

Такое решение является более предпочтительным, если все тесты должны быть написаны на одном языке программирования. Шаблон *удаленный тест хранимой процедуры* (Remoted Stored Procedure Test) упрощает запуск всех тестов при каждом включении изменений в код приложения.

ТЕСТИРОВАНИЕ ХРАНИМЫХ ПРОЦЕДУР С ПОМОЩЬЮ JUNIT

В одном из первых проектов на основе методологии экстремального программирования автору пришлось разрабатывать приложение, которое должно было использовать хранимые процедуры, созданные другой группой разработчиков. Каждый раз при интеграции кода Java с кодом PL/SQL от другой группы разработчиков обнаруживались серьезные ошибки в фундаментальном поведении хранимых процедур. Автоматизированные тесты для кода приложения создавались на базе инфраструктуры JUnit. Несмотря на уверенность в ясности

контракта интерфейса и в повышении качества кода процедур в результате создания модульных тестов, так и не удалось заставить другую группу создавать модульные тесты для своих процедур. А пакета utPLSQL на тот момент не существовало даже в планах.

Было принято решение писать тесты для хранимых процедур на основе известной реализации xUnit: JUnit. Поскольку все равно приходилось писать код JDBC для доступа к хранимым процедурам, тесты для каждой хранимой процедуры были определены через самостоятельно написанные классы `PreparedStatement`. Тесты вызывали базовое поведение хранимых процедур и проверяли несколько очевидных ошибочных ситуаций. При получении каждой новой версии хранимых процедур тесты JUnit запускались еще до попыток вызвать процедуры из кода приложения. Даже не стоит говорить, что многие тесты завершались неудачно.

На общем собрании разработчики процедур показали тесты и результаты их запуска. Им было стыдно за результаты своей работы, но с правильностью тестов они согласились. После исправления обнаруженных дефектов была представлена новая версия. Она была лучше предыдущей, но все равно содержала ошибки. После этого произошло важное событие: другая группа разработчиков попросила копию готовых тестов с инструкциями по их запуску. Через некоторое время они уже самостоятельно создавали модульные тесты для языка PL/SQL с использованием инфраструктуры JUnit!

Эта возможность оказывается особенно полезной, если хранимые процедуры создаются и/или модифицируются разработчиками основного клиентского кода. Кроме того, *удаленные тесты хранимой процедуры* (Remoted Stored Procedure Test) должны использоваться в ситуациях, когда хранимые процедуры предоставляются другими разработчиками и нет уверенности в их способности создавать код без дефектов (например, из-за их отказа создавать тесты хранимых процедур внутри баз данных). В такой ситуации *удаленные тесты хранимой процедуры* (Remoted Stored Procedure Test) используются в качестве приемочных тестов для получаемого кода. Применение такого подхода в реальном проекте показано выше, во врезке “Тестирование хранимых процедур с помощью JUnit” на с. 664.

Одним из недостатков использования *удаленных тестов хранимой процедуры* (Remoted Stored Procedure Test) является замедление работы, так как тестам нужна база данных. Тесты хранимых процедур можно разместить в отдельном *наборе с подмножеством* (Subset Suite), чтобы они не запускались автоматически с остальными тестами. Это может значительно ускорить проверку и позволяет избежать появления *медленных тестов* (Slow Tests, с. 289).

Удаленные тесты хранимой процедуры (Remoted Stored Procedure Test) полезны при попытке перенести логику из приложения в базу данных (когда для логики приложения модульные тесты уже существуют). Используя *удаленные тесты хранимой процедуры* (Remoted Stored Procedure Test), можно избежать переписывания тестов на другом языке программирования для другой *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332), что в результате позволяет сэкономить время и деньги. К тому же этот шаблон позволяет избежать ошибок перевода при переписывании логики, а значит, можно быть уверенным в правильности ее результата.

Мотивирующий пример

Ниже показан пример хранимой процедуры, написанной на языке программирования PL/SQL.

```
CREATE OR REPLACE PROCEDURE calc_secs_between (
    date1 IN DATE,
    date2 IN DATE,
    secs OUT NUMBER
)
IS
BEGIN
    secs := (date2 - date1) * 24 * 60 * 60;
END;
/
```

Этот код взят из набора примеров, которые предоставляются вместе с инфраструктурой utPLSQL. В действительности тестировать этот код не имеет смысла из-за его простоты (а с другой стороны, почему бы нет?), но его достаточно для демонстрации подходов к тестированию.

Замечания по рефакторингу

В данном примере основное внимание уделяется не рефакторингу, а добавлению пропущенного теста. Попытаемся найти способ его добавления. Для демонстрации будут использоваться два основных варианта: *хранившийся в базе данных тест хранимой процедуры* (In-Database Stored Procedure Test) и *удаленный тест хранимой процедуры* (Remoted Stored Procedure Test).

Пример: хранившийся в базе данных тест хранимой процедуры (In-Database Stored Procedure Test)

В этом примере используется инфраструктура utPLSQL, которая является реализацией xUnit для языка программирования PL/SQL. Данная инфраструктура предназначена для автоматизации тестов, работающих внутри базы данных.

```
CREATE OR REPLACE PACKAGE BODY ut_calc_secs_between
IS
    PROCEDURE ut_setup
    IS
    BEGIN
        NULL;
    END;
    PROCEDURE ut_teardown
    IS
    BEGIN
        NULL;
    END;
    - Для каждой тестируемой программы...
    PROCEDURE ut_CALC_SECS_BETWEEN
    IS
        secs PLS_INTEGER;
    BEGIN
```

```

CALC_SECS_BETWEEN (
    DATE1 => SYSDATE
    '
    DATE2 => SYSDATE
    '
    SECS => secs
);
utAssert.eq (
    'Same dates',
    secs,
    0
);
END ut_CALC_SECS_BETWEEN;
END ut_calc_secs_between;
/

```

В данном тесте имеются знакомые структуры xUnit. Это один из нескольких тестов, которые обычно создаются для такой хранимой процедуры — по одному тесту для каждого возможного сценария. (Этот пример был взят из документации к инфраструктуре utPLSQL. Форматирование кода соответствует оригиналу.)

Пример: удаленный тест хранимой процедуры (Remoted Stored Procedure Test)

Для проверки этой процедуры на обычном языке программирования в среде разработки сначала необходимо найти или создать *удаленный прокси* (Remote Proxy) для выбранной среды модульного тестирования. После этого модульные тесты можно создавать как обычно.

В следующем примере инфраструктура JUnit используется для автоматизации тестов, работающих за пределами базы данных и проверяющих хранимую процедуру, которая написана на языке программирования PL/SQL.

```

public class StoredProcedureTest extends TestCase {
    public void testCalcSecsBetween_SameTime() {
        // Настройка
        TimeCalculatorProxy SUT = new TimeCalculatorProxy();
        Calendar cal = new GregorianCalendar();
        long now = cal.getTimeInMillis();
        // Вызов
        long timeDifference = SUT.calc_secs_between(now, now);
        // Проверка
        assertEquals(0, timeDifference);
    }
}

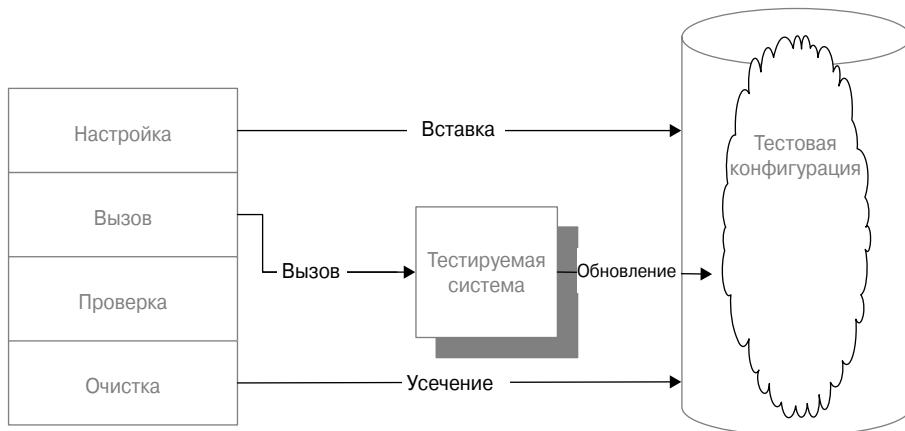
```

Исходный тест был преобразован в простой тест функции за счет скрытия JDBC/ODBC CallableStatement за *фасадом служб* (Service Facade). Рассматривая этот пример, сложно определить, что тестируется не метод Java. Скорее всего, для проверки неудачных подключений и других проблем потребуются дополнительные *тесты на ожидаемое исключение* (Expected Exception Test).

Очистка усечением таблиц (Table Truncation Teardown)

Как очистить тестовую конфигурацию, если она находится в реляционной базе данных?

Для очистки тестовой конфигурации обрезаются таблицы, модифицированные во время теста.



Повторяемость тестов в значительной мере зависит от отчистки тестовой конфигурации после завершения каждого теста. Оставшиеся объекты и записи в базе данных, а также открытые файлы и подключения могут привести как минимум к снижению быстродействия, а как максимум — к неудачному завершению тестов и аварийному завершению работы системы. Хотя некоторые из этих ресурсов можно освободить автоматически с помощью сборки мусора, остальные ресурсы остаются занятыми, если не освободить их явно.

Чтобы создать надежный код для правильной очистки, необходимы значительные усилия и много времени. При этом следует понимать, что может остаться после получения каждого возможного результата теста, и создать отдельный код для обработки каждого из вариантов. Подобная *сложная очистка* (Complex Teardown) приводит к появлению *условной логики теста* (Conditional Test Logic) и, что еще хуже, к появлению большого объема *нетестируемого кода тестов* (Untestable Test Code).

При проверке системы, использующей реляционную базу данных, можно воспользоваться возможностями базы данных, вызывая команду TRUNCATE для удаления всех данных из модифицированной таблицы.

Как это работает

Если постоянная тестовая конфигурация больше не нужна, для каждой таблицы конфигурации вызывается команда TRUNCATE. Она очень быстро удаляет из таблиц все данные без побочных эффектов (например, срабатываний триггеров).

Когда это использовать

К очистке усечением таблиц (Table Truncation Teardown) обычно прибегают при использовании *постоянной новой тестовой конфигурации* (Persistent Fresh Fixture) вместе с тестируемой системой, которая обращается к базе данных. Но этот подход редко выбирается первым. Обычно сначала применяется *очистка откатом транзакции* (Transaction Rollback Teardown, с. 675). В то же время очистка усечением таблиц (Table Truncation Teardown) является наилучшим выбором при использовании *общей тестовой конфигурации* (Shared Fixture, с. 350), поскольку такая конфигурация переживает тест по определению. С другой стороны, очистка откатом транзакции (Transaction Rollback Teardown) вместе с *общей тестовой конфигурацией* (Shared Fixture) потребует очень длительной транзакции. Хотя это и возможно, при длительных транзакциях могут возникать дополнительные проблемы.

Перед использованием очистки усечением таблиц (Table Truncation Teardown) необходимо удовлетворить несколько условий. Первым является допустимость удаления всех данных в указанных таблицах, вторым — предоставление отдельной “песочницы” с базой данных (Database Sandbox, с. 658) для каждой программы запуска тестов (Test Runner, с. 405). Очистка усечением таблиц (Table Truncation Teardown) не работает при использовании схемы разбиения базы данных (Database Partitioning Scheme) для изоляции пользователей или тестов друг от друга. Такой механизм идеально подходит для применения вместе со схемой базы данных на программу запуска тестов (DB-Schema per Test-Runner), особенно при реализации *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture) как отдельной совместно используемой схемы базы данных. В результате возникает возможность удаления всех данных *новой тестовой конфигурации* (Fresh Fixture) в собственной “песочнице” с базой данных (Database Sandbox) без изменения *немодифицируемой общей тестовой конфигурации* (Immutable Shared Fixture).

Если используемая база данных не поддерживает транзакции, наилучшим приближением является *автоматическая очистка* (Automated Teardown, с. 521), которая удаляет только те записи, которые были созданы тестом. Автоматическая очистка (Automated Teardown) не зависит от транзакций в базе данных, но требует дополнительных усилий со стороны разработчика. Кроме того, можно полностью отказаться от очистки, если воспользоваться *дельта-утверждениями* (Delta Assertion, с. 505).

Замечания по реализации

Кроме стандартного вопроса “Где разместить код очистки?”, при реализации очистки усечением таблиц (Table Truncation Teardown) необходимо ответить еще на ряд вопросов.

- Как фактически удалять данные, т.е. какие команды базы данных использовать?
- Как обрабатывать ограничения на внешние ключи и триггеры?
- Как обеспечить непротиворечивость при использовании объектно-реляционного отображения (object-relation mapping — ORM)?

В некоторых базах данных (например, в Oracle) команда TRUNCATE поддерживается непосредственно. В этом случае использование данной команды является очевидным решением. Иначе может потребоваться использовать команду DELETE * FROM tablename. Команды TRUNCATE или DELETE могут вызываться при использовании *встроенной очистки* (In-line Teardown, с. 527) из *тестового метода* (Test Method) или при использо-

вании *неявной очистки* (Implicit Teardown, с. 533) из метода `tearDown`. Некоторые разработчики предпочитают вызывать эту команду при “ленивой” очистке (Lazy Teardown), поскольку в такой ситуации пустота таблиц обеспечивается в начале теста, если работа тестов может быть нарушена лишними данными.

Ограничения внешних ключей могут превратиться в проблему при *очистке усечением таблиц* (Table Truncation Teardown), если база данных не поддерживает аналога команды `ON DELETE CASCADE` из СУБД Oracle. В Oracle присутствие в команде очистки таблицы строки `ON DELETE CASCADE` вынудит базу данных удалить и те строки в других таблицах, которые зависят от строк в очищаемой таблице. Если база данных не поддерживает каскадное удаление, очистку таблиц в определенном схемой порядке придется обеспечивать самостоятельно. Изменения в схеме могут нарушить этот порядок, что приведет к ошибкам при работе кода очистки. К счастью, такие ошибки легко обнаружить: ошибка в работе теста показывает, что код очистки требует модификации. Внести исправления достаточно просто: обычно необходимо только изменить порядок команд `TRUNCATE`. Конечно, можно найти способ динамически упорядочивать вызов команд `TRUNCATE` на основе зависимостей между таблицами, но обычно логику очистки достаточно скрыть внутри *вспомогательного метода теста* (Test Utility Method, с. 610).

Чтобы избежать побочных эффектов от триггеров и других сложностей в базах данных без поддержки команды `TRUNCATE`, на время выполнения теста можно отключить ограничения и/или триггеры. Этот шаг должен предприниматься, только если другие тесты вызывают тестируемую систему с включенными ограничениями и триггерами.

Если используется уровень объектно-реляционного отображения, например Toplink, (N)Hibernate или EJB 3.0, возможно, придется очистить кэш объектов, уже извлеченных из базы данных, чтобы поиск не приводил к использованию удаленных объектов. Например, в пакете NHibernate для этих целей предоставляется метод `ClearAllCaches` объекта `TransactionManager`.

Вариант: “ленивая” очистка (Lazy Teardown)

Этот способ очистки работает лишь с некоторыми вариантами *общей тестовой конфигурации* (Shared Fixture). В этом случае тестовая конфигурация должна поддерживать удаление в любое время. Таким образом, нельзя полагаться на “запоминание” удаляемых объектов; список удаляемых объектов должен быть очевидным в любой момент. *Очистка усечением таблиц* (Table Truncation Teardown) удовлетворяет этому условию, так как она всегда происходит одинаково. Команды очистки таблиц вызываются на этапе создания тестовой конфигурации перед созданием новой конфигурации.

Мотивирующий пример

В следующем teste для удаления всех созданных записей используется *гарантированная встроенная очистка* (Guaranteed In-line Teardown).

```
[Test]
public void TestGetFlightsByOrigin_NoInboundFlights()
{
    // Настройка тестовой конфигурации
    long OutboundAirport = CreateTestAirport("10F");
    long InboundAirport = CreateTestAirport("11F");
```

```

FlightDto ExpFlightDto = null;
try
{
    ExpFlightDto =
        CreateTestFlight(OutboundAirport, InboundAirport);
    // Вызов тестируемой системы
    IList FlightsAtDestination1 =
        Facade.GetFlightsByOriginAirport(InboundAirport);
    // Проверка результата
    Assert.AreEqual(0, FlightsAtDestination1.Count);
}
finally
{
    Facade.RemoveFlight(ExpFlightDto.FlightNumber);
    Facade.RemoveAirport(OutboundAirport);
    Facade.RemoveAirport(InboundAirport);
}
}
}

```

Этот код сложно писать и сложно исправлять! (Ошибки данного подхода рассматриваются в разделе, посвященном *встроенной очистке* (In-line Teardown).) Очень сложно без последствий отследить все созданные объекты с дальнейшим их удалением.

Замечания по рефакторингу

Большинства проблем координации *встроенной очистки* (In-line Teardown) нескольких ресурсов можно безопасно избежать, если воспользоваться *очисткой усечением таблиц* (Table Truncation Teardown) и удалить все объекты аэропортов одним легким движением. (Предполагается, что в начале и в конце теста в базе данных не должно быть никаких записей объектов аэропортов. Если необходимо удалить только конкретные объекты аэропортов, *очистку усечением таблиц* (Table Truncation Teardown) использовать нельзя.) Большая часть усилий по рефакторингу направлена на удаление существующего кода очистки из раздела `finally` и добавление вызова метода `clearDatabase`. После этого необходимо реализовать данный метод с помощью команд удаления.

Пример: тест с использованием делегированной очистки усечением таблиц (Table Truncation Teardown)

Ниже представлен тест после завершения рефакторинга.

```

public void TestGetFlightsByOrigin_NoInboundFlight_TTTD()
{
    // Настройка тестовой конфигурации
    long OutboundAirport = CreateTestAirport("1OF");
    long InboundAirport = 0;
    FlightDto ExpectedFlightDto = null;
    try
    {
        InboundAirport = CreateTestAirport("1IF");
        ExpectedFlightDto =
            CreateTestFlight(OutboundAirport, InboundAirport);
        // Вызов тестируемой системы
        IList FlightsAtDestination1 =

```

```

        Facade.GetFlightsByOriginAirport (InboundAirport) ;
        // Проверка результата
        Assert.AreEqual (0, FlightsAtDestination1.Count) ;
    }
    finally
    {
        CleanDatabase () ;
    }
}

```

В этом примере для сохранения видимости кода очистки используется *делегированная очистка* (Delegated Teardown). Обычно в таких ситуациях применяется *неявная очистка* (Implicit Teardown) и логика размещается в методе tearDown. Блок try/catch обеспечивает вызов метода cleanDatabase, но не гарантирует, что очистка завершится при появлении ошибки внутри метода cleanDatabase.

Пример: тест с использованием “ленивой” очистки (Lazy Teardown)

Ниже показан тот же пример, но модифицированный для использования “ленивой” очистки (Lazy Teardown).

```

[Test]
    public void TestGetFlightsByOrigin_NoInboundFlight_LTD ()
{
    // "Ленивая" очистка
    CleanDatabase () ;
    // Настройка тестовой конфигурации
    long OutboundAirport = CreateTestAirport ("1OF") ;
    long InboundAirport = 0 ;
    FlightDto ExpectedFlightDto = null ;
    InboundAirport = CreateTestAirport ("1IF") ;
    ExpectedFlightDto =
        CreateTestFlight (OutboundAirport, InboundAirport) ;
    // Вызов тестируемой системы
    IList FlightsAtDestination1 =
        Facade.GetFlightsByOriginAirport (InboundAirport) ;
    // Проверка результата
    Assert.AreEqual (0, FlightsAtDestination1.Count) ;
}

```

Если вынести вызов cleanDatabase в начало *тестового метода* (Test Method), можно гарантировать ожидаемое состояние базы данных. Данный код очищает результаты работы предыдущего теста независимо от правильности предыдущей очистки. Кроме того, удаляются все записи, добавленные в интересующие таблицы после запуска предыдущего теста. При таком подходе исчезает необходимость в использовании блока try/finally, что значительно упрощает чтение и понимание теста.

Пример: очистка усечением таблиц (Table Truncation Teardown) с использованием SQL

В данной реализации метода cleanDatabase используется создаваемый кодом оператор SQL.

```

public static void CleanDatabase() {
    string[] tablesToTruncate =
        new string[] {"Airport", "City", "Airline_Cd", "Flight"};
    IDbConnection conn = getCurrentConnection();
    IDbTransaction txn = conn.BeginTransaction();
    try {
        foreach (string eachTableToTruncate in tablesToTruncate)
        {
            TruncateTable(txn, eachTableToTruncate);
        }
        txn.Commit();
        conn.Close();
    } catch (Exception e) {
        txn.Rollback();
    } finally {
        conn.Close();
    }
}
private static void TruncateTable(IDbTransaction txn,
                                  string tableName)
{
    const string C_DELETE_SQL = "DELETE FROM {0}";
    IDbCommand cmd = txn.Connection.CreateCommand();
    cmd.Transaction = txn;
    cmd.CommandText = string.Format(C_DELETE_SQL, tableName);
    cmd.ExecuteNonQuery();
}

```

В связи с использованием в качестве СУБД пакета SQL Server приходится реализовывать собственный метод TruncateTable, который выводит команду Delete * from.... Этот метод не потребовался бы, если бы команда TRUNCATE была реализована непосредственно в базе данных.

Пример: очистка усечением таблиц (Table Truncation Teardown) с использованием объектно-реляционного отображения

Ниже показана реализация метода cleanDatabase с использованием NHibernate в качестве уровня объектно-реляционного отображения.

```

public static void CleanDatabase() {
    ISession session =
        TransactionManager.Instance.CurrentSession;
    TransactionManager.Instance.BeginTransaction();
    try {
        // Необходимо удалить только корневые классы,
        // поскольку каскадные правила удалят все дочерние сущности
        session.Delete("from Airport");
        session.Delete("from City");
        session.Flush();
        TransactionManager.Instance.Commit();
    } catch (Exception e) {
        Console.WriteLine(e);
        throw e;
    }
}

```

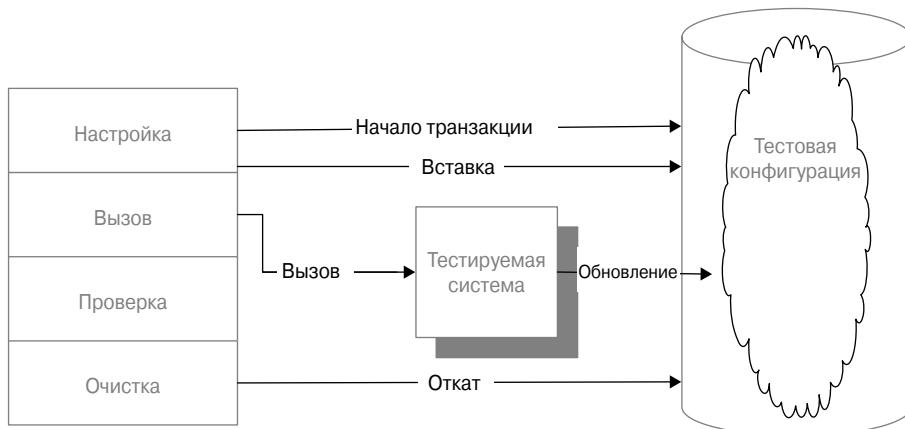
```
    } finally {
        TransactionManager.Instance.CloseSession();
    }
}
```

При использовании объектно-реляционного отображения происходит чтение, запись и удаление объектов предметной области. Инфраструктура автоматически определяет необходимые таблицы и выполняет соответствующие операции. Поскольку объекты City и Airport являются “корневыми” (родительскими), любой подчиненный (дочерний) объект, например Flight, удаляется автоматически при удалении корневого объекта. Такой подход еще больше ослабляет связь с реализацией таблиц.

Очистка откатом транзакции (Transaction Rollback Teardown)

Как очистить тестовую конфигурацию, если она находится в реляционной базе данных?

Несохраненная транзакция теста откатывается на этапе очистки тестовой конфигурации.



Повторяемость тестов в значительной мере зависит от очистки тестовой конфигурации после завершения каждого теста. Оставшиеся объекты и записи в базе данных, а также открытые файлы и подключения могут привести как минимум к снижению быстродействия, а как максимум — к неудачному завершению тестов и аварийному завершению работы системы. Хотя некоторые из этих ресурсов можно освободить автоматически с помощью сборки мусора, остальные ресурсы остаются занятыми, если не освободить их явно.

Чтобы создать надежный код для правильной очистки, во всех ситуациях нужны значительные усилия и много времени. При этом следует понимать, что может остаться после получения каждого возможного результата теста, и написать отдельный код для обработки каждого из вариантов. Подобная *сложная очистка* (Complex Teardown) приводит к появлению *условной логики теста* (Conditional Test Logic) и, что еще хуже, к появлению большого объема *нетестируемого кода тестов* (Untestable Test Code).

Внесения изменений в содержимое базы данных можно избежать, если не сохранять транзакцию теста и воспользоваться возможностью отката, реализованной в базе данных.

Как это работает

Тест начинает новую транзакцию, настраивает конфигурацию, вызывает тестируемую систему и проверяет результат. На любом из этих этапов он может взаимодействовать с базой данных. В конце выполнения теста начатая транзакция откатывается к предыдущему состоянию, что позволяет отказаться от записи внесенных изменений.

Когда это использовать

Очистка откатом транзакции (Transaction Rollback Teardown) может использоваться совместно с новой тестовой конфигурацией (Fresh Fixture, с. 344), если тестируемая система включает базу данных с поддержкой транзакций. Но существует ряд предварительных требований. *Очистка откатом транзакции* (Transaction Rollback Teardown) может использоваться только при их удовлетворении.

В частности, тестируемая система должна предоставлять методы, которые обычно вызываются в контексте существующей транзакции с помощью *минимального контроллера транзакций* (Humble Transaction Controller), т.е. методы не должны начинать собственную транзакцию и никогда не должны ее завершать. При разработке на основе тестов такой дизайн станет результатом применения шаблона *очистки откатом транзакции* (Transaction Rollback Teardown) в процессе написания кода. При интеграции тестов в существующее программное обеспечение перед использованием *очистки откатом транзакции* (Transaction Rollback Teardown) может потребоваться рефакторинг для вставки *минимального контроллера транзакций* (Humble Transaction Controller).

Положительным качеством *очистка откатом транзакции* (Transaction Rollback Teardown) является сохранение исходного состояния базы данных независимо от изменений, которые вносились в процессе работы теста. В результате не нужно создавать список удаляемых объектов. Изменения в схеме базы данных или в содержимом также не влияют на логику очистки. Очевидно, что этот шаблон применять значительно проще, чем *очистку усечением таблиц* (Table Truncation Teardown, с. 668).

Не забывайте о стандартных особенностях тестов, работающих с настоящей базой данных: такие тесты работают приблизительно в 50 (да, в 50!) раз медленнее, чем тесты без доступа к базе данных. Если настоящую базу данных не заменить *базой данных в памяти* (In-Memory Database) для большинства тестов, подобный подход практически гарантированно приводит к появлению *медленных тестов* (Slow Tests, с. 289). Поскольку тесты зависят от транзакционных свойств базы данных, простая *поддельная база данных* (Fake Database) в большинстве случаев не сможет решать поставленные задачи.

Еще одним требованием является недопустимость завершения транзакции в пределах теста или вызываемого кода. Примеры скрытного завершения транзакций с последующими проблемами приводятся во врезке “Мучения с откатом транзакций”.

МУЧЕНИЯ С ОТКАТОМ ТРАНЗАКЦИЙ

Джон Херст прислал по электронной почте описание проблем, с которыми столкнулась его команда при использовании *очистки откатом транзакции* (Transaction Rollback Teardown). Вот что он пишет.

Некоторое время после обсуждения на конференции TheServerSide для интеграционных тестов базы данных использовалась очистка откатом транзакции (Transaction Rollback Teardown), которую предложил Род Джонсон. Как я понял, основным аргументом за использование этого подхода было быстродействие. Обычно откат транзакции происходит намного быстрее, чем восстановление содержимого базы данных в новой транзакции. Действительно, тесты стали работать немного быстрее. Мы использовали отличный базовый класс AbstractTransactionalDataSourceTestTests из пакета Spring, который изначально поддерживает большинство операций данного шаблона.

Но через несколько месяцев было принято решение отказаться от использования шаблона. Вот список причин, по которым это было сделано.

1. Утрачивается изолированность тестов. При использованной у нас реализации шаблона каждый тест предполагал, что база данных находится в некотором базовом состоянии и откат вернет ее в это состояние. В текущей модели тест сам отвечает (обычно через метод `setUp()` базового класса) за перевод базы данных в известное состояние.
2. Нельзя заглянуть в базу данных, если что-то пошло не так. Если тест завершается неудачно, обычно возникает желание просмотреть содержимое базы данных и разобраться, что произошло. При откате всех изменений найти ошибку может оказаться сложнее, чем обычно.
3. Необходимо внимательно следить за неожиданными завершениями транзакции во время работы теста. Да, код в пределах теста обеспечивает декларативное управление транзакциями и не делает ничего неожиданного. Но иногда в процессе настройки тестовой конфигурации приходится выполнять некоторые операции, например удаление и повторное создание последовательности для сброса ее значения. Учитывая, что при этом используется `DDL`, все незавершенные транзакции завершаются. Результат — ничего не понимающий разработчик.
4. Сложно комбинировать этот подход с тестами, для которых завершение транзакций необходимо. В последнее время были добавлены тесты и хранимые процедуры `PL/SQL`. Некоторые из хранимых процедур явно завершают транзакции. Невозможно разместить в одном наборе эти тесты и тесты, требующие от базы данных сохранения определенного состояния.

Приношу свои извинения за несовпадение используемой у нас терминологии с терминологией в вашей книге. Кроме того, наш опыт достаточно ограничен, поскольку этот подход рассматривался только в среде Spring и большинство операций мы предпочитаем выполнять в соответствии с “идеологией” Spring. Наконец, уверен, что эти ограничения можно обойти различными способами. Просто для нашей команды разработчиков данный шаблон оказался не стоящим затраченных усилий.

Не поймите меня превратно: я считаю, что данный шаблон должен присутствовать в книге. Просто последствия его выполнения должны быть описаны, и подходит он не для каждой группы разработчиков.

Замечания по реализации

Не многие реализации xUnit непосредственно поддерживают шаблон очистка откатом транзакции (Transaction Rollback Teardown). Для других реализаций могут существовать расширения с открытым исходным кодом. В противном случае придется реализовать несложную логику очистки. Более значительным свойством реализации является предоставление тестам доступа к нетранзакционным методам тестируемой системы. Большинство объектов предметной области не поддерживают транзакции, поэтому данное требование не станет проблемой при тестировании объектов предметной области. Скорее всего, проблемы возникнут при создании “подкожных” тестов (Subcutaneous Test) фасада служб (Service Facade), так как обычно они выполняют контроль транзакций. В таком случае придется раскрывать доступ к нетранзакционным версиям методов через рефакторинг в направлении шаблона *минимальный контроллер транзакций* (Humble Transaction Controller). Можно или использовать транзакционный декоратор в качестве отдельного объекта, или вынудить транзакционные методы де-

легировать выполнение нетранзакционным версиям методов того же экземпляра. Такой подход называется *простейшим минимальным объектом* (Poor Man's Humble Object).

Если методы существуют, но не видны клиенту, возможно, их придется открыть для доступа со стороны теста. Для этого тестируемые методы можно сделать открытыми или обеспечить опосредованный доступ через *связанный с тестом подкласс* (Test-Specific Sub-class, с. 591). Кроме того, можно воспользоваться рефакторингом *выделение тестируемого компонента* (Extract Testable Component) для переноса нетранзакционных версий методов в другой класс, сделав их видимыми для теста из нового места.

Любое чтение обновленных данных в базе данных должно происходить в контексте одной и той же транзакции. Это не проблема, кроме случаев имитации или тестирования одновременного доступа. При использовании уровня объектно-реляционного отображения, например Toplink, (N)Hibernate или EJB 3.0, от этого уровня может потребоваться запись внесенных в объект изменений в базу данных, чтобы изменения были видны методам, непосредственно читающим состояние из базы данных (в пределах того же контекста транзакции). Например, в инфраструктуре EJB 3.0 предоставляется статический метод EntityManager.flush, обеспечивающий именно такую функциональность.

Мотивирующий пример

В следующем тесте предпринята попытка использовать *гарантированную встроенную очистку* (Guaranteed In-line Teardown) для удаления всех созданных записей.

```
public void testGetFlightsByOriginAirport_NoInboundFlights()
    throws Exception {
    // Настройка тестовой конфигурации
    BigDecimal outboundAirport = createTestAirport("1OF");
    BigDecimal inboundAirport = createTestAirport("1IF");
    FlightDto expFlightDto = null;
    try {
        expFlightDto = createTestFlight(outboundAirport, inboundAirport);
        // Вызов тестируемой системы
        List flightsAtDestination1 =
            facade.getFlightsByOriginAirport(inboundAirport);
        // Проверка результата
        assertEquals(0, flightsAtDestination1.size());
    } finally {
        facade.removeFlight(expFlightDto.getFlightNumber());
        facade.removeAirport(outboundAirport);
        facade.removeAirport(inboundAirport);
    }
}
```

Этот код сложно написать и сложно понять! (Ошибки данного подхода рассматриваются в разделе, посвященном *встроенной очистке* (In-line Teardown).) Очень сложно без последствий отследить все созданные объекты с дальнейшим их удалением.

Замечания по рефакторингу

Большинства проблем координации *встроенной очистки* (In-line Teardown) можно избежать, если воспользоваться *очисткой откатом транзакции* (Transaction Rollback Tear-

down) и удалять все изменения одним движением. Значительная доля рефакторинга включается в удалении кода очистки из блока `finally` и добавления вызова метода `abortTransaction`. Кроме того, необходимо вызывать метод `beginTransaction` перед началом создания тестовой конфигурации и модифицировать *методы создания* (Creation Method, с. 441), чтобы они не завершали транзакцию. Для этого они должны вызывать нетранзакционные версии методов *facade служб* (Service Facade).

Пример: очистка откатом транзакции через объекты

Ниже показан тест после внесения соответствующих изменений.

```
public void testGetFlightsByOrigin_NoInboundFlight_TRBTD()
    throws Exception {
    // Настройка тестовой конфигурации
    TransactionManager.beginTransaction();
    BigDecimal outboundAirport = createTestAirport("1OF");
    BigDecimal inboundAirport = null;
    FlightDto expectedFlightDto = null;
    try {
        inboundAirport = createTestAirport("1IF");
        expectedFlightDto =
            createTestFlight(outboundAirport, inboundAirport);
        // Вызов тестируемой системы
        List flightsAtDestination1 =
            facade.getFlightsByOriginAirport(inboundAirport);
        // Проверка результата
        assertEquals(0, flightsAtDestination1.size());
    } finally {
        TransactionManager.abortTransaction();
    }
}
```

В полученном teste несколько строк кода очистки в блоке `finally` заменены единственным вызовом метода `abortTransaction`. Блок `finally` все еще необходим, так как в этом примере используется *встроенная очистка* (In-line Teardown). Благодаря универсальности этот вызов объекта `TransactionManager` можно легко перенести в метод `tearDown`.

В данном примере *очистка откатом транзакции* (Transaction Rollback Teardown) отменяет настройку тестовой конфигурации, выполняемую различными *методами создания* (Creation Method), которые вызываются в начале теста. Объекты тестовой конфигурации еще не были внесены в базу данных, но так как метод `getFlightsFromAirport` вызывается в контексте транзакции, он возвращает добавленные, но еще не записанные объекты перелетов.

```
private BigDecimal createTestAirport(String airportName)
    throws FlightBookingException {
    BigDecimal newAirportId =
        facade._createAirport(airportName,
            "Airport" + airportName,
            "City" + airportName);
    return newAirportId;
}
```

Метод создания вызывает нетранзакционную версию метода фасада (пример *простейшего минимального объекта*, Poor Man's Humble Object).

```
public BigDecimal createAirport(String airportCode,
                                String name,
                                String nearbyCity)
    throws FlightBookingException{
    TransactionManager.beginTransaction();
    BigDecimal airportId = _createAirport(airportCode, name, nearbyCity);
    TransactionManager.commitTransaction();
    return airportId;
}
// Закрытая нетранзакционная версия для вызова тестами
BigDecimal _createAirport(String airportCode,
                           String name,
                           String nearbyCity)
    throws DataException, InvalidArgumentException {
    Airport airport =
        dataAccess.createAirport(airportCode, name, nearbyCity);
    logMessage("CreateFlight", airport.getCode());
    return airport.getId();
}
```

Если вызываемый метод (например, `getFlightsFromAirport`) модифицировал состояние тестируемой системы, а также начал и закончил собственную транзакцию, его придется подвергнуть аналогичному рефакторингу.

Пример: очистка откатом транзакции через базу данных

В первом примере база данных была скрыта от кода за уровнем доступа к данным, который возвращал и принимал объекты. Подобная практика распространена при организации бизнес-логики на основе шаблона Domain Model. *Очистка откатом транзакции* (Transaction Rollback Teardown) обычно используется при непосредственном управлении базой данных из логики приложения (через так называемый **сценарий транзакции**, transaction script). В следующем примере такой подход продемонстрирован на основе наборов строк .NET.

```
[TestFixture]
public class TransactionRollbackTearDownTest
{
    private SqlConnection _Connection;
    private SqlTransaction _Transaction;
    public TransactionRollbackTearDownTest()
    {
    }
    [SetUp]
    public void Setup()
    {
        string dbConnectionString = ConfigurationSettings.
            AppSettings.Get("DbConnectionString");
        _Connection = new SqlConnection(dbConnectionString);
        _Connection.Open();
        _Transaction = _Connection.BeginTransaction();
    }
}
```

```
[TearDown]
    public void TearDown()
{
    _Transaction.Rollback();
    _Connection.Close();
    // Обход ошибки NUnit "поведение экземпляра"
    _Transaction = null;
    _Connection = null;
}
[Test]
    public void AnNUnitTest()
{
    const string C_INSERT_SQL =
        "INSERT INTO Invoice(Amount, Tax, CustomerId) " +
        " VALUES({0}, {1}, {2})";
    SqlCommand cmd = _Connection.CreateCommand();
    cmd.Transaction = _Transaction;
    cmd.CommandText = string.Format(
        C_INSERT_SQL,
        new object[] {"100.00", "7.00", 2001});
    // Вызов тестируемой системы
    cmd.ExecuteNonQuery();
    // Проверка результата
    // И т.д.
}
}
```

В этом примере для создания подключения и начала транзакции используется *неявная настройка* (Implicit Setup, с. 449).

После запуска *тестового метода* (Test Method, с. 378) *неявная очистка* (Implicit Tear-down, с. 533) используется для отката транзакции и завершения подключения. Переменным экземпляра присваивается значение null, так как NUnit, в отличие от других реализаций xUnit, не создает отдельный *объект теста* (Testcase Object, с. 410) для каждого *тестового метода* (Test Method). Дополнительная информация приведена во врезке “Всегда есть исключения” на с. 411.

Глава 26

Шаблоны проектирования с учетом тестов

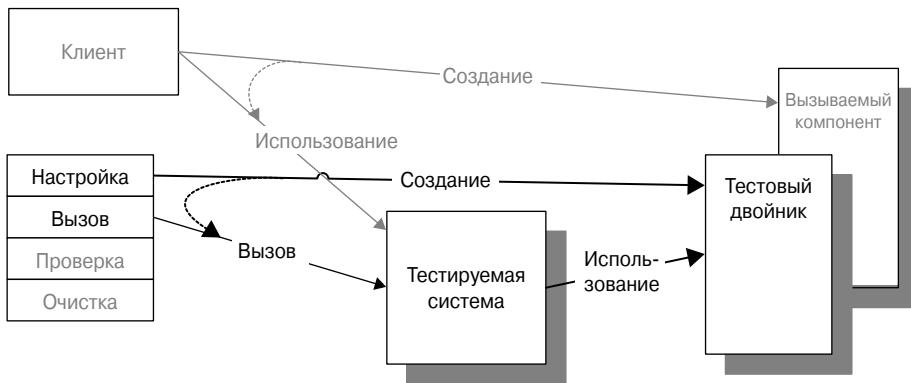
Шаблоны в этой главе:

Вставка зависимости (Dependency Injection)	684
Поиск зависимости (Dependency Lookup).....	692
Минимальный объект (Humble Object)	700
Ловушка для теста (Test Hook)	713

Вставка зависимости (Dependency Injection)

Как спроектировать тестируемую систему для замены зависимостей во время выполнения?

Вызываемый компонент предоставляется тестируемой системе клиентом.



Практически каждый фрагмент кода зависит от других классов, объектов, модулей или процедур. Для правильного модульного тестирования код необходимо изолировать от его зависимостей. Если такие зависимости зафиксированы в коде в виде имен классов, подобная изоляция может быть затруднена.

Вставка зависимости (Dependency Injection) является способом разрыва связи между тестируемой системой и ее зависимостями во время автоматизированного тестирования.

Как это работает

Необходимо избегать использования имен классов в коде и применять другие способы настройки клиента или системы для предоставления тестируемой системе объектов, от которых она зависит во время выполнения. При проектировании тестируемой системы принимается решение о передаче зависимости через “главный ход”, т.е. возможность указания зависимости становится частью программного интерфейса тестируемой системы. Указатель на используемый компонент может передаваться в качестве аргумента вызываемого метода, аргумента конструктора или значения атрибута (свойства).

Когда это использовать

Возможность замены вызываемого компонента необходима для более простого использования *тестовых двойников* (Test Double, с. 538) при проверке кода. **Статическое связывание** (static binding) — т.е. указание конкретных типов или классов на этапе компиляции — существенно сокращает возможности настройки программного обеспечения во время работы. **Динамическое связывание** (dynamic binding) позволяет создавать значительно более гибкое программное обеспечение за счет откладывания решений о выборе класса или типа до момента выполнения. *Вставка зависимости* (Dependency Injection)

является хорошим механизмом выбора классов при проектировании программного обеспечения “с нуля”. Данный шаблон обеспечивает естественный способ проектирования кода при разработке на основе тестов, так как множество создаваемых тестов пытается заменить вызываемый компонент *тестовым двойником* (Test Double).

Если тестируемый код контролируется не полностью, например при интеграции тестов в существующий код (в такой ситуации распространено ограничение “Если работает, не трогай (даже для упрощения тестирования)”), могут потребоваться другие способы ввода *тестовых двойников* (Test Double). Если для обнаружения вызываемых компонентов тестируемая система использует *поиск зависимости* (Dependency Lookup, с. 692), механизм поиска можно переопределить для возврата *тестового двойника* (Test Double). Кроме того, если обращение к вызываемому компоненту скрыто внутри метода для возврата *тестового двойника* (Test Double), можно использовать *связанный с тестом подкласс* (Test-Specific Subclass, с. 591).

Замечания по реализации

Для добавления механизма *вставки зависимости* (Dependency Injection) следует решить две проблемы. Во-первых, необходимо иметь возможность использовать *тестовый двойник* (Test Double) вместо настоящего вызываемого компонента. Такое ограничение обычно характерно для статически типизированных языков, поскольку необходимо убедить компилятор передать *тестовый двойник* (Test Double) вместо настоящего объекта. Во-вторых, тестируемую систему необходимо заставить использовать *тестовый двойник* (Test Double).

Совместимость типов

Независимо от способа установки зависимости в тестируемую систему необходимо обеспечить “совместимость по типу” *тестового двойника* (Test Double) и кода, который будет его использовать. Проще всего, если *тестовый двойник* (Test Double) и настоящий компонент реализуют один и тот же интерфейс (в статически типизированных языках) или имеют одну и ту же сигнатуру (в динамически типизированных языках). Быстрым способом вставки *тестового двойника* (Test Double) в существующий код является применение рефакторинга *выделение интерфейса* (Extract Interface) к настоящему вызываемому компоненту с последующей реализацией интерфейса в *тестовом двойнике* (Test Double).

Установка тестового двойника

Существует несколько способов вынудить тестируемую систему использовать *тестовый двойник* (Test Double), но все они подразумевают замену фиксированного имени механизма определения типа объекта во время выполнения. Ниже приведены три основных варианта.

- *Вставка параметра* (Parameter Injection). Зависимость передается в тестируемую систему во время вызова.
- *Вставка конструктора* (Constructor Injection). Тестируемая система получает информацию о вызываемом компоненте на этапе создания экземпляра.
- *Вставка метода установки* (Setter Injection). Тестируемая система получает информацию о вызываемом компоненте в какой-то момент между созданием и использованием.

Каждый из вариантов *вставки зависимости* (Dependency Injection) можно реализовать вручную. Альтернативный вариант предполагает использование инфраструктуры **инвертирования управления** (inversion of control) для связывания различных компонентов на этапе выполнения. Такая схема позволяет избежать излишнего разнообразия в реализации *вставки зависимости* (Dependency Injection) в пределах приложения и может упростить процесс перенастройки приложения для различных моделей распространения.

Вариант: вставка параметра (Parameter Injection)

Вставка параметра (Parameter Injection) является вариантом *вставки зависимости* (Dependency Injection), при котором тестируемая система не хранит и не инициализирует ссылку на вызываемый компонент. Вместо этого ссылка передается в качестве аргумента методов тестируемой системы. Все клиенты (как тестовые так и настоящие) предоставляют вызываемый компонент. В результате тестируемая система становится более независимой от контекста, поскольку не делается предположений о зависимостях за пределами интерфейса использования. Основным недостатком является требование от клиента знаний о зависимости, что желательно не во всех ситуациях. В большинстве других вариантов *вставки зависимости* (Dependency Injection) это знание выносится за пределы клиента или данное требование считается необязательным.

Вставка параметра (Parameter Injection) рассматривалась в исходной статье о *подставных объектах* (Mock Object, с. 558) [ET]. Максимальная эффективность данного шаблона достигается при разработке на основе тестов, поскольку в такой ситуации обеспечивается максимальный контроль над проектированием программного обеспечения. *Вставку параметра* (Parameter Injection) можно реализовать как необязательный механизм, предоставив альтернативную сигнатуру метода с дополнительным параметром. После этого традиционный метод может создавать экземпляр зависимости и вызывать метод, принимающий зависимость в качестве параметра.

Вариант: вставка конструктора (Constructor Injection)

И *вставка конструктора* (Constructor Injection), и *вставка метода установки* (Setter Injection) предполагают хранение ссылки на вызываемый компонент в качестве атрибута (поля или переменной экземпляра) тестируемой системы. При *вставке конструктора* (Constructor Injection) это поле инициализируется аргументом конструктора. Тестируемая система может предоставить более простой конструктор, который вызывает альтернативный конструктор со значением, обычно используемым при промышленной эксплуатации. Если тесту требуется замена настоящего вызываемого компонента *тестовым двойником* (Test Double), двойник передается конструктору при создании экземпляра тестируемой системы.

Такой подход ко вводу *вставки зависимости* (Dependency Injection) хорошо работает при использовании кода с одним или двумя конструкторами и небольшим списком аргументов. *Вставку конструктора* (Constructor Injection) является единственным допустимым подходом, если вызываемый компонент является активным объектом, который создает собственный поток выполнения в процессе создания. Такое поведение, скорее всего, приведет к появлению *сложного в тестировании кода* (Hard-to-Test Code, с. 251), поэтому имеет смысл рассмотреть использование *минимального выполняемого файла* (Humble Executable). Если аргументы конструктора содержат большое количество зависимостей, потребуется рефакторинг кода для избавления от этого запаха.

Вариант: вставка метода установки (Setter Injection)

Как и в случае *вставки конструктора* (Constructor Injection), тестируемая система содержит ссылку на вызываемый компонент в виде атрибута (поля), который инициализируется в конструкторе. Главным отличием является предоставление доступа к этой ссылке в виде открытого атрибута или через метод установки. Если тесту необходимо заменить настоящий вызываемый компонент *тестовым двойником* (Test Double), экземпляр двойника присваивается открытому атрибуту или передается в метод установки. Такой подход вполне оправдан, если настоящий вызываемый компонент не имеет нежелательных побочных эффектов и ничего не может произойти автоматически между вызовом конструктора и вызовом метода установки со стороны теста. *Вставка метода установки* (Setter Injection) не может использоваться, если тестируемая система выполняет значительную долю работы в конструкторе, зависящем от вызываемого компонента. В таком случае необходимо использовать *вставку конструктора* (Constructor Injection). Если создание настоящего вызываемого компонента имеет неприятные побочные эффекты, можно избежать его создания через конструктор, модифицировав тестируемую систему для использования *“ленивой” инициализации* (lazy initialization). В таком случае экземпляр вызываемого компонента будет создаваться при первой необходимости.

Добавление вставки зависимости

Если ни один из этих вариантов не поддерживается “из коробки”, эту возможность придется интегрировать с помощью *связанного с тестом подкласса* (Test-Specific Subclass). Если используемый класс получается на основе конфигурационных данных, это должно выполняться другим компонентом, а класс должен передаваться в тестируемую систему с помощью *вставки зависимости* (Dependency Injection). Такое использование *минимального объекта* (Humble Object) для настройки ослабляет связь тестируемой системы со средой и делает ненужным создание внешней зависимости (конфигурационного файла) для вставки *тестового двойника* (Test Double).

Еще один вариант предполагает использование **аспектно-ориентированного программирования** (aspect-oriented programming) для добавления механизма *вставки зависимости* (Dependency Injection) в среду разработки. Например, решение использовать *тестовый двойник* (Test Double) или связанную с тестом логику можно вставить непосредственно в тестируемую систему.

Мотивирующий пример

Следующий тест не может завершиться успешно в показанном виде.

```
public void testDisplayCurrentTime_AtMidnight() {
    // Настройка тестовой конфигурации
    TimeDisplay sut = new TimeDisplay();
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка непосредственного вывода
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals(expectedTimeString, result);
}
```

Такой тест практически всегда завершается неудачно, поскольку он зависит от текущего времени, возвращаемого вызываемым компонентом в тестируемую систему. Тест не управляет возвращаемыми значениями компонента `DefaultTimeProvider`. Таким образом, он будет завершаться успешно, только если системные часы показывают ровно полночь.

```
public String getCurrentTimeAsHtmlFragment() {
    Calendar currentTime;
    try {
        currentTime = new DefaultTimeProvider().getTime();
    } catch (Exception e) {
        return e.getMessage();
    }
    // И т.д.
}
```

Тестируемая система содержит фиксированную ссылку на определенный класс для получения времени, поэтому заменить вызываемый компонент *тестовым двойником* (Test Double) не возможно. Это ограничение делает тест неопределенным и практически бесполезным. Необходимо найти способ получения контроля над опосредованным вводом тестируемой системы.

Замечания по рефакторингу

Для получения контроля над временем можно воспользоваться рефакторингом *заменить зависимость тестовым двойником* (Replace Dependency with Test Double, с. 740). Механизм *вставки метода установки* (Setter Injection) можно добавить в существующий код, если этот метод применяется не слишком широко и над кодом есть достаточный контроль. Также для этих целей можно воспользоваться инструментами рефакторинга, поддерживающими рефакторинг *введение параметра* (Introduce parameter). В противном случае можно воспользоваться рефакторингом *выделение метода* (Extract Method) для создания новой сигнатуры метода, принимающего зависимость в качестве аргумента, оставив старый метод в качестве *адаптера* (Adapter), вызывающего новый метод.

Пример: вставка параметра (Parameter Injection)

Ниже показан тест, переписанный для использования *вставки параметра* (Parameter Injection).

```
public void testDisplaycurrentTime_AtMidnight_PI() {
    // Настройка тестовой конфигурации
    // Установка тестового двойника
    TimeProvider tpStub = new MidnightTimeProvider();
    // Создание экземпляра тестируемой системы
    TimeDisplay sut = new TimeDisplay();
    // Вызов тестируемой системы с использованием двойника
    String result = sut.getCurrentTimeAsHtmlFragment(tpStub);
    // Проверка результата
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}
```

В этом случае только тест использует новую сигнатуру. Существующий код может использовать старую сигнатуру, а адаптер создает настоящий объект зависимости перед его передачей.

```
public String getCurrentTimeAsHtmlFragment(
    TimeProvider timeProviderArg) {
    Calendar currentTime;
    try {
        currentTime = timeProviderArg.getTime();
    } catch (Exception e) {
        return e.getMessage();
    }
    // И т.д.
}
```

Пример: вставка конструктора (Constructor Injection)

Ниже показан тот же тест, но переписанный для использования *вставки конструктора* (Constructor Injection).

```
public void testDisplayCurrentTime_AtMidnight_CI()
    throws Exception {
    // Настройка тестовой конфигурации
    // Установка тестового двойника
    TimeProvider tpStub = new MidnightTimeProvider();
    // Создание тестируемой системы с передачей двойника
    TimeDisplay sut = new TimeDisplay(tpStub);
    // Вызов тестируемой системы
    String expectedTimeString =
        "<span class=\"tinyBoldText\">12:01 AM</span>";
    // Проверка результата
    assertEquals("12:01 AM",
        expectedTimeString,
        sut.getCurrentTimeAsHtmlFragment());
}
```

Чтобы преобразовать тестируемую систему в соответствии с требованиями механизма *вставки конструктора* (Constructor Injection), можно воспользоваться рефакторингом *введение поля* (Introduce Field) для хранения вызываемого компонента в поле, которое инициализируется в существующем конструкторе. После этого можно воспользоваться рефакторингом *введение параметра* (Introduce Parameter) для модификации всех вызовов существующего конструктора с передачей настоящего вызываемого компонента в качестве параметра конструктора. Если нельзя или нежелательно модифицировать все существующие вызовы конструктора, следует определить новый конструктор, который будет создавать экземпляр настоящего вызываемого компонента и передавать его в новый конструктор.

```
public class TimeDisplay {
    private TimeProvider timeProvider;
    public TimeDisplay() { // Обратно совместимый конструктор
        timeProvider = new DefaultTimeProvider();
    }
    public TimeDisplay(TimeProvider timeProvider) { // Новый конструктор
```

```

        this.timeProvider = timeProvider;
    }
}

```

Еще один конструктор предполагает использование рефакторинга *выделение метода* (Extract Method) к вызову конструктора с последующим рефакторингом *перемещение метода* (Move Method) для его перемещения в *фабрику объектов* (Object Factory). В результате будет получена реализация *поиска зависимости* (Dependency Lookup).

Пример: вставка метода установки (Setter Injection)

Ниже показан тот же тест, переписанный для использования *вставки метода установки* (Setter Injection).

```

public void testDisplayCurrentTime_AtMidnight_SI()
    throws Exception {
    // Настройка тестовой конфигурации
    // Создание экземпляра тестового двойника
    TimeProvider tpStub = new MidnightTimeProvider();
    // Создание экземпляра тестируемой системы
    TimeDisplay sut = new TimeDisplay();
    // Установка тестового двойника
    sut.setTimeProvider(tpStub);
    // Создание тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка результата
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals("Midnight", expectedTimeString, result);
}

```

Обратите внимание на вызов метода setTimeProvider для установки *фиксированного тестового двойника* (Hard-Coded Test Double, с. 581). Если бы использовался *настраиваемый тестовый двойник* (Configurable Test Double, с. 571), его настройка происходила бы непосредственно перед вызовом метода setTimeProvider.

Переработка тестируемой системы для поддержки *вставки метода установки* (Setter Injection) возможна за счет использования рефакторинга *вставки метода установки* (Setter Injection) для хранения вызываемого компонента в переменной, которая инициализируется в существующем конструкторе, и вызова компонента через это поле. После этого поле можно сделать доступным непосредственно или через метод, который может переопределять его значение. Ниже показана переработанная версия тестируемой системы.

```

public class TimeDisplay {
    private TimeProvider timeProvider;
    public TimeDisplay() {
        timeProvider = new DefaultTimeProvider();
    }
    public void setTimeProvider(TimeProvider provider) {
        this.timeProvider = provider;
    }
    public String getCurrentTimeAsHtmlFragment()
        throws TimeProviderEx {
        Calendar currentTime;
}

```

```
try {
    currentTime = getTimeProvider().getTime();
} catch (Exception e) {
    return e.getMessage();
}
// и т.д.
```

В данном случае для получения вызываемого компонента используется метод доступа. С тем же успехом поле timeProvider можно было использовать непосредственно.

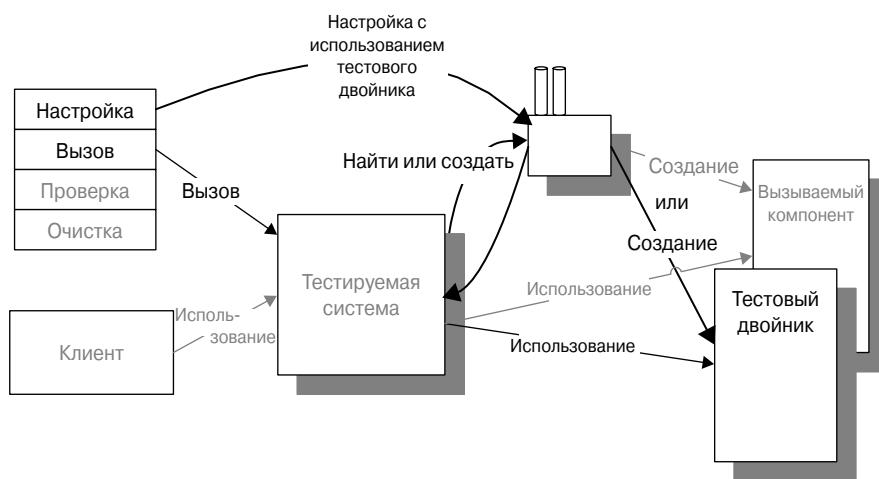
Поиск зависимости (Dependency Lookup)

Также известен как:

*Локатор служб (Service Locator),
Фабрика объектов (Object Factory), Брокер компонентов
(Component Broker), Регистр компонентов (Component Registry)*

Как спроектировать тестируемую систему для замены зависимостей во время выполнения?

Ссылка на вызываемый компонент запрашивается у другого объекта.



Практически каждый фрагмент кода зависит от других классов, объектов, модулей или процедур. Для правильного модульного тестирования код необходимо изолировать от его зависимостей. Если такие зависимости зафиксированы в коде в виде имен классов, подобная изоляция может быть затруднена.

Поиск зависимости (Dependency Lookup) является способом разрыва связи между тестируемой системой и ее зависимостями во время автоматизированного тестирования.

Как это работает

Необходимо избегать использования имен классов, от которых зависит тестируемая система, в коде, так как статическое связывание значительно ограничивает настройку программного обеспечения в процессе выполнения. Вместо этого фиксируется “брокер компонентов”, который возвращает готовые к использованию объекты. Брокер компонентов обеспечивает клиентскому программному обеспечению или диспетчеру системной конфигурации возможность сообщить тестируемой системе, какой объект использовать для каждого запроса компонента.

Когда это использовать

Поиск зависимости (Dependency Lookup) лучше всего подходит при получении вызываемого компонента глубоко в недрах системы, когда очень сложно передать *тестовый*

двойник (Test Double, с. 538) непосредственно от клиента. Хорошим примером такой ситуации является замена уровня доступа к данным *поддельной базой данных* (Fake Database) или *базой данных в памяти* (In-Memory Database) для ускорения работы автоматизированных приемочных тестов. Было бы слишком сложно передавать *поддельную базу данных* (Fake Database) через *фасад служб* (Service Facade) при каждом обращении теста, предлагающем использование уровня доступа к данным. *Поиск зависимости* (Dependency Lookup) позволяет тесту или даже *декоратору настройки* (Setup Decorator, с. 471) использовать “фасад конфигурации” для установки *поддельной базы данных* (Fake Database), которая будет применяться тестируемой системой автоматически без дополнительных действий. Вот что по этому поводу пишет Джереми Миллер.

Нельзя недооценивать значение локатора служб (Service Locator) при автоматизированном тестировании. Альтернативные зависимости при тестировании используются регулярно для решения проблем со сложными зависимостями и для повышения быстродействия тестов. Например, для повышения быстродействия в функциональном тесте Web-сайт и поддерживающий его сервер приложений объединяются в единый процесс.

Механизм *поиска зависимости* (Dependency Lookup) значительно проще интегрировать в существующее и унаследованное программное обеспечение, так как он касается только тех фрагментов, в которых происходит создание объектов. Модификация каждого промежуточного объекта или метода, как при использовании *вставки зависимости* (Dependency Injection, с. 684), не нужна. Кроме того, “завернув” существующие тесты в *декоратор настройки* (Setup Decorator), намного проще заставить их использовать *поддельный объект* (Fake Object). При такой схеме модификация каждого теста не требуется. Вместо этого в каждом тесте создается новый экземпляр тестируемой системы, а *поддельный объект* (Fake Object) используется тот же, так как *локатор служб* (Service Locator) сохраняется между тестами¹.

Основной альтернативой *поиску зависимости* (Dependency Lookup) является представление механизма замены внутри тестируемой системы через *вставку зависимости* (Dependency Injection). Этот подход обычно более предпочтителен для модульных тестов, так как замена вызываемого компонента становится более очевидной и непосредственно связанной с вызовом тестируемой системы. Еще один вариант предполагает использование **аспектно-ориентированного программирования** (aspect-oriented programming) для вставки логики теста средствами разработки, а не модификацией дизайна программного обеспечения. Наименее желательным решением является использование *ловушек для теста* (Test Hook, с. 713) внутри тестируемой системы для отказа от использования вызываемого компонента или внутри вызываемого компонента для поведения в соответствии с требованиями теста.

Широкоизвестный промежуточный компонент может называться *локатором служб* (Service Locator), *фабрикой объектов* (Object Factory), *брокером компонентов* (Component Broker) или *реестром компонентов* (Component Registry). Хотя каждое из названий предполагает свою семантику (возврат новых или существующих объектов), такое деление не является обязательным. Для повышения производительности может приниматься решение о возврате нового объекта из *локатора служб* (Service Locator) или “бывшего в употреблении”.

¹ Такие тесты называются многомодальными, поскольку они одновременно работают с настоящим и поддельным вызываемыми компонентами.

реблении” — из *фабрики объектов* (Object Factory). Для простоты здесь используется термин *брокер компонентов* (Component Broker).

Замечания по реализации

Для использования *тестовых двойников* (Test Double) при тестировании кода необходим механизм замены вызываемых компонентов. Такое ограничение делает недопустимой фиксацию имен вызываемых классов в коде тестируемой системы, так как статическое связывание значительно сокращает возможности настройки программного обеспечения в процессе работы. Одним из способов решения этой проблемы является делегирование создания вызываемых компонентов другому объекту. Конечно, при такой схеме необходим способ получения ссылки на этот объект. Данная рекурсивная проблема решается за счет использования широкоизвестного объекта, который служит посредником между тестом и вызываемым компонентом. Для обращения к широкоизвестному объекту используется фиксированное имя класса. Для использования такого объекта при установке *тестового двойника* (Test Double) должен существовать механизм, с помощью которого можно выбрать возвращаемый объект.

Поиск зависимости (Dependency Lookup) имеет следующие характеристики.

- В качестве реализации используется шаблон Singleton [GOF], Registry [PEAA] или Thread-specific storage [POSA2].
- Интерфейс полностью скрывает используемую реализацию.
- Встроенный механизм подмены используется для замены возвращаемого объекта *тестовым двойником* (Test Double).
- Доступен через широкоизвестное имя.

Механизм *поиска зависимости* (Dependency Lookup) возвращает объект, который может непосредственно использоваться клиентом. Природа возвращаемого объекта определяет выбор между названиями *локатор служб* (Service Locator) и *фабрика объектов* (Object Factory). После получения объекта тестируемая система использует его непосредственно. В процессе тестирования тест настраивает механизм *поиска зависимости* (Dependency Lookup) для возврата необходимых тесту объектов.

Сокрытие реализации

Основным требованием к *поиску зависимости* (Dependency Lookup) является существование широкоизвестного объекта, которому можно делегировать запросы на получение вызываемых компонентов. Такой объект может быть реализацией одного из следующих шаблонов: Singleton, Registry или одного из типов Thread-Specific Storage.²

Брокер компонентов (Component Broker) должен скрывать свою реализацию от клиента (тестируемой системы), т.е. предоставляемый им интерфейс не должен показывать признаки реализации шаблона Singleton, Registry или Thread-Specific Storage. На самом деле в тес-

² Основным отличием между ними является гарантированная единственность экземпляра Singleton. Шаблон Registry такой единственности не обещает. Шаблон Thread-Specific Storage позволяет объектам получать доступ к “глобальным” данным через широкоизвестный объект, при этом данные связаны с конкретным потоком; тот же самый объект может возвращать другие данные в зависимости от потока, в котором он находится.

товой среде может использоваться другая реализация, например *заменяемый единственный экземпляр класса* (Substitutable Singleton), специально предназначенная для обхода проблем, которые возникают при тестировании с использованием объектов Singleton.

Механизм замены

Если тест пытается заменить настоящий вызываемый компонент *тестовым двойником* (Test Double), должен существовать способ настройки *брокера компонентов* (Component Broker) на возврат *тестового двойника* (Test Double) вместо обычного объекта. *Брокер компонентов* (Component Broker) может иметь конфигурационный интерфейс для настройки возвращаемого объекта или тест может заменить реестр компонентов подходящим *связанным с тестом подклассом* (Test-Specific Subclass). Кроме того, может потребоваться способ восстановления исходной или принятой по умолчанию конфигурации, чтобы используемая в данном teste конфигурация не “протекала” в другие тесты, превращая *брокер компонентов* (Component Broker) в *общую тестовую конфигурацию* (Shared Fixture, с. 350).

Менее желательным вариантом настройки является чтение имен классов из файла конфигурации. С таким подходом связано несколько проблем. Во-первых, тест должен записывать данные в файл на этапе создания тестовой конфигурации (если нет возможности заменить доступ к файлам другим механизмом). Это гарантированно приведет к появлению *медленных тестов* (Slow Tests, с. 289). Во-вторых, такая схема не будет работать для *настраиваемых тестовых двойников* (Configurable Test Double), если конфигурационный файл не будет содержать данные для инициализации объекта. Наконец, запись в файл открывает двери для *взаимодействующих тестов* (Interacting Tests), так как разные тесты могут использовать разную информацию о конфигурации.

Если *брокер компонентов* (Component Broker) должен возвращать объекты в соответствии с конфигурационными данными, более подходящим решением является использование *минимального объекта* (Humble Object, с. 700), который будет читать данные из файла и вызывать конфигурационный интерфейс *брокера компонентов* (Component Broker). После этого тест сможет использовать этот же интерфейс, чтобы настроить брокер для удовлетворения своих нужд.

Мотивирующий пример

Следующий тест не может работать в приведенном виде.

```
public void testDisplayCurrentTime_AtMidnight() {
    // Настройка тестовой конфигурации
    TimeDisplay sut = new TimeDisplay();
    // Вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // Проверка непосредственного вывода
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals(expectedTimeString, result);
}
```

Этот тест практически всегда завершается неудачно, так как текущее время должно возвращаться в тестируемую систему из вызываемого компонента. Тест не контролирует

возвращаемые значения этого компонента (`DefaultTimeProvider`), поэтому будет завершаться успешно только при запуске ровно в полночь.

```
public String getCurrentTimeAsHtmlFragment() {
    Calendar currentTime;
    try {
        currentTime = new DefaultTimeProvider().getTime();
    } catch (Exception e) {
        return e.getMessage();
    }
    // И т.д.
}
```

Поскольку в тестируемой системе для получения значения времени используется фиксированный класс, вызываемый компонент нельзя заменить *тестовым двойником* (`Test Double`). В результате тест является неопределенным и практически бесполезным. Необходимо обеспечить контроль над опосредованным вводом тестируемой системы.

Замечания по рефакторингу

Первым шагом к тестированию этого поведения является замена фиксированного имени класса вызовом *локатора служб* (`Service Locator`).

```
public String getCurrentTimeAsHtmlFragment() {
    Calendar currentTime;
    try {
        TimeProvider timeProvider =
            (TimeProvider) ServiceLocator.getInstance().
                findService("Time");
        currentTime = timeProvider.getTime();
    } catch (Exception e) {
        return e.getMessage();
    }
    // И т.д.
}
```

Цепочки вызовов методов можно было избежать за счет использования метода класса (через перенос метода `getInstance` на уровень класса). Следующий этап рефакторинга зависит от наличия конфигурационного интерфейса в *локаторе служб* (`Service Locator`). Если конфигурационный интерфейс имеет смысл при нормальном использовании, его можно ввести непосредственно в *локатор служб* (`Service Locator`), как показано в следующем примере. В противном случае можно просто переопределить возвращаемый объект через *связанный с тестом подкласс* (`Test-Specific Subclass`), как показано во втором примере.

Пример: настраиваемый реестр (Configurable Registry)

В данной версии тест модифицирован для использования конфигурационного интерфейса *локатора служб* (`Service Locator`), обеспечивающего установку *тестового двойника* (`Test Double`).

```
public void testDisplayCurrentTime_AtMidnight CSL() {
    // Настройка тестовой конфигурации
    //      Настройка тестового двойника
    MidnightTimeProvider tpStub = new MidnightTimeProvider();
```

```

// Создание экземпляра тестируемой системы
TimeDisplay sut = new TimeDisplay();
// Установка тестового двойника
ServiceLocator.getInstance().registerServiceForName(tpStub, "Time");
// Вызов тестируемой системы
String result = sut.getCurrentTimeAsHtmlFragment();
// Проверка результата
String expectedTimeString =
    "<span class=\"tinyBoldText\">Midnight</span>";
assertEquals("Midnight", expectedTimeString, result);
}

```

Код тестируемой системы был приведен выше. Ниже представлен код *конфигурационного интерфейса* (Configuration Interface) для *настраиваемого реестра* (Configurable Registry).

```

public class ServiceLocator {
    protected ServiceLocator() {};
    protected static ServiceLocator soleInstance = null;
    public static ServiceLocator getInstance() {
        if (soleInstance==null)
            soleInstance = new ServiceLocator();
        return soleInstance;
    }
    private HashMap providers = new HashMap();
    public ServiceProvider findService(String serviceName) {
        return (ServiceProvider) providers.get(serviceName);
    }
    // Конфигурационный интерфейс
    public void registerServiceForName(ServiceProvider provider,
                                      String serviceName) {
        providers.put(serviceName, provider);
    }
}

```

Интересной особенностью данного примера является добавление конфигурационного интерфейса в класс продукта, а не в *тестовый двойник* (Test Double). Фактически *настраиваемый реестр* (Configurable Registry) позволяет избежать использования *тестового двойника* (Test Double), предоставив механизм замены возвращаемого служебного компонента.

Пример: замененный единственный экземпляр класса (Substituted Singleton)

В данной версии теста используется не поддерживающий настройку механизм *поиска зависимости* (Dependency Lookup). Объект soleInstance локатора служб заменяется *заменным единственным экземпляром класса* (Substituted Singleton). Чтобы обеспечить повторное использование конфигурационного интерфейса, в метод overrideSoleInstance в качестве аргумента передается *тестовая заглушка* (Test Stub, с. 544) TimeProvider.

```

public void testDisplayCurrentTime_AtMidnight_TSS() {
    // Настройка тестовой конфигурации
    //      Настройка тестового двойника
    MidnightTimeProvider tpStub = new MidnightTimeProvider();
    // Создание тестируемой системы
}

```

```

TimeDisplay sut = new TimeDisplay();
// Установка тестового двойника
// заменяет целый локатор служб вариантом,
// всегда возвращающим заданную тестовую заглушку
ServiceLocatorTestSingleton.overrideSoleInstance(tpStub);
// Вызов тестируемой системы
String result = sut.getCurrentTimeAsHtmlFragment();
// Проверка результата
String expectedTimeString =
    "<span class=\"tinyBoldText\">Midnight</span>";
assertEquals("Midnight", expectedTimeString, result);
}

```

Обратите внимание, как тест переопределяет объект, который обычно возвращается методом `getInstance`, на экземпляр *связанного с тестом подкласса* (Test-Specific Sub-class). Ниже показан код класса с единственным экземпляром.

```

public class ServiceLocator {
    protected ServiceLocator() {};
    protected static ServiceLocator soleInstance = null;
    public static ServiceLocator getInstance() {
        if (soleInstance==null)
            soleInstance = new ServiceLocator();
        return soleInstance;
    }
    private HashMap providers = new HashMap();
    public ServiceProvider findService(String serviceName) {
        return (ServiceProvider) providers.get(serviceName);
    }
}

```

Конструктор и метод `soleInstance` пришлось квалифицировать как `protected`, а не как `private`, чтобы обеспечить возможность их переопределения в подклассе. Наконец, ниже приведен и код *заменившего единственного экземпляра класса* (Substituted Singleton).

```

public class ServiceLocatorTestSingleton extends ServiceLocator {
    private ServiceProvider tpStub;
    private ServiceLocatorTestSingleton(TimeProvider newTpStub) {
        this.tpStub = newTpStub;
    }
    // Интерфейс установки
    static ServiceLocatorTestSingleton
        overrideSoleInstance(TimeProvider tpStub) {
        // Можно сохранить настоящий экземпляр перед присвоением
        // полю soleInstance для последующего восстановления,
        // но в данном примере этой сложности решено избежать
        soleInstance = new ServiceLocatorTestSingleton( tpStub );
        return (ServiceLocatorTestSingleton) soleInstance;
    }
    // Переопределенный метод суперкласса
    public ServiceProvider findService(String serviceName) {
        return tpStub; // Фиксировано; игнорирует serviceName
    }
}

```

Поскольку код не может видеть закрытый объект `HashMap`, он просто возвращает содержимое поля `tpStub`, сохраненное в конструкторе.

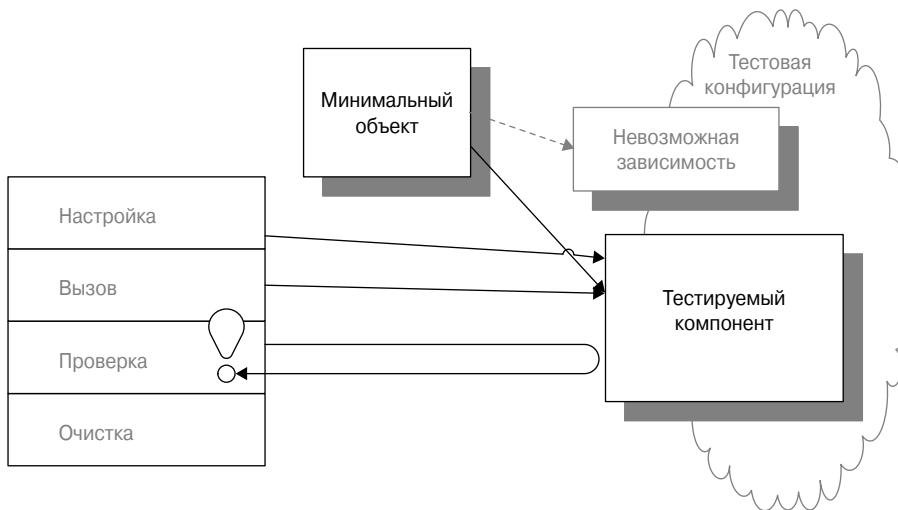
Об имени

Выбор имени для данного шаблона оказался серьезной проблемой. *Локатор служб* (Service Locator) и *брюкер компонентов* (Component Broker) уже широко использовались. Оба хорошо описывали соответствующие сущности. К сожалению, ни один из них не включал в себя второй шаблон. Пришлось придумать другое имя, которое позволило бы объединить два основных варианта. Имя *вставка зависимости* (Dependency Injection) уже использовалось для описания альтернативного шаблона; стремление к единобразию имен подтолкнуло к использованию имени *поиск зависимости* (Dependency Lookup). Дополнительная информация о процессе принятия таких решений приводится во врезке “Что в имени шаблона?” на с. 588.

Минимальный объект (Humble Object)

Как обеспечить тестирование кода, если он тесно связан со средой?

Логика выделяется в отдельный простой в тестировании компонент, который не связан со средой.



Часто возникает необходимость тестировать программное обеспечение, которое тесно связано с некоторой инфраструктурой. В качестве примеров можно назвать визуальные компоненты (например, графические элементы, диалоговые окна) и встраиваемые транзакционные модули. Тестирование таких объектов затруднено, так как создание всех объектов, с которыми взаимодействует тестируемая система, может оказаться слишком дорогим или вообще невозможным. В других случаях объекты могут оказаться сложными в тестировании из-за асинхронного взаимодействия; в качестве примеров можно назвать активные объекты (потоки, процессы, Web-серверы) и пользовательские интерфейсы. Асинхронность таких объектов вносит долю неуверенности, требования к межпроцессной конфигурации и необходимость добавления задержек внутри тестов. Иногда из-за всех этих проблем разработчики просто отказываются тестировать подобный код. Результатом становятся *ошибки в продукте* (Production Bugs, с. 303), вызванные *нетестированным кодом* (Untested Code) и *нетестованными требованиями* (Untested Requirement).

Минимальный объект (Humble Object) позволяет протестировать логику сложно создаваемых объектов без значительных затрат.

Как это работает

Вся логика сложного в тестировании компонента выносится в компонент, поддающийся проверке **синхронными тестами** (synchronous test). В таком компоненте реализуется служебный интерфейс, состоящий из методов, которые обеспечивают доступ к логике нетестируемого компонента. Единственным отличием является доступность этих методов через

синхронные вызовы. В результате компонент *минимального объекта* (*Humble Object*) выступает в роли тонкого слоя адаптера с минимальным объемом кода. При каждом вызове *минимального объекта* (*Humble Object*) со стороны инфраструктуры объект делегирует запрошенные операции тестируемому компоненту. Если тестируемый компонент нуждается в информации из контекста, *минимальный объект* (*Humble Object*) извлекает эту информацию и передает тестируемому компоненту. Код *минимального объекта* (*Humble Object*) обычно настолько простой, что даже не требует создания отдельных тестов.

Когда это использовать

Минимальный объект (*Humble Object*) можно и нужно вводить при наличии нетривиальной логики в компоненте, экземпляр которого сложно создать из-за зависимости от инфраструктуры или доступный только асинхронно. Существует множество причин сложности тестирования объектов. Следовательно есть множество вариантов разрыва этих зависимостей. Следующие варианты являются самыми распространенными примерами *минимальных объектов* (*Humble Object*), но не стоит удивляться, что время от времени придется придумывать собственный вариант.

Вариант: минимальный диалог (*Humble Dialog*)

Инфраструктуры графического интерфейса пользователя требуют предоставления страниц и элементов управления в виде объектов. Такие объекты содержат логику преобразования пользовательских действий в действия основной системы, а ответов системы — в видимое пользователю поведение. Такая логика может подразумевать вызов приложения за пользовательским интерфейсом и/или модификацию состояния одного или нескольких визуальных объектов.

Визуальные объекты плохо поддаются тестированию, так как они тесно связаны с презентационной инфраструктурой. Чтобы эффективно выполнять тестирование, тесту приходится эмулировать среду для предоставления визуальных объектов со всей информацией и необходимыми служебными механизмами. Дополнительная сложность связана с тем, что графическая инфраструктура часто работает в отдельном потоке управления, а значит, тесты должны быть асинхронными. Такие тесты сложно создавать и в результате всегда получаются *медленные* (Slow Tests, с. 289) и *неопределенные тесты* (Nondeterministic Test). В такой ситуации можно попытаться использовать *минимальный объект* (*Humble Object*) для переноса всей логики контроллера и обновления представления из зависящего от инфраструктуры объекта в тестируемый объект.

Вариант: минимальный выполняемый файл (*Humble Executable*)

Многие приложения содержат активные объекты. Активные объекты имеют собственный поток выполнения, позволяющий выполнять операции одновременно с другими действиями системы. К активным объектам относится все, что выполняется в отдельном процессе (например, приложения Windows в файлах .exe) или потоке (в языке Java это любой объект, реализующий интерфейс Runnable). Эти объекты могут запускаться непосредственно клиентом или автоматически, обрабатывая запросы из очереди и отправляя ответы через возвращаемые сообщения. В любом случае для проверки их поведения приходится разрабатывать асинхронные тесты (включая межпроцессную координацию и/или явные задержки со *всегда успешными тестами*, Neverfail Test).

Шаблон *минимальный выполняемый файл* (Humble Executable) позволяет перенести логику выполняемого файла в область применимости теста без формирования задержек, которые в противном случае привели бы к появлению *медленных* (Slow Test) и *неопределенных тестов* (Nondeterministic Test). Логика выполняемого файла выделяется в компонент, который можно тестировать с помощью синхронных тестов. В этом компоненте реализуется служебный интерфейс, состоящий из методов, обеспечивающих выполняемому файлу доступ ко всей логике. Единственным отличием является доступность методов через синхронные вызовы. В качестве тестируемого компонента может выступать динамическая библиотека (DLL) Windows, контейнер Java JAR, содержащий класс *фасада служб* (Service Facade), или другой языковой компонент или класс, предоставляющий доступ к службам выполняемого файла в пригодном для тестирования виде.

Сам компонент *минимального выполняемого файла* (Humble Executable) содержит минимальный объем кода. В его потоке управления только загружается тестируемый компонент (если это *истинный минимальный объект*, True Humble Object) и ему делегируются все операции. В результате *минимальный выполняемый файл* (Humble Executable) требует не более одного-двух тестов для проверки загрузки и/или делегирования функций. Хотя эти тесты и потребуют несколько секунд на выполнение, общее время выполнения набора тестов не увеличится, так как подобных тестов будет очень мало. Учитывая, что код минимального выполняемого файла меняется достаточно редко, такие тесты можно вообще исключить из набора, запускаемого перед включением изменений в хранилище, что позволит еще больше ускорить выполнение тестов. Конечно, тесты *минимального выполняемого файла* (Humble Executable) желательно всегда запускать в процессе автоматической компиляции.

Вариант: минимальный контроллер транзакций (Humble Transaction Controller)

Во многих приложениях для хранения состояния используются базы данных. Настройка тестовой конфигурации в базе данных происходит сложно и долго, а остатки конфигурации могут нарушить работу последующих тестов и даже тех же самых тестов при последующих запусках. При использовании *общей тестовой конфигурации* (Shared Fixture, с. 350) сохранение конфигурации может привести к появлению *нестабильных тестов* (Erratic Test). *Минимальный контроллер транзакций* (Humble Transaction Controller) обеспечивает тестирование логики, которая работает внутри транзакции, предоставляя тесту возможность управлять транзакциями. В результате можно вызывать логику, проверять результат и отменять транзакцию, не оставляя следов в базе данных.

Для реализации *минимального контроллера транзакций* (Humble Transaction Controller) используется рефакторинг *выделение метода* (Extract Method). Вся проверяемая логика переносится из кода управления транзакцией в отдельный метод, который ничего не знает об управлении транзакциями и может вызываться тестом. Поскольку транзакцией управляет вызывающая сторона, тест может начинать, сохранять (при необходимости) и откатывать (чаще всего) транзакцию. В таком случае поведение (а не зависимости) заставляет обходить *минимальный объект* (Humble Object) при тестировании бизнес-логики. А значит, с большой вероятностью можно обойтись *простейшим минимальным объектом* (Poor Man's Humble Object).

Что касается самого *минимального объекта* (Humble Object), то он бизнес-логики не содержит. Таким образом, приходится проверять только правильность сохранения или отката транзакции в зависимости от вызываемых методов. Можно написать тест, который заменя-

ет проверяемый компонент *тестовой заглушкой* (Test Stub, с. 544), генерирующей исключение. После этого тест проверяет откат транзакции в ответ на исключение от заглушки. При использовании *простейшего минимального объекта* (Poor Man's Humble Object) заглушка реализуется с помощью *тестового двойника в виде подкласса* (Subclassed Test Double), определяющего “настоящие” методы на версии, генерирующие исключения.

Большинство основных технологий серверов приложений прямо или опосредованно поддерживают такой шаблон, вынося контроль транзакций из бизнес-объектов. Если программное обеспечение создается без использования инфраструктуры управления транзакциями, может потребоваться реализация собственного *минимального контроллера транзакций* (Humble Transaction Controller). Дополнительная информация о подобном разделении приводится ниже в разделе “Замечания по реализации”.

Вариант: минимальный адаптер контейнера (Humble Container Adapter)

В контексте “контейнеров” часто приходится реализовывать конкретные интерфейсы для запуска объектов внутри сервера приложений (например, интерфейс EJB Session Bean). Еще один вариант *минимального объекта* (Humble Object) предполагает проектирование объектов с учетом независимости от контейнера с последующим использованием *минимального адаптера контейнера* (Humble Container Adapter) для их приспособления к требуемому интерфейсу контейнера. Такая стратегия упрощает тестирование компонентов с логикой за пределами контейнера, что значительно сокращает цикл “редактирование, компиляция, тестирование”.

Замечания по реализации

Существует несколько способов тестирования логики, которая обычно работает внутри *минимального объекта* (Humble Object). Все они имеют общую особенность: доступ к логике открывается для синхронных тестов. Но каждый из вариантов отличается решениями по обеспечению доступа к логике. Независимо от использованного подхода апологеты разработки на основе тестов предпочитают проверять с помощью тестов правильность вызова логики из *минимального объекта* (Humble Object). Это можно сделать, заменив методы с логикой одной из реализаций *тестового двойника* (Test Double, с. 538).

Вариант: простейший минимальный объект (Poor Man's Humble Object)

Самым простым способом изоляции и предоставления доступа к каждому фрагменту проверяемой логики является его размещение в отдельном методе. Для этого можно воспользоваться рефакторингом *выделение метода* (Extract Method) с последующим предоставлением тесту доступа к методу. Конечно, метод не должен зависеть от содержимого контекста. В идеальном случае вся необходимая для работы информация должна предоставляться через аргументы, но вполне допустимо и использование полей экземпляра. Но если тестируемый компонент должен вызывать методы для доступа к необходимой информации, эти методы зависят от (несуществующего и/или подделанного) контекста, возникают проблемы, так как подобная зависимость усложняет процесс написания тестов.

Подобный подход, являющийся основой “простейшего” *минимального объекта* (Humble Object), вполне оправдан, если ничего не мешает создать экземпляр *минимального объекта* (Humble Object) (например, когда поток создается автоматически, а также отсутствуют открытый конструктор и неудовлетворенные зависимости). Разорвать такие зави-

симости можно с помощью *связанного с тестом подкласса* (Test-Specific Subclass), через который тесту становятся доступными конструктор и закрытые методы.

При тестировании *минимального объекта на основе подкласса* (Subclassed Humble Object) или *простейшего минимального объекта* (Poor Man's Humble Object) *тестовый агент* (Test Spy, с. 552) можно создавать как *тестовый двойник в виде подкласса* (Subclassed Test Double) минимального объекта. Этот агент будет записывать вызываемые методы, которые будут проверяться с помощью утверждений внутри *тестового метода* (Test Method, с. 378).

Вариант: истинный минимальный объект (True Humble Object)

В диаметрально противоположном случае можно выделить тестируемую логику в отдельный класс и заставить *минимальный объект* (Humble Object) делегировать запросы его экземпляру. Подобный подход подразумевался в описании данного шаблона. Он работает практически во всех ситуациях при наличии полного контроля над кодом.

Иногда инфраструктура требует, чтобы на объект возлагались обязанности, которые нельзя перенести в другое место. Например, инфраструктура графического интерфейса ожидает, что в объектах представления хранятся данные для элементов графического интерфейса, а также данные, отображаемые на экране. В таком случае тестируемый объект должен содержать ссылку на *минимальный объект* (Humble Object) для управления его данными или в *минимальной объект* (Humble Object) придется включить логику обновления, не покрытую автоматизированными тестами. Первый вариант практически всегда возможен и является более предпочтительным.

Переработка в направлении *истинного минимального объекта* (True Humble Object) обычно состоит из последовательного применения рефакторинга *выделение метода* (Extract Method) для разделения открытого интерфейса *минимального объекта* (Humble Object) и делегируемой логики реализации. После этого с помощью рефакторинга *выделение класса* (Extract Class) все методы (кроме публичного интерфейса минимального объекта) выносятся в новый “тестируемый” класс. Затем добавляется атрибут (поле), в котором хранится ссылка на экземпляр нового класса. Присвоение экземпляра атрибуту происходит в конструкторе или в каждом методе интерфейса с помощью механизма “ленивой” инициализации.

При тестировании *истинного минимального объекта* (True Humble Object) (когда *минимальный объект* (Humble Object) делегирует все операции отдельному классу), для проверки правильности вызова выделенного класса обычно используется *ленивый подставной объект* (Lazy Mock Object) или *тестовый агент* (Test Spy). Использование активного *подставного объекта* (Mock Object) в такой ситуации может привести к проблемам, так как утверждения проверяются не в том потоке, в котором работает *объект теста* (Testcase Object, с. 410), поэтому ложные утверждения не будут обнаруживаться, пока отсутствует механизм их передачи в поток теста.

Чтобы обеспечить правильное создание экземпляра тестируемого компонента для создания выделенного компонента, можно воспользоваться наблюдаемой *фабрикой объектов* (Object Factory). Тест может зарегистрироваться слушателем и наблюдать за правильностью вызова методов фабрики. Кроме того, можно воспользоваться нормальным объектом фабрики, а на время тестирования заменять его *подставным объектом* (Mock Object) или *тестовой заглушкой* (Test Stub), чтобы контролировать правильность вызова методов фабрики.

Вариант: минимальный объект на основе подкласса (Subclassed Humble Object)

Между крайними случаями *простейшего минимального объекта* (Poor Man's Humble Object) и *истинного минимального объекта* (True Humble Object) существуют подходы, основанные на подклассах, выносящих логику в другие классы, но оставляющие ее в пределах одного объекта. Выбор вариантов реализации зависит от необходимости наследования конкретного класса инфраструктуры классом *минимального объекта* (Humble Object). Здесь этот вариант рассматриваться не будет, так как техника очень зависит от используемого языка программирования и среды выполнения. Несмотря на это необходимо знать основные варианты.

1. Зависящий от инфраструктуры класс должен наследовать тестируемую логику от суперкласса.
2. Класс должен делегировать вызов абстрактному методу, реализованному в подклассе.

Мотивирующий пример на основе минимального выполняемого файла (Humble Executable)

В этом примере проверяется логика, работающая в собственном потоке и обрабатывающая запросы по мере поступления. В каждом тесте создается поток, отправляются сообщения и формируется задержка, обеспечивающая истинность утверждений. К сожалению, для создания и инициализации потока, а также для обработки первого запроса требуется несколько секунд. Таким образом, если после создания потока не вставить задержку в две секунды, тест случайным образом завершится неудачно.

```
public class RequestHandlerThreadTest extends TestCase {
    private static final int TWO_SECONDS = 3000;
    public void testWasInitialized_Async()
        throws InterruptedException {
        // Настройка
        RequestHandlerThread sut = new RequestHandlerThread();
        // Вызов
        sut.start();
        // Проверка
        Thread.sleep(TWO_SECONDS);
        assertTrue(sut.initializedSuccessfully());
    }
    public void testHandleOneRequest_Async()
        throws InterruptedException {
        // Настройка
        RequestHandlerThread sut = new RequestHandlerThread();
        sut.start();
        // Вызов
        enqueueRequest(makeSimpleRequest());
        // Проверка
        Thread.sleep(TWO_SECONDS);
        assertEquals(1, sut.getNumberOfRequestsCompleted());
        assertEquals(makeSimpleResponse(), sut.getResponse());
    }
}
```

В идеальном случае поток должен проверяться на каждой операции отдельно. Это позволит достичь максимально точной локализации дефектов (Defect Localization, с. 78). К сожалению, в этом случае набор тестов будет выполняться несколько минут, так как в каждом teste присутствует задержка на несколько секунд. Еще одна проблема связана с невозможностью неудачного завершения, если активный объект генерирует исключение в собственном потоке.

Двухсекундная задержка не кажется чем-то значительным, но представьте, что таких тестов десяток. В таком случае набор тестов будет выполнятся полминуты. Сравните такое быстродействие с быстродействием обычных тестов, несколько сотен которых выполняется в течение секунды. Тестирование с использованием выполняемого файла значительно снижает производительность разработчика. Для полноты картины ниже показан код выполняемого файла.

```
public class RequestHandlerThread extends Thread {
    private boolean _initializationCompleted = false;
    private int _numberOfRequests = 0;
    public void run() {
        initializeThread();
        processRequestsForever();
    }
    public boolean initializedSuccessfully() {
        return _initializationCompleted;
    }
    void processRequestsForever() {
        Request request = nextMessage();
        do {
            Response response = processOneRequest(request);
            if (response != null) {
                putMsgOntoOutputQueue(response);
            }
            request = nextMessage();
        } while (request != null);
    }
}
```

Для простоты бизнес-логика уже была выделена в метод `processOneRequest`. Здесь также не показана фактическая логика инициализации. Достаточно сказать, что в случае успешной инициализации устанавливается переменная `_initializationCompleted`.

Замечания по рефакторингу

Для создания *простейшего минимального объекта* (Poor Man's Humble Object) тесту предоставляется доступ к методам. (Если логика скрыта внутри метода, сначала необходимо воспользоваться рефакторингом *выделение метода* (Extract Method).) Если контекст содержит зависимости, придется использовать рефакторинг *введение параметра* (Introduce parameter) или *введение поля* (Introduce field), чтобы методу `processOneRequest` не пришлось обращаться к информации из контекста.

Для создания *истинного минимального объекта* (True Humble Object) можно использовать рефакторинг *выделение класса* (Extract Class) по отношению к выполняемому файлу для выделения тестируемого компонента. В результате рефакторинга от выполняемого файла останется пустая оболочка, которая будет играть роль *минимального объекта*.

(Humble Object). Обычно в процессе применяется рефакторинг *выделение метода* (Extract Method), позволяющий разделить тестируемую логику (например, методы `initializeThread` и `processOneRequest`) и логику взаимодействия с контекстом выполняемого файла. После этого с помощью рефакторинга *выделение класса* (Extract Class) создается класс тестируемого компонента (фактически единственный объект Strategy) и в него переносятся все методы, кроме открытых методов интерфейса. В процессе рефакторинга обеспечивается создание экземпляра класса и добавляется поле для хранения ссылки на созданный экземпляр. Кроме того, необходимо исправить все открытые методы, чтобы они вызывали методы нового тестируемого класса.

Пример: простейший минимальный объект (Poor Man's Humble Object)

Ниже показан тот же тест с использованием *простейшего минимального объекта* (Poor Man's Humble Object).

```
public void testWasInitialized_Sync()
    throws InterruptedException {
    // Настройка
    RequestHandlerThread sut = new RequestHandlerThread();
    // Вызов
    sut.initializeThread();
    // Проверка
    assertTrue(sut.initializedSuccessfully());
}

public void testHandleOneRequest_Sync()
    throws InterruptedException {
    // Настройка
    RequestHandlerThread sut = new RequestHandlerThread();
    // Вызов
    Response response = sut.processOneRequest(makeSimpleRequest());
    // Проверка
    assertEquals(1, sut.getNumberOfRequestsCompleted());
    assertEquals(makeSimpleResponse(), response);
}
```

В данном случае методы `initializeThread` и `processOneRequest` являются открытыми, что позволяет синхронно вызывать их из теста. Обратите внимание на отсутствие задержки. Такой подход допустим, пока существует простая возможность создать экземпляр выполняемого компонента.

Пример: истинный минимальный выполняемый файл (True Humble Executable)

В данном случае код тестируемой системы переработан для использования *истинного минимального выполняемого файла* (True Humble Executable).

```
public class HumbleRequestHandlerThread extends Thread
    implements Runnable {
    public RequestHandler requestHandler;
    public HumbleRequestHandlerThread() {
        super();
        requestHandler = new RequestHandlerImpl();
    }
    public void run() {
```

```

    requestHandler.initializeThread();
    processRequestsForever();
}
public boolean initializedSuccessfully() {
    return requestHandler.initializedSuccessfully();
}
public void processRequestsForever() {
    Request request = nextMessage();
    do {
        Response response =
            requestHandler.processOneRequest(request);
        if (response != null) {
            putMsgOntoOutputQueue(response);
        }
        request = nextMessage();
    } while (request != null);
}

```

Здесь метод `processOneRequest` вынесен в отдельный класс, экземпляр которого просто создать. Ниже показан тот же тест, но переписанный для использования выделенного компонента. Обратите внимание на отсутствие задержки.

```

public void testNotInitialized_Sync()
throws InterruptedException {
    // Настройка/вызов
    RequestHandler sut = new RequestHandlerImpl();
    // Проверка
    assertFalse("init", sut.initializedSuccessfully());
}
public void testWasInitialized_Sync()
throws InterruptedException {
    // Настройка
    RequestHandler sut = new RequestHandlerImpl();
    // Вызов
    sut.initializeThread();
    // Проверка
    assertTrue("init", sut.initializedSuccessfully());
}
public void testHandleOneRequest_Sync()
throws InterruptedException {
    // Настройка
    RequestHandler sut = new RequestHandlerImpl();
    // Вызов
    Response response = sut.processOneRequest(makeSimpleRequest());
    // Проверка
    assertEquals(1, sut.getNumberOfRequestsDone());
    assertEquals(makeSimpleResponse(), response);
}

```

Поскольку вызовы делегируются другому объекту, возможно, придется проверить правильность делегирования. Следующий тест проверяет правильность инициализации метода `initializeThread` и `processOneRequest` из *минимального объекта* (*Humble Object*).

```

public void testLogicCalled_Sync()
throws InterruptedException {
    // Настройка

```

```

RequestHandlerRecordingStub mockHandler =
    new RequestHandlerRecordingStub();
HumbleRequestHandlerThread sut = new HumbleRequestHandlerThread();
// Установка подставного объекта
sut.setHandler(mockHandler);
sut.start();
// Вызов
enqueueRequest(makeSimpleRequest());
// Проверка
Thread.sleep(TWO_SECONDS);
assertTrue("init", mockHandler.initializedSuccessfully());
assertEquals(1, mockHandler.getNumberOfRequestsDone());
}

```

Обратите внимание, что данному тесту требуется небольшая задержка для запуска нового потока. Но она короче, так как настоящий компонент с логикой заменен *тестовым двойником* (Test Double), который отвечает немедленно и в задержке нуждается только сам тест. Этот тест даже можно вынести в отдельный набор, который запускается реже остальных (например, только во время автоматической компиляции), что позволит быстро выполнять все тесты перед включением изменений в общее хранилище кода.

Еще одной особенностью является использование *тестового агента* (Test Spy) вместо *подставного объекта* (Mock Object). Поскольку утверждения *подставного объекта* (Mock Object) проверялись бы в новом потоке, поток *тестового метода* (Test Method), а значит, и *инфраструктура автоматизации тестов* (Test Automation Framework) (в данном случае — JUnit) не получили бы информации о ложных утверждениях. В результате тест завершался бы успешно даже с ложными утверждениями внутри *подставного объекта* (Mock Object). Проверяя утверждения внутри *тестового метода* (Test Method), можно отказаться от специального механизма передачи исключений из *подставного объекта* (Mock Object) в поток, в котором работает *тестовый метод* (Test Method).

В предыдущем teste проверялась правильность делегирования из *минимального объекта* (Humble Object) в установленный *тестовый агент* (Test Spy). Кроме того, желательно проверить правильность инициализации переменной, в которой содержится ссылка на класс для делегирования. Ниже показан простой вариант такой проверки.

```

public void testConstructor() {
    // Вызов
    HumbleRequestHandlerThread sut = new HumbleRequestHandlerThread();
    // Проверка
    String actualDelegateClass = sut.requestHandler.getClass().getName();
    assertEquals(RequestHandlerImpl.class.getName(),
                 actualDelegateClass);
}

```

Тест конструктора (Constructor Test) проверяет инициализацию конкретного атрибута.

Пример: минимальный диалог (Humble Dialog)

Во многих средах разработки поддерживается создание пользовательского интерфейса визуальными средствами с помощью перетаскивания различных объектов (“элементов управления”) в пустое окно. После размещения элементам управления можно назначать поведение, выбирая одно из возможных действий или событий, допустимых для данного объекта, и вводя реализацию логики в окно кода, предоставленное средой разработки.

Данная логика может обращаться к приложению, скрытому за пользовательским интерфейсом, или модифицировать состояние этого или других визуальных объектов.

Эффективное тестирование визуальных объектов значительно затруднено, так как они тесно связаны с презентационной инфраструктурой. Для предоставления визуальному объекту всех необходимых данных тесту придется моделировать подобную среду, что само по себе является сложной задачей. Эти тесты могут стать настолько сложными, что многие разработчики даже не будут пытаться тестировать презентационную логику. Не удивительно, что недостаток тестирования часто приводит к появлению *ошибок в продукте* (Production Bugs), являющихся результатом *нетестированных требований* (Untested Requirements).

Для создания *минимального диалога* (Humble Dialog) сначала вся логика компонента представления выделяется в невизуальный компонент, поддерживающий тестирование через синхронные тесты. Если от компонента требуется обновление визуального компонента (*минимального диалога*, Humble Dialog), он передается в виде аргумента. При тестировании невизуальных компонентов *минимальный диалог* (Humble Dialog) обычно заменяется *подставным объектом* (Mock Object), который настроен на подходящий опосредованный ввод и ожидаемое поведение (опосредованный вывод). В графических инфраструктурах, требующих от *минимального диалога* (Humble Dialog) регистрации для получения возникающих событий, невизуальный компонент может зарегистрироваться вместо *минимального диалога* (Humble Dialog) (если это не приводит к появлению неуправляемых зависимостей от контекста). Подобная гибкость позволяет сделать *минимальный диалог* (Humble Dialog) еще проще, так как события попадают непосредственно в невизуальный компонент и не требуют дополнительной логики делегирования.

Следующий пример взят из визуального компонента Visual Basic (.ct1), содержащего нетривиальную логику. Это часть собственного встраиваемого модуля, который был создан для утилиты TestDirector компании *Mercury Interactive*.

```
' Метод интерфейса, TestDirector вызывает этот метод
' для вывода результата
Public Sub ShowResultEx(TestSetKey As TdTestSetKey, _
    TSTestKey As TdTestKey,
    ResultKey As TdResultKey)
    Dim RpbFiles As OcsRpbFiles
    Set RpbFiles = getTestResultFileNames(ResultKey)
    ResultsFileName = RpbFiles.ActualResultFileName
    ShowFileInBrowser ResultsFileName
End Sub

Function getTestResultFileNames(ResultKey As Variant) As OcsRpbFiles
    On Error GoTo Error
    Dim Attachments As Collection
    Dim thisTest As Run
    Dim RpbFiles As New OcsRpbFiles
    Call EnsureConnectedToTd
    Set Attachments = testManager.GetAllAttachmentsOfRunTest(ResultKey)
    Call RpbFiles.LoadFromCollection(Attachments, "RunTest")
    Set getTestResultFileNames = RpbFiles
    Exit Function
Error:
    ' сделать что-то
End Function
```

Эту логику необходимо протестировать. К сожалению, передаваемые в качестве параметров объекты невозможно создать, так как они не имеют открытых конструкторов. Передать объекты другого типа также невозможно, поскольку типы параметров функций вписаны в конкретные классы.

С помощью рефакторинга *выделение тестируемого компонента* (Extract Testable Component) на основе выполняемого файла можно создать тестируемый компонент, оставив только пустую оболочку *минимального диалога* (Humble Dialog). Обычно такой подход предполагает последовательное применение рефакторинга *выделение метода* (Extract Method) (в исходном примере это преобразование уже выполнено для простоты понимания) к каждому элементу перемещаемой логики. После этого рефакторинг *выделение класса* (Extract Class) позволяет создать новый класс тестируемого компонента. Вместе с рефакторингом *выделение класса* (Extract Class) могут использоваться *перемещение метода* (Move Method) и *перемещение поля* (Move Field) для переноса необходимых логики и данных из *минимального диалога* (Humble Dialog) в новый тестируемый компонент.

Ниже показано то же самое представление после преобразования в *минимальный диалог* (Humble Dialog).

```
' Метод интерфейса, TestDirector вызывает этот метод
' для вывода результатов
Public Sub ShowResultEx(TestSetKey As TdTestSetKey, _
    TSTestKey As TdTestKey, _
    ResultKey As TdResultKey)
    Dim RpbFiles As OcsRpbFiles
    Call EnsureImplExists
    Set RpbFiles = Implementation.getTestResultFileNames(ResultKey)
    ResultsFileName = RpbFiles.ActualResultFileName
    ShowFileInBrowser ResultsFileName
End Sub
Private Sub EnsureImplExists()
    If Implementation Is Nothing Then
        Set Implementation = New OcsScriptViewerImpl
    End If
End Sub
```

Ниже показан тестируемый компонент OcsScriptViewerImpl, который вызывается *минимальным объектом* (Humble Object).

```
' Реализация ResultViewer:
Public Function getTestResultFileNames(ResultKey As Variant) As
OcsRpbFiles
    On Error GoTo Error
    Dim Attachments As Collection
    Dim thisTest As Run
    Dim RpbFiles As New OcsRpbFiles
    Call EnsureConnectedToTd
    Set Attachments = testManager.GetAllAttachmentsOfRunTest(ResultKey)
    Call RpbFiles.LoadFromCollection(Attachments, "RunTest")
    Set getTestResultFileNames = RpbFiles
    Exit Function
Error:
    ' сделать что-то
End Function
```

После этого можно просто создать экземпляр класса `OcsScriptViewerImpl` и написать для него тесты `VbUnit`. Тесты здесь не показаны для экономии места, так как в них нет ничего интересного.

Пример: минимальный контроллер транзакций (Humble Transaction Controller)

В разделе, посвященном очистке откатом транзакции (*Transaction Rollback Tear-down*, с. 675), приведен пример теста, обходящего *минимальный контроллер транзакций* (*Humble Transaction Controller*).

Источник дополнительной информации

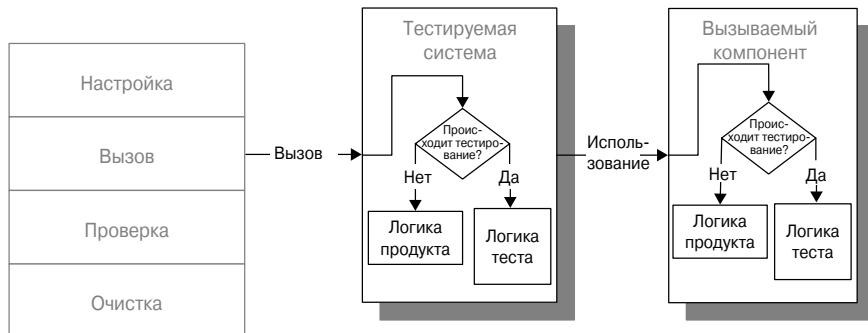
Оригинальная статья Майкла Фезерса о *минимальных диалогах* (*Humble Dialog*) доступна по следующему адресу:

<http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf>

Ловушка для теста (Test Hook)

Как проектировать test-ируемую систему для замены зависимостей во время выполнения?

Тестируемая система модифицируется для изменения поведения во время теста.



Практически каждый элемент кода зависит от других классов, объектов, модулей или процедур. Для правильного модульного тестирования элемента кода его необходимо изолировать от зависимостей. Такая изоляция может оказаться сложной, если зависимости использованы в коде в виде фиксированных имен классов.

Ловушка для теста (Test Hook) является “последним решением” по вставке относящегося к тесту поведения на время автоматизированного тестирования.

Как это работает

Через добавление “ловушки” непосредственно в тестируемую систему или вызываемый компонент поведение тестируемой системы модифицируется в соответствии с потребностями тестов. Такой подход предполагает существование некоторого флага `testing`, который проверяется в соответствующих местах кода.

Когда это использовать

Иногда приходится прибегать к “шаблону крайних мер”, если не удается воспользоваться *вставкой зависимости* (Dependency Injection, с. 684) или *поиском зависимости* (Dependency Lookup, с. 692). В такой ситуации *ловушка для теста* (Test Hook) используется из-за недоступности других способов избавиться от *нетестированного кода* (Untested Code), вызванного *фиксированной зависимостью* (Hard-Coded Dependency).

Ловушка для теста (Test Hook) может оказаться единственным способом вставки поведения *тестового двойника* (Test Double, с. 538) при разработке на процедурном языке без поддержки объектов, указателей на функцию или другого варианта динамического связывания.

Ловушки для теста (Test Hook) могут использоваться в качестве переходной стратегии по интеграции тестов в существующий код. Сначала тесты интегрируются через *ловушки для теста* (Test Hook), а затем полученный набор тестов используется в качестве *страховочной*

сети (Safety Net) для рефакторинга и интеграции более корректных тестов. На некотором этапе придется отказаться от начальных тестов, требующих применения ловушек для теста (Test Hook), так как “современных” тестов будет достаточно для полноценной проверки.

Замечания по реализации

Суть ловушки для теста (Test Hook) заключается во вставке кода в тестируемую систему. Независимо от способа добавления, код может:

- передавать управление *тестовому двойнику* (Test Double) вместо настоящего объекта;
- выступать в качестве *тестового двойника* (Test Double) внутри настоящего объекта;
- выступать в роли декоратора, делегирующего запросы настоящему объекту.

Выступающий в качестве индикатора тестирования флаг может устанавливаться как константа на этапе компиляции, что может вынудить компилятор к оптимизации путем удаления всей тестовой логики. В языках с поддержкой препроцессоров или макросов компилятора такие константы могут использоваться для удаления ловушек для теста (Test Hook) перед передачей кода в промышленную эксплуатацию. Кроме того, значение флага может указываться в конфигурационных данных или храниться в непосредственно устанавливаемой тестом глобальной переменной.

Мотивирующий пример

Следующий тест не может завершиться успешно без модификаций.

```
public void testDisplayCurrentTime_AtMidnight() {
    // настройка тестовой конфигурации
    TimeDisplay sut = new TimeDisplay();
    // вызов тестируемой системы
    String result = sut.getCurrentTimeAsHtmlFragment();
    // проверка непосредственного вывода
    String expectedTimeString =
        "<span class=\"tinyBoldText\">Midnight</span>";
    assertEquals(expectedTimeString, result);
}
```

Данный тест практически всегда завершается неудачно, так как он зависит от вызываемого компонента, который возвращает тестируемой системе текущее время. Тест не управляет возвращаемыми значениями компонента DefaultTimeProvider и в результате завершается успешно только при запуске ровно в полночь.

```
public String getCurrentTimeAsHtmlFragment() {
    Calendar currentTime;
    try {
        currentTime = new DefaultTimeProvider().getTime();
    } catch (Exception e) {
        return e.getMessage();
    }
    // И т.д.
}
```

Поскольку для получения значения времени в тестируемой системе используется фиксированный класс, заменить вызываемый компонент *тестовым двойником* (Test

`Double`) не возможно. В результате тест является неопределенным и практически бесполезным. Необходимо найти способ получения контроля над опосредованным вводом тестируемой системы.

Замечания по рефакторингу

Можно создать флаг, который проверяется тестируемой системой, тем самым интегрировав в нее *ловушку для теста* (Test Hook). После этого проверяемый код заключается в структуру управления `if/then/else` и относящаяся к тесту логика размещается в разделе `then`.

Пример: ловушка для теста в тестируемой системе

Ниже показан код продукта после модификации для внедрения *ловушки для теста* (Test Hook).

```
public String getCurrentTimeAsHtmlFragment() {
    Calendar theTime;
    try {
        if (TESTING) {
            theTime = new GregorianCalendar();
            theTime.set(Calendar.HOUR_OF_DAY, 0);
            theTime.set(Calendar.MINUTE, 0);
        } else {
            theTime = new DefaultTimeProvider().getTime();
        }
    } catch (Exception e) {
        return e.getMessage();
    }
    // И т.д.
```

В данном случае флаг тестирования реализован в виде глобальной константы, которая редактируется при необходимости. Подобная гибкость подразумевает компиляцию отдельной версии системы для тестирования. Такая стратегия немного безопаснее, чем динамически настраиваемый параметр или переменная, так как многие компиляторы просто удаляют эту ловушку из кода объекта.

Пример: ловушка для теста в вызываемом компоненте

Ловушку для теста (Test Hook) можно внедрить в вызываемый компонент, а не в тестируемую систему.

```
public Calendar getTime() throws TimeProviderEx {
    Calendar theTime = new GregorianCalendar();
    if (TESTING) {
        theTime.set(Calendar.HOUR_OF_DAY, 0);
        theTime.set(Calendar.MINUTE, 0);
    } else {
        // Просто вернуть календарь
    }
    return theTime;
};
```

Такое решение немного лучше, так как тестируемая система не модифицируется в процессе проверки.

Глава 27

Шаблоны значений

Шаблоны в этой главе:

Точное значение (Literal Value)	718
Вычисляемое значение (Derived Value)	722
Сгенерированное значение (Generated Value).....	726
Объект-заглушка (Dummy Object)	730

Точное значение (Literal Value)

Также известен как:

Фиксированное значение (Hard-Coded Value), Константное значение (Constant Value)

Как описать значения, используемые в тестах?

Для атрибутов объектов и утверждений используются точные значения.

```
BigDecimal expectedTotal = new BigDecimal("99.95");
```

Значения атрибутов объектов в тестовой конфигурации и ожидаемый результат выполнения тестов часто связаны друг с другом в соответствии с требованиями, предъявляемыми к тестируемой системе. Установить правильные значения (и в частности, правильное отношение между пред- и постусловиями) очень важно, так как это обеспечивает правильное поведение тестируемой системы.

Точные значения (Literal Value) достаточно часто используются для описания атрибутов объектов в пределах теста.

Как это работает

Точные константы соответствующего типа присваиваются каждому атрибуту объекта или используются в качестве аргументов вызовов методов тестируемой системы либо *методов с утверждением* (Assertion Method, с. 390). Ожидаемые значения вычисляются вручную, на калькуляторе или с помощью электронной таблицы, после чего записываются в код теста в виде *точного значения* (Literal Value).

Когда это использовать

Применение *точного значения* (Literal Value) в коде теста делает используемое значение очевидным; не остается никаких сомнений в значении, так как оно вписано в код. К сожалению, с помощью *точных значений* (Literal Value) можно скрыть отношение между значениями в разных местах тестов, что, в свою очередь, приводит к появлению *непонятных тестов* (Obscure Test, с. 230). Точные значения (Literal Value) имеет смысл использовать, если они явно указаны в требованиях к тесту. (Иногда, чтобы избежать ошибок копирования значений в тест, лучше воспользоваться *управляемыми данными тестами* (Data-Driven Test, с. 322).)

Одним из недостатков использования *точных значений* (Literal Value) является возможность присвоения одного и того же значения двум не относящимся один к другому атрибутам. Если тестируемая система воспользуется неправильным атрибутом, тест может завершиться успешно, даже если этого не должно было произойти. Если в качестве *точного значения* (Literal Value) используется имя файла или ключ доступа к базе данных, смысл значения теряется — на самом деле поведением системы управляет содержимое файла или запись в базе данных. Использование *точного значения* (Literal Value) в качестве ключа ни о чем не говорит читателю теста и в результате приводит к появлению *непонятного теста* (Obscure Test).

Если значение в описании ожидаемого результата можно вычислить на основе значений в логике настройки конфигурации, использование *вычисляемого значения* (Derived Value, с. 722) значительно повышает ценность *тестов как документации* (Tests as Docu-

mentation, с. 79). С другой стороны, если значения не важны для описания тестируемой логики, лучше воспользоваться *сгенерированными значениями* (Generated Value, с. 726).

Замечания по реализации

Чаще всего *точные значения* (Literal Value) используются в качестве констант в коде. Если одно и то же значение должно использоваться в нескольких местах теста (обычно при настройке конфигурации и проверке результата), такой подход может скрыть отношение между пред- и постусловиями теста. Введя символьную константу с описательным именем, можно сделать отношение более явным. Точно так, если очевидное значение использовать нельзя, код можно сделать понятнее, определив именованную символьную константу и использовав ее везде, где использовалось *точное значение* (Literal Value).

Вариант: символьная константа (Symbolic Constant)

Если *точное значение* (Literal Value) должно использоваться в нескольких местах одного *тестового метода* (Test Method, с. 378) или в нескольких разных тестах, желательно вместо *точного значения* (Literal Value) использовать *символьную константу* (Symbolic Constant). *Символьная константа* (Symbolic Constant) является функциональным эквивалентом *точного значения* (Literal Value), но сокращает вероятность *высокой стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300).

Вариант: описательное значение (Self-Describing Value)

Если нескольким атрибутам объекта необходимо присвоить похожие значения, то, используя разные значения, можно доказать, что тестируемая система обрабатывает правильный атрибут. Если атрибут или аргумент имеет строковый тип и не накладывает ограничений на присваиваемую строку, можно воспользоваться значением, которое описывает свою роль в teste. Например, строка “несуществующий клиент” в качестве имени клиента может сделать тест более понятным, чем простое “Joe Blow”, особенно в процессе отладки или при выводе значений атрибутов в журнал неудачных завершений тестов.

Пример: точное значение (Literal Value)

Поскольку *точное значение* (Literal Value) обычно выступает в роли отправной точки при написании тестов, мотивирующий пример здесь опущен и сразу приведен пример использования *точного значения* (Literal Value) в обычном teste. Обратите внимание, что *точные значения* (Literal Value) используются как в логике настройки тестовой конфигурации, так и в утверждениях.

```
public void testAddItemQuantity_1() throws Exception {
    Product product = new Product("Widget", 19.95);
    Invoice invoice = new Invoice();
    // Вызов
    invoice.addItemQuantity(product, 1);
    // Проверка
    List lineItems = invoice.getLineItems();
    LineItem actualItem = (LineItem)lineItems.get(0);
    assertEquals(new BigDecimal("19.95"),
                actualItem.getExtendedPrice());
}
```

Конструктору объекта `Product` нужно знать имя и стоимость. Утверждение относительно атрибута `extendedCost` объекта `lineItem` требует общей стоимости продукта. В этом примере значения указаны в виде фиксированных констант. В следующем примере используются символьные константы.

Замечания по рефакторингу

С помощью рефакторинга замена магического числа символьской константой (Replace Magic Number with Symbolic Constant) можно сократить дублирование тестового кода (Test Code Duplication, с. 254), которое проявляется в многократном использовании точного значения (Literal Value) 19.95.

Пример: символьная константа (Symbolic Constant)

В переработанной версии исходного теста дублированное точное значение (Literal Value) цены изделия (19.95) заменяется соответствующим образом именованной символьной константой (Symbolic Constant), которая используется во время настройки тестовой конфигурации и при проверке результата.

```
public void testAddItemQuantity_1s() throws Exception {
    BigDecimal widgetPrice = new BigDecimal("19.95");
    Product product = new Product("Widget", widgetPrice);
    Invoice invoice = new Invoice();
    // Вызов
    invoice.addItemQuantity(product, 1);
    // Проверка
    List lineItems = invoice.getLineItems();
    LineItem actualItem = (LineItem)lineItems.get(0);
    assertEquals(widgetPrice, actualItem.getExtendedPrice());
}
```

Пример: описательное значение (Self-Describing Value)

В данной версии теста описательное значение (Self-Describing Value) используется для аргумента обязательного имени, который передается в конструктор `Product`. Это значение не используется тестируемым методом. Оно просто хранится для последующего доступа со стороны другого метода, который здесь не проверяется.

```
public void testAddItemQuantity_1b() throws Exception {
    BigDecimal widgetPrice = new BigDecimal("19.95");
    Product product = new Product("Irrelevant product name",
        widgetPrice);
    Invoice invoice = new Invoice();
    // Вызов
    invoice.addItemQuantity(product, 1);
    // Проверка
    List lineItems = invoice.getLineItems();
    LineItem actualItem = (LineItem)lineItems.get(0);
    assertEquals(widgetPrice, actualItem.getExtendedPrice());
}
```

Пример: отдельное значение (Distinct Value)

Данный тест проверяет, получено ли имя элемента из имени продукта. В качестве имени используются *отдельное значение* (Distinct Value) и строка “SKU”, что позволяет разделить имя на элементы.

```
public void testAddItemQuantity_1c() throws Exception {  
    BigDecimal widgetPrice = new BigDecimal("19.95");  
    String name = "Product name";  
    String sku = "Product SKU";  
    Product product = new Product(name, sku, widgetPrice);  
    Invoice invoice = new Invoice();  
    // Вызов  
    invoice.addItemQuantity(product, 1);  
    // Проверка  
    List lineItems = invoice.getLineItems();  
    LineItem actualItem = (LineItem)lineItems.get(0);  
    assertEquals(name, actualItem.getName());  
}
```

Также это один из примеров *описательного значения* (Self-Describing Value).

Вычисляемое значение (Derived Value)

Также известный как Вычисляемое значение <i>(Calculated Value)</i>	Как описать значения, используемые в тестах? Одни значения выражаются в виде вычислений на основе других значений.
--	---

```
BigDecimal expectedTotal = itemPrice.multiply(QUANTITY);
```

Значения атрибутов объектов в тестовой конфигурации и ожидаемый результат тестирования часто связаны друг с другом в соответствии с требованиями, предъявляемыми к тестируемой системе. Установить правильные значения (и в частности, правильное отношение между пред- и постусловиями) очень важно, так как это обеспечивает правильное поведение тестируемой системы.

Часто одни значения можно вычислить на основе других значений в пределах того же теста. В таком случае повышается ценность *тестов как документации* (Tests as Documentation, с. 79), поскольку через вычисление выражений показана связь между значениями.

Как это работает

Компьютеры хорошо справляются с вычислением значений и составлением строк. Можно избежать вычисления в уме (или на калькуляторе), если реализовать вычисление ожидаемых результатов непосредственно в teste. Кроме того, *вычисляемое значение* (Derived Value) может использоваться в качестве аргумента при создании объектов тестовой конфигурации и в качестве аргумента методов тестируемой системы.

Природа *вычисляемого значения* (Derived Value) требует использования переменных или символьных констант для хранения значения. Переменные и/или константы можно инициализировать на этапе компиляции (константы), в процессе инициализации класса или *объекта теста* (Testcase Object, с. 410), а также на этапе создания тестовой конфигурации или в теле *тестового метода* (Test Method, с. 378).

Когда это использовать

Вычисляемое значение (Derived Value) должно использоваться каждый раз, когда одни значения можно определенным образом вычислить на основе других значений в пределах теста. Основным недостатком использования *вычисляемых значений* (Derived Value) является одновременное появление одной и той же математической ошибки (например, округления) в teste и в тестируемой системе. Чтобы защититься от таких ошибок, некоторые патологические случаи можно реализовать с помощью *точных значений* (Literal Value, с. 718). Если используемые значения должны быть уникальны или не влияют на логику тестируемой системы, лучше остановиться на использовании *сгенерированных значений* (Generated Value, с. 726).

Вычисляемое значение (Derived Value) может использоваться в процессе создания тестовой конфигурации (*вычисленные входные данные*, Derived Input, или *единственный дефектный атрибут*, One Bad Attribute) или при определении ожидаемых результатов для сравнения с выводом тестируемой системы (*вычисленное ожидание*, Derived Expectation). Ниже приводится более подробная информация.

Вариант: вычисляемые входные данные (Derived Input)

Иногда тестовая конфигурация содержит похожие значения, которые тестируемая система может сравнивать и основывать свою логику на разнице между ними. Например, *вычисляемые входные данные* (Derived Input) могут формироваться на этапе создания конфигурации через добавление разницы к базовому значению. Такая операция делает явным отношение между двумя значениями. Добавляемое значение можно даже указать в виде символьной константы с описательным именем, например MAXIMUM_ALLOWABLE_TIME_DIFFERENCE.

Вариант: единственный дефектный атрибут (One Bad Attribute)

Вычисляемые входные данные (Derived Input) часто используются для проверки методов, принимающих в качестве аргумента сложный объект. Например, для полноценного тестиования “проверки ввода” нужно поочередно вызывать метод с объектами, атрибутам которых присвоены заведомо некорректные значения. Тест проверяет реакцию метода на каждое изменение. Поскольку первое некорректное значение приводит к завершению работы метода, каждый из некорректных атрибутов необходимо проверять в отдельном вызове тестируемой системы. Каждый из вызовов должен выполняться в пределах отдельного тестового метода, который является *тестом одного условия* (Single Condition Test, с. 99). Для создания некорректного объекта достаточно сначала создать правильный объект, а затем заменить один из атрибутов некорректным значением. Правильный объект лучше создавать с помощью *метода создания* (Creation Method, с. 441), что позволит избежать *дублирования тестового кода* (Test Code Duplication, с. 254).

Вариант: вычисленное ожидание (Derived Expectation)

Если возвращаемое тестируемой системой значение должно быть связано с одним или несколькими аргументами тестируемой системы или конфигурации, ожидаемое значение можно вычислить на основе входных параметров в процессе работы теста без *точных значений* (Literal Value). Результат можно использовать в качестве ожидаемого значения в *утверждении равенства* (Equality Assertion).

Мотивирующий пример

В следующем teste *вычисляемые значения* (Derived Value) не используются. Обратите внимание на *точные значения* (Literal Value) в логике создания тестовой конфигурации и в утверждениях.

```
public void testAddItemQuantity_2a() throws Exception {
    BigDecimal widgetPrice = new BigDecimal("19.99");
    Product product = new Product("Widget", widgetPrice);
    Invoice invoice = new Invoice();
    // Вызов
    invoice.addItemQuantity(product, 5);
    // Проверка
    List lineItems = invoice.getLineItems();
    LineItem actualItem = (LineItem)lineItems.get(0);
    assertEquals(new BigDecimal("99.95"),
                actualItem.getExtendedPrice());
}
```

Чтобы установить соотношения между значениями конфигурации и проверяемым результатом, читателю придется несколько вычислений выполнить в уме.

Замечания по рефакторингу

Для простоты чтения теста фактически вычисляемые *точные значения* (Literal Value) можно заменить формулами их расчета.

Пример: вычисленное ожидание (Derived Expectation)

В исходном примере содержался только один элемент для пяти экземпляров продукта. Таким образом, ожидаемое значение общей цены вычислялось как произведение цены элемента на количество, что делало связь между значениями более явной.

```
public void testAddItemQuantity_2b() throws Exception {
    BigDecimal widgetPrice = new BigDecimal("19.99");
    BigDecimal numberOfUnits = new BigDecimal("5");
    Product product = new Product("Widget", widgetPrice);
    Invoice invoice = new Invoice();
    // Вызов
    invoice.addItemQuantity(product, numberOfUnits);
    // Проверка
    List lineItems = invoice.getLineItems();
    LineItem actualItem = (LineItem)lineItems.get(0);
    BigDecimal totalPrice = widgetPrice.multiply(numberOfUnits);
    assertEquals(totalPrice, actualItem.getExtendedPrice());
}
```

Обратите внимание, что для цены элемента и количества введены символьные константы, что делает смысл выражения еще более очевидным и снижает затраты на модификацию значений в дальнейшем.

Пример: единственный дефектный атрибут (One Bad Attribute)

Предположим, что существует следующий метод фабрики, который в качестве аргумента принимает объект *CustomerDto*. Необходимо создать тесты, которые проверяют поведение при передаче некорректных атрибутов объекта *CustomerDto*. Можно создавать объект *CustomerDto* внутри *тестового метода* (Test Method) и там же инициализировать один из аргументов некорректным значением.

```
public void testCreateCustomerFromDto_BadCredit() {
    // Настройка тестовой конфигурации
    CustomerDto customerDto = new CustomerDto();
    customerDto.firstName = "xxx";
    customerDto.lastName = "yyy";
    // И т.д.
    customerDto.address = createValidAddress();
    customerDto.creditRating = CreditRating.JUNK;
    // Вызов тестируемой системы
    try {
        sut.createCustomerFromDto(customerDto);
        fail("Expected an exception");
    } catch (InvalidInputException e) {
```

```

        assertEquals("Field", "Credit", e.field);
    }
}
public void testCreateCustomerFromDto_NullAddress() {
    // Настройка тестовой конфигурации
    CustomerDto customerDto = new CustomerDto();
    customerDto.firstName = "xxx";
    customerDto.lastName = "yyy";
    // И т.д.
    customerDto.address = null;
    customerDto.creditRating = CreditRating.AAA;
    // вызов тестируемой системы
    try {
        sut.createCustomerFromDto(customerDto);
        fail("Expected an exception");
    } catch (InvalidInputException e) {
        assertEquals("Field", "Address", e.field);
    }
}

```

Очевидным недостатком этого примера является *дублирование тестового кода* (Test Code Duplication), поскольку для каждого атрибута создается как минимум один тест. При инкрементной разработке проблема становится еще серьезнее: не только потребуются новые тесты для каждого нового атрибута, но и придется возвращаться к уже существующим тестам и добавлять новый атрибут в сигнатуру метода фабрики.

Правильным решением является определение *метода создания* (Creation Method) правильного экземпляра CustomerDto с последующим вызовом метода создания из каждого теста. После этого каждый тест содержит объект с *единственным дефектным атрибутом* (One Bad Attribute) (своим для каждого теста).

```

public void testCreateCustomerFromDto_BadCredit_OBA() {
    CustomerDto customerDto = createValidCustomerDto();
    customerDto.creditRating = CreditRating.JUNK;
    try {
        sut.createCustomerFromDto(customerDto);
        fail("Expected an exception");
    } catch (InvalidInputException e) {
        assertEquals("Field", "Credit", e.field);
    }
}
public void testCreateCustomerFromDto_NullAddress_OBA() {
    CustomerDto customerDto = createValidCustomerDto();
    customerDto.address = null;
    try {
        sut.createCustomerFromDto(customerDto);
        fail("Expected an exception");
    } catch (InvalidInputException e) {
        assertEquals("Field", "Address", e.field);
    }
}

```

Сгенерированное значение (Generated Value)

Как описать значения, используемые в тестах?

Подходящее значение генерируется при каждом запуске теста.

```
BigDecimal uniqueCustomerNumber = getUniqueNumber();
```

При инициализации объектов в тестовой конфигурации необходимо учесть, что большинство объектов имеют атрибуты (поля), значения которых передаются в качестве аргументов конструктора. Иногда конкретные значения влияют на результат работы теста. Но чаще достаточно, чтобы каждый объект использовал свое значение. Если конкретные значения тесту не нужны, важно, чтобы они не фигурировали в коде теста!

Сгенерированные значения (Generated Value) используются совместно с *методами создания* (Creation Method, с. 441) для вынесения потенциально отвлекающей информации из кода теста.

Как это работает

Вместо определения на этапе написания теста используемые значения генерируются во время выполнения теста. При этом можно выбирать значения, которые соответствуют конкретным критериям, например “уникальность в пределах базы данных”, которые имеют смысл только на этапе выполнения.

Когда это использовать

Сгенерированные значения (Generated Value) должны использоваться каждый раз, когда нельзя или нежелательно указывать значения до запуска теста. Возможно, значение атрибута не влияет на результат запуска теста и у разработчика нет желания определять *точное значение* (Literal Value, с. 718), а в некоторых случаях определенное свойство атрибута можно определить только на этапе выполнения. Иногда тестируемая система требует, чтобы значение атрибута было уникальным. С помощью *сгенерированных значений* (Generated Value) можно удовлетворить этот критерий и избежать появления *неповторяющихся тестов* (Unrepeatable Test) и “войн” запуска тестов (Test Run War) за счет сокращения вероятности конфликтов при параллельном запуске. При необходимости уникальное значение можно использовать для всех атрибутов объекта, что значительно упростит идентификацию объекта при просмотре с помощью отладчика.

Стоит обратить внимание, что разные значения позволяют выявить разные ошибки. Например, число из одной цифры может форматироваться правильно, а число из нескольких цифр — нет, и наоборот. Сгенерированные значения могут стать причиной появления *неопределенных тестов* (Nondeterministic Test). Если выявлена неопределенность (иногда тест завершается успешно, а при следующем запуске — неудачно), необходимо проверить код тестируемой системы и определить, не могут ли различные значения стать основной причиной проблемы.

Обычно не рекомендуется использовать *сгенерированные значения* (Generated Value), если значение не должно быть уникальным, так как это приводит к неопределенности. Очевидной альтернативой является использование *точного значения* (Literal Value). Ме-

нее очевидной альтернативой является применение *вычисляемого значения* (Derived Value, с. 722), особенно если ожидаемый результат определяется в самом тесте.

Замечания по реализации

Существует несколько способов генерации значений. Выбор подходящего способа зависит от ситуации.

Вариант: отдельное сгенерированное значение (Distinct Generated Value)

Если каждый тест или объект должен использовать уникальное значение, можно воспользоваться *отдельным сгенерированным значением* (Distinct Generated Value). В таком случае будет создан набор вспомогательных функций, которые возвращают уникальные значения различных типов (например, целые, строки, числа с плавающей точкой). Различные методы `getUnique` можно построить на основе генератора последовательности целых чисел. Для чисел, уникальных в пределах совместно используемой базы данных, можно воспользоваться последовательностями базы данных или таблицей последовательности. Для чисел, которые должны сохранять уникальность в пределах запуска теста, можно воспользоваться хранящимся в памяти генератором последовательности (например, статической переменной Java, которая увеличивается перед каждым использованием). Хранящаяся в памяти последовательность чисел, которая начинается с 1, при каждом запуске набора тестов имеет одно полезное качество: для каждого теста генерируются одинаковые значения при каждом запуске, что заметно упрощает отладку.

Вариант: случайное сгенерированное значение (Random Generated Value)

Одним из способов получения хорошего покрытия тестами без длительного анализа поведения и генерации тестовых условий является использование различных значений при каждом запуске тестов. Одним из решений на пути к этой цели является использование *случайных сгенерированных значений* (Random Generated Value). Хотя такой подход может показаться хорошей идеей, тесты становятся неопределенными, что значительно усложняет отладку неудачно завершившихся тестов. В идеальном случае при неудачном завершении теста должна иметься возможность повторить неудачное завершение. Для этого *случайное сгенерированное значение* (Random Generated Value) можно занести в журнал работы теста и показать вместе с отчетом о неудачном завершении. После этого тест необходимо вынуть еще раз воспользоваться этим значением, пока разработчик занимается отладкой. В ольшинстве случаев затраченные усилия перевешивают потенциальную выгоду. Конечно, если это действительно нужно, данный способ применять необходимо.

Вариант: связанное сгенерированное значение (Related Generated Value)

Один из возможных вариантов этого шаблона предполагает комбинирование *сгенерированного значения* (Generated Value) и *вычисляемого значения* (Derived Value) за счет использования одного и того же сгенерированного целого числа в качестве основы для всех атрибутов одного объекта. Для достижения такого результата необходимо один раз вызвать `getUniqueInt` и воспользоваться полученным значением для создания уникальных строк, чисел с плавающей точкой и других значений. При использовании *связанного сгенерированного значения* (Related Generated Value) все поля объекта содержат “связанные” данные, что

упрощает обнаружение объекта при отладке. Еще одним вариантом реализации является разделение генерации корневого значения и генерации остальных значений с помощью явного вызова сначала `generateNewUniqueRoot` и только после этого — `getUniqueInt`, `getUniqueString` и т.д.

Для строк имеет смысл пересыпать в функцию **описывающий роль аргумент** (role-describing argument), который комбинируется с уникальным ключом в виде целого числа, что позволяет проще описать намерение теста. Хотя такие аргументы можно передавать и в другие функции, их встраивание в целочисленные аргументы невозможно.

Мотивирующий пример

В следующем teste в качестве аргументов конструктора используются *точные значения* (Literal Value).

```
public void testProductPrice_HCV() {
    // Настройка
    Product product =
        new Product(88,                               // ID
                    "Widget",                         // Название
                    new BigDecimal("19.99")); // Цена
    // Вызов
    // ...
}
```

Замечания по рефакторингу

Тест можно переработать для использования *отдельного сгенерированного значения* (Distinct Generated Value), заменив *точные значения* (Literal Value) вызовами соответствующего метода `getUnique`. Эти методы просто увеличивают значение счетчика при каждом вызове и используют его в качестве корня для создания значения подходящего типа.

Пример: отдельное сгенерированное значение (Distinct Generated Value)

Ниже показан тот же тест, переписанный для использования *отдельного сгенерированного значения* (Distinct Generated Value). В метод `getUniqueString` передается строка, описывающая роль значения (“Widget Name”).

```
public void testProductPrice_DVG() {
    // Настройка
    Product product =
        new Product(getUniqueInt(),           // ID
                    getUniqueString("Widget"), // Имя
                    getUniqueBigDecimal()); // Цена
    // Вызов
    // ...
}
static int counter = 0;
int getUniqueInt() {
    counter++;
    return counter;
}
BigDecimal getUniqueBigDecimal() {
```

```

        return new BigDecimal(getUniqueInt());
    }
String getUniqueString(String baseName) {
    return baseName.concat(String.valueOf(getUniqueInt()));
}

```

В данном тесте для аргументов конструктора используются разные значения. Генерированные таким образом числа оказываются последовательными, но для получения общего представления читатель теста должен обратить внимание на конкретный атрибут. Скорее всего, не стоит генерировать значение цены, если тестируемая логика связана с его вычислением, так как это заставит логику проверки приспособливаться к разным общим суммам.

Пример: связанное сгенерированное значение (Related Generated Value)

Разделив генерацию корневого значения и создание отдельных значений, можно сделать очевидной связь между всеми значениями, которые используются в teste. В следующем примере генерация корневого значения вынесена в метод `setUp`, что позволяет каждому тесту получать корневое значение только один раз. Возвращающий разные значения метод (например, `getUniqueString`) просто использует ранее сгенерированный корень для генерации нового значения.

```

public void testProductPrice_DRVG() {
    // Настройка
    Product product =
        new Product(getUniqueInt(),           // ID
                    getUniqueString("Widget"), // Имя
                    getUniqueBigDecimal()); // Цена
    // Вызов
    // ...
}

static int counter = 0;
public void setUp() {
    counter++;
}
int getUniqueInt() {
    return counter;
}
String getUniqueString(String baseName) {
    return baseName.concat(String.valueOf(counter));
}
BigDecimal getUniqueBigDecimal() {
    return new BigDecimal(counter);
}

```

Если рассмотреть этот объект через инспектор объектов, дамп базы данных или при сохранении части объекта в журнал, его можно быстро идентифицировать независимо от рассматриваемого поля.

Объект-заглушка (Dummy Object)

Также известен как:

Кукла (*Dummy*), Параметр-заглушка (*Dummy Parameter*), Значение-заглушка (*Dummy Value*), Заменитель (*Placeholder*), Заглушка (*Stub*)

Как указывать используемые в тесте значения, если единственное их назначение — применяться в качестве ненужных аргументов метода тестируемой системы?

В качестве аргумента вызываемого метода тестируемой системы передается объект без реализации.

```
Invoice inv = new Invoice(new DummyCustomer());
```

Перевод тестируемой системы в подходящее для теста состояние часто требует вызова и других ее методов. Обычно такие методы в качестве аргументов принимают объекты, сохраняемые в переменных экземпляра для дальнейшего использования. Часто полученные объекты (или как минимум некоторые из их атрибутов) никогда не используются в тестируемом коде. Вместо этого они создаются исключительно для соответствия сигнатуре метода, который вызывается для перевода тестируемой системы в необходимое для теста состояние. Создание таких объектов может оказаться нетривиальной задачей и усложнить тест.

В таких случаях в качестве аргумента передается *объект заглушки* (*Dummy Object*), что избавляет от необходимости создавать настоящий объект.

Как это работает

Сначала создается экземпляр класса, позволяющего быстро создать экземпляр без дополнительных зависимостей. После этого полученный экземпляр используется в качестве аргумента метода тестируемой системы. Поскольку объект в самой системе не используется, реализация ему не требуется. В случае вызова методов *объекта-заглушки* (*Dummy Object*) тест должен завершаться с ошибкой, которая обычно возникает при попытке вызова несуществующего метода.

Когда это использовать

Объект-заглушка (*Dummy Object*) может использоваться каждый раз, когда объекты передаются в качестве атрибутов других объектов или аргументов методов тестируемой системы. Он позволяет избежать появления *непонятных тестов* (*Obscure Test*, с. 230), исключив не относящийся к тесту код, который в противном случае потребовался бы для создания настоящего объекта. В результате читателю теста становится ясно, какие объекты и значения не используются тестируемой системой.

Для того чтобы управлять опосредованным вводом или проверять опосредованный вывод тестируемой системы, скорее всего придется использовать *тестовую заглушку* (*Test Stub*, с. 544) или *подставной объект* (*Mock Object*, с. 558). Если объект используется тестируемой системой, но нет возможности предоставить настоящий объект, стоит рассмотреть вариант создания *поддельного объекта* (*Fake Object*, с. 565), который предоставляет достаточное подмножество поведения для успешного завершения теста.

Если тестируемая система все-таки использует объект, можно воспользоваться одним из шаблонов значений. В зависимости от ситуации может подойти *точное значение* (Literal Value, с. 718), *сгенерированное значение* (Generated Value, с. 726) или *вычисляемое значение* (Derived Value, с. 722).

Вариант: аргумент-заглушка (Dummy Argument)

Аргумент-заглушка (Dummy Argument) может использоваться каждый раз, когда методы тестируемой системы принимают объекты в качестве аргументов, но эти объекты не имеют значения в пределах теста¹.

Вариант: атрибут заглушки (Dummy Attribute)

Атрибут-заглушка (Dummy Attribute) может использоваться при создании объектов, которые являются частью тестовой конфигурации или выступают в роли аргументов методов тестируемой системы, но некоторые атрибуты этих объектов не имеют значения для теста.

Замечания по реализации

Самая простая реализация *объекта-заглушки* (Dummy Object) заключается в передаче значения `null` в качестве аргумента. Такой подход работает даже в статически типизированных языках, например в Java (если вызываемый метод не проверяет, имеют ли аргументы значение `null`). Если метод выводит сообщение об ошибке при передаче значения `null` в качестве аргумента, потребуется немного более сложная реализации. Наибольшим недостатком использования значения `null` является его недостаточная описательная нагрузка.

В динамически типизированных языках (например, в Ruby, Perl и Python) фактический тип объекта никогда не проверяется (так как он никогда не используется), поэтому можно воспользоваться любым классом, например `String` или `Object`. В таком случае объекту желательно присвоить *описательное значение* (Self-Describing Value), например “*Dummy Customer*”.

В статически типизированных языках (например, в Java, C# и C++) *объект-заглушка* (Dummy Object) должен быть совместим по типу с соответствующим параметром. Значительно проще достигнуть совместимости типов, если параметр имеет абстрактный тип (например, `Interface` в языке Java), поскольку в таком случае можно создать собственную тривиальную реализацию типа или передать подходящий *псевдообъект* (Pseudo-Object). Если в качестве типа параметра выступает конкретный класс, можно попытаться создать тривиальный экземпляр или экземпляр *связанного с тестом подкласса* (Test-Specific Subclass, с. 591) внутри теста.

¹ Информация с сайта Wikipedia. Параметры также часто называются аргументами, хотя аргументы правильнее воспринимать как фактические значения или ссылки, присвоенные переменным параметров при вызове подпрограммы во время выполнения. При рассмотрении кода, вызывающего подпрограмму, передаваемые в подпрограмму значения и ссылки являются аргументами, а место перечисления этих значений или ссылок называется списком параметров. При рассмотрении кода внутри определения подпрограммы переменные из списка параметров подпрограммы называются параметрами, в то время как значения параметров во время выполнения являются аргументами.

В некоторых инфраструктурах *подставных объектов* (Mock Object) существуют *вспомогательные методы теста* (Test Utility Method, с. 610), которые генерируют *объект-заглушку* (Dummy Object) для указанного класса. Сгенерированный объект принимает аргумент типа *String* для создания *описательного значения* (Self-Describing Value).

Хотя *объект-заглушка* (Dummy Object) может представлять собой значение *null*, он отличается от Null Object [PLOPD3]. *Объект-заглушка* (Dummy Object) не используется тестируемой системой, поэтому его поведение не имеет значения или при попытке использования должно генерироваться исключение. С другой стороны, Null Object используется тестируемой системой, но предназначен для “ничегонеделания”. Небольшое, но очень важное отличие!

Мотивирующий пример

В приведенном ниже примере тестируется класс *Invoice*, но для создания его экземпляра требуется объект *Customer*. Объекту *Customer* нужен объект *Address*, которому, в свою очередь, необходим объект *City*. Таким образом, только для настройки тестовой конфигурации создается несколько дополнительных объектов. Но если тестируемое поведение вообще не пытается получить доступ к объекту *Customer*, зачем создавать его и объекты, от которых он зависит?

```
public void testInvoice_addLineItem_noECS() {
    final int QUANTITY = 1;
    Product product = new Product(getUniqueNumberAsString(),
                                   getUniqueNumber());
    State state = new State("West Dakota", "WD");
    City city = new City("Centreville", state);
    Address address = new Address("123 Blake St.", city, "12345");
    Customer customer= new Customer(getUniqueNumberAsString(),
                                      getUniqueNumberAsString(),
                                      address);
    Invoice inv = new Invoice(customer);
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    assertLineItemsEqual("",expItem, actual);
}
```

В результате создания нескольких дополнительных объектов тест получился несколько запутанным. Как проверяемое поведение относится к объектам *Address* и *City*? Анализируя код теста, можно подумать, что какая-то связь все-таки существует. Но это просто заблуждение!

Замечания по рефакторингу

Если объекты тестовой конфигурации не имеют прямого отношения к тесту, в пределах теста они не должны быть видны. Таким образом, нужно избавиться от необходимости создавать эти объекты. Можно попытаться вместо объекта *Customer* передать зна-

чение `null`. В таком случае конструктор проверит полученный аргумент и откажется его принимать, поскольку он равен `null`. Поэтому придется искать другой способ.

Правильное решение предполагает замену неважного объекта *объектом-заглушки* (*Dummy Object*). В динамически типизированных языках достаточно передать строку. В статически типизированных языках, например в Java и C#, передается совместимый по типу объект. В такой ситуации по отношению к объекту *Customer* применяется рефакторинг *выделение интерфейса* (*Extract Interface*) [Ref], что позволяет создать новый интерфейс и новый класс реализации, который называется *DummyCustomer*. Конечно, в процессе рефакторинга необходимо заменить все ссылки на объект *Customer* новым именем интерфейса. Способ с минимальной модификацией кода заключается в использовании *связанного с тестом подкласса* (*Test-Specific Subclass*) для класса *Customer*, в котором добавляется удобный для теста конструктор.

Пример: значения-заглушки (*Dummy Value*) и объекты-заглушки (*Dummy Object*)

Ниже показан тот же тест, но использующий *объект-заглушку* (*Dummy Object*) вместо имени продукта и объекта *Customer*. Обратите внимание, насколько упростилось создание тестовой конфигурации!

```
public void testInvoice_addLineItem_DO() {
    final int QUANTITY = 1;
    Product product = new Product("Dummy Product Name",
                                   getUniqueNumber());
    Invoice inv = new Invoice(new DummyCustomer());
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    // Вызов
    inv.addItemQuantity(product, QUANTITY);
    // Проверка
    List lineItems = inv.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actual = (LineItem)lineItems.get(0);
    assertLineItemsEqual("", expItem, actual);
}
```

Достаточно просто использовать *объект-заглушку* (*Dummy Object*) в качестве имени продукта, так как это строка без требования уникальности. Таким образом, удалось воспользоваться *описательным значением* (*Self-Describing Value*). Воспользоваться *объектом-заглушки* (*Dummy Object*) в качестве номера продукта не удалось, так как номер должен быть уникальным, поэтому в качестве номера использовано *сгенерированное значение* (*Generated Value*). С объектом *Customer* ситуация немного сложнее, так как конструктор объекта *LineItem* ожидает отличного от `null` объекта. Поскольку данный пример написан на языке Java, параметр метода является сильно типизированным. По этой причине пришлось создать альтернативную реализацию интерфейса *ICustomer* с конструктором без аргументов, что позволило упростить создание объекта в пределах теста. Поскольку объект *DummyCustomer* никогда не используется, он создается на месте, а не сохраняется в переменной. Это сокращает код создания тестовой конфигурации на одну строку, а присутствие вызова конструктора в вызове конструктора *Invoice* дополнительно показывает, что *объект-заглушка* (*Dummy Object*) нужен только для конструктора. Ниже показан код класса *DummyCustomer*.

```
public class DummyCustomer implements ICustomer {  
    public DummyCustomer() {  
        // Все просто; нечего инициализировать!  
    }  
    public int getZone() {  
        throw new RuntimeException("This should never be called!");  
    }  
}
```

Здесь был реализован класс `DummyCustomer`, методы которого соответствуют интерфейсу. Поскольку каждый метод генерирует исключение, легко определить, какой из них был вызван. Вместо `DummyCustomer` можно было воспользоваться *псевдообъектом* (Pseudo-Object). В других ситуациях можно было просто передать значение `null` или создать простейший экземпляр настоящего класса. Основным недостатком второго варианта является невозможность определить, был ли использован *объект-заглушка* (Dummy Object).

Источники дополнительной информации

Когда в книге [UTwJ] делаются ссылки на *объект-заглушку* (Dummy Object), имеется в виду описанная здесь *тестовая заглушка* (Test Stub). Более подробно терминология тестовых двойников сравнивается в приложении Б. В инфраструктурах JMock и NMock поддерживается автоматическая генерация *объектов-заглушек* (Dummy Object) для тестирования с использованием *подставных объектов* (Mock Object).

Часть IV

Приложения

Приложение А

Рефакторинг тестов

Выделение тестируемого компонента (Extract Testable Component)

Необходимо разработать простой способ тестирования логики, но компонент слишком тесно связан с контекстом, что делает тестирование невозможным.

Следует выделить тестируемую логику в отдельный компонент, который специально спроектирован для облегчения тестирования и не зависит от контекста, в котором запускается.

Также известен как:

Отращиваемый класс (Sprout Class) [WEwLC]

Замечания по реализации

Логика из нетестируемого компонента выделяется в компонент, доступный для синхронных тестов. Новый компонент не связан со своим контекстом. Обычно это означает, что все необходимые элементы контекста извлекаются прежним компонентом и передаются новому компоненту в качестве аргументов тестируемых методов или конструкторов. В результате исходный компонент практически не содержит кода и может считаться *минимальным объектом* (Humble Object, с. 700). Он просто извлекает из контекста информацию, необходимую тестируемому компоненту. Все взаимодействие с новым компонентом выполняется через синхронные вызовы методов.

Тестируемый компонент может представлять собой библиотеку Windows DLL или Java JAR, содержащую класс *фасада служб* (Service Facade) [CJ2EPP] или другой языковой компонент, предоставляющий службы выполняемого файла методам тестов. Нетестируемый код может быть выполняемым файлом, диалоговым окном или другим презентационным компонентом, логикой внутри транзакции или даже сложным методом теста. После выделения тестируемого компонента должен оставаться *минимальный объект* (Humble Object), который не требует тестирования.

В зависимости от природы нетестируемого компонента может потребоваться создание тестов для логики делегирования, но иногда это совершенно невозможно из-за тесной связанныности логики с контекстом. Если создать тесты можно, потребуется только пара тестов, проверяющих правильность создания экземпляра и делегирования. Поскольку код будет меняться не слишком часто, эти тесты менее критичны и даже могут

быть исключены из набора тестов, запускаемого разработчиками перед включением кода в общее хранилище. Конечно, эти тесты имеет смысл всегда запускать во время автоматической компиляции.

Источники дополнительной информации

Этот рефакторинг напоминает *выделение интерфейса* (Extract Interface) и *выделение реализатора* (Extract Implementer), но *выделение тестируемого компонента* (Extract Testable Component) не требует сохранения интерфейса. Также его можно рассматривать как специальный случай рефакторинга *выделение класса* (Extract Class).

Встраивание ресурса (In-line Resource)

Зависящие от невидимого внешнего ресурса тесты создают проблему таинственного гостя (Mystery Guest).

Необходимо перенести содержимое внешнего ресурса в логику создания тестовой конфигурации.

Из [RTC]:

Для устранения зависимости тестового метода от внешнего ресурса данный ресурс встраивается в код теста. Это достигается путем создания тестовой конфигурации, содержащей те же данные, что и ресурс. Затем созданная тестовая конфигурация используется вместо ресурса. Простым примером такого рефакторинга является помещение содержимого файла в строку тестового кода.

Если объем содержимого ресурса слишком велик, высока вероятность того, что рассматривается “энергичный” тест (Eager Test; см. Рулетка утверждений, Assertion Roulette, с. 264). В таком случае рассмотрите использование выделения метода (Extract Method) и минимизации данных (Minimize Data, с. 739).

Замечания по реализации

Проблема тестов, зависящих от внешнего ресурса, заключается в невозможности контролировать предварительные условия работы теста. В качестве ресурса может выступать файл в файловой системе, содержимое базы данных или другой объект, создаваемый за пределами теста. Ни одна из этих *предварительно созданных тестовых конфигураций* (Prebuilt Fixture) не видна читателю теста. Сделайте ее видимой, встроив ресурс в тест. Самый простой способ сделать это — создать ресурс средствами самого теста. Например, файл можно записать, а не просто сделать ссылку на существующий файл. Если удалить файл в конце теста, можно перейти от использования *предварительно созданной тестовой конфигурации* (Prebuilt Fixture) к применению *постоянной новой тестовой конфигурации* (Persistent Fresh Fixture; см. Новая тестовая конфигурация, Fresh Fixture, с. 344). В результате тесты будут работать немного дольше.

Более прогрессивным способом встраивания внешнего ресурса является замена реального ресурса *тестовой заглушкой* (Test Stub, с. 544), инициализируемой из теста. Таким образом, читатель может видеть содержимое ресурса. Когда тестируемая система выполняется, она использует *тестовую заглушку* (Test Stub) вместо реального ресурса.

Еще одним решением является рефакторинг тестируемой системы для упрощения написания тестов. К фрагменту тестируемой системы, использующему содержимое ресурса, можно применить рефакторинг *выделение тестируемого компонента* (Extract Testable Component, с. 737). В результате фрагмент можно будет тестировать непосредственно без доступа к внешнему ресурсу, т.е. тест будет передавать содержимое ресурса использующей его логике. Кроме того, можно проверить *минимальный объект* (Humble Object, с. 700), который читает содержимое ресурса независимо, заменяя выделенный компонент *тестовой заглушкой* (Test Stub) или *подставным объектом* (Mock Object, с. 558).

Создание уникального ресурса (Make Resource Unique)

Некоторые тесты случайно создают или используют один и тот же ресурс в общей тестовой конфигурации (Shared Fixture).

Сделайте уникальными используемые тестом имена ресурсов.

Из [RTC]:

Пересекающиеся имена ресурсов становятся причиной множества проблем. Это может случаться как в разных тестах, запущенных одним пользователем, так и при одновременном запуске теста разными пользователями.

Такие проблемы легко предотвратить (или исправить) с помощью уникальных идентификаторов всех выделяемых ресурсов (для обеспечения уникальности можно использовать метку времени). Если в идентификатор можно включить и имя теста, отвечающего за выделения ресурса, значительно упростится поиск теста, который не освобождает свои ресурсы.

Замечания по реализации

Имя любого используемого ресурса необходимо сделать уникальным в пределах всех тестов с помощью *отдельного сгенерированного значения* (Distinct Generated Value; см. *Сгенерированное значение*, Generated Value, с. 726) как части имени. В идеальном случае имя ресурса должно включать в себя имя теста, которому “принадлежит” ресурс. Во избежание появления *взаимодействующих тестов* (Interacting Tests; см. *Нестабильный тест*, Erratic Test, с. 267) в имя каждого созданного тестом ресурса включается метка времени. Для удаления ресурса используется *автоматическая очистка* (Automated Teardown, с. 521).

Минимизация данных (Minimize Data)

Тестовая конфигурация слишком большая и препятствует пониманию теста.

Необходимо удалять элементы тестовой конфигурации, пока не будет получена минимальная тестовая конфигурация (Minimal Fixture).

Из [RTC]:

Уменьшайте объем данных в тестовой конфигурации до тех пор, пока не останутся только абсолютно необходимые. Это даст два преимущества: тесты будут более пригодны для использования в качестве документации и менее чувствительны к изменениям.

Замечания по реализации

Сократив объем данных в тестовой конфигурации до абсолютного минимума, можно получить *минимальную тестовую конфигурацию* (Minimal Fixture, с. 336). В результате можно будет использовать *тесты как документацию* (Tests as Documentation, с. 79). Способ реализации зависит от организации *тестовых методов* (Test Method, с. 378) в *классы теста* (Testcase Class, с. 401).

Если *тестовые методы* (Test Method) организованы в виде *класса теста для каждой тестовой конфигурации* (Testcase Class per Fixture, с. 639) и есть подозрение, что создана *тестовая конфигурация общего характера* (General Fixture; см. *Непонятный тест*, Obscure Test, с. 230), можно удалить логику создания для всех элементов тестовой конфигурации, которые предположительно не используются тестами. Лучше удалять элементы логики постепенно, чтобы при отказе теста можно было вернуть последнее изменение и проверить тест еще раз.

Если *тестовые методы* (Test Method) организованы в виде *класса теста для каждой функции* (Testcase Class per Feature, с. 633) или *класса теста для каждого класса* (Testcase Class per Class, с. 627), рефакторинг *минимизация данных* (Minimize Data) может потребовать копирование логики создания тестовой конфигурации из метода `setUp`, принадлежащего классу теста (Testcase Class) или *декоратору настройки* (Setup Decorator, с. 471), в каждый тест, которому нужна эта конфигурация. Предположив, что коллекция объектов в виде *общей тестовой конфигурации* (Shared Fixture, с. 350) является слишком большой для любого отдельно взятого теста, можно последовательно применять рефакторинг *выделение метода* (Extract Method) для создания набора *методов создания* (Creation Method, с. 441), которые впоследствии будут вызываться тестами. После этого вызовы *методов создания* (Creation Method) удаляются из метода `setUp` и помещаются только в те *тестовые методы* (Test Method), которые нуждаются в соответствующем элементе исходной тестовой конфигурации. Последним штрихом будет преобразование всех переменных, хранящих тестовую конфигурацию, в локальные переменные.

Замена зависимости тестовым двойником (Replace Dependency with Test Double)

Зависимости *тестируемого объекта* *препятствуют запуску тестов.*

**Устраните зависимость, заменив вызываемый компонент
тестовым двойником (Test Double).**

Замечания по реализации

Первым шагом является выбор варианта замены зависимости. *Вставка зависимости* (Dependency Injection, с. 684) является наилучшим вариантом для модульных тестов, а *поиск зависимости* (Dependency Lookup, с. 692) часто неплохо решает задачу в приемочных тестах. После этого тестируемая система переделывается для поддержки такого выбора или проектирование этой функциональности выполняется в процессе разработки на основе тестов. Следующим решением является выбор между *поддельным объектом* (Fake Object, с. 565), *тестовой заглушкой* (Test Stub, с. 544), *тестовым агентом* (Test Spy, с. 552) или *подставным объектом* (Mock Object, с. 558) в зависимости от способа взаимодействия теста с *тестовым двойником* (Test Double). Данное решение более подробно рассматривается в главе 11, “Использование тестовых двойников”.

При использовании *тестовой заглушки* (Test Stub) или *подставного объекта* (Mock Object) придется выбирать между *фиксированным тестовым двойником* (Hard-Coded Test Double, с. 581) и *настраиваемым тестовым двойником* (Configurable Test Double, с. 571). Положительные и отрицательные стороны каждого решения рассматриваются в главе 11 и в подробном описании шаблонов. Принятое решение определяет форму теста, например при использовании *подставного объекта* (Mock Object) тест больше “загружен” созданием этого объекта.

Наконец, тест модифицируется для создания, возможной настройки и установки *подставного объекта* (Mock Object). Кроме того, для некоторых типов *подставных объектов* (Mock Object) может потребоваться вызов метода *verification*. В статически типизированных языках перед внесением подставной реализации может потребоваться рефакторинг *выделение интерфейса* (Extract Interface). После этого данный интерфейс используется в качестве типа переменной, хранящей ссылку на подменяемую зависимость.

Настройка внешнего ресурса (Setup External Resource)

Тестируемая система зависит от содержимого внешнего ресурса, который с точки зрения теста выглядит как *таинственный гость* (Mystery Guest).

Не используйте предопределенный ресурс, а создайте внешний ресурс с помощью логики настройки тестовой конфигурации.

Из [RTC]:

Если тест должен зависеть от внешних ресурсов, например каталогов, баз данных или файлов, убедитесь, что они явно создаются или выделяются тестом перед началом работы и освобождаются после ее завершения (обеспечьте освобождение ресурса и в случае неудачного завершения теста).

Замечания по реализации

Если тестируемая система должна использовать такой внешний ресурс, как файл, и нет никакой возможности заменить механизм доступа *тестовой заглушкой* (Test Stub, с. 544) или *поддельным объектом* (Fake Object, с. 565), придется смириться с использованием внешнего ресурса. Очевидно, что с такими ресурсами связан ряд проблем: при чтении тестов неясно содержимое ресурса и ресурс неожиданно может исчезнуть, приведя к неудачному завершению теста в результате *оптимизма по отношению к ресурсу* (Resource Optimism; см. *Нестабильный тест*, Erratic Test, с. 267). Кроме того, ресурсы могут привести к появлению *взаимодействующих тестов* (Interacting Tests; см. *Нестабильный тест*, Erratic Test) и “войнам” запуска тестов (Test Run War; см. *Нестабильный тест*, Erratic Test). Данный рефакторинг не поможет решить последнюю проблему, но позволит избежать проявления *таинственного гостя* (Mystery Guest; см. *Непонятный тест*, Obscure Test, с. 230) и *оптимизма по отношению к ресурсу* (Resource Optimism).

Для реализации рефакторинга *настройка внешнего ресурса* (Setup External Resource) содержимое внешнего ресурса просто переносится в *тестовый метод* (Test Method, с. 378), метод *setUp* или во *вспомогательный метод теста* (Test Utility Method, с. 610). На основе содержимого кода теста создает ресурс, делая его очевидным для читателя теста. Также этот подход гарантирует существование ресурса, поскольку он создается при каждом запуске теста.

Приложение Б

Терминология xUnit

Подставной объект (Mock Object), поддельный объект (Fake Object), тестовая заглушка (Test Stub) и объект-заглушка (Dummy Object)

Вводят ли вас в заблуждение термины “тестовая заглушка” и “подставной объект”? Не кажется ли вам иногда, что ваш собеседник пользуется совсем другой терминологией? Если кажется, то вы не одиноки!

Терминология для различных типов *тестовых двойников* (Test Double, с. 538) очень туманна. Разные авторы используют различные термины для описания одних и тех же понятий. А иногда под одними и теми же терминами подразумеваются разные понятия! (Дополнительные рассуждения о важности имен приводятся во врезке “Что в имени шаблона?” на с. 588.)

Одна из причин, по которым была написана эта книга, — желание навести порядок в терминологии, чтобы разработчики могли использовать набор имен с точными определениями. В этом приложении приведены текущие источники терминологии и перекрестные ссылки для терминов, которые используются в этой книге и в источниках.

Описание ролей

В следующей таблице описываются имена основных шаблонов семейства *тестовый двойник* (Test Double).

Шаблон	Описание	Имеет ли поведение	Вставляет ли опосредованные входные данные в тестируемую систему	Обрабатывает ли опосредованный вывод тестируемой системы	Значения, предоставленные тестом	Примеры
<i>Тестовый двойник</i> (Test Double, с. 538)	Общее имя семейства					

Окончание таблицы

Шаблон	Описание	Имеет ли поведение	Вставляет ли опосредованные входные данные в тестируемую систему	Обрабатывает ли опосредованный вывод тестируемой системы	Значения, предоставленные тестом	Примеры
<i>Объект-заглушка</i> (<i>Dummy Object</i> , с. 730)	Параметр атрибута или метода	Нет	Нет, никогда не вызывается	Нет, никогда не вызывается	Нет	Null, “Игнорируемая строка”, new <i>Object</i> ()
<i>Тестовая заглушка</i> (<i>Test Stub</i> , с. 544)	Проверка опосредованных входных данных тестируемой системы	Да	Да	Игнорирует	Входные значения	
<i>Тестовый агент</i> (<i>Test Spy</i> , с. 552)	Проверка опосредованного вывода тестируемой системы	Да	Не обязательно	Перехватывает для последующей проверки	Входные значения (не обязательно)	
<i>Подставной объект</i> (<i>Mock Object</i> , с. 558)	Проверка опосредованного вывода тестируемой системы	Да	Не обязательно	Проверяет правильность путем сравнения с ожидаемым значением	Входные значения (не обязательно) и ожидаемые выходные значения	
<i>Поддельный объект</i> (<i>Fake Object</i> , с. 565)	(Быстрый) запуск (незапускаемых) тестов	Да	Нет	Использует	Нет	Эмулятор базы данных в памяти
<i>Временная тестовая заглушка</i> (<i>Temporary Test Stub</i> ; см. <i>Тестовая заглушка</i> , <i>Test Stub</i>)	Замена написанного кода	Да	Нет	Использует	Нет	Эмулятор базы данных в памяти

Перекрестные ссылки на терминологию

В следующей таблице перечислены некоторые источники конфликтующих определений для имен шаблонов, использующихся в этой книге.

Шаблон	Astels	Beck	Feathers	Fowler	jMock	UTWJ	OMG	Pragmatic	Recipes
<i>Тестовый двойник</i> (Test Double)								Double или Stand-in	
<i>Объект-заглушка</i> (Dummy Object)					Stub	Dummy			Stub
<i>Тестовая заглушка</i> (Test Stub)	Fake		Fake	Stub		Stub	Dummy	Mock	Fake
<i>Тестовый агент</i> (Test Spy)						Dummy			Fake
<i>Подставной объект</i> (Mock Object)	Mock		Mock	Mock	Mock	Mock	Mock	Mock	Mock
<i>Поддельный объект</i> (Fake Object)						Dummy			
<i>Временная тестовая заглушка</i> (Temporary Test Stub)						Stub			
OMG CORBA Stub								Stub	

- В книге “Unit Testing with Java” [UTwJ] термин “Dummy Object” используется для описания того же понятия, что и *поддельный объект* (Fake Object) в этой книге.
- В книге “Pragmatic Unit Testing” [PUT] термин “Stub” определен как пустая реализация метода. Это распространенная интерпретация в мире процедурного про-

граммирования. Но в области объектно-ориентированного программирования эта сущность обычно называется “Null Object” [PLOPD3].

- В ранних источниках термины “Stub” и “Mock Object” рассматривались как эквивалентные. Позднее различие между ними было описано в [MRNO] и [MAS].
- В стандарте CORBA (Common Object Request Broker Architecture; этот стандарт определен организацией Object Management Group) и других спецификациях удаленного вызова процедур термины “Stub” и “Skeleton” используются для описания автоматически генерированного кода локальной и удаленной реализации интерфейса, определенного в спецификации IDL. (Эта информация приводится здесь потому, что данная интерпретация терминов также широко используется среди разработчиков автоматизированных тестов.)

В следующей таблице приведены источники, процитированные в предыдущей таблице.

Источник	Описание	Ссылка	Издательство
Astels	Книга “Test-Driven Development: A Practical Guide”	[TDD-APG]	Prentice Hall
Beck	Книга “Test-Driven Development: By Example”	[TDD-BE]	Addison-Wesley
Feathers	Книга “Working Effectively with Legacy Code”	[WEwLC]	Prentice Hall
Fowler	Блог “Mocks Aren’t Stubs”	[MAS]	martinfowler.com
jMock	Публикация “Mock Roles, Not Objects”	[MRNO]	ACM (OOPSLA)
UTWJ	Книга “Unit Testing in Java”	[UTwJ]	Morgan Kaufmann
OMG	Спецификация CORBA от Object Management Group		OMG
Pragmatic Recipes	Книга “Pragmatic Unit Testing in C# with NUnit” [PUT] Книга “JUnit Recipes”		Pragmatic Bookshelf Manning

Перекрестные ссылки на терминологию xUnit

В следующей таблице приводится соответствие терминов из настоящей книги терминам различных членов семейства xUnit. Этот список не является исчерпывающим, но он демонстрирует адаптацию стандартной терминологии xUnit к идиомам и культуре различных языков и сообществ.

Пакет		Термин из данной книги					
Язык	Пакет xUnit	Класс <i>TestCase</i> (Testcase Class)	Фабрика наборов тестов (Test Suite Factory)	Тестовый метод (Test Method)	Настройка тестовой конфигурации	Очистка тестовой конфигурации набора (Suite Fixture Setup)	Настройка тестовой конфигурации набора (Suite Fixture Teardown)
Java 1.4	JUnit 3.8.2	Подкласс TestCase	static suite()	testXXX()	setUp()	tearDown()	Нет
Java 5	JUnit 4.0+	import org.junit.Test import static org.junit.Assert. suite()	@Test	@Before	@After	Класс @Before	Подкласс Expected Exception Test
.NET	CsUnit	[Test Fixture]	[Suite]	[Test]	[SetUp]	[TearDown]	Нет
.NET	NUnit 2.0	[Test Fixture]	[Suite]	[Test]	[SetUp]	[TearDown]	Нет
.NET	NUnit 2.1+	[Test Fixture]	[Suite]	[Test]	[SetUp]	[TearDown]	[TestFixture TearDown]
.NET	MbUnit 2.0	[Test Fixture]	[Suite]	[Test]	[SetUp]	[TearDown]	[Fixture TearDown]
.NET	MSTest	[Test Class]	Нет	[Test Method]	[TestInitialize]	[TestCleanup]	[Expected Exception Test]
PHP	PHPUnit	Подкласс TestCase	static suite()	testXXX()	setUp()	tearDown()	Нет
Python	PyUnit	Подкласс unittest.TestCase	Test Loader()	testXXX	setUp	tearDown	Нет
Ruby	Test::Unit	Подкласс Test::Unit::TestCase	Classname.suite()	testXXX()	setUp()	tearDown	Нет
						assert raise	assert _raise

Окончание таблицы

Пакет		Термины из данной книги					
Язык	Пакет xUnit	Класс <i>теста</i> (Testcase Class)	Фабрика наборов тестов (Test Suite Factory)	Тестовый метод (Test Method)	Настройка тестовой конфигурации	Очистка тестовой конфигурации набора (Suite Fixture Setup)	Тест на ожидаемое исключение (Expected Exception Test)
Smalltalk	SUnit	Super class: TestCase	TestSuite	TestXXX	setUp	tearDown	Не определено
VB 6	VbUnit	Реализует IFixture	Реализует ISuite	TestXXX()	IFixture_Setup()	IFixture_TearDown	IFixture_Frame_Create()
SAP ABAP	ABAP Unit	FOR TESTING	Автоматические скрипты	Любой	setup	tearDown	class_setup tearDown

Приложение В

Пакеты семейства xUnit

Ниже описаны пакеты семейства xUnit для иллюстрации разнообразия этого семейства и распространенности поддержки автоматизированного модульного тестирования в различных языках программирования. Также здесь комментируются конкретные возможности некоторых членов семейства. Более полная информация приведена по адресу:

<http://xprogramming.com/software.htm>

ABAP Object Unit

Пакет семейства xUnit для языка программирования SAP ABAP. Данный пакет является непосредственной реализацией пакета JUnit на языке программирования ABAP, но не поддерживает перехват исключений, возникающих в тестируемой системе.

Пакет ABAP Object Unit доступен на сайте www.abapunittests.com. Там же приводятся сведения о написании тестов в ABAP. Пакет для SAP/ABAP начиная с версии 6.40 рассматривается в следующем разделе.

ABAP Unit

Данный пакет семейства xUnit предназначен для языка программирования SAP ABAP начиная с версии 6.40 (NetWeaver 2004s). Наиболее заметным отличием ABAP Unit является специальная поддержка удаления кода из тестов при переносе кода из тестовой среды в рабочую.

Пакет ABAP Unit распространяется непосредственно компанией SAP AG в составе средств разработки NetWeaver 2004s. Дополнительная информация о модульном тестировании в языке ABAP приведена в документации компании SAP и по адресу www.abapunittests.com. Для продукта SAP/ABAP до версии 6.40 (NetWeaver 2004s) используется пакет ABAP Object Unit (рассматривался в предыдущем разделе).

CppUnit

Это пакет семейства xUnit для языка программирования C++. Пакет доступен на сайте <http://cppunit.sourceforge.net>. Разработчики для платформы .NET иногда используют пакет NUnit.

CsUnit

Данный пакет семейства xUnit предназначен для языка программирования C#. Пакет доступен на сайте www.csunit.org. Разработчики для платформы .NET иногда используют пакет NUnit.

CUnit

Этот пакет семейства xUnit предназначен для языка программирования С. Дополнительная информация приводится по адресу:

<http://cunit.sourceforge.net/doc/index.html>

DbUnit

Это расширение пакета JUnit, предназначенное для упрощения тестирования баз данных. Расширение доступно по адресу www.dbunit.org.

IeUnit

Пакет семейства xUnit предназначен для тестирования Web-страниц, выводимых обозревателем Microsoft Internet Explorer с помощью языка JavaScript и DHTML. Пакет доступен по адресу <http://ieunit.sourceforge.net/>.

JBehave

Один из первых пакетов нового поколения семейства xUnit, облегчающий использование *тестов как спецификации* (Tests as Specification) при разработке на основе тестов. Главным отличием пакета JBehave и традиционных пакетов семейства xUnit является отказ от традиционной “тестовой” терминологии и переход к терминам, характерным для спецификации. Таким образом, “fixture” превращается в “context”, “assert” — в “should” и т.д. Пакет JBehave доступен на сайте <http://jbehave.codehaus.org>. Эквивалентом данного пакета для языка Ruby является RSpec.

JUnit

Пакет семейства xUnit для языка Java. Данный пакет был переписан в конце 2005 года для использования аннотаций, введенных в языке Java версии 1.5. Пакет доступен по адресу www.junit.org.

MbUnit

Пакет семейства xUnit для языка программирования C#. Основным преимуществом пакета MbUnit является непосредственная поддержка *параметризованных тестов* (Parametrized Test). Пакет доступен по адресу www.nunit.org/mgbunit.com. Разработчики для платформы .NET могут также использовать пакеты NUnit, CsUnit и MSTest.

MSTest

Этот член семейства xUnit не имеет формального имени, кроме пространства имен Microsoft.VisualStudio.TestTools.UnitTesting, но большинство разработчиков ссылаются на него, как на MSTest. Технически это имя *программы запуска тестов для командной строки* (Command-Line Test Runner) `mstest.exe`. Основным преимуществом пакета MSTest является распространение вместе с пакетом Visual Studio 2005 Team System. В более дешевых версиях Visual Studio этот пакет не распространяется. Пакет MSTest включает в себя ряд прогрессивных функций, например непосредственная поддержка *управляемых данными тестов* (Data-Driven Test). Дополнительная информация о пакете доступна в базе знаний MSDN по адресу:

<http://msdn.microsoft.com/en-us/library/ms182516.aspx>

В качестве альтернативных (и более дешевых) вариантов разработчики для платформы .NET могут использовать пакеты NUnit, CsUnit и MbUnit.

NUnit

Пакет семейства xUnit для языков программирования платформы .NET. Пакет доступен по адресу www.nunit.org. Разработчики на языке C# могут использовать пакеты CsUnit, MbUnit и MSTest.

PHPUnit

Член семейства xUnit для языка программирования PHP. Согласно Себастьяну Бергманну, “PHPUnit является полной реализацией пакета JUnit 3.8. Кроме исходного набора функций добавленастроенная поддержка *подставных объектов* (Mock Object), *покрытия кода* (Code Coverage), *гибкой документации* (Agile Documentation) и *неполных и пропущенных тестов* (Incomplete and Skipped Tests)”. Дополнительная информация о PHPUnit приведена на сайте <http://www.phpunit.de>. Там же доступна книга о PHPUnit.

PyUnit

Этот пакет семейства xUnit предназначен для языка Python. Это полноценная реализация JUnit. Дополнительная информация о данном пакете приведена на сайте <http://pyunit.sourceforge.net/>.

RSpec

Один из первых пакетов нового поколения семейства xUnit, облегчающий использование *тестов как спецификации* (Tests as Specification) при разработке на основе тестов. Главным отличием пакета RSpec и традиционных пакетов семейства xUnit является отказ от традиционной “тестовой” терминологии и переход к терминам, характерным для спецификации. Таким образом, “fixture” превращается в “context”, “assert” — в “should” и т.д. Пакет RSpec доступен на сайте <http://rspec.rubyforge.org>. Эквивалентом данного пакета для языка Java является JBehave.

runit

Еще один пакет семейства xUnit для языка Ruby. Это оболочка поверх Test::Unit, обеспечивающая дополнительную функциональность. Данный пакет доступен на сайте www.rubypeople.org.

SUnit

Самопровозглашенный “предок всех инфраструктур модульного тестирования”. Пакет SUnit предназначен для языка программирования Smalltalk. Он доступен на сайте <http://sunit.sourceforge.net>.

Test::Unit

Член семейства xUnit для языка программирования Ruby. Пакет доступен по адресу www.rubypeople.org и является частью модуля “Ruby Development Tools” интегрированной среды разработки Eclipse.

TestNG

Пакет семейства xUnit для языка программирования Java. Имеет несколько отличающееся от JUnit поведение. Пакет TestNG явно поддерживает зависимости между тестами и совместное использование тестовой конфигурации *тестовыми методами* (Test Method). Дополнительная информация о данном пакете приведена на сайте <http://testng.org>.

utPLSQL

Член семейства xUnit для языка программирования баз данных PL/SQL. Дополнительная информация и исходный код пакета доступны по адресу <http://utplsql.sourceforge.net/>. Встраиваемый модуль для интеграции пакета utPLSQL в инструментарий Oracle доступен по адресу www.ounit.com.

VB Lite Unit

Еще один пакет семейства xUnit, предназначенный для языков программирования Visual Basic и VBA (Visual Basic for Applications). “VB Lite Unit является надежным и простым инструментом модульного тестирования для языков Visual Basic и VBA. Пакет написан Стивом Йоргенсеном. Мотивирующим принципом при создании VB Lite Unit было создание самого простого и надежного инструмента для модульного тестирования, который сможет выполнять все функции, необходимые для разработки на основе тестов с использованием языков Visual Basic 6 и VBA. Функции, которые не работают или работают ненадежно, в Visual Basic или VBA не реализованы, например не предпринимается попытка интроспекции для идентификации тестовых методов.” Разработчики на языках Visual Basic и VBA могут также использовать пакет VbUnit. Разработчики на языке VB.NET могут использовать пакеты NUnit, CsUnit и MbUnit.

VbUnit

Пакет семейства xUnit для языка программирования Visual Basic 6.0. В этом пакете была впервые реализована функция настройки *тестовой конфигурации набора* (Suite Fixture Setup) и введена концепция “тестовая конфигурация” для описания *класса теста* (Testcase Class).

Одним из основных отличий VbUnit является поведение *метода с утверждением* (Assertion Method). Если метод приводит к неудачному завершению теста, сообщение записывается в журнал ошибок немедленно, а не после перехвата ошибки *программой запуска тестов* (Test Runner). Практически это означает сложности при проверке *специального утверждения* (Custom Assertion), так как сообщения в журнале не предотвращаются средствами обычной конструкции *тест на ожидаемое исключение* (Expected Exception Test). Обходным решением этой проблемы является запуск теста со *специальным утверждением* (Custom Assertion) внутри *встроенной программы запуска тестов* (Encapsulated Test Runner).

Это платный пакет семейства xUnit. Пакет доступен на сайте www.vbunit.org. Раньше существовала и бесплатная версия; возможно, она появится снова. Разработчики на языках Visual Basic и VBA могут использовать пакет VB Lite Unit. Разработчики на языке VB.NET могут использовать пакеты NUnit, CsUnit и MbUnit.

xUnit

Это общее название для всех *инфраструктур автоматизации тестов* (Test Automation Framework) для модульного тестирования, основанных на пакетах SUnit и JUnit. Пакеты инфраструктуры xUnit для большинства языков программирования доступна на сайтах <http://xprogramming.com> и <http://en.wikipedia.org/wiki/XUnit>. Еще одним источником средств для работы с модульными и приемочными тестами является сайт www.opensourcetesting.org.

Приложение Г

Инструментарий

Следующие утилиты упоминаются в тех или иных разделах настоящей книги. В данном приложении описывается назначение утилит, а также их отношение к автоматизации тестов с помощью xUnit.

Ant

Утилита автоматической компиляции, применяемая в сообществе разработчиков Java. Эквивалентом для платформы .NET является пакет NAnt.

AntHill

Утилита непрерывной интеграции, применяемая разработчиками Java.

BPT

Коммерческая утилита для создания *тестов на основе сценария* (Scripted Test), позволяющая нетехническому персоналу создавать тесты на основе повторного использования компонентов, полученных в результате рефакторинга *записанных тестов* (Recorded Test). Кроме того, утилита позволяет составить список повторно используемых компонентов тестов, которые должны быть написаны более квалифицированными разработчиками. Дополнительная информация приведена на сайте компании Mercury Interactive. На момент передачи данной книги в печать компания Mercury Interactive находилась в процессе поглощения компанией Hewlett-Packard, поэтому адрес сайта мог измениться.

Canoo WebTest

Это инфраструктура для подготовки *тестов на основе сценария* (Scripted Test), написанных на языке XML. Концептуально пакет Canoo WebTest напоминает пакет Fit, так как позволяет определить специфичный для проблемной области язык тестов и уже с его помощью определить сами тесты. Дополнительная информация приведена на сайтах <http://webtest.canoo.com> и <http://webtest-community.canoo.com>.

Cruise Control

Утилита непрерывной интеграции, используемая разработчиками на языке Java. Эквивалентом для платформы .NET является пакет Cruise Control.net.

DDSteps

Это расширение Data-Driven Test для пакета JUnit. “DDSteps является расширением пакета JUnit, предназначенным для создания управляемых данными тестов. В принципе, пакет DDSteps позволяет параметризовать тесты и запускать их с разными данными.” Дополнительная информация о данном пакете приведена по адресу www.ddsteps.org.

EasyMock

Это инструментарий для генерации *подставных объектов* (Mock Object) для использования в тестах Java. Поскольку пакет EasyMock использует *режим настройки* (Configuration Mode) для описания ожиданий, тесты выглядят немного странно и к ним нужно привыкнуть. Дополнительная информация приведена на сайте www.easymock.org.

eCATT

Это утилита для создания *записанных тестов* (Recorded Test), которая предоставляется вместе со средствами разработки SAP. Дополнительная информация о данном пакете приведена на сайтах www.sap.com и www.sdn.sap.com.

Eclipse

Это интегрированная среда разработки для языка Java. Изначально среда Eclipse была создана компанией IBM, но сейчас разработкой занимается Eclipse Foundation. Некоторые специфичные для языка модули были интегрированы вместе с соответствующим модулем xUnit. Например, среда разработки Java включает в себя JUnit, а среда разработки Ruby Development Tools — Test::Unit. Пакет Eclipse доступен для загрузки по адресу www.eclipse.org.

Fit

Уорд Каннингем разработал инфраструктуру, позволяющую клиентам создавать автоматизированные тесты. Пакет Fit разделяет задачу определения тестов с помощью таблиц на Web-страницах или электронных таблиц и задачу программирования вызовов тестируемой системы. Хотя в свое время Fit был отдельной утилитой, теперь это спецификация для целого семейства утилит, реализованных на нескольких языках, включая Java, .NET, Ruby и Python. Одни пакеты семейства представляют собой просто инфраструктуру запуска тестов. Другие, например Fitnessse, включают в себя инструментарий для написания тестов и управления версиями. Каждый пакет должен реализовывать один и тот же набор стандартных тестовых конфигураций. Дополнительная информация приведена на

сайте <http://fit.c2.com> и в книге [FitB], которую Каннингем написал совместно с Риком Магриджем.

FitNesse

Это утилита для создания тестов Fit, написанная Бобом Мартином из компании Object Mentor. FitNesse обеспечивает подобную Wiki функциональность для написания тестов на основе предопределенных тестовых конфигураций. В результате клиенты могут самостоятельно создавать и запускать автоматизированные тесты. Дополнительная информация о данном пакете приведена на сайте www.fitnesse.org.

HttpUnit

Интерфейс поверх пакета JUnit для написания тестов, взаимодействующих с Web-приложениями по протоколу HTTP. Пакет HttpUnit не использует обозреватель, поэтому он не подходит для тестирования приложений с большим количеством сценариев на стороне клиента (например, AJAX). Дополнительная информация приведена на сайте <http://httpunit.sourceforge.net>.

Idea

Интегрированная среда разработки на языке Java, поддерживающая множество функций рефакторинга. На сайте Idea [JBrains] приводится подробное описание многих рефакторингов. На этом же сайте предлагается модуль рефакторинга для Visual Studio, который называется ReSharper.

JFCUnit

Это интерфейс для пакета JUnit, которые работает поверх пакета HttpUnit и позволяет взаимодействовать с Web-приложениями по протоколу HTTP. Пакет JFCUnit содержит набор *вспомогательных методов теста* (Test Utility Method), формирующих язык *высокого уровня* (Higher-Level Language) для описания тестов Web-приложений. Поскольку пакет работает поверх пакета HttpUnit, он также не использует обозреватель. Таким образом, пакет JFCUnit не подходит для тестирования приложений с большим количеством сценариев на стороне клиента (например, AJAX). Дополнительная информация приведена на сайте <http://jfcunit.sourceforge.net>.

JMock

Широко применяемая инфраструктура создания *подставных объектов* (Mock Object) при написании тестов на языке Java. *Конфигурационный интерфейс* (Configuration Interface) используется для описания ожиданий и делает тесты очень простыми для чтения. Дополнительная информация приведена на сайте www.jmock.org.

NMock

Широко применяемая инфраструктура создания *подставных объектов* (Mock Object) при написании тестов для платформы .NET. *Конфигурационный интерфейс* (Configuration Interface) используется для описания ожиданий и делает тесты очень простыми для чтения. Дополнительная информация приведена на сайте <http://nmock.org>.

QTP (QuickTest Professional)

Коммерческая утилита для записи тестов, позволяющая непрофессиональным пользователям записывать тесты в процессе использования приложений. В комбинации с режимом “Expert View” для просмотра записанных тестов утилита QTP может использоваться для рефакторинга тестов в повторно используемые компоненты. Такие компоненты смогут применять и рядовые пользователи. Дополнительная информация об этой утилите приведена на сайте компании Mercury Interactive. На момент передачи данной книги в печать компания Mercury Interactive находилась в процессе поглощения компанией Hewlett-Packard, поэтому адрес сайта мог измениться.

ReSharper

Встраиваемый модуль поддержки рефакторинга от компании JetBrains (разработчик среды Idea) для среды разработки Visual Studio. Дополнительное описание многих рефакторингов приводится на сайте разработчиков [JBrains].

Visual Studio

Интегрированная среда разработки от компании Microsoft, предназначенная для платформы .NET. Среда разработки распространяется в нескольких версиях (с разной ценой). В некоторые версии включены пакет MSTest и поддержка рефакторинга кода и тестов. Существуют модули сторонних производителей, поддерживающие как рефакторинг (см. [JBrains]), так и работу с тестами xUnit (см. CsUnit, MbUnit и NUnit).

Watir

Расшифровывается как “Web Application Testing in Ruby”. Это набор компонентов для управления обозревателем Internet Explorer с помощью *тестов на основе сценария* (Scripted Test), написанных на языке программирования Ruby. Дополнительная информация о данном пакете приведена на сайте <http://wtr.rubyforge.org/>.

Приложение Д

Цели и принципы

Имя	Страница	Отношение	Базовое имя	Глава
“Репеллент” для ошибок (Bug Repellent)	78		“Репеллент” для ошибок (Bug Repellent)	3
Донести намерение (Communicate Intent)	95		Донести намерение (Communicate Intent)	5
Локализация дефектов (Defect Localization)	78		Локализация дефектов (Defect Localization)	3
Проектирование с учетом тестирования (Design for Testability)	94		Проектирование с учетом тестирования (Design for Testability)	5
Не навреди (Do No Harm)	79		Не навреди (Do No Harm)	5
Не модифицировать тестируемую систему (Don’t Modify the SUT)	95		Не модифицировать тестируемую систему (Don’t Modify the SUT)	5
Обеспечить адекватные усилия и ответственность (Ensure Commensurate Effort and Responsibility)	101		Обеспечить адекватные усилия и ответственность (Ensure Commensurate Effort and Responsibility)	5
Исполнимая спецификация (Executable Specification)	77	Псевдоним	Тест как спецификация (Tests as Specification)	3
Выразительные тесты (Expressive Tests)	83		Выразительные тесты (Expressive Tests)	3
Сначала через главный вход (Front Door First)	94	Псевдоним	Сначала использовать главный вход (Use the Front Door First)	5
Полностью автоматизированные тесты (Fully Automated Tests)	81		Полностью автоматизированные тесты (Fully Automated Tests)	3
Язык высокого уровня (Higher-Level Language)	95	Псевдоним	Донести намерение (Communicate Intent)	5
Независимые тесты (Independent Tests)	96	Псевдоним	Сохранять независимость тестов (Keep Tests Independent)	5

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Изолировать тестируемую систему</i> (Isolate the SUT)	97		<i>Изолировать тестируемую систему</i> (Isolate the SUT)	5
<i>Не вносить логику тестов в код продукта</i> (Keep Test Logic Out of Production Code)	99		<i>Не вносить логику тестов в код продукта</i> (Keep Test Logic Out of Production Code)	5
<i>Сохранять независимость тестов</i> (Keep Tests Independent)	96		<i>Сохранять независимость тестов</i> (Keep Tests Independent)	5
<i>Минимизировать пересечения тестов</i> (Minimize Test Overlap)	98		<i>Минимизировать пересечения тестов</i> (Minimize Test Overlap)	5
<i>Минимизировать нетестированный код</i> (Minimize Untestable Code)	98		<i>Минимизировать нетестированный код</i> (Minimize Untestable Code)	5
<i>Не вставлять тесты в код продукта</i> (No Test Logic in Production Code)	99		<i>Не вносить логику тестов в код продукта</i> (Keep Test Logic Out of Production Code)	5
<i>Не вносить риски с помощью тестов</i> (No Test Risk)	79	Псевдоним	<i>Не навреди</i> (Do No Harm)	5
<i>Повторяемые тесты</i> (Repeatable Tests)	81		<i>Повторяемые тесты</i> (Repeatable Tests)	3
<i>Жизнеспособный тест</i> (Robust Test)	84		<i>Жизнеспособный тест</i> (Robust Test)	3
<i>Страховочная сеть</i> (Safety Net)	79	Псевдоним	<i>Тесты как страховка</i> (Tests as Safety Net)	3
<i>Самопроверяющийся тест</i> (Self-Checking Test)	81		<i>Самопроверяющийся тест</i> (Self-Checking Test)	3
<i>Разделение интереса</i> (Separation of Concerns)	83		<i>Разделение интереса</i> (Separation of Concerns)	3
<i>Простые тесты</i> (Simple Tests)	83		<i>Простые тесты</i> (Simple Tests)	3
<i>Тест одного условия</i> (Single Condition Test)	99	Псевдоним	<i>Один тест проверяет одно условие</i> (Verify One Condition per Test)	5
<i>Понимание с одного взгляда</i> (Single Glance Readable)	95	Псевдоним	<i>Доносить намерение</i> (Communicate Intent)	5
<i>Тестируовать аспекты отдельно</i> (Test Concerns Separately)	101		<i>Тестируовать аспекты отдельно</i> (Test Concerns Separately)	5
<i>Разработка на основе тестов</i> (Test-Driven Development)	94	Псевдоним	<i>Сначала писать тесты</i> (Write the Tests First)	5

Окончание таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Разработка с тестами в начале</i> (Test First Development)	94	Псевдоним	<i>Сначала писать тесты</i> (Write the Tests First)	5
<i>Тесты как документация</i> (Tests as Documentation)	79		<i>Тесты как документация</i> (Tests as Documentation)	3
<i>Тесты как страховка</i> (Tests as Safety Net)	79		<i>Тесты как страховка</i> (Tests as Safety Net)	3
<i>Тест как спецификация</i> (Tests as Specification)	77		<i>Тест как спецификация</i> (Tests as Specification)	3
<i>Сначала использовать главный вход</i> (Use the Front Door First)	94		<i>Сначала использовать главный вход</i> (Use the Front Door First)	5
<i>Один тест проверяет одно условие</i> (Verify One Condition per Test)	99		<i>Один тест проверяет одно условие</i> (Verify One Condition per Test)	5
<i>Сначала писать тесты</i> (Write the Tests First)	94		<i>Сначала писать тесты</i> (Write the Tests First)	5

Приложение E

Запахи, псевдонимы и причины

Имя	Страница	Отношение	Базовое имя	Глава
<i>Рулетка утверждений</i> (Assertion Roulette)	264		<i>Рулетка утверждений</i> (Assertion Roulette)	16
<i>Асинхронный код</i> (Asynchronous Code)	252	Причина для	<i>Сложный в тестировании код</i> (Hard-to-Test Code)	15
<i>Асинхронный тест</i> (Asynchronous Test)	291	Причина для	<i>Медленные тесты</i> (Slow Tests)	16
<i>Чувствительность к поведению</i> (Behavior Sensitivity)	279	Причина для	<i>“Хрупкий” тест</i> (Fragile Test)	16
<i>Тест с ошибками</i> (Buggy Test)	296		<i>Тест с ошибками</i> (Buggy Test)	17
<i>Сложная очистка</i> (Complex Teardown)	248	Причина для	<i>Условная логика теста</i> (Conditional Test Logic)	15
<i>Сложный тест</i> (Complex Test)	230	Псевдоним	<i>Непонятный тест</i> (Obscure Test)	15
<i>Условная логика теста</i> (Conditional Test Logic)	243		<i>Условная логика теста</i> (Conditional Test Logic)	15
<i>Условная логика проверки</i> (Conditional Verification Logic)	246	Причина для	<i>Условная логика теста</i> (Conditional Test Logic)	15
<i>Чувствительность к контексту</i> (Context Sensitivity)	282	Причина для	<i>“Хрупкий” тест</i> (Fragile Test)	16
<i>Повторное использование через копирование</i> (Cut-and-Paste Code Reuse)	255	Причина для	<i>Дублирование тестового кода</i> (Test Code Duplication)	15
<i>Чувствительность к данным</i> (Data Sensitivity)	280	Причина для	<i>“Хрупкий” тест</i> (Fragile Test)	16
<i>Разработчики не пишут тесты</i> (Developers Not Writing Tests)	298		<i>Разработчики не пишут тесты</i> (Developers Not Writing Tests)	17
<i>“Энергичный” тест</i> (Eager Test)	231	Причина для	<i>Рулетка утверждений</i> (Assertion Roulette)	16
<i>Засорение равенства</i> (Equality Pollution)	260	Причина для	<i>Логика теста в продукте</i> (Test Logic in Production)	15

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Нестабильный тест</i> (Erratic Test)	267		<i>Нестабильный тест</i> (Erratic Test)	16
<i>Гибкий тест</i> (Flexible Test)	245	Причина для	<i>Условная логика теста</i> (Conditional Test Logic)	15
<i>Только для тестов</i> (For Tests Only)	259	Причина для	<i>Логика теста в продукте</i> (Test Logic in Production)	15
<i>“Хрупкая” тестовая конфигурация</i> (Fragile Fixture)	284	Причина для	<i>“Хрупкий” тест</i> (Fragile Test)	16
<i>“Хрупкий” тест</i> (Fragile Test)	277		<i>“Хрупкий” тест</i> (Fragile Test)	16
<i>Частая отладка</i> (Frequent Debugging)	285		<i>Частая отладка</i> (Frequent Debugging)	16
<i>Тестовая конфигурация общего характера</i> (General Fixture)	234	Причина для	<i>Непонятный тест</i> (Obscure Test)	15
<i>Сложный в тестировании код</i> (Hard-to-Test Code)	251		<i>Сложный в тестировании код</i> (Hard-to-Test Code)	15
<i>Фиксированная зависимость</i> (Hard-Coded Dependency)	252	Псевдоним	<i>Сложный в тестировании код</i> (Hard-to-Test Code)	15
<i>Фиксированные данные теста</i> (Hard-Coded Test Data)	238	Причина для	<i>Непонятный тест</i> (Obscure Test)	15
<i>Высокая стоимость обслуживания тестов</i> (High Test Maintenance Cost)	300		<i>Высокая стоимость обслуживания тестов</i> (High Test Maintenance Cost)	17
<i>Тесно связанный код</i> (Highly Coupled Code)	252	Причина для	<i>Сложный в тестировании код</i> (Hard-to-Test Code)	15
<i>Код теста со смещением</i> (Indented Test Code)	243	Псевдоним	<i>Условная логика теста</i> (Conditional Test Logic)	15
<i>Опроседованное тестирование</i> (Indirect Testing)	239	Причина для	<i>Непонятный тест</i> (Obscure Test)	15
<i>Нечасто запускаемые тесты</i> (Infrequently Run Tests)	303	Причина для	<i>Ошибки в продукте</i> (Production Bugs)	17
<i>Взаимодействующие наборы тестов</i> (Interacting Test Suites)	270	Причина для	<i>Нестабильный тест</i> (Erratic Test)	16
<i>Взаимодействующие тесты</i> (Interacting Tests)	268	Причина для	<i>Нестабильный тест</i> (Erratic Test)	16
<i>Чувствительность к интерфейсу</i> (Interface Sensitivity)	278	Причина для	<i>“Хрупкий” тест</i> (Fragile Test)	16
<i>Неуместная информация</i> (Irrelevant Information)	235	Причина для	<i>Непонятный тест</i> (Obscure Test)	15
<i>Одинокий тест</i> (Lonely Test)	271	Причина для	<i>Нестабильный тест</i> (Erratic Test)	16

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Длинный тест</i> (Long Test)	230	Псевдоним	<i>Непонятный тест</i> (Obscure Test)	15
<i>Потерянный тест</i> (Lost Test)	304	Причина для	<i>Ошибки в продукте</i> (Production Bugs)	17
<i>Отладка вручную</i> (Manual Debugging)	285	Псевдоним	<i>Частая отладка</i> (Frequent Debugging)	16
<i>Ручная генерация события</i> (Manual Event Injection)	288	Причина для	<i>Ручное вмешательство</i> (Manual Intervention)	16
<i>Ручная настройка тестовой конфигурации</i> (Manual Fixture Setup)	287	Причина для	<i>Ручное вмешательство</i> (Manual Intervention)	16
<i>Ручное вмешательство</i> (Manual Intervention)	287		<i>Ручное вмешательство</i> (Manual Intervention)	16
<i>Ручная проверка результата</i> (Manual Result Verification)	288	Причина для	<i>Ручное вмешательство</i> (Manual Intervention)	16
<i>Отсутствующее сообщение для утверждения</i> (Missing Assertion Message)	265	Причина для	<i>Рулетка утверждений</i> (Assertion Roulette)	16
<i>Отсутствующий модульный тест</i> (Missing Unit Test)	305	Причина для	<i>Ошибки в продукте</i> (Production Bugs)	17
<i>Несколько тестовых условий</i> (Multiple Test Conditions)	249	Причина для	<i>Условная логика теста</i> (Conditional Test Logic)	15
<i>Таинственный гость</i> (Mystery Guest)	232	Причина для	<i>Непонятный тест</i> (Obscure Test)	15
<i>Всегда успешный тест</i> (Neverfail Test)	308	Причина для	<i>Ошибки в продукте</i> (Production Bugs)	17
<i>Неопределенный тест</i> (Nondeterministic Test)	275	Причина для	<i>Нестабильный тест</i> (Erratic Test)	16
<i>Недостаточно времени</i> (Not Enough Time)	298	Причина для	<i>Разработчики не пишут тесты</i> (Developers Not Writing Tests)	17
<i>Непонятный тест</i> (Obscure Test)	230		<i>Непонятный тест</i> (Obscure Test)	15
<i>Слишком связанный тест</i> (Overcoupled Test)	283	Псевдоним	<i>“Хрупкий” тест</i> (Fragile Test)	16
<i>Зарегулированная программа</i> (Overspecified Software)	283	Причина для	<i>“Хрупкий” тест</i> (Fragile Test)	16
<i>Ошибки в продукте</i> (Production Bugs)	303		<i>Ошибки в продукте</i> (Production Bugs)	17
<i>Логика продукта внутри теста</i> (Production Logic in Test)	247	Причина для	<i>Условная логика теста</i> (Conditional Test Logic)	15
<i>Повторное изобретение колеса</i> (Reinventing the Wheel)	256	Причина для	<i>Дублирование тестового кода</i> (Test Code Duplication)	15

Окончание таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Утечка ресурсов</i> (Resource Leakage)	271	Причина для	<i>Нестабильный тест</i> (Erratic Test)	16
<i>Оптимизм по отношению к ресурсу</i> (Resource Optimism)	272	Причина для	<i>Нестабильный тест</i> (Erratic Test)	16
<i>Чувствительное сравнение</i> (Sensitive Equality)	283	Причина для	<i>“Хрупкий” тест</i> (Fragile Test)	16
<i>Использование медленного компонента</i> (Slow Component Usage)	290	Причина для	<i>Медленные тесты</i> (Slow Tests)	16
<i>Медленные тесты</i> (Slow Tests)	289		<i>Медленные тесты</i> (Slow Tests)	16
<i>Дублирование тестового кода</i> (Test Code Duplication)	254		<i>Дублирование тестового кода</i> (Test Code Duplication)	15
<i>Зависимость продукта от теста</i> (Test Dependency in Production)	260	Причина для	<i>Логика теста в продукте</i> (Test Logic in Production)	15
<i>Логика теста в продукте</i> (Test Logic in Production)	257		<i>Логика теста в продукте</i> (Test Logic in Production)	15
<i>“Война” запуска тестов</i> (Test Run War)	274	Причина для	<i>Нестабильный тест</i> (Erratic Test)	16
<i>Слишком много тестов</i> (Too Many Tests)	292	Причина для	<i>Медленные тесты</i> (Slow Tests)	16
<i>Неповторяемый тест</i> (Unrepeatable Test)	272	Причина для	<i>Нестабильный тест</i> (Erratic Test)	16
<i>Нетестируемый код теста</i> (Untestable Test Code)	253	Причина для	<i>Сложный в тестировании код</i> (Hard-to-Test Code)	15
<i>Не тестированный код</i> (Untested Code)	306	Причина для	<i>Ошибки в продукте</i> (Production Bugs)	17
<i>Не тестированное требование</i> (Untested Requirement)	307	Причина для	<i>Ошибки в продукте</i> (Production Bugs)	17
<i>Подробный тест</i> (Verbose Test)	230	Псевдоним	<i>Непонятный тест</i> (Obscure Test)	15
<i>Неправильная стратегия автоматизации тестов</i> (Wrong Test Automation Strategy)	299	Причина для	<i>Разработчики не пишут тесты</i> (Developers Not Writing Tests)	17

Приложение Ж

Шаблоны, псевдонимы и варианты

Имя	Страница	Отношение	Базовое имя	Глава
<i>Абстрактный декоратор настройки</i> (Abstract Setup Decorator)	472	Вариант	<i>Декоратор настройки</i> (Setup Decorator)	20
<i>Абстрактная тестовая конфигурация</i> (Abstract Test Fixture)	646	Псевдоним	<i>Суперкласс теста</i> (Testcase Superclass)	24
<i>Абстрактный тест</i> (Abstract Testcase)	646	Псевдоним	<i>Суперкласс теста</i> (Testcase Superclass)	24
<i>Набор всех тестов</i> (AllTests Suite)	605	Вариант	<i>Именованный набор тестов</i> (Named Test Suite)	24
<i>Анонимный метод создания</i> (Anonymous Creation Method)	443	Вариант	<i>Метод создания</i> (Creation Method)	20
<i>Сообщение с описанием аргументов</i> (Argument-Describing Message)	399	Вариант	<i>Сообщение для утверждения</i> (Assertion Message)	19
<i>Идентифицирующее утверждение сообщение</i> (Assertion-Identifying Message)	399	Вариант	<i>Сообщение для утверждения</i> (Assertion Message)	19
<i>Сообщение для утверждения</i> (Assertion Message)	398		<i>Сообщение для утверждения</i> (Assertion Message)	19
<i>Метод с утверждением</i> (Assertion Method)	390		<i>Метод с утверждением</i> (Assertion Method)	19
<i>Подключаемый метод</i> (Attachment Method)	444	Вариант	<i>Метод создания</i> (Creation Method)	20
<i>Автоматическое удаление результатов</i> (Automated Exercise Teardown)	523	Вариант	<i>Автоматическая очистка</i> (Automated Teardown)	22
<i>Автоматическое удаление тестовой конфигурации</i> (Automated Fixture Teardown)	522	Вариант	<i>Автоматическая очистка</i> (Automated Teardown)	22

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Автоматическая очистка</i> (Automated Teardown)	521		<i>Автоматическая очистка</i> (Automated Teardown)	22
<i>Автоматизированный модульный тест</i> (Automated Unit Test)	318	Псевдоним	<i>Тест на основе сценария</i> (Scripted Test)	18
<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	359		<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	18
<i>Настройка через “черный ход”</i> (Back Door Setup)	360	Вариант	<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	18
<i>Очистка через “черный ход”</i> (Back Door Teardown)	361	Вариант	<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	18
<i>Проверка через “черный ход”</i> (Back Door Verification)	362	Вариант	<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	18
<i>Раскрывающий поведение подкласс</i> (Behavior-Exposing Subclass)	592	Вариант	<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	23
<i>Модифицирующий поведение подкласс</i> (Behavior-Modifying Subclass)	592	Вариант	<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	23
<i>Проверка поведения</i> (Behavior Verification)	489		<i>Проверка поведения</i> (Behavior Verification)	21
<i>Заказное утверждение</i> (Bespoke Assertion)	495	Псевдоним	<i>Специальное утверждение</i> (Custom Assertion)	21
<i>Встроенная запись тестов</i> (Built-in Test Recording)	313	Вариант	<i>Записанный тест</i> (Recorded Test)	18
<i>Вычисляемые значения</i> (Calculated Values)	722	Псевдоним	<i>Вычисляемое значение</i> (Derived Value)	27
<i>Тест вида “запись/воспроизведение”</i> (Capture/Playback Test)	311	Псевдоним	<i>Записанный тест</i> (Recorded Test)	18
<i>Цепочки тестов</i> (Chained Tests)	477		<i>Цепочки тестов</i> (Chained Tests)	20
<i>Метод очистки</i> (Cleanup Method)	613	Вариант	<i>Вспомогательный метод теста</i> (Test Utility Method)	24
<i>Программа запуска тестов для командной строки</i> (Command-Line Test Runner)	407	Вариант	<i>Программа запуска тестов</i> (Test Runner)	19
<i>Брокер компонентов</i> (Component Broker)	692	Псевдоним	<i>Поиск зависимости</i> (Dependency Lookup)	26

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Регистр компонентов</i> (Component Registry)	692	Псевдоним	<i>Поиск зависимости</i> (Dependency Lookup)	26
<i>Тест компонента</i> (Component Test)	371	Вариант	<i>Тест уровня</i> (Layer Test)	18
<i>Настраиваемый подставной объект</i> (Configurable Mock Object)	571	Псевдоним	<i>Настраиваемый тестовый двойник</i> (Configurable Test Double)	23
<i>Настраиваемый тестовый двойник</i> (Configurable Test Double)	571		<i>Настраиваемый тестовый двойник</i> (Configurable Test Double)	23
<i>Настраиваемый тестовый агент</i> (Configurable Test Spy)	571	Псевдоним	<i>Настраиваемый тестовый двойник</i> (Configurable Test Double)	23
<i>Настраиваемая тестовая заглушка</i> (Configurable Test Stub)	571	Псевдоним	<i>Настраиваемый тестовый двойник</i> (Configurable Test Double)	23
<i>Конфигурационный интерфейс</i> (Configuration Interface)	573	Вариант	<i>Настраиваемый тестовый двойник</i> (Configurable Test Double)	23
<i>Режим настройки</i> (Configuration Mode)	573	Вариант	<i>Настраиваемый тестовый двойник</i> (Configurable Test Double)	23
<i>Константное значение</i> (Constant Value)	718	Псевдоним	<i>Точное значение</i> (Literal Value)	27
<i>Вставка конструктора</i> (Constructor Injection)	686	Вариант	<i>Вставка зависимости</i> (Dependency Injection)	26
<i>Тест конструктора</i> (Constructor Test)	381	Вариант	<i>Тестовый метод</i> (Test Method)	19
<i>Метод создания</i> (Creation Method)	441		<i>Метод создания</i> (Creation Method)	20
<i>Специальное утверждение</i> (Custom Assertion)	495		<i>Специальное утверждение</i> (Custom Assertion)	21
<i>Тест специального утверждения</i> (Custom Assertion Test)	498	Вариант	<i>Специальное утверждение</i> (Custom Assertion)	21
<i>Специальное утверждение равенства</i> (Custom Equality Assertion)	497	Вариант	<i>Специальное утверждение</i> (Custom Assertion)	21

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Схема БД на программу запуска тестов</i> (DB-Schema per Test-Runner)	659	Вариант	“Песочница” с базой данных (Database Sandbox)	25
<i>Загрузчик данных</i> (Data Loader)	362	Вариант	<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	18
<i>Получатель данных</i> (Data Retriever)	363	Вариант	<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	18
<i>Управляемый данными тест</i> (Data-Driven Test)	322		<i>Управляемый данными тест</i> (Data-Driven Test)	18
<i>Инфраструктура управляемых данными тестов</i> (Data-Driven Test Framework) (Fit)	324	Вариант	<i>Управляемый данными тест</i> (Data-Driven Test)	18
<i>Инфраструктура управляемых данными тестов</i> (Data-Driven Test Framework)	334	Вариант	<i>Инфраструктура автоматизации тестов</i> (Test Automation Framework)	18
<i>Сценарий экспорта из базы данных</i> (Database Extraction Script)	363	Вариант	<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	18
<i>Схема разбиения базы данных</i> (Database Partitioning Scheme)	660	Вариант	“Песочница” с базой данных (Database Sandbox)	25
<i>Сценарий наполнения базы данных</i> (Database Population Script)	362	Вариант	<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	18
“Песочница” с базой данных (Database Sandbox)	658		“Песочница” с базой данных (Database Sandbox)	25
“Ленивая” настройка с декоратором (Decorated Lazy Setup)	473	Вариант	Декоратор настройки (Setup Decorator)	20
Выделенная “песочница” с базой данных (Dedicated Database Sandbox)	659	Вариант	“Песочница” с базой данных (Database Sandbox)	25
<i>Делегированная настройка</i> (Delegated Setup)	437		<i>Делегированная настройка</i> (Delegated Setup)	20
<i>Делегированная очистка</i> (Delegated Teardown)	529	Вариант	<i>Встроенная очистка</i> (In-line Teardown)	22
<i>Дельта-утверждение</i> (Delta Assertion)	505		<i>Дельта-утверждение</i> (Delta Assertion)	21

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Тест инициализации зависимости</i> (Dependency Initialization Test)	381	Вариант	<i>Тестовый метод</i> (Test Method)	19
<i>Вставка зависимости</i> (Dependency Injection)	684		<i>Вставка зависимости</i> (Dependency Injection)	26
<i>Поиск зависимости</i> (Dependency Lookup)	692		<i>Поиск зависимости</i> (Dependency Lookup)	26
<i>Вычисленное ожидание</i> (Derived Expectation)	723	Вариант	<i>Вычисляемое значение</i> (Derived Value)	27
<i>Вычисленные входные данные</i> (Derived Input)	723	Вариант	<i>Вычисляемое значение</i> (Derived Value)	27
<i>Вычисляемое значение</i> (Derived Value)	722		<i>Вычисляемое значение</i> (Derived Value)	27
<i>Диагностическое утверждение</i> (Diagnostic Assertion)	497	Вариант	<i>Специальное утверждение</i> (Custom Assertion)	21
<i>Непосредственный вызов метода теста</i> (Direct Test Method Invocation)	427	Вариант	<i>Перечисление тестов</i> (Test Enumeration)	19
<i>Отдельное сгенерированное значение</i> (Distinct Generated Value)	727	Вариант	<i>Сгенерированное значение</i> (Generated Value)	27
<i>Утверждение для предметной области</i> (Domain Assertion)	497	Вариант	<i>Специальное утверждение</i> (Custom Assertion)	21
<i>Заглушка</i> (Dummy)	730	Вариант	<i>Объект-заглушка</i> (Dummy Object)	27
<i>Аргумент-заглушка</i> (Dummy Argument)	731	Вариант	<i>Объект-заглушка</i> (Dummy Object)	27
<i>Атрибут-заглушка</i> (Dummy Attribute)	731	Вариант	<i>Объект-заглушка</i> (Dummy Object)	27
<i>Объект-заглушка</i> (Dummy Object)	730		<i>Объект-заглушка</i> (Dummy Object)	27
<i>Параметр-заглушка</i> (Dummy Parameter)	730	Псевдоним	<i>Объект-заглушка</i> (Dummy Object)	27
<i>Значение-заглушка</i> (Dummy Value)	730	Псевдоним	<i>Объект-заглушка</i> (Dummy Object)	27
<i>Динамически генерируемый тестовый двойник</i> (Dynamically Generated Test Double)	574	Вариант	<i>Настраиваемый тестовый двойник</i> (Configurable Test Double)	23
<i>Обрезание цепочки сущностей</i> (Entity Chain Snipping)	546	Вариант	<i>Тестовая заглушка</i> (Test Stub)	23
<i>Утверждение равенства</i> (Equality Assertion)	393	Вариант	<i>Метод с утверждением</i> (Assertion Method)	19

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Описывающее ожидание сообщение</i> (Expectation-Describing Message)	399	Вариант	<i>Сообщение для утверждения</i> (Assertion Message)	19
<i>Ожидаемое поведение</i> (Expected Behavior)	491	Псевдоним	<i>Проверка поведения</i> (Behavior Verification)	21
<i>Спецификация ожидаемого поведения</i> (Expected Behavior Specification)	491	Вариант	<i>Проверка поведения</i> (Behavior Verification)	21
<i>Утверждение об ожидаемом исключении</i> (Expected Exception Assertion)	394	Вариант	<i>Метод с утверждением</i> (Assertion Method)	19
<i>Тест на ожидаемое исключение</i> (Expected Exception Test)	380	Вариант	<i>Тестовый метод</i> (Test Method)	19
<i>Ожидаемый объект</i> (Expected Object)	486	Псевдоним	<i>Проверка состояния</i> (State Verification)	21
<i>Спецификация ожидаемого состояния</i> (Expected State Specification)	486	Вариант	<i>Проверка состояния</i> (State Verification)	21
<i>Внешняя запись теста</i> (External Test Recording)	314	Вариант	<i>Записанный тест</i> (Recorded Test)	18
<i>Поддельная база данных</i> (Fake Database)	566	Вариант	<i>Поддельный объект</i> (Fake Object)	23
<i>Поддельный объект</i> (Fake Object)	565		<i>Поддельный объект</i> (Fake Object)	23
<i>Поддельный уровень обслуживания</i> (Fake Service Layer)	567	Вариант	<i>Поддельный объект</i> (Fake Object)	23
<i>Поддельная Web-служба</i> (Fake Web Service)	567	Вариант	<i>Поддельный объект</i> (Fake Object)	23
<i>Программа запуска тестов с файловой системы</i> (File System Test Runner)	498	Вариант	<i>Программа запуска тестов</i> (Test Runner)	19
<i>Тест с настройкой тестовой конфигурации</i> (Fixture Setup Testcase)	479	Вариант	<i>Цепочки тестов</i> (Chained Tests)	20
<i>Четырехфазный тест</i> (Four-Phase Test)	387		<i>Четырехфазный тест</i> (Four-Phase Test)	19
<i>Вызываемая инфраструктурой настройка</i> (Framework-Invoked Setup)	449	Псевдоним	<i>Неявная настройка</i> (Implicit Setup)	20
<i>Вызываемая инфраструктурой очистка</i> (Framework-Invoked Teardown)	533	Псевдоним	<i>Неявная очистка</i> (Implicit Teardown)	22
<i>Новый контекст</i> (Fresh Context)	344	Псевдоним	<i>Новая тестовая конфигурация</i> (Fresh Fixture)	18

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Новая тестовая конфигурация</i> (Fresh Fixture)	344		<i>Новая тестовая конфигурация</i> (Fresh Fixture)	18
<i>Нечеткое утверждение равенства</i> (Fuzzy Equality Assertion)	393	Вариант	<i>Метод с утверждением</i> (Assertion Method)	19
<i>Очистка со сборкой мусора</i> (Garbage-Collected Teardown)	518		<i>Очистка со сборкой мусора</i> (Garbage-Collected Teardown)	22
<i>Сгенерированное значение</i> (Generated Value)	726		<i>Сгенерированное значение</i> (Generated Value)	27
<i>Глобальная тестовая конфигурация</i> (Global Fixture)	455	Вариант	<i>Предварительно созданная тестовая конфигурация</i> (Prebuilt Fixture)	20
<i>Программа запуска тестов с графическим интерфейсом</i> (Graphical Test Runner)	406	Вариант	<i>Программа запуска тестов</i> (Test Runner)	19
<i>Сторожевое утверждение</i> (Guard Assertion)	510		<i>Сторожевое утверждение</i> (Guard Assertion)	21
<i>Созданный вручную тестовый двойник</i> (Hand-Built Test Double)	573	Вариант	<i>Настраиваемый тестовый двойник</i> (Configurable Test Double)	23
<i>Созданный вручную сценарий теста</i> (Hand-Scripted Test)	319	Псевдоним	<i>Тест на основе сценария</i> (Scripted Test)	18
<i>Написанный вручную тест</i> (Hand-Written Test)	319	Псевдоним	<i>Тест на основе сценария</i> (Scripted Test)	18
<i>Фиксированный подставной объект</i> (Hard-Coded Mock Object)	581	Псевдоним	<i>Фиксированный тестовый двойник</i> (Hard-Coded Test Double)	23
<i>Фиксированный декоратор настройки</i> (Hard-Coded Setup Decorator)	473	Вариант	<i>Декоратор настройки</i> (Setup Decorator)	20
<i>Фиксированный тестовый двойник</i> (Hard-Coded Test Double)	581		<i>Фиксированный тестовый двойник</i> (Hard-Coded Test Double)	23
<i>Фиксированная тестовая заглушка</i> (Hard-Coded Test Stub)	581	Псевдоним	<i>Фиксированный тестовый двойник</i> (Hard-Coded Test Double)	23
<i>Фиксированное значение</i> (Hard-Coded Value)	718	Псевдоним	<i>Точное значение</i> (Literal Value)	27
<i>Настройка через ловушки</i> (Hooked Setup)	449	Псевдоним	<i>Неявная настройка</i> (Implicit Setup)	20

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Очистка через ловушки</i> (Hooked Teardown)	533	Псевдоним	<i>Неявная очистка</i> (Implicit Teardown)	22
<i>Минимальный адаптер контейнера</i> (Humble Container Adapter)	703	Вариант	<i>Минимальный объект</i> (Humble Object)	26
<i>Минимальный диалог</i> (Humble Dialog)	701	Вариант	<i>Минимальный объект</i> (Humble Object)	26
<i>Минимальный исполняемый файл</i> (Humble Executable)	701	Вариант	<i>Минимальный объект</i> (Humble Object)	26
<i>Минимальный объект</i> (Humble Object)	700		<i>Минимальный объект</i> (Humble Object)	26
<i>Минимальный контроллер транзакций</i> (Humble Transaction Controller)	702	Вариант	<i>Минимальный объект</i> (Humble Object)	26
<i>Немодифицируемая общая тестовая конфигурация</i> (Immutable Shared Fixture)	355	Вариант	<i>Общая тестовая конфигурация</i> (Shared Fixture)	18
<i>Неявная настройка</i> (Implicit Setup)	449		<i>Неявная настройка</i> (Implicit Setup)	20
<i>Неявная очистка</i> (Implicit Teardown)	533		<i>Неявная очистка</i> (Implicit Teardown)	22
<i>Самозванец</i> (Imposter)	538	Псевдоним	<i>Тестовый двойник</i> (Test Double)	23
<i>Хранящийся в базе данных тест хранимой процедуры</i> (In-Database Stored Procedure Test)	663	Вариант	<i>Тест хранимой процедуры</i> (Stored Procedure Test)	25
<i>База данных в памяти</i> (In-Memory Database)	567	Вариант	<i>Поддельный объект</i> (Fake Object)	23
<i>Инкрементный табличный тест</i> (Incremental Tabular Test)	620	Вариант	<i>Параметризованный тест</i> (Parametrized Test)	24
<i>Инкрементные тесты</i> (Incremental Tests)	353	Вариант	<i>Общая тестовая конфигурация</i> (Shared Fixture)	18
<i>Регистр опосредованного вывода</i> (Indirect Output Registry)	555	Вариант	<i>Тестовый агент</i> (Test Spy)	23
<i>Встроенная настройка</i> (In-line Setup)	434		<i>Встроенная настройка</i> (In-line Setup)	20
<i>Встроенная очистка</i> (In-line Teardown)	527		<i>Встроенная очистка</i> (In-line Teardown)	22
<i>Внутренний тестовый двойник</i> (Inner Test Double)	583	Вариант	<i>Фиксированный тестовый двойник</i> (Hard-Coded Test Double)	23
<i>Тестирование взаимодействия</i> (Interaction Testing)	489	Псевдоним	<i>Проверка поведения</i> (Behavior Verification)	21

<i>Продолжение таблицы</i>				
Имя	Страница	Отношение	Базовое имя	Глава
<i>Тест уровня</i> (Layer Test)	368		<i>Тест уровня</i> (Layer Test)	18
<i>Тест с пересечением уровней</i> (Layer-Crossing Test)	359	Псевдоним	<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	18
<i>Одноуровневый тест</i> (Layered Test)	368	Псевдоним	<i>Тест уровня</i> (Layer Test)	18
“Ленивая” настройка (Lazy Setup)	460		“Ленивая” настройка (Lazy Setup)	20
“Ленивая” очистка (Lazy Teardown)	670	Вариант	<i>Очистка усечением таблиц</i> (Table Truncation Teardown)	25
<i>Оставшаяся тестовая конфигурация</i> (Leftover Fixture)	350	Псевдоним	<i>Общая тестовая конфигурация</i> (Shared Fixture)	18
<i>Точное значение</i> (Literal Value)	718		<i>Точное значение</i> (Literal Value)	27
<i>Тест на основе цикла</i> (Loop-Driven Test)	621	Вариант	<i>Параметризованный тест</i> (Parametrized Test)	24
<i>Минимальная тестовая конфигурация</i> (Minimal Fixture)	336		<i>Минимальная тестовая конфигурация</i> (Minimal Fixture)	18
<i>Минимальный контекст</i> (Minimal Context)	336	Псевдоним	<i>Минимальная тестовая конфигурация</i> (Minimal Fixture)	18
<i>Подставной объект</i> (Mock Object)	558		<i>Подставной объект</i> (Mock Object)	23
<i>Простейшая встроенная очистка</i> (Naive In-line Teardown)	529	Вариант	<i>Встроенная очистка</i> (In-line Teardown)	22
<i>Простейший интерпретатор тестов</i> xUnit (Naive xUnit Test Interpreter)	326	Вариант	<i>Управляемый данными тест</i> (Data-Driven Test)	18
<i>Метод, достигающий указанного состояния</i> (Named State Reaching Method)	443	Вариант	<i>Метод создания</i> (Creation Method)	20
<i>Именованный набор тестов</i> (Named Test Suite)	604		<i>Именованный набор тестов</i> (Named Test Suite)	24
<i>Утверждение равенства атрибутов объекта</i> (Object Attribute Equality Assertion)	497	Вариант	<i>Специальное утверждение</i> (Custom Assertion)	21
<i>Фабрика объектов</i> (Object Factory)	692	Псевдоним	<i>Поиск зависимости</i> (Dependency Lookup)	26

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Инкубатор объектов (Object Mother)</i>	652	Вариант	<i>Вспомогательный класс теста (Test Helper)</i>	24
<i>Единственный дефектный атрибут (One Bad Attribute)</i>	723	Вариант	<i>Вычисляемое значение (Derived Value)</i>	27
<i>Вставка параметра (Parameter Injection)</i>	686	Вариант	<i>Вставка зависимости (Dependency Injection)</i>	26
<i>Параметризованный анонимный метод создания (Parameterized Anonymous Creation Method)</i>	443	Вариант	<i>Метод создания (Creation Method)</i>	20
<i>Параметризованный метод создания (Parameterized Creation Method)</i>	442	Вариант	<i>Декоратор настройки (Setup Decorator)</i>	20
<i>Параметризованный тест (Parameterized Test)</i>	618		<i>Параметризованный тест (Parameterized Test)</i>	21
<i>Тестовая конфигурация для каждого запуска (Per-Run Fixture)</i>	355	Вариант	<i>Общая тестовая конфигурация (Shared Fixture)</i>	18
<i>Тест уровня хранения (Persistence Layer Test)</i>	370	Вариант	<i>Тест уровня (Layer Test)</i>	18
<i>Постоянная новая тестовая конфигурация (Persistent Fresh Fixture)</i>	347	Вариант	<i>Новая тестовая конфигурация (Fresh Fixture)</i>	18
<i>Заменитель (Placeholder)</i>	730	Псевдоним	<i>Объект-заглушка (Dummy Object)</i>	27
<i>Простейший минимальный объект (Poor Man's Humble Object)</i>	703	Вариант	<i>Минимальный объект (Humble Object)</i>	26
<i>Предварительно созданный контекст (Prebuilt Context)</i>	454	Псевдоним	<i>Предварительно созданная тестовая конфигурация (Prebuilt Fixture)</i>	20
<i>Предварительно созданная тестовая конфигурация (Prebuilt Fixture)</i>	454		<i>Предварительно созданная тестовая конфигурация (Prebuilt Fixture)</i>	20
<i>Тест презентационного уровня (Presentation Layer Test)</i>	369	Вариант	<i>Тест уровня (Layer Test)</i>	18
<i>Закрытая тестовая конфигурация (Private Fixture)</i>	343	Псевдоним	<i>Новая тестовая конфигурация (Fresh Fixture)</i>	18
<i>Процедурная проверка поведения (Procedural Behavior Verification)</i>	491	Вариант	<i>Проверка поведения (Behavior Verification)</i>	21
<i>Процедурная проверка состояния (Procedural State Verification)</i>	485	Вариант	<i>Проверка состояния (State Verification)</i>	21

<i>Продолжение таблицы</i>				
Имя	Страница	Отношение	Базовое имя	Глава
<i>Процедурная тестовая заглушка</i> (Procedural Test Stub)	546	Вариант	<i>Тестовая заглушка</i> (Test Stub)	23
<i>Программный тест</i> (Programmatic Test)	319	Псевдоним	<i>Тест на основе сценария</i> (Scripted Test)	18
<i>Псевдообъект</i> (Pseudo-Object)	584	Вариант	<i>Фиксированный тестовый двойник</i> (Hard-Coded Test-Double)	23
<i>Декоратор низкого уровня</i> (Pushdown Decorator)	473	Вариант	<i>Декоратор настройки</i> (Setup Decorator)	20
<i>Случайное сгенерированное значение</i> (Random Generated Value)	727	Вариант	<i>Сгенерированное значение</i> (Generated Value)	27
<i>Тест на основе записи и воспроизведения</i> (Record and Playback Test)	312	Псевдоним	<i>Записанный тест</i> (Recorded Test)	18
<i>Переработанный записанный тест</i> (Refactored Recorded Test)	314	Вариант	<i>Записанный тест</i> (Recorded Test)	18
<i>Связанное сгенерированное значение</i> (Related Generated Value)	727	Вариант	<i>Сгенерированное значение</i> (Generated Value)	27
<i>Удаленный тест хранимой процедуры</i> (Remoted Stored Procedure Test)	664	Вариант	<i>Тест хранимой процедуры</i> (Stored Procedure Test)	25
<i>Генератор ответов</i> (Responder)	545	Вариант	<i>Тестовая заглушка</i> (Test Stub)	23
<i>Интерфейс извлечения</i> (Retrieval Interface)	554	Вариант	<i>Тестовый агент</i> (Test Spy)	23
<i>Повторное использование теста для настройки тестовой конфигурации</i> (Reuse Test for Fixture Setup)	444	Вариант	<i>Метод создания</i> (Creation Method)	20
<i>Повторно используемая тестовая конфигурация</i> (Reused Fixture)	350	Псевдоним	<i>Общая тестовая конфигурация</i> (Shared Fixture)	18
<i>Тест “роботом”-пользователем</i> (Robot User Test)	312	Псевдоним	<i>Записанный тест</i> (Recorded Test)	18
<i>Инфраструктура тестирования “роботом”-пользователем</i> (Robot User Test Framework)	333	Вариант	<i>Инфраструктура автоматизации тестов</i> (Test Automation Framework)	18
<i>Строковый тест</i> (Row Test)	620	Псевдоним	<i>Параметризованный тест</i> (Parameterized Test)	24
<i>Инкапсуляция программного интерфейса testируемой системы</i> (SUT API Encapsulation)	612	Псевдоним	<i>Вспомогательный метод теста</i> (Test Utility Method)	24
<i>Метод инкапсуляции testируемой системы</i> (SUT Encapsulation Method)	612	Вариант	<i>Вспомогательный метод теста</i> (Test Utility Method)	24

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Диверсант (Saboteur)</i>	545	Вариант	<i>Тестовая заглушка (Test Stub)</i>	23
<i>Тест на основе сценария (Scripted Test)</i>	319		<i>Тест на основе сценария (Scripted Test)</i>	18
<i>Тестовый шунт (Self Shunt)</i>	583	Вариант	<i>Фиксированный тестовый двойник (Hard-Coded Test Double)</i>	23
<i>Описательное значение (Self-Describing Value)</i>	719	Вариант	<i>Точное значение (Literal Value)</i>	27
<i>Тест служебного уровня (Service Layer Test)</i>	370	Вариант	<i>Тест уровня (Layer Test)</i>	18
<i>Локатор служб (Service Locator)</i>	692	Псевдоним	<i>Поиск зависимости (Dependency Lookup)</i>	26
<i>Вставка метода установки (Setter Injection)</i>	687	Вариант	<i>Вставка зависимости (Dependency Injection)</i>	26
<i>Декоратор настройки (Setup Decorator)</i>	471		<i>Декоратор настройки (Setup Decorator)</i>	20
<i>Общий контекст (Shared Context)</i>	350	Псевдоним	<i>Общая тестовая конфигурация (Shared Fixture)</i>	18
<i>Общая тестовая конфигурация (Shared Fixture)</i>	350		<i>Общая тестовая конфигурация (Shared Fixture)</i>	18
<i>Утверждение о состоянии общей тестовой конфигурации (Shared Fixture State Assertion)</i>	511	Вариант	<i>Сторожевое утверждение (Guard Assertion)</i>	21
<i>Общий метод настройки (Shared Setup Method)</i>	449	Псевдоним	<i>Неявная настройка (Implicit Setup)</i>	20
<i>Простой тест успешности (Simple Success Test)</i>	379	Вариант	<i>Тестовый метод (Test Method)</i>	19
<i>Тест одного уровня (Single-Layer Test)</i>	368	Псевдоним	<i>Тест уровня (Layer Test)</i>	18
<i>Утверждение с единственным результатом (Single Outcome Assertion)</i>	394	Вариант	<i>Метод с утверждением (Assertion Method)</i>	19
<i>Набор из одного теста (Single Test Suite)</i>	605	Вариант	<i>Именованный набор тестов (Named Test Suite)</i>	24
<i>Медленный тест (Slow Test)</i>	351	Вариант	<i>Общая тестовая конфигурация (Shared Fixture)</i>	18
<i>Агент (Spy)</i>	552	Псевдоним	<i>Тестовый агент (Test Spy)</i>	23

<i>Продолжение таблицы</i>				
Имя	Страница	Отношение	Базовое имя	Глава
<i>Устаревшая тестовая конфигурация</i> (Stale Fixture)	350	Псевдоним	<i>Общая тестовая конфигурация</i> (Shared Fixture)	18
<i>Стандартный контекст</i> (Standard Context)	338	Псевдоним	<i>Стандартная тестовая конфигурация</i> (Standard Fixture)	18
<i>Стандартная тестовая конфигурация</i> (Standard Fixture)	338		<i>Стандартная тестовая конфигурация</i> (Standard Fixture)	18
<i>Проверка состояния</i> (State Verification)	484		<i>Проверка состояния</i> (State Verification)	21
<i>Тестирование на основе состояния</i> (State-Based Testing)	484	Псевдоним	<i>Проверка состояния</i> (State Verification)	21
<i>Открывающий состояние подкласс</i> (State-Exposing Subclass)	592	Вариант	<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	23
<i>Утверждение с заявленным результатом</i> (Stated Outcome Assertion)	393	Вариант	<i>Метод с утверждением</i> (Assertion Method)	19
<i>Статически сгенерированный тестовый двойник</i> (Statically Generated Test Double)	574	Вариант	<i>Настраиваемый тестовый двойник</i> (Configurable Test Double)	23
<i>Тест хранимой процедуры</i> (Stored Procedure Test)	662		<i>Тест хранимой процедуры</i> (Stored Procedure Test)	25
<i>Заглушка</i> (Stub)	544	Псевдоним	<i>Тестовая заглушка</i> (Test Stub)	23
<i>Заглушка</i> (Stub)	730	Псевдоним	<i>Объект-заглушка</i> (Dummy Object)	27
<i>Минимальный объект подкласса</i> (Subclassed Humble Object)	705	Вариант	<i>Минимальный объект</i> (Humble Object)	26
<i>Единственный экземпляр подкласса</i> (Subclasred Singleton)	593	Псевдоним	<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	23
<i>Тестовый двойник в виде подкласса</i> (Subclassed Test Double)	592	Псевдоним	<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	23
<i>“Подкожный” тест</i> (Subcutaneous Test)	370	Вариант	<i>Тест уровня</i> (Layer Test)	18
<i>Набор с подмножеством</i> (Subset Suite)	605	Вариант	<i>Именованный набор тестов</i> (Named Test Suite)	24
<i>Заменяемый единственный экземпляр класса</i> (Substitutable Singleton)	593	Псевдоним	<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	23

Продолжение таблицы

Имя	Страница	Отношение	Базовое имя	Глава
<i>Замененный единственный экземпляр класса</i> (Substituted Singleton)	593	Вариант	<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	23
<i>Набор наборов</i> (Suite of Suites)	415	Вариант	<i>Объект набора тестов</i> (Test Suite Object)	19
<i>Настройка тестовой конфигурации набора</i> (Suite Fixture Setup)	465		<i>Настройка тестовой конфигурации набора</i> (Suite Fixture Setup)	20
<i>Символьная константа</i> (Symbolic Constant)	719	Вариант	<i>Точное значение</i> (Literal Value)	27
<i>Очистка усечением таблиц</i> (Table Truncation Teardown)	668		<i>Очистка усечением таблиц</i> (Table Truncation Teardown)	25
<i>Табличный тест</i> (Tabular Test)	620	Вариант	<i>Параметризованный тест</i> (Parameterized Test)	24
<i>Пункт охраны очистки</i> (Teardown Guard Clause)	528	Вариант	<i>Встроенная очистка</i> (In-line Teardown)	22
<i>Временная тестовая заглушка</i> (Temporary Test Stub)	545	Вариант	<i>Тестовая заглушка</i> (Test Stub)	23
<i>Инфраструктура автоматизации тестов</i> (Test Automation Framework)	332		<i>Инфраструктура автоматизации тестов</i> (Test Automation Framework)	18
<i>Тестовый стенд</i> (Test Bed)	454	Псевдоним	<i>Предварительно созданная тестовая конфигурация</i> (Prebuilt Fixture)	20
<i>Обнаружение тестов</i> (Test Discovery)	420		<i>Обнаружение тестов</i> (Test Discovery)	19
<i>Тестовый двойник</i> (Test Double)	538		<i>Тестовый двойник</i> (Test Double)	23
<i>Класс тестового двойника</i> (Test Double Class)	582	Вариант	<i>Фиксированный тестовый двойник</i> (Hard-Coded Test Double)	23
<i>Подкласс как тестовый двойник</i> (Test Double Subclass)	592	Вариант	<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	23
<i>Тестовый двойник как “черный ход”</i> (Test Double as Back Door)	363	Вариант	<i>Манипуляция через “черный ход”</i> (Back Door Manipulation)	18
<i>Перечисление тестов</i> (Test Enumeration)	425		<i>Перечисление тестов</i> (Test Enumeration)	19
<i>Тестовая конфигурация</i> (Test Fixture)	401	Псевдоним	<i>Класс теста</i> (Testcase Class)	19

<i>Продолжение таблицы</i>				
Имя	Страница	Отношение	Базовое имя	Глава
<i>Регистр тестовых конфигураций</i> (Test Fixture Registry)	652	Вариант	<i>Вспомогательный класс теста</i> (Test Helper)	24
<i>Вспомогательный класс теста</i> (Test Helper)	651		<i>Вспомогательный класс теста</i> (Test Helper)	24
<i>Вспомогательный класс теста</i> (Test Helper Class)	653	Вариант	<i>Вспомогательный класс теста</i> (Test Helper)	24
<i>Вспомогательный миксин</i> (Test Helper Mixin)	647	Вариант	<i>Суперкласс теста</i> (Testcase Superclass)	24
<i>Вспомогательный объект</i> (Test Helper Object)	653	Вариант	<i>Вспомогательный класс теста</i> (Test Helper)	24
<i>Ловушка для теста</i> (Test Hook)	713		<i>Ловушка для теста</i> (Test Hook)	26
<i>Тестовый метод</i> (Test Method)	378		<i>Тестовый метод</i> (Test Method)	19
<i>Обнаружение тестовых методов</i> (Test Method Discovery)	421	Вариант	<i>Обнаружение тестов</i> (Test Discovery)	19
<i>Перечисление тестовых методов</i> (Test Method Enumeration)	426	Вариант	<i>Перечисление тестов</i> (Test Enumeration)	19
<i>Выбор тестовых методов</i> (Test Method Selection)	430	Вариант	<i>Выбор тестов</i> (Test Selection)	19
<i>Регистр тестовых объектов</i> (Test Object Registry)	521	Псевдоним	<i>Автоматическая очистка</i> (Automated Teardown)	22
<i>Программа запуска тестов</i> (Test Runner)	405		<i>Программа запуска тестов</i> (Test Runner)	19
<i>Выбор тестов</i> (Test Selection)	429		<i>Выбор тестов</i> (Test Selection)	19
<i>Тестовый агент</i> (Test Spy)	552		<i>Тестовый агент</i> (Test Spy)	23
<i>Тестовая заглушка</i> (Test Stub)	544		<i>Тестовая заглушка</i> (Test Stub)	23
<i>Перечисление наборов тестов</i> (Test Suite Enumeration)	426	Вариант	<i>Перечисление тестов</i> (Test Enumeration)	19
<i>Фабрика наборов тестов</i> (Test Suite Factory)	425	Псевдоним	<i>Перечисление тестов</i> (Test Enumeration)	19
<i>Объект набора тестов</i> (Test Suite Object)	414		<i>Объект набора тестов</i> (Test Suite Object)	19
<i>Генератор объекта набора тестов</i> (Test Suite Object Generator)	326	Вариант	<i>Управляемый данными тест</i> (Data-Driven Test)	18

<i>Продолжение таблицы</i>				
Имя	Страница	Отношение	Базовое имя	Глава
<i>Имитатор объекта набора тестов</i> (Test Suite Object Simulator)	327	Вариант	<i>Управляемый данными тест</i> (Data-Driven Test)	19
<i>Обозреватель набора тестов</i> (Test Tree Explorer)	408	Вариант	<i>Программа запуска тестов</i> (Test Runner)	19
<i>Вспомогательный метод теста</i> (Test Utility Method)	610		<i>Вспомогательный метод теста</i> (Test Utility Method)	24
<i>Тест вспомогательного метода теста</i> (Test Utility Test)	614	Вариант	<i>Вспомогательный метод теста</i> (Test Utility Method)	24
<i>Связанное с тестом расширение</i> (Test-Specific Extension)	591	Псевдоним	<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	23
<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	591		<i>Связанный с тестом подкласс</i> (Test-Specific Subclass)	23
<i>Класс теста</i> (Testcase Class)	401		<i>Класс теста</i> (Testcase Class)	19
<i>Обнаружение классов теста</i> (Testcase Class Discovery)	421	Вариант	<i>Обнаружение тестов</i> (Test Discovery)	19
<i>Класс теста для каждого класса</i> (Testcase Class per Class)	627		<i>Класс теста для каждого класса</i> (Testcase Class per Class)	24
<i>Класс теста для каждой функции</i> (Testcase Class per Feature)	633		<i>Класс теста для каждой функции</i> (Testcase Class per Feature)	24
<i>Класс теста для каждой тестовой конфигурации</i> (Testcase Class per Fixture)	639		<i>Класс теста для каждой тестовой конфигурации</i> (Testcase Class per Fixture)	24
<i>Класс теста для каждого метода</i> (Testcase Class per Method)	634	Вариант	<i>Класс теста для каждой функции</i> (Testcase Class per Feature)	24
<i>Класс теста для каждого пользовательского сценария</i> (Testcase Class per User Story)	634	Вариант	<i>Класс теста для каждой функции</i> (Testcase Class per Feature)	24
<i>Выбор классов теста</i> (Testcase Class Selection)	430	Вариант	<i>Выбор тестов</i> (Test Selection)	19
<i>Набор классов тестов</i> (Testcase Class Suite)	414	Вариант	<i>Объект набора тестов</i> (Test Suite Object)	19
<i>Объект теста</i> (Testcase Object)	410		<i>Объект теста</i> (Testcase Object)	19
<i>Суперкласс теста</i> (Testcase Superclass)	646		<i>Суперкласс теста</i> (Testcase Superclass)	24

<i>Окончание таблицы</i>				
Имя	Страница	Отношение	Базовое имя	Глава
<i>Тестирование уровней</i> (Testing by Layers)	369	Псевдоним	<i>Тест уровня</i> (Layer Test)	18
Семейство xUnit	334	Вариант	<i>Инфраструктура автоматизации тестов</i> (Test Automation Framework)	18
<i>Очистка откатом транзакции</i> (Transaction Rollback Teardown)	675		<i>Очистка откатом транзакции</i> (Transaction Rollback Teardown)	25
<i>Временная новая тестовая конфигурация</i> (Transient Fresh Fixture)	347	Вариант	<i>Новая тестовая конфигурация</i> (Fresh Fixture)	18
<i>Истинный минимальный объект</i> (True Humble Object)	704	Вариант	<i>Минимальный объект</i> (Humble Object)	26
<i>Утверждение незаконченного теста</i> (Unfinished Test Assertion)	514		<i>Утверждение незаконченного теста</i> (Unfinished Test Assertion)	21
<i>Метод проверки</i> (Verification Method)	498	Вариант	<i>Специальное утверждение</i> (Custom Assertion)	21

Словарь терминов

Словарь терминов включает авторские определения терминов, используемых в настоящей книге.

ACID

Четыре характеристики транзакции, обеспечиваемые современными базами данных.

- *Атомарность* (atomic). Транзакция выполняется полностью или не выполняется вообще.
- *Единообразие* (consistent). Все операции внутри транзакции работают с одинаковым представлением внешнего мира.
- *Независимость* (independent). Транзакции независимы одна от другой.
- *Постоянство* (durable). После завершения транзакции внесенные изменения являются постоянными (они не могут просто так исчезнуть по неизвестной причине!).

BDUF

“Изначально полный проект” (“Big design up front”) является классическим “каскадным” подходом к проектированию программного обеспечения. При этом все требования должны быть обозначены на начальных стадиях проекта, а дизайн программного продукта должен быть завершен на единственном “этапе проектирования”. Противоположностью является *возникающий дизайн* (emergent design), характерный для проектов с гибкой разработкой.

DfT

См. *проектирование с учетом тестирования* (design for testability).

DTO

Сокращение от названия шаблона *объект передачи данных* (Data Transfer Object) [CJ2EEP].

EDD

См. *разработка на основе примеров* (example-driven development).

Автоматизатор тестов (test automater)

Исполнитель (или роль в проекте), задачей которого является написание тестов. Иногда тесты для автоматизации предоставляются экспертом в предметной области.

Анализ причин (root cause analysis)

Процесс обнаружения всех факторов, связанных с причиной возникновения ошибки или отказа. Анализ причин позволяет избежать лечения симптомов и обнаружить реальные причины проблемы. Существует множество способов обнаружения причин, включая пять “Почему?”, применяемых в компании Toyota.

Аннотация (annotation)

Способ сообщить что-то о чем-то. В пакете JUnit 4.0 аннотации используются для перечисления классов *теста* (Testcase Class) и тестовых методов (Test Method). В пакете NUnit для этого предназначены атрибуты .NET.

Анонимный внутренний класс (anonymous inner class)

Внутренний класс в Java, который определяется без присвоения уникального имени. Часто анонимные внутренние классы используются при определении *фиксированных тестовых двойников* (Hard-Coded Test Double).

Антишаблон (anti-pattern)

Шаблон, который не должен использоваться, так как известно, что он дает неоптимальный результат. Запахи кода и причины их возникновения являются типами антишаблонов.

Асинхронный тест (asynchronous test)

Тест, который работает в отдельном потоке управления относительно тестируемой системы и взаимодействует с ней с помощью асинхронных (или “настоящих”) сообщений. Асинхронный тест должен координировать свои действия с действиями тестируемой системы, так как система времени выполнения не может управлять этим взаимодействием. Асинхронный тест может включать в себя задержки для предоставления тестируемой системе времени на выполнение запросов перед проверкой результата. Сравните это с поведением синхронного теста, который взаимодействует с системой через вызовы методов.

Аспектно-ориентированное программирование (aspect-oriented programming)

Усложненная методика разделения программного обеспечения на модули, позволяющая распределять задачи и “вплетать” пересекающиеся функции в код после компиляции программного продукта, но перед его запуском.

Атрибут (attribute)

Характеристика чего-то. Пакеты семейства xUnit для платформы .NET используют атрибуты методов и классов для перечисления *классов теста* (Testcase Class) и *тестовых методов* (Test Method). Иногда термин “атрибут” является синонимом термина “переменная экземпляра класса”.

Атрибут класса (class attribute)

Атрибут, добавленный в исходный код класса и сообщающий компилятору или среде выполнения, что класс является “специальным”. В некоторых пакетах семейства xUnit атрибуты используются для выделения *классов тестов* (Testcase Class).

Атрибут метода (method attribute)

Атрибут, который присваивается методу в исходном коде для выделения метода как “специального” с точки зрения компилятора или среды выполнения. В некоторых пакетах семейства xUnit атрибуты методов используются для перечисления *тестовых методов* (Test Method).

Бизнес-логика (business logic)

Логика ядра системы, связанная с проблемной областью. Поскольку бизнес-логика обычно отражает результаты многих независимых бизнес-решений, она может быть чем угодно, но только не логикой!

Блок (block)

Блок кода, который можно запустить. Блоки (также известны как **блочные конструкции**) используются во многих языках программирования (особенно в Smalltalk и Ruby). Блоки используются в качестве способа передачи фрагмента кода методу. После этого метод выполняет полученный код в собственном контексте. В языке Java без непосредственной поддержки блоков такого поведения можно достичь с помощью анонимных внутренних классов. В языке C# для этого используются делегаты.

Блочная конструкция (block closure)

См. *блок* (block).

Внутренний класс (inner class)

Класс на языке Java, который определен внутри другого класса. Анонимные внутренние классы определяются внутри методов, а внутренние классы определяются за пределами методов. Часто внутренние классы используются для определения *фиксированного тестового двойника* (Hard-Coded Test Double).

Возникающий дизайн (emergent design)

Противоположность BDUF (полный проект перед началом разработки). Возникающий дизайн предполагает развитие правильного дизайна в процессе медленной эволюции программного продукта по мере удовлетворения тестов.

Встроенный автотест (built-in self-test)

Способ организации тестового кода, в котором тесты существуют в том же модуле или классе, что и код продукта. При этом тесты запускаются в момент инициализации системы.

Вызов тестируемой системы (exercise SUT)

Этап тестирования, на котором тест стимулирует логику тестируемой системы после настройки *тестовой конфигурации* (fixture setup).

Вызываемый компонент (depended-on component)

Отдельный класс или крупный компонент, от которого зависит тестируемая система. Обычно такая зависимость принимает форму делегирования через вызов методов. При автоматизации тестов вызываемый компонент рассматривается с точки зрения его взаимодействия с тестируемой системой.

Гвоздь (spike)

В гибких методиках разработки (например, eXtreme Programming) — ограниченный во времени эксперимент для получения оценки трудозатрат на реализацию новой функциональности.

Гибкий метод (agile method)

Метод работы над проектами (обычно по созданию программных продуктов, но не только), снижающий стоимость изменений и позволяющий потребителю продукта получить больше контроля над расходами и результатом. К гибким методам относятся eXtreme Programming, SCRUM, Feature-Driven Development (FDD) и Dynamic Systems Development Method (DSDM). Основным элементом большинства гибких методов является автоматизированный модульный тест.

Главный вход (front door)

Открытый программный интерфейс для программного продукта. Противоположность **черному ходу** (back door).

Глобальная переменная (global variable)

Переменная, которая считается глобальной с точки зрения всего приложения. Она доступна из любой точки программы и никогда не выходит из области видимости, хотя выделенная для нее память может быть явно освобождена.

Границочное значение (boundary value)

Входящее значение тестируемой системы, которое находится непосредственно возле границы двух **классов эквивалентности** (equivalence class). Тесты на основе соседних граничных значений позволяют убедиться, что поведение меняется только при передаче правильных входных данных.

Динамическое связывание (dynamic binding)

Ситуация, когда решение о выборе программного компонента, которому будет передаваться управление, откладывается до времени выполнения. Одно и то же имя метода может использоваться для вызова различных поведений (тел метода) в зависимости от класса объекта, в котором он вызывается. При этом класс выбирается только на этапе выполнения. Динамическое связывание является противоположностью **статического связывания** (static binding). Еще одно название — **полиморфизм** (polymorphism; в переводе с латыни означает “иметь много форм”).

Живое обеспечение (liveware)

Люди, которые используют программные продукты. Обычно считается, что они намного умнее программного обеспечения или аппаратных средств, но иногда они могут действовать совершенно непредсказуемо.

Задача (task)

Единица выделения трудозатрат в экстремальном программировании. Для реализации **пользовательской истории** (user story) или **функции** (feature), возможно, придется выполнить одну или более задач.

Задолженность по тестам (test debt)

С концепцией различных задолженностей автор впервые столкнулся в списке рассылки по индустриальному экстремальному программированию. Концепция задолженности является метафорой недостаточного выполнения каких-либо действий. Для выхода из состояния задолженности необходимо приложить больше усилий в той области, где они были недостаточны. Задолженность по тестам возрастает, если все необходимые тесты не написаны. В результате появляется “незащищенный код”, который может некорректно работать даже при успешном завершении всех тестов.

Заменяемая зависимость (substitutable dependency)

Программный компонент может зависеть от любого количества других компонентов. Если приходится тестировать данный компонент, должна существовать возможность заменить другие компоненты *тестовым двойником* (Test Double), т.е. каждый компонент должен быть заменяемой зависимостью. Для получения заменяемой зависимости можно воспользоваться рядом подходов, включая *вставку зависимости* (Dependency Injection), *поиск зависимости* (Dependency Lookup) и *связанный с тестом подкласс* (Test-Specific Subclass).

Запах (smell)

Симптом проблемы. Запах не обязательно указывает на конкретную проблему, так как причин его появления может быть несколько. Запах имеет свойство ясно указывать на наличие проблемы. Для определения причины придется провести дополнительный анализ.

Запахи классифицируются по месту их обнаружения. Наиболее распространенными запахами продукта являются **запах кода** (code smell), **запах поведения** (behavior smell) и **запах проекта** (project smell). **Запахи тестов** (test smell) могут проявляться как **запахи кода** (code smell) или **запах поведения** (behavior smell).

Запах кода (code smell)

Классический “неприятный” запах, описанный Мартином Фаулером. Специалисты по автоматизации тестов должны распознавать этот запах в процессе обслуживания кода тестов. Обычно такой запах отрицательно влияет на стоимость обслуживания, но может стать и ранним предупреждением возможности появления запахов поведения в будущем.

См. *запах теста* (test smell), *запах поведения* (behavior smell), *запах проекта* (project smell)

Запах поведения (behavior smell)

Запах теста, проявляющийся при компиляции и запуске теста. Для выявления запахов поведения особая наблюдательность не нужна, так как они проявляются в виде ошибок компиляции или неудачных завершений тестов.

Запах проекта (project smell)

Признак присутствия в проекте недостатков. Причиной его появления, скорее всего, служит один или несколько **запахов кода** (code smell) или **запахов поведения** (behavior smell). Поскольку руководители проектов редко запускают или создают тесты, обнаруженные ими запахи проектов являются первым признаком ошибок в области автоматизации тестов.

Запах теста (test smell)

Симптом проблем, существующих в **коде теста** (test code). Запах не всегда указывает на конкретную причину, так как причин может быть несколько.

Запрос (pull)

Концепция производства компонентов в фактически затребованных объемах. В такой системе линии сборки компонентов производят максимум столько единиц, сколько было изъято со склада, который выступает в роли буфера для использующих компонент линий сборки. В разработке программного обеспечения эту идею можно сформулировать следующим образом: “Имеет смысл писать только те методы, которые уже вызывались другими компонентами, и решать те проблемы, которые действительно влияют на работу других компонентов”. Этот подход позволяет избежать написания ненужного программного кода, который в процессе разработки рассматривается как склад (и в некоторых концепциях производства считается напрасно потраченным ресурсом).

Запуск теста (test run)

Тест (test) или **набор тестов** (test suite) может запускаться несколько раз, каждый раз возвращая новый результат. Некоторые коммерческие инструменты автоматизации тестов записывают результат каждого запуска для последующего сравнения.

Зеленая полоса (green bar)

Многие программы запуска тестов с графическим интерфейсом (Graphical Test Runner) описывают текущее состояние набора тестов с помощью полосы состояния. Если все тесты завершились удачно, полоса остается зеленой. Если хотя бы один тест завершился неудачно, полоса становится **красной** (red bar).

Инвертирование управления (inversion of control — IOC)

Парадигма управления, различающая программные инфраструктуры и “наборы инструментов” или компоненты. Инфраструктура вызывает встраиваемый модуль (а не наоборот). В реальном мире инвертирование управления обычно называется принципом Голливуда. С ростом популярности автоматизированных модульных тестов инфраструктура инвертирования управления появилась специально для обеспечения простоты замены вызываемого компонента *тестовым двойником* (Test Double).

Инкрементная доставка (incremental delivery)

Метод поэтапной компиляции и развертывания программной системы по мере завершения работы над каждым этапом (инкрементом). Такой подход позволяет обеспечить раннюю доставку работающей системы пользователям. При этом возможности системы растут со временем. При использовании гибких методов разработки инкрементом функциональности является **функция** (feature) или **пользовательская история** (user story). Инкрементная доставка расширяет итеративную разработку и инкрементную разработку, фактически регулярно добавляя функции в готовый продукт. Эта идея выражается принципом “быстрой и частой доставки”.

Инкрементная разработка (incremental development)

Метод поэтапного создания программного продукта, подразумевающий тестирование имеющейся функциональности перед переходом к следующему этапу. Такой подход позволяет обеспечить раннюю доставку работающей системы потребителю. При этом возможности системы будут расти со временем (за счет инкрементной доставки). При использовании гибких методов разработки инкрементом функциональности является **функция** (feature) или **пользовательская история** (user story). Инкрементная разработка расширяет итеративную разработку, так как позволяет создавать работающий, протестированный и потенциально готовый для доставки программный продукт на каждой итерации. Вместе с инкрементной доставкой этот подход позволяет обеспечить “быструю и частую доставку”.

Интегрированная среда разработки (IDE)

Среда, обеспечивающая средства для редактирования, компиляции, запуска и (обычно) тестирования кода в пределах одного приложения.

Интерфейс (interface)

Обычно это полностью абстрактный класс, определяющий только открытые методы, которые должны быть реализованы при наследовании интерфейса. В языке программирования Java интерфейс представляет собой определение типа, не содержащее реализаций. В большинстве языков программирования без множественного наследования класс может реализовывать любое количество интерфейсов, но быть подклассом только одного класса.

Интроспекция (reflection)

Способность программы просматривать собственную структуру во время выполнения. Интроспекция часто используется в инструментах разработки для поддержки добавления новых возможностей.

Исправление ошибок с использованием тестов (test-driven bug fixing)

Способ исправления ошибок, подразумевающий создание и автоматизацию модульных тестов, воспроизводящих каждую ошибку, еще до отладки кода и исправления ошибки. Это расширение процесса разработки на основе тестов, используемое для исправления ошибок.

Исследовательское тестирование (exploratory testing)

Интерактивное тестирование приложения без автоматизированных сценариев. Тестер “исследует” систему, разрабатывая теории о ее поведении в зависимости от действий приложения. После этого теории проверяются с помощью тестов. Хотя такое тестирование не имеет четкого плана, такой подход позволяет обнаруживать реальные ошибки с большей вероятностью, чем тщательно закодированные сценарии.

Исходящий интерфейс (outgoing interface)

Интерфейс для доступа к компонентам (например, классам или наборам классов), которые часто полагаются на другие компоненты в реализации собственного поведения. Входящие и исходящие данные, передающиеся через такие интерфейсы, называются **опосредованным вводом** (indirect input) и **опосредованным выводом** (indirect output). Исходящий интерфейс может состоять из вызовов методов или функций других компонентов, сообщений в очереди или записей, вставленных в базу данных или в файл. Тестирование с использованием исходящих интерфейсов требует специальных способов, например применения *подставных объектов* (Mock Object), для перехвата и проверки использования исходящих интерфейсов.

Итеративная разработка (iterative development)

Метод создания программного продукта ограниченными временем “итерациями”. Каждая итерация сначала планируется, а затем выполняется. В конце запланированного периода проделанная работа оценивается и планируется следующая итерация. Точное соблюдение временных рамок позволяет избежать разработки с “убегающим” последним сроком, при которой состояние системы никогда не оценивается, так как ничто на самом деле не завершено. В отличие от инкрементной разработки итеративная разработка не требует рабочего продукта в конце каждой итерации.

Класс эквивалентности (equivalence class)

Способ идентификации **условий тестов** (test condition), снижающий количество необходимых тестов за счет группирования входных данных, приводящих к одному и тому же результату или вызывающих одну и ту же логику тестируемой системы. Такая организация позволяет сконцентрировать тесты на **границых значениях** (boundary value), приводящих к изменению ожидаемого вывода.

Код продукта (production code)

В индустрии ИТ среда, в которой будет работать приложение, часто называется **промышленным использованием** (production). Под кодом продукта подразумевается код, который будет работать в этой среде. Противоположное понятие — **тестовый код** (test code).

Код теста (test code)

Код, специально написанный для тестирования другого кода (кода продукта или кода других тестов).

Компонент (component)

Достаточно большая часть системы, которая может распространяться самостоятельно. Разработка на основе компонентов предусматривает разбиение функциональности на

набор отдельных компонентов. Компоненты разрабатываются и распространяются по-отдельности. В результате приложения с одинаковой функциональностью могут совместно использовать компоненты. Каждый компонент является результатом одного или нескольких решений на этапе проектирования. Кроме того, его поведение можно проследить до одного из элементов исходных требований.

Компоненты могут иметь различные форматы в зависимости от используемых технологий. В операционной системе Windows в качестве компонентов используются динамически подключаемые библиотеки DLL или сборки. На платформе Java для этого применяются архивы Java (JAR). Архитектура на основе служб в качестве компонентов выделяет Web-службы. Компонент может содержать реализацию взаимодействия с пользователем (диалоговое окно открытия файла) или с базой данных (подготовка данных клиента к длительному хранению). Компонент должен проверяться с помощью **тестов компонентов** (Component Test), прежде чем приложение будет проверяться с помощью приемочных тестов.

Конструктор (constructor)

Специальный метод создания нового объекта в некоторых объектно-ориентированных языках программирования. Часто он имеет то же имя, что и класс, и вызывается автоматически системой выполнения при каждом использовании специального оператора new. *Полный метод конструктора* (Complete Constructor Method) [SBPP] возвращает готовый к использованию объект, не требующий дополнительной настройки. Часто это означает, что конструктору должны передаваться необходимые аргументы.

Контекст теста (test context)

Все необходимые элементы тестируемой системы, которые должны быть подготовлены перед вызовом тестируемой системы для проверки ее поведения. По этой причине в пакете RSpec термин “контекст” используется для обозначения *тестовой конфигурации* (Test Fixture) (этот термин применяется в инфраструктуре xUnit).

```
Context: a set fruits with
         contents = {apple, orange, pear}
Exercise: remove orange from the fruits set
Verify: fruits set contents = {apple, pear}
```

В этом примере тестовая конфигурация состоит из единственного набора и создается в процессе работы теста. Способ создания тестовой конфигурации влияет практически на все аспекты написания и обслуживания тестов.

Красная полоса (red bar)

Многие программы запуска тестов с графическим интерфейсом (Graphical Test Runner) описывают текущее состояние набора тестов с помощью полосы состояния. Если все тесты завершились удачно, полоса остается зеленой. Если хотя бы один тест завершился неудачно, полоса становится красной.

Ложное несрабатывание (false negative)

Ситуация, в которой тест завершается успешно, но тестируемая система работает некорректно. Такой тест приводит к ложному успешному завершению.

См. также *ложное срабатывание* (false positive).

Ложное срабатывание (false positive)

Ситуация, в которой тест завершается неудачно, но тестируемая система работает корректно. Такой тест приводит к ложному неудачному завершению. Данная терминология взята из статистики и связана с попытками подсчитать вероятность появления некоторой ошибки наблюдения. Например, в медицине тесты используются для выявления медицинских состояний. Если состояние присутствует, тест считается “положительным”. Важно знать вероятность выявления определенного состояния (например, диабета), когда на самом деле такого состояния нет, т.е. когда происходит ложное срабатывание. Если рассматривать тесты программного обеспечения как способ выявления состояния (определенного дефекта или ошибки), сообщающий о несуществующем дефекте тест приведет к появлению ложных срабатываний.

См. также *ложное несрабатывание* (false negative). Более подробная информация доступна на сайте Wikipedia в статье “Type I and type II errors”.

Локальная переменная (local variable)

Переменная, связанная с блоком кода, а не с объектом или классом. Локальная переменная доступна только из данного блока кода. Переменная исчезает из области видимости, когда блок кода завершает работу.

Локальная переменная с тестовой конфигурацией (fixture holding local variable)

Локальная переменная *тестового метода* (Test Method), используемая для хранения ссылки на *тестовую конфигурацию* (Test Fixture). Обычно она содержит ссылку на *новую тестовую конфигурацию* (Fresh Fixture), которая была создана с использованием *встроенной настройки* (In-line Setup) или получена в результате *делегированной настройки* (Delegated Setup).

Метаобъект (meta object)

Объект, который содержит данные, контролирующие поведение другого объекта. Протокол метаобъекта является интерфейсом, через который создается или настраивается метаобъект.

Метатест (meta test)

Тест, который проверяет поведение одного или нескольких тестов. Обычно используется при разработке на основе тестов. Если тесты используются в качестве примеров или

обучающего материала, возможно, придется проверить неудачное завершение тестов на проблеме определенного типа.

Метод доступа (accessor)

Метод, который обеспечивает доступ к переменной экземпляра класса, возвращая текущее значение переменной или позволяя установить новое значение.

Метод класса (class method)

Метод, связанный с классом, а не с объектом. Он может вызываться в формате `имя_класса.имя_метода` (например, `Assert.assertEquals(message, expected, actual)`) и не требует создания экземпляра класса перед вызовом. Методы класса не могут обращаться к **методам экземпляров** (`instance method`) или **переменным экземпляров** (`instance variables`), т.е. они не имеют доступа к указателям `self` и `this`. В языке Java метод класса называется **статическим методом** (`static method`). В других языках для этого могут использоваться другие названия и ключевые слова.

Метод установки (setter)

Метод объекта, предназначенный для установки значений одного из атрибутов. Обычно в соглашении об именовании предполагается использование одноименных с атрибутом методов установки или добавление префикса `set` (например, `setName`).

Метод экземпляра (instance method)

Метод, который связан с объектами, а не с классами объектов. Метод экземпляра доступен только из экземпляра класса или через него. Обычно он используется для получения информации, которая отличается для каждого экземпляра класса.

Конкретный синтаксис вызова метода экземпляра зависит от языка программирования. Чаще всего это `ссылкаНаОбъект.имяМетода()`. Если вызов происходит из другого метода объекта, в некоторых языках требуется явно указать ссылки на объект (например, `this.имяМетода()` или `self.имяМетода`). В некоторых языках предполагается, что все неквалифицированные ссылки на методы касаются методов экземпляра.

Миксин (mixin)

Функциональность, предназначенная для наследования другим классом в виде фрагмента реализации, но без специализации (отношения типа) исходного класса.

“Термин **миксин** (mixin) возник в магазине мороженого в Сомервиле (штат Массачусетс), в котором конфеты и пирожные подмешивались в стандартные типы мороженого. Эта метафора понравилась некоторым разработчикам с опытом объектно-ориентированного программирования, которые захаживали в магазин. Особенно метафора показалась подходящей для объектно-ориентированного языка SCOOPS”. (Из книги *SAMS Teach Yourself C++ in 21 Days, 4th edition.*)

Модель предметной области (domain model)

Модель, которая может лежать в основе объектной модели уровня бизнес-логики программного продукта. Дополнительная информация приводится в книге Эрика Эванса “Domain-Driven Design” [DDD].

Модуль (module)

В устаревших средах разработки (и, возможно, в некоторых современных) — независимо компилируемый фрагмент исходного кода (например, “модуль файлового ввода-вывода”), который позднее связывается с финальным выполняемым файлом. В отличие от компонента, модуль такого типа не может распространяться отдельно. Модуль может иметь или не иметь соответствующий набор модульных и компонентных тестов.

При описании функциональности программной системы или приложения модуль представляет собой полный вертикальный фрагмент приложения, обеспечивающий определенную функциональность (например, “модуль управления клиентами”). Такой модуль может использоваться независимо от других модулей. Он может иметь соответствующий набор приемочных тестов и стать объектом инкрементной доставки.

Модульный тест (unit test)

Тест, который проверяет поведение небольшого фрагмента большой системы. Тест считается модульным, если тестируемая система представляет собой очень небольшой фрагмент целой системы (может не рассматриваться как отдельная часть неспециалистами в разработке программного обеспечения). В качестве тестируемой системы может восприниматься один объект или метод. Выбор тестируемой системы является следствием одного или нескольких решений на этапе проектирования, хотя поведение тестируемой системы может прослеживаться до одного из аспектов функциональных требований. От модульных тестов не требуется простота чтения и проверки со стороны заказчика или эксперта в предметной области. Это отличает их от **приемочных тестов** (customer test), которые практически полностью основаны на требованиях и должны поддаваться проверке со стороны заказчика.

Набор тестов (test suite)

Способ именования коллекции тестов, которые должны запускаться вместе.

Настройка тестовой конфигурации (fixture setup)

Перед вызовом проверяемой логики тестируемой системы необходимо создать предварительные условия работы теста. В общем, все объекты (и их состояния) называются **тестовой конфигурацией** (Test Fixture) (или **контекстом теста**, test context), а фаза теста, создающая **тестовую конфигурацию** (Test Fixture), называется **настройкой тестовой конфигурации** (fixture setup).

Непосредственный ввод (direct input)

Тест может взаимодействовать с тестируемой системой через “главный вход”, или открытый программный интерфейс, либо через “черный ход”. Стимулы, вводимые в тестируемую систему через “главный вход”, являются непосредственным вводом. Он может состоять из вызовов методов или функций других компонентов либо сообщений, помещенных в очередь.

Непосредственный вывод (direct output)

Тест может взаимодействовать с тестируемой системой через “главный вход”, или открытый программный интерфейс, либо через “черный ход”. Ответы, полученные через “главный вход”, могут состоять из возвращаемых значений метода или функции, обновленных аргументов, переданных по ссылке, исключений в тестируемой системе или сообщений, полученных через общую очередь сообщений.

Непрерывная интеграция (continuous integration)

Практика непрерывной интеграции изменений в программный продукт характерна для гибких процессов разработки. Обычно разработчики выполняют интеграцию каждые несколько часов или дней. Непрерывная интеграция часто включает в себя механизм автоматической компиляции после каждого включения изменений в общее хранилище кода. Обычно в процессе компиляции запускаются и все автоматизированные тесты. Кроме того, могут запускаться тесты, которые не запускаются перед включением кода в хранилище, так как, с точки зрения разработчика, они работают слишком долго. Компиляция считается неудачной, если хотя бы один тест завершился неудачно. В случае неудачной компиляции исправление выявленных ошибок имеет наибольший приоритет. На этом этапе в хранилище принимается только код для исправления обнаруженных ошибок.

Неудачное завершение теста (test failure)

Завершение теста, которое происходит, если при запуске теста полученный результат не совпадает с ожидаемым.

Обслуживающий объект (service object)

Объект, который предоставляет службы другим объектам. Обычно обслуживающий объект не имеет собственного жизненного цикла. Содержащееся в нем состояние обычно является агрегированным состоянием обслуживаемых объектов сущностей. Интерфейс такого объекта определяется с помощью класса *фасада служб* (Service Facade) [CJ2EEP]. Примером обслуживающего объекта может служить EJB Session Bean.

Объект, не имеющий состояния (stateless)

Объект, не сохраняющий состояние между вызовами: каждый запрос является независимым и не требует использования одного и того же серверного объекта для обработки последовательности запросов.

Объект сущности (entity object)

Объект, представляющий концепцию сущности из предметной области. Объекты сущностей обычно имеют жизненный цикл, представленный в виде их состояния. Противоположным понятием является **служебный объект** (service object), не имеющий состояния. Примером объекта сущности является EJB Entity Bean.

Объектно-реляционное отображение (object-relational mapping — ORM)

Компонент среднего уровня, который выполняет трансляцию объектно-ориентированной модели предметной области в таблично-ориентированное представление, используемое системами управления базами данных.

Ожидаемый результат (expected outcome)

Проверяемый результат работы тестируемой системы. *Самопроверяющийся тест* (Self-Checking Test) проверяет ожидаемый результат с помощью вызовов *методов с утверждением* (Assertion Method).

Ожидание (expectation)

Тест ожидает от тестируемой системы некоторых действий. При использовании *подставного объекта* (Mock Object) для проверки **опосредованного вывода** (indirect output) каждый *подставной объект* (Mock Object) загружается со списком ожидаемых вызовов методов (включая ожидаемые аргументы). Эта информация и является ожиданием.

Опосредованный ввод (indirect input)

Значения, возвращаемые другим компонентом, предоставляющим услуги системе, если от них (значений) зависит поведение тестируемой системы. Такой ввод может состоять из возвращаемых значений функций, обновленных параметров процедур или подпрограмм, а также любых ошибок или исключений, возникших в вызываемом компоненте. Проверка поведения тестируемой системы с помощью опосредованного ввода требует соответствующей контрольной точки на “обратной стороне” тестируемой системы. Часто для вставки опосредованного ввода используется *тестовая заглушка* (Test Stub).

Опосредованный вывод (indirect output)

Действия, за которыми нельзя наблюдать через открытый программный интерфейс, но которые можно проконтролировать со стороны других подсистем или компонентов, если они (действия) входят в поведение тестируемой системы. Такой вывод может состоять из вызовов методов и функций других компонентов, сообщений в очереди или записей в базе данных или в файле. Проверка поведения на основе опосредованного вывода требует соответствующей **точки наблюдения** (observation point) на “обратной стороне” тестируемой системы. Часто для реализации точки наблюдения используется *подставной*

объект (Mock Object), который перехватывает опосредованный вывод и сравнивает его с ожидаемыми значениями.

См. также *исходящий интерфейс* (outgoing interface)

Ответственный за тесты (test maintainer)

Лицо (или роль в проекте), ответственное за обслуживание тестов с развитием системы или приложения. Чаще всего этот человек добавляет в систему новую функциональность или исправляет ошибки. Ответственным за тесты может стать любой разработчик, который получил уведомление о неудачном завершении автоматизированного теста.

Очистка тестовой конфигурации (fixture teardown)

Этап тестирования, на котором созданная тестовая конфигурация должна быть уничтожена после завершения работы теста.

Ошибка теста (test error)

Ошибка, которая не позволяет тесту полностью завершить работу. Она может создаваться явно или генерироваться тестируемой системой либо самим тестом. Кроме того, источником ошибки может служить система выполнения (например, операционная система или виртуальная машина). Обычно ошибку теста намного проще диагностировать, чем **неудачное завершение теста** (test failure), так как причина проблемы обычно находится намного ближе к точке возникновения ошибки.

Пакет теста (test package)

В языках с поддержкой пакетов или пространств имен — пакет или пространство имен, которое может использоваться для хранения *классов теста* (Testcase Class).

Переменная класса (class variable)

Связанная с классом (а не с объектом) переменная. Обычно такая переменная совместно используется всеми экземплярами класса. В некоторых языках программирования доступ к переменным класса можно получить с помощью синтаксиса вида `имя_класса.имя_переменной` (например, `TestHelper.lineFeedCharacter;`), т.е. для доступа к этим переменным указатели `self` и `this` не нужны. В языке Java переменная класса называется **статической переменной** (static variable). В других языках для этого могут использоваться другие названия и ключевые слова.

Переменная класса с тестовой конфигурацией (fixture holding class variable)

Переменная класса теста (Testcase Class), используемая для хранения ссылки на *тестовую конфигурацию* (Test Fixture). Обычно в такой переменной хранится ссылка на *общую тестовую конфигурацию* (Shared Fixture).

Переменная экземпляра (instance variable)

Переменная, которая связана с объектом, а не с классом объектов. Переменная экземпляра доступна только из экземпляра класса или через него. Обычно переменная используется для хранения информации, которая отличается для каждого экземпляра.

Переменная экземпляра с тестовой конфигурацией (fixture holding instance variable)

Переменная объекта теста (Testcase Object), в которой хранится ссылка на *тестовую конфигурацию* (Test Fixture). Обычно она используется для хранения ссылки на *новую тестовую конфигурацию* (Fresh Fixture), созданную в процессе *неявной настройки* (Implicit Setup).

Плавный интерфейс (fluent interface)

Стиль интерфейса объектных конструкторов, позволяющий создать простые для понимания операторы. *Конфигурационный интерфейс* (Configuration Interface), предоставленный средствами разработки JMock для работы с *подставными объектами* (Mock Object), является примером плавного интерфейса.

Полиморфизм (polymorphism)

Динамическое связывание (dynamic binding). В переводе с латыни означает “иметь много форм”.

Пользовательские истории (user story)

Предмет *инкрементной разработки* (incremental development) в экстремальном программировании. Каждая пользовательская история должна иметь следующие свойства: независимость, возможность обсуждения, ценность, прогнозируемость, малый размер и возможность тестирования [XP123]. Пользовательская история примерно соответствует **функции** (feature) в терминологии за пределами методологии экстремального программирования. Обычно пользовательская история делится на **задачи** (task), которые должны быть выполнены участниками проекта.

Презентационная логика (presentation logic)

Логика на презентационном уровне бизнес-системы. Данная логика определяет отображаемые на экране данные, список элементов меню, список доступных элементов управления и т.д.

Презентационный уровень (presentation layer)

Часть многоуровневой архитектуры [DDD, PEAA, WWW], содержащая презентационную логику.

Приемочный тест (customer test)

Тест, проверяющий видимую функциональность системы. Тестируемая система может включать в себя как весь продукт, так и отдельный модуль. Приемочные тесты не должны зависеть от решений, принятых при проектировании и создании тестируемой системы. Другими словами, один и тот же набор приемочных тестов должен работать с любым вариантом тестируемой системы. (Конечно, способ взаимодействия тестов с тестируемой системой может зависеть от высокочувствительных характеристик архитектуры продукта.)

Пример использования (use case)

Способ описания функциональности системы в терминах целей пользователей и действий системы для достижения этих целей. В отличие от пользовательских историй (user story) пример использования может описывать различные сценарии, но не тестируется отдельно.

Принцип Голливуда (Hollywood principle)

Режиссеры из Голливуда при массовых кастингах сообщают актерам: “Не звоните нам, мы сами вам позвоним (если вы нас заинтересуете)”. В программном обеспечении такая концепция называется **инвертированием управления** (inversion of control — IOC).

Проверка результата (result verification)

В *четырехфазном тесте* (Four-Phase Test) после фазы вызова тестируемой системы проверяется получение ожидаемого (правильного) результата. Эта фаза тестирования называется **проверкой результата** (result verification).

Проверка результата (verify outcome)

После фазы вызова тестируемой системы тест сравнивает фактический результат (включая возвращаемые значения, опосредованный вывод и состояние системы после теста) с ожидаемым результатом. Это фаза проверки результата *четырехфазного теста* (Four-Phase Test).

Прогонный тест (acceptance test)

Тест, который потребитель программного продукта планирует использовать, прежде чем принять проделанную над продуктом работу. Обычно прогонные тесты выполняются вручную после всех автоматизированных приемочных тестов. Они проверяют все уровни системы начиная от пользовательского интерфейса и заканчивая базой данных. Кроме того, тесты проверяют интеграцию с другими системами, от которых зависит приложение.

Программный интерфейс (application programming interface)

Способ взаимодействия программного обеспечения с фрагментом функциональности. В объектно-ориентированном программном обеспечении программный интерфейс состоит из классов и их публичных методов. В процедурном программном обеспечении интерфейс состоит из имени модуля или пакета, а также публично доступных процедур.

Проектирование с учетом тестирования (design for testability)

Способ обеспечения простоты тестирования, основанный на учете требований тестов на этапе проектирования кода. При разработке на основе тестов такое проектирование становится положительным побочным эффектом процесса разработки.

Промышленное использование (production)

В индустрии ИТ — среда, в которой приложение будет применяться конечными пользователями. Эта среда отличается от различных тестовых сред, таких как “приемка”, “интеграция”, “разработка” и “контроль качества”.

Простейшее решение (The simplest thing that could possibly work — STTCPW)

“Простейшее рабочее решение”. Подход, который обычно используется в проектах на основе экстремального программирования в противоположность избыточному проектированию продукта с учетом возможных требований в будущем.

Процедурная переменная (procedure variable)

Переменная, которая ссылается на процедуру или функцию, а не просто на фрагмент данных. Она позволяет отложить выбор вызываемого кода до этапа выполнения (динамическая привязка), избавляя от необходимости принять это решение на этапе компиляции. Фактически вызываемая процедура присваивается переменной на этапе инициализации приложения или в процессе выполнения. Процедурные переменные предшествовали появлению истинных объектно-ориентированных языков программирования. Ранние языки из этой группы (например, C++) применяли таблицы (массивы) структур данных, содержащих процедурные переменные, для реализации таблиц диспетчеризации методов классов.

Разработка на основе документов (document-driven development)

Процесс разработки, основное внимание в ходе которого уделяется созданию документов с описанием структуры кода, на основе которых фактически создается код. Разработка на основе документов обычно ассоциируется с “каскадным” процессом. Противоположностью является разработка на основе тестов, когда основное внимание уделяется написанию кода для обеспечения последовательного успешного завершения тестов.

Разработка на основе поведения (behavior-driven development)

Вариант разработки на основе тестов (test-driven development), при котором основная задача тестов заключается в ясном описании ожидаемого поведения тестируемой системы.

При разработке на основе поведения могут применяться традиционные пакеты семейства xUnit. Существуют и новые пакеты семейства, специально рассчитанные на смещение внимания. В них используется соответствующим образом модифицированная терминология (например, “тест” становится “спецификацией”, а “тестовая конфигурация” — “контекстом”) и инфраструктурой обеспечивается более явная поддержка ясности спецификации.

Разработка на основе потребностей (need-driven development)

Вариант процесса разработки на основе тестов, когда код пишется в направлении “извне вовнутрь”, а все вызываемые компоненты заменяются *подставными объектами* (Mock Object), которые проверяют ожидаемый опосредованный вывод написанного кода. Такой подход обеспечивает полное понимание области ответственности каждого программного компонента еще перед его реализацией. Причиной этому являются модульные тесты, основанные на примерах реального использования. Внешний уровень программного продукта создается в ходе разработки на основе тестов историй (storytest-driven development). При этом, кроме приемочных тестов, должны присутствовать примеры использования реальными клиентами (например, пользовательский интерфейс, управляющий *фасадом служб* (Service Facade) [CJ2EEP]).

Разработка на основе примеров (example-driven development — EDD)

Вариант разработки на основе тестов, при котором основное внимание уделяется аспекту тестов “выполняемая спецификация”. Работа с примерами является интуитивно понятной большинству. При этом не приходится “тестировать” еще ненаписанное программное обеспечение.

Разработка на основе тестов (test-driven development)

Процесс разработки, подразумевающий написание и автоматизацию модульных тестов еще до разработки соответствующих модулей. Данный процесс обеспечивает понимание обязанностей каждого модуля программного продукта еще до написания кода. В отличие от разработки с первоочередным написанием тестов (test-first development) разработка на основе тестов обычно предполагает написание кода продукта для последова-

тельного удовлетворения требований тестов (эта характеристика называется **возникающим дизайном**, emergent design).

См. также *разработка на основе тестов историй* (storytest-driven development).

Разработка на основе тестов историй (storytest-driven development — STDD)

Вариант процесса разработки на основе тестов, подразумевающий написание автоматизированных приемочных тестов до разработки соответствующей функциональности. При этом предполагается, что интеграция проверенных с помощью модульных тестов фрагментов продукта позволит получить стабильное целое решение. Термин “разработка на основе тестов историй” был впервые предложен Джошуа Кериевски в описании методологии “Industrial XP” [IXP].

Разработка с написанием тестов после кода (test-last development)

Процесс разработки, подразумевающий запуск **модульных тестов** (unit test) после завершения работы над соответствующим модулем. В отличие от разработки с **первоочередным написанием тестов** (test-first development) такой процесс просто требует запуска тестов перед началом промышленного использования написанного кода. Требования к автоматизации тестов не предъявляются. Традиционное тестирование для контроля качества является характерным примером такого подхода к разработке (кроме случаев, когда тесты создаются на этапе формализации требований и используются командой разработчиков).

Разработка с первоочередным написанием тестов (test-first development)

Процесс разработки, предполагающий написание и автоматизацию модульных тестов (unit test) перед написанием соответствующих модулей. В ходе данного процесса обязанности каждого модуля понятны еще до написания кода. В отличие от **разработки на основе тестов** (test-driven development) данный процесс просто обеспечивает написание тестов перед кодом продукта. Последовательное удовлетворение условий тестов не требуется. Такой процесс может применяться как для **модульных тестов** (unit test), так и для **приемочных тестов** (customer test) в зависимости от цели автоматизации.

Регрессионный тест (regression test)

Тест, проверяющий неизменность поведения тестируемой системы. Большинство регрессионных тестов изначально создавались как модульные или приемочные, но впоследствии были включены в набор регрессионных тестов для защиты функциональности от неожиданной модификации.

Результат выполнения теста (test result)

Тест (test) или **набор тестов** (test suite) может запускаться несколько раз, каждый раз возвращая новый результат.

Ретроспективная оценка (retrospective)

Процедура оценки процессов и производительности группы разработчиков с целью поиска более эффективных методов работы. Ретроспективная оценка обычно выполняется в конце проекта для сбора данных и формирования рекомендаций для будущих проектов. Такая оценка имеет больший эффект, если выполняется регулярно на протяжении всего проекта. В проектах с гибкой методикой разработки ретроспективная оценка формируется как минимум после каждой версии и часто — после каждой итерации.

Рефакторинг (refactoring)

Модификация структуры существующего кода без изменения его поведения. Рефакторинг используется для улучшения дизайна существующего кода. Часто выполняется перед добавлением новой функциональности. Авторитетным источником информации о рефакторинге является книга Мартина Фаулера [Ref].

Сборка мусора (garbage collection)

Механизм автоматического освобождения памяти с удалением недоступных объектов. Многие современные объектно-ориентированные среды разработки обеспечивают автоматическую сборку мусора.

Синхронный тест (synchronous test)

Тест, взаимодействующий с тестируемой системой с помощью нормальных (синхронных) вызовов методов. Методы возвращают результат, который проверяется тестом. Синхронный тест не требует координации действий с тестируемой системой. Такая координация осуществляется автоматически системой выполнения. Противоположное понятие — **асинхронный тест** (asynchronous test), который работает в отдельном потоке выполнения.

Специфическое для теста равенство (test-specific equality)

Тесты и тестируемая система могут иметь разные представления о равенстве двух объектов. На самом деле такое представление может отличаться даже для двух разных тестов. Нежелательно модифицировать определение равенства в тестируемой системе в соответствии с ожиданиями теста, так как подобная практика приводит к *засорению равенства* (Equality Pollution). Также не является решением использование *утверждения равенства* (Equality Assertion) для каждого атрибута объекта, так как это может привести к появлению *непонятных тестов* (Obscure Test) и *дублированию тестового кода* (Test Code Duplication). Вместо этого желательно создать одно или несколько *специальных утверждений* (Custom Assertion), соответствующих потребностям теста.

Статическая переменная (static variable)

В языке программирования Java — переменная, которая фиксируется на этапе компиляции, а не на этапе выполнения. Статическая переменная также является **переменной**

класса (class variable), поскольку только переменные класса могут фиксироваться на этапе компиляции в языке Java. Не во всех языках статические члены класса связываются с классом (в противоположность связыванию с объектами).

Статический метод (static method)

В языке Java — метод, который выбирается на этапе компиляции (а не на этапе выполнения с использованием динамического связывания). Такое поведение противоположно **динамическим** (dynamic) (или **виртуальным** (virtual) в C++) методам. Кроме того, статический метод является **методом класса** (class method), так как только методы классов могут связываться на этапе компиляции языка Java. Статический метод не во всех языках является методом класса. Ниже представлен пример статического метода.

```
Assert.assertEquals(message, expected, actual);
```

Статическое связывание (static binding)

Определение фрагмента программного обеспечения, которому будет передаваться управление, на этапе компиляции. Статическое связывание является противоположностью **динамического связывания** (dynamic binding).

Счастливый маршрут (happy path)

“Нормальный” путь выполнения пользовательской истории или реализующего ее программного обеспечения. Также известен как сценарий “солнечного дня”. Ничего плохого или необычного не происходит, поэтому сразу достигается необходимая цель.

Тест (test case)

В инфраструктуре xUnit под этим названием может рассматриваться *класс теста* (Testcase Class), представляющий собой *фабрику наборов тестов* (Test Suite Factory) и место для хранения связанных *тестовых методов* (Test Method).

Тест (test)

Автоматизированная или запускаемая вручную процедура, сравнивающая существующее поведение тестируемой системы с ожидаемым.

Тест Fit

Тест на основе инфраструктуры Fit. Чаще всего это *приемочный тест* (customer test).

Тест истории (storytest)

Приемочный тест, являющийся “подтверждением” в “трилогии” **пользовательской истории** (user story): карточка, общение, подтверждение [ХРС]. Если тесты историй раз-

рабатываются до программного обеспечения, процесс можно назвать основанным на тестах историй.

Тест компонента (component test)

Тест, проверяющий поведение одного из компонентов системы. Каждый компонент является результатом одного или нескольких решений на этапе проектирования. Кроме того, его поведение можно проследить до одного из элементов исходных требований. Тесты компонентов не обязательно должны быть простыми для чтения или проверки экспертом в проблемной области или представителем клиента. Эта особенность отличает тест компонента от приемочных тестов, которые создаются исключительно на основе требований и должны проверяться заказчиком, а также от **модульных тестов** (unit test), которые проверяют компоненты значительно меньшего масштаба. Тесты компонентов являются чем-то средним между тестами этих типов.

Тесты компонентов в процессе разработки на основе тестов создаются после написания приемочных тестов и фиксации общего дизайна продукта, сразу после принятия архитектурных решений, но перед проектированием и программированием отдельных модулей. Обычно тесты компонентов автоматизируются средствами одного из пакетов семейства xUnit.

Тест разработчика (developer test)

Еще одно название автоматизированных **модульных тестов** (unit test), подготовленных человеком, играющим роль разработчика в проекте с использованием методологии экстремального программирования (eXtreme Programming).

Тест с внедрением ошибки (fault insertion test)

Специальный тест, намеренно внедряющий ошибку в один из компонентов системы, чтобы проверить реакцию другого компонента на ошибку. Изначально ошибки могли быть связаны с аппаратными сбоями, но теперь такой подход используется и в случае ошибок в программном обеспечении. Замена вызываемого компонента генерирующими исключение диверсантом (Saboteur) является примером теста с внедрением ошибки.

Тест с пересечением уровней (layer-crossing test)

Тест, который создает тестовую конфигурацию или проверяет результат с помощью **манипуляции через “черный ход”** (Back Door Manipulation). При этом в качестве “черного хода” может использоваться база данных или другой компонент. Противоположностью является использование **теста через открытый интерфейс** (round-trip test).

Тест через открытый интерфейс (round-trip test)

Тест, который взаимодействует с тестируемой системой только через открытый интерфейс. Противоположное понятие — **тест с пересечением уровней** (layer-crossing test).

Тест, выполняемый вручную (manual test)

Тест, который выполняется человеком, взаимодействующим с тестируемой системой. Пользователь может выполнять “сценарий теста” (не путайте с *тестом на основе сценария*, Scripted Test) или произвольное либо исследовательское тестирование.

Тестируемая система (system under test — SUT)

То, что тестируется. Тестируемая система всегда определяется с точки зрения теста. При написании **модульных тестов** (unit test) в качестве тестируемой системы выступает класс (тестируемый класс), объект (тестируемый объект) или метод (тестируемый метод). При написании **приемочных тестов** (customer test) в качестве тестируемой системы выступает все приложение или его крупная подсистема. Остальные подсистемы приложения могут выступать в качестве **вызываемого компонента** (depended-on component).

Тестовая база данных (test database)

Экземпляр базы данных, который используется исключительно для запуска тестов. Он не должен использоваться готовым продуктом.

Тестовая конфигурация (test fixture)

С точки зрения семейства xUnit — все элементы, которые необходимы для запуска теста и получения определенного результата. Тестовая конфигурация содержит все предварительные условия для выполнения теста, т.е. состояние тестируемой системы и ее контекста “до запуска”.

С точки зрения пакетов NUnit и VbUnit — *класс теста* (Testcase Class).

С точки зрения пакета Fit — адаптер, интерпретирующий таблицу Fit и вызывающий методы тестируемой системы, реализуя управляемый данными тест (Data-Driven Test).

Тестовая конфигурация (test fixture) (в пакете NUnit)

В пакете NUnit (а также в VbUnit и большинстве реализаций xUnit для платформы .NET) — *класс теста* (Testcase Class), для которого реализуются *тестовые методы* (Test Method). Классу, хранящему тестовые методы, присваивается атрибут [TestFixture].

Некоторые члены семейства xUnit предполагают, что *класс теста* (Testcase Class) “является” *контекстом теста* (test context). Характерный пример — пакет NUnit. Такая интерпретация предполагает, что при организации тестов используется подход *класс теста для каждой тестовой конфигурации* (Testcase Class per Fixture). Если требуется другая организация тестов, например *класс теста для каждого класса* (Testcase Class per Class) или *класс теста для каждой функции* (Testcase Class per Feature), объединение концепций контекста теста и класса теста может оказаться сложным в понимании. В данной книге термин **тестовая конфигурация** (test fixture) используется в смысле “предварительные условия теста” (или **контекст теста**, test context), а *класс теста* (Testcase Class) — “класс, содержащий *тестовые методы* (Test Method) и код, который необходим для создания контекста теста”.

Тестовая конфигурация (test fixture) (в семействе xUnit)

В семействе xUnit — все элементы, которые необходимы для запуска теста и получения определенного результата, т.е. **контекст теста** (test context). В некоторых вариантах xUnit концепция контекста теста отделена от *класса теста* (Testcase Class), который создает контекст. В JUnit и созданных на его основе пакетах используется именно такое соглашение. Создание тестовой конфигурации является первой фазой *четырехфазного теста* (Four-Phase Test). Значение термина **тестовая конфигурация** (test fixture) в других контекстах приводилось выше.

Тестовая конфигурация Fit

В пакете Fit шаблон Adapter [GOF] интерпретирует таблицу Fit и вызывает методы тестируемой системы, реализуя **управляемый данными тест** (Data-Driven Test).

Точка взаимодействия (interaction point)

Точка, в которой тест взаимодействует с тестируемой системой. В качестве точки взаимодействия может использоваться **точка управления** (control point) или **точка наблюдения** (observation point).

Точка наблюдения (observation point)

Используемая тестом точка для контроля поведения тестируемой системы. Этот тип **точки взаимодействия** (interaction point) может использоваться для получения состояния системы после обращения со стороны теста или для отслеживания взаимодействий тестируемой системы и вызываемых компонентов. Некоторые точки наблюдения созданы специально для тестов. Они не должны использоваться кодом продукта, так как могут открыть доступ к закрытым подробностям реализации. Опасность таких подробностей заключается в том, что нельзя расчитывать на их неизменность в будущем.

Точка управления (control point)

Определяет, как тест запрашивает у тестируемой системы выполнение определенной операции. Точка управления может использоваться для создания или очистки тестовой конфигурации, а также в процессе вызова тестируемой системы. Точка управления является одним из видов **точки взаимодействия** (interaction point). Некоторые контрольные точки создаются исключительно для тестирования. Они не должны использоваться кодом продукта, так как они не проверяют входные данные или нарушают нормальный жизненный цикл тестируемой системы либо другого объекта, от которого она зависит.

Указатель функции (function pointer)

“Один из типов указателей в языках C, C++, D и других языках с аналогичным синтаксисом. При разыменовании указателя вызывается функция без параметров или с несколькими параметрами, как и обычная функция” (из статьи на сайте Wikipedia [Wp]).

Унаследованное программное обеспечение (legacy software)

В программисты, практикующие разработку на основе тестов, называют так любой программный продукт, не имеющий *страховочной сети* (Safety Net) из *полностью автоматизированных тестов* (Fully Automated Tests).

Унифицированный язык моделирования (UML)

“Открытый язык описания спецификаций для объектного моделирования. UML является языком моделирования общего назначения и включает стандартные графические обозначения для создания абстрактной модели системы, называемой моделью UML” (из статьи на сайте Wikipedia [Wp]).

Уровень доступа к данным (data access layer)

Способ отделения логики доступа к данным от кода приложения. Обычно такой код помещается в отдельный компонент, скрывающий функции работы с базой данных.

Уровень предметной области (domain layer)

Один из уровней многоуровневой архитектуры [DDD, PEAA, WWW], соответствующий модели предметной области. Дополнительная информация приводится в книге Эрика Эванса “Domain-Driven Design” [DDD].

Условие теста (test condition)

Определенное поведение тестируемой системы, которое необходимо проверить. Его можно описать как такую последовательность промежуточных состояний:

- если тестируемая система находится в состоянии S1 и
- тестируемая система вызывается способом X, то
- тестируемая система должна выдать ответ R и
- тестируемая система должна находиться в состоянии S2.

Успешное завершение теста (test success)

Ситуация, в которой после выполнения теста все полученные результаты совпали с ожидаемыми.

Утверждение (assertion)

Утверждение об истинности чего-либо. В *инфраструктуре автоматизации тестов* (Test Automation Framework) в стиле xUnit утверждение принимает форму *метода с утверждением* (Assertion Method), который завершается неудачно, если передаваемый ему фактический результат не совпадает с ожидаемым.

Фабрика (factory)

Метод, объект или класс, предназначенный для создания других объектов.

Фильтр для тестов (test stripper)

Этап или программа компиляции, удаляющая весь код тестов из откомпилированного и связанного выполняемого файла.

Функциональный тест (functional test)

Тест “черного ящика” для пользовательской функциональности приложения. Сообщество сторонников гибкой методологии разработки пытается избегать использования функциональных тестов подобного рода из-за потенциальной путаницы при обсуждении проверки функциональных (в отличие от нефункциональных или сверхфункциональных) свойств **модуля** (unit) или **компоненты** (component). В этой книге для описания функциональных тестов всего приложения используется термин “приемочный тест”. Для описания функциональных тестов модулей приложения здесь употребляется термин “модульный тест”.

Функция (feature)

Поддающаяся тестированию функциональность, которую можно добавить к развивающемуся программному продукту. В экстремальном программировании *пользовательские истории* (user story) примерно соответствуют функциям.

Черный ход (back door)

Альтернативный интерфейс к тестируемой системе, который может использоваться тестами для опосредованной вставки входных данных. База данных является распространенным примером “черного хода”, но в этом качестве может использоваться любой компонент, которым можно манипулировать, чтобы получить значения для теста, или который можно заменить *тестовым двойником* (Test Double). Противоположным понятием является **главный вход** (front door): **программный интерфейс** (application programming interface).

Черный ящик (black box)

Фрагмент программного продукта, рассматриваемый как непрозрачный объект с неизвестным внутренним содержимым. Тесты для “черного ящика” могут проверять только видимое внешнее поведение и не зависят от реализации тестируемой системы.

Читатель теста (test reader)

Любой разработчик, которому нужно читать код теста, включая ответственного за тесты. Одна из причин чтения тестов может быть связана с необходимостью понять, что должна делать тестируемая система (*тесты как документация*, Tests as Documentation). Кроме того, чтение кода тестов может потребоваться в процессе обслуживания тестов или разработки кода программного продукта.

Шаблон (pattern)

Решение периодически возникающей проблемы. Для каждого шаблона существует контекст, в котором он обычно применяется, и факторы, позволяющие выбрать один шаблон из нескольких в зависимости от контекста. Шаблоны проектирования (design patterns) являются одним из типов шаблонов. Организационные шаблоны в данной книге не рассматриваются.

Шаблон проектирования (design pattern)

Шаблон, который обычно используется для решения конкретной проблемы проектирования программного обеспечения. Большинство шаблонов не зависят от языка программирования. Зависящие от языка шаблоны обычно называются идиомами кодирования. Шаблоны проектирования стали популярными после выхода книги “Design Patterns” [GOF].

Экстремальное программирование (eXtreme Programming)

Методология гибкой разработки программного обеспечения, в которой активно применяются методики парного программирования, автоматизированные тесты и короткие итерации разработки.

Эндоскопическое тестирование (endoscopic testing)

Способ тестирования, впервые примененный авторами документа о *подставных объектах* (Mock Object) [ET] и предполагающий тестирование программного обеспечения изнутри.

Язык шаблонов (pattern language)

Набор шаблонов, комбинация которых позволяет показать переход от проблемы очень высокого уровня к очень подробному решению, подходящему для конкретного контекста. Если язык шаблонов достигает своей цели, он считается генерирующим. Эта характеристика отличает язык шаблонов от простого набора шаблонов. Дополнительная информация по этой теме приведена в статье [APLfPW].

Источники информации

[AP]

AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis

Издательство: John Wiley (1998)

ISBN: 0-471-19713-0

Авторы: Уильям Браун (William J. Brown) и др.

В этой книге рассматриваются распространенные проблемы, возникающие при разработке программного обеспечения, и описываются способы их решения через изменение архитектуры или организации проекта.

[APLfPW]

A Pattern Language for Pattern Writing

Опубликовано в книге *Pattern Languages of Program Design 3* [PLoPD3], с. 529–574.

Издательство: Addison-Wesley (1998)

Авторы: Джерард Месарош (Gerard Meszaros) и Джеймс Добл (James Doble)

Накопление опыта в написании и выделении шаблонов и языков шаблонов в сообществе разработчиков позволило разработать формальные способы и подходы создания шаблонов. Разработанные способы оказались особенно эффективны при решении некоторых часто встречающихся проблем. При создании данного языка шаблонов использовались некоторые из лучших способов написания шаблонов, которые были показаны как при описании самих шаблонов, так и при создании примеров на их основе. В результате этот язык шаблонов сам по себе является отличным примером процесса создания языка шаблонов.

Источники дополнительной информации

Полный текст этой статьи доступен в Интернете в формате PDF по адресу:

<http://patternwritingpatterns.gerardmeszaros.com>

и в формате HTML с гиперссылками в содержании по адресу:

<http://hillside.net/patterns/writing/patternwritingpaper.htm>

[ARTRP]

Agile Regression Testing Using Record and Playback

<http://AgileRegressionTestPaper.gerardmeszaros.com>

Авторы: Джерард Месарош (Gerard Meszaros) и Ральф Боне (Ralph Boenert)

Эта статья была представлена на конференции XP/Agile Universe в 2003 году. В ней рассматривается создание механизма тестирования на основе “записи и воспроизведения” в приложение с критичными требованиями к безопасности. Механизм предназначался для упрощения регрессионного тестирования во время переноса приложения с операционной системы OS/2 на операционную систему Windows.

[CJ2EEP]

Core J2EE Patterns, Second Edition: Best Practices and Design Strategies

“Образцы J2EE. Лучшие решения и стратегии проектирования”

Издательство: Prentice Hall (2003), Лори (2004)

ISBN: 0-131-42246-4 (англ.), 5-85582-216-8 (рус.)

Авторы: Дипак Алур (Deepak Alur), Дэн Малкс (Dan Malks) и Джон Крупи (John Crupi)

В этой книге приводится каталог основных шаблонов использования Enterprise Java Beans (EJB), которые являются ключевым элементом Java 2 Enterprise Edition. В качестве одного из примеров рассматриваемых шаблонов можно назвать *фасад сеансов* (Session Facade) [CJ2EEP].

[DDD]

Domain-Driven Design: Tackling Complexity in the Heart of Software

Издательство: Addison-Wesley (2004)

ISBN: 0-321-12521-5

Автор: Эрик Эванс (Eric Evans)

Данная книга может служить хорошим введением в процесс использования модели предметной области в качестве основы программной системы.

Читателям предлагается изучить использование модели предметной области для ус-корения динамики сложного процесса разработки. Список рекомендованных практик и стандартных шаблонов лежит в основе общего языка для всех членов команды разработки.

[ET]

Endo-Testing

<http://www.connextra.com/aboutUs/mockobjects.pdf>

Авторы: Тим Маккиннон (Tim Mackinnon), Стив Фриман (Steve Freeman) и Филип Крейг (Philip Craig)

В этой статье (которая была представлена на конференции XP в 2000 году) рассматривается использование *подставных объектов* (Mock Object, с. 558) для тестирования поведения объектов через мониторинг поведения во время выполнения.

*Модульное тестирование лежит в основе экстремального программирования, но нетривиальный код сложно тестировать в изоляции. Тяжело избежать создания сложных, неполных и трудных в обслуживании наборов тестов. Использование подставных объектов (*Mock Object*) при модульном тестировании помогает улучшить как код продукта, так и наборы тестов. Такие объекты позволяют создавать модульные тесты для чего угодно, упрощать структуру тестов и избегать засорения кода продуктов инфраструктурой тестирования.*

[FaT] Frameworks and Testing

Опубликовано в *Proceedings of XP2002*.

<http://www.agilealliance.org/articles/roockstefanframeworks/file>

Автор: Стефан Рук (Stefan Roock)

Эта статья обязательна для прочтения перед созданием собственной инфраструктуры. В ней рассматриваются четыре типа автоматизированного тестирования, которые существуют в связи с инфраструктурой, включая возможность проверки совместимости подключаемого модуля с протоколом инфраструктуры, а также инфраструктуры тестирования, позволяющие проверять приложения на базе создаваемой инфраструктуры.

[FitB] Fit for Developing Software

Издательство: Addison-Wesley (2005)

ISBN: 0-321-26934-9

Авторы: Рик Магридж (Rick Mugridge) и Уорд Каннингем (Ward Cunningham)

Эта книга может служить отличным введением в использование управляемых данными тестов (Data-Driven Test, с. 322) в процессе подготовки приемочных тестов. Описанные подходы могут применяться как для гибкой, так и для традиционной методики разработки. Вот выдержка из моей рекомендации для этой книги.

Жалко, что этой книги не было, когда пришлось разрабатывать первый проект на основе тестов пользовательских историй. В книге рассматривается лежащая в основе инфраструктуры Fit философия, а также процесс использования этой инфраструктуры для взаимодействия с клиентами во время определения требований к проекту. Инфраструктура Fit описывается в книге настолько понятно, что первые тесты FitNesse я написал еще до того, как закончил читать книгу.

Источники дополнительной информации

Дополнительная информация об инфраструктуре Fit доступна на сайте Уорда по адресу:

<http://fit.c2.com>

[GOF]

Design Patterns: Elements of Reusable Object-Oriented Software

“Приемы объектно-ориентированного проектирования”

Издательство: Addison-Wesley (1995), Питер (2005)

ISBN: 0-201-63361-2 (англ.), 5-469-01136-4 (рус.)

Авторы: Эрих Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидес (John Vlissides)

Именно с этой книги началось движение по использованию шаблонов проектирования. В ней “банда четырех” описала 23 шаблона, применяемых при проектировании объектно-ориентированных программных систем. В качестве примеров таких шаблонов можно привести Composite, Factory Method и Facade.

[HoN]

Hierarchy of Needs

Из Wikipedia [Wp]:

Иерархия потребностей Маслоу является теорией в области психологии, предложенной Абрахамом Маслоу (Abraham Maslow) в 1943 году в статье “Теория человеческой мотивации”. Впоследствии Маслоу расширил свою теорию. В ней утверждается, что после удовлетворения “базовых потребностей” человек пытается удовлетворить “потребности более высокого уровня”, организованные в виде иерархии.

Иерархия потребностей Маслоу обычно изображается в виде пирамиды из пяти уровней: четыре нижних уровня сгруппированы в потребности недостатка, связанные с психологическими потребностями, а верхний уровень называется потребностью роста. В процессе удовлетворения потребности недостатка постоянно оказывают влияние на наше поведение. Основной принцип заключается в том, что потребностям более высокого уровня внимание уделяется только после удовлетворения всех потребностей нижних уровней пирамиды. Силы роста создают движение в направлении вершины иерархии, а регressive силы сдвигают преобладающие потребности на более низкие уровни иерархии.

[IEAT]

Improving the Effectiveness of Automated Tests

<http://FasterTestsPaper.gerardmeszaros.com>

Авторы: Джерард Месарош (Gerard Meszaros), Шон Смит (Shaun Smith) и Дженнита Андреа (Jennita Andrea)

Эта статья впервые была представлена на конференции XP2001. В ней рассмотрены проблемы снижения скорости и эффективности автоматизированных модульных тестов, а также показаны способы решения этих проблем.

[IXP] Industrial XP

<http://ixp.industriallogic.com>

Industrial XP — это “популяризованный” вариант среди экстремального программирования (eXtreme Programming), созданный Джошуа Кериевски в компании Industrial Logic. Методология включает набор практик, позволяющих масштабировать инструменты экстремального программирования до уровня крупных предприятий.

[JBrains] JetBrains

<http://www.jetbrains.com>

Компания JetBrains разрабатывает инструменты для разработки программного обеспечения, позволяющие автоматизировать рефакторинг (кроме всего прочего). На их сайте приводится список вариантов рефакторинга, поддерживаемых различными инструментами компании, включая даже те, которые не описаны в книге [Ref].

[JNI] JUnit New Instance

<http://www.martinfowler.com/bliki/JunitNewInstance.html>

В этой статье Мартина Фаулера обсуждается необходимость создания нового экземпляра класса *теста* (Testcase Class, с. 401) для каждого *тестового метода* (Test Method, с. 378) при использовании инфраструктуры JUnit и остальных адаптированных версий.

[JuPG] JUnit Pocket Guide

Издательство: O'Reilly

ISBN: 0-596-00743-4

Автор: Кент Бек (Kent Beck)

Эта 80-страничная книга небольшого формата считается отличным справочником по ключевым функциям пакета JUnit и рекомендуемым подходам при написании тестов. Учитывая, что книга легко помещается в кармане, слишком подробных объяснений в ней нет, но общее представление о возможных решениях и источниках более подробной информации можно получить.

[LSD] Lean Software Development : An Agile Toolkit

Издательство: Addison-Wesley (2003)

ISBN: 0-321-15078-3

Авторы: Мэри Попpendик (Mary Poppendieck) и Том Попpendик (Tom Poppendieck)

В этой отличной книге рассматривается 22 “мыслительных инструмента”, которые позволяют быстро и эффективно работать во множестве предметных областей. Авторы описывают применение этих инструментов к разработке программного обеспечения. Обязательно прочитайте эту книгу, чтобы понять, почему методы гибкой разработки работают!

[MAS] **Mocks Aren't Stubs**

<http://www.martinfowler.com/articles/mocksArentStubs.html>

Автор: Мартин Фаулер (Martin Fowler)

В этой статье рассматривается разница между *подставными объектами* (Mock Object, с. 559) и *тестовыми заглушками* (Test Stub, с. 544). Кроме этого, здесь показаны два фундаментально разных подхода к разработке на основе тестов: классическая разработка на основе тестов и разработка на основе тестов с подставными объектами.

[MRNO] **Mock Roles, Not Objects**

Статья опубликована на конференции OOPSLA 2004

Авторы: Стив Фриман (Steve Freeman), Тим Маккиннон (Tim Mackinnon), Нат Прайс (Nat Pryce) и Джо Уолнес (Joe Walnes)

В этой статье рассматривается использование *подставных объектов* (Mock Object, с. 558) для выявления сигнатур объектов, от которых зависит проектируемый и тестируемый класс. Это позволяет отложить проектирование классов поддержки до завершения реализации и тестирования клиентских классов. Члены ACM могут получить эту статью по адресу:

http://portal.acm.org/ft_gateway.cfm?id=1028765&type=pdf

Для остальных статья доступна по адресу:

<http://joe.truemesh.com/MockRoles.pdf>

[PEAA] **Patterns of Enterprise Application Architecture**

“Архитектура корпоративных программных приложений”

Издательство: Addison-Wesley (2003), Вильямс (2004)

ISBN: 0-321-12742-0 (англ.), 978-5-8459-0579-6 (рус.)

Автор: Мартин Фаулер (Martin Fowler)

В этой книге приводятся архитектурные шаблоны, которые подходят для любой платформы приложений уровня предприятия. Книга позволяет понять, чем отличаются подходы к разработке больших бизнес-систем.

[PiJV1]**Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML**

“Шаблоны проектирования в JAVA. Каталог популярных шаблонов проектирования, проиллюстрированных при помощи UML”

Издательство: Wiley Publishing (2002), Новое знание (2004)

ISBN: 0-471-22729-3 (англ.), 5-94735-047-5 (рус.)

Автор: Марк Гранд (Mark Grand)

Каталог шаблонов проектирования, часто применявшихся в языке Java.

Источник дополнительной информации

<http://www.markgrand.com/id1.html>

[PLoPD3]**Pattern Languages of Program Design 3**

Издательство: Addison-Wesley (1998)

ISBN: 0-201-31011-2

Редакторы: Роберт Мартин (Robert C. Martin), Дирк Рихле (Dirk Riehle) и Франк Бушман (Frank Buschmann)

Коллекция шаблонов, которые были разработаны на конференциях *Pattern Languages of Programs* (PLoP).

[POSA2]**Pattern-Oriented Software Architecture, Volume 2:
Patterns for Concurrent and Networked Objects**

Издательство: Wiley & Sons (2000)

ISBN: 0-471-60695-2

Авторы: Дуглас Шмидт (Douglas Schmidt), Майкл Стал (Michael Stal), Ганс Роберт (Hans Robert) и Франк Бушман (Frank Buschmann)

Это второй том популярной серии Pattern-Oriented Software Architecture (POSA). Первый том был опубликован в 1996 году. В этой книге показаны 17 взаимосвязанных шаблонов, лежащих в основе параллельных и сетевых систем: доступ к службам и их настройка, обработка событий, синхронизация и параллельная обработка.

[PUT]**Pragmatic Unit Testing**

Издательство: Pragmatic Bookshelf

ISBN: 0-9745140-2-0 (In C# with NUnit)

ISBN: 0-9745140-1-2 (In Java with JUnit)

Авторы: Энди Хант (Andy Hunt) и Дейв Томас (Dave Thomas)

В этой книге “прагматичных программистов” рассматривается концепция автоматизированного модульного тестирования. В обеих версиях книги барьер входа снижается через описание только основных концепций. Кроме этого, здесь присутствуют хорошие разделы с рекомендациями по определению необходимых тестов для конкретного класса или метода.

[RDb]

Refactoring Databases: Evolutionary Database Design

“Рефакторинг баз данных: эволюционное проектирование”

Издательство: Addison-Wesley (2006), Вильямс (2007)

ISBN: 0-321-29353-3 (англ.), 978-5-8459-1157-5 (рус.)

Авторы: Прамодкумар Садаладж (Pramodkumar J. Sadalage) и Скотт Эмблер (Scott W. Ambler)

Данная книга может служить хорошим введением в варианты применения гибких принципов при разработке приложений, зависящих от базы данных. В книге рассматриваются способы отказа от получения полного проекта базы данных еще до начала написания коды. Книга вполне заслужила место на полке каждого разработчика, применяющего гибкие методы для создания приложений на основе базы данных. Ее краткое содержание приводится на странице по адресу:

<http://www.ambysoft.com/books/refactoringDatabases.html>

[Ref]

Refactoring: Improving the Design of Existing Code

“Рефакторинг. Улучшение существующего кода”

Издательство: Addison-Wesley (1999), Символ-Плюс (2007)

ISBN: 0-201-48567-2 (англ.), 5-93286-045-6 (рус.)

Авторы: Мартин Фаулер (Martin Fowler) и др.

В этой книге приводится описание процесса рефакторинга программного обеспечения, рассматривается несколько “запахов кода” и предлагаются способы переработки кода для избавления от этих запахов.

[RTC]

Refactoring Test Code

Статья была представлена на конференции XP2001.

Авторы: Арие ван Дерсен (Arie van Deursen), Леон Монен (Leon Moonen), Алекс ван ден Берг (Alex van den Bergh) и Джерард Кок (Gerard Kok)

В этой статье впервые концепция “запахов” кода была применена к коду тестов. В ней рассмотрены 12 “запахов” тестов и предложен набор вариантов рефакторинга, которые могут использоваться для улучшения характеристик кода. Исходную статью можно найти по адресу:

<http://homepages.cwi.nl/~leon/papers/xp2001/xp2001.pdf>

[RtP] Refactoring to Patterns

“Рефакторинг с использованием шаблонов”

Издательство: Addison-Wesley (2005), Вильямс (2006)
 ISBN: 0-321-21335-1 (англ.), 978-5-8459-1087-5 (рус.)

Автор: Джошуа Кериевски (Joshua Kerievsky)

В этой книге рассматривается связь рефакторинга (процесса улучшения дизайна существующего кода) с шаблонами (классическими решениями повторяющихся проблем проектирования). Название книги подсказывает, что использование шаблонов для улучшения существующего дизайна выгоднее, чем использование шаблонов на ранних этапах проектирования (вне зависимости от возраста улучшаемого кода). Улучшение дизайна с использованием шаблонов заключается в применении последовательности низкоуровневых трансформаций, известных как рефакторинг.

[SBPP] Smalltalk Best Practice Patterns

Издательство: Prentice Hall (1997)
 ISBN: 0-13-476904-X

Автор: Кент Бек (Kent Beck)

В этой книге рассматриваются низкоуровневые шаблоны программирования, которые применяются в хорошем объектно-ориентированном ПО. На обложке приведен следующий отзыв Мартина Фаулера.

Стиль Кента в использовании Smalltalk является стандартом, которому я пытаюсь следовать в своей работе. Эта книга не только устанавливает подобный стандарт, но и объясняет, почему он является стандартом. Эта книга должна быть под рукой у каждого разработчика на языке Smalltalk.

Хотя Smalltalk уже не является доминирующим языком в объектно-ориентированной разработке, многие шаблоны, созданные разработчиками на Smalltalk, были приняты как стандартные решения и в других объектно-ориентированных языках. Шаблоны в этой книге остаются применимыми несмотря на то, что примеры написаны на языке Smalltalk.

[SCMP] Software Configuration Management Patterns: Effective Teamwork, Practical Integration

Издательство: Addison-Wesley (2003)
 ISBN: 0-201-74117-1

Авторы: Стив Берчук (Steve Berczuk) и Брэд Эпплтон (Brad Appleton)

В этой книге в форме шаблонов показано, как и зачем использовать системы управления конфигурацией исходного кода для синхронизации действий нескольких разраб-

ботчиков. Описанные подходы одинаково пригодны как для проектов на основе гибких методов, так и для традиционных проектов.

Источники дополнительной информации

<http://www.scmpatterns.com>
<http://www.scmpatterns.com/book/pattern-summary.html>

[SoC]

Secrets of Consulting: A Guide to Giving and Getting Advice Successfully

Издательство: Dorset House (1985)

ISBN: 0-932633-01-3

Автор: Джеральд Вайнберг (Gerald Weinberg)

Книга полна правил и законов Джерри, например “Закон малинового варенья: чем дальше размазываешь, тем тощее становится”.

[TAM]

Test Automation Manifesto

<http://TestAutomationManifesto.gerardmeszaros.com>

Авторы: Шон Смит (Shaun Smith) и Джерард Месарош (Gerard Meszaros)

Эта статья была представлена на конференции XP/Agile Universe в 2003 году в Новом Орлеане. В ней рассматривается ряд принципов, позволяющих сделать автоматизированные модульные тесты на базе xUnit эффективными с точки зрения затрат.

[TDD-APG]

Test-Driven Development: A Practical Guide

Издательство: Prentice Hall (2004)

ISBN: 0-13-101649-0

Автор: Дэвид Астелс (David Astels)

Книга может служить хорошим введением в процесс управления разработкой программного обеспечения с помощью модульных тестов. В третьей части книги приводится полный пример использования тестов для управления небольшим проектом на языке Java.

[TDD-BE]

Test-Driven Development: By Example

“Экстремальное программирование: разработка через тестирование”

Издательство: Addison-Wesley (2003), Питер (2003)

ISBN: 0-321-14653-0 (англ.), 5-8046-0051-6 (рус.)

Автор: Кент Бек (Kent Beck)

Книга может служить хорошим введением в процесс управления разработкой программного обеспечения с помощью модульных тестов. Во второй части книги Кент демонстрирует применение разработки на основе тестов на примере создания *инфраструктуры автоматизации тестов* (Test Automation Framework, с. 332) на языке Python. Кент любит называть такой подход “хирургической операцией на собственном мозгу”. Создаваемая инфраструктура используется для запуска тестов для каждой новой функции инфраструктуры. Это хороший пример разработки на основе тестов и последовательного добавления новых функций на основе уже существующих.

[TDD.Net]

Test-Driven Development in Microsoft .NET

Издательство: Microsoft Press (2004)

ISBN: 0-735-61948-4

Авторы: Джеймс Ньюкирк (James W. Newkirk) и Алексей Воронцов (Alexei A. Vorontsov)

Эта книга является хорошим введением в процесс разработки на основе тестов и инструментарий, позволяющий вести разработку на платформе Microsoft.NET.

[TI]

Test Infected

<http://junit.sourceforge.net/doc/testinfected/testing.htm>

Авторы: Эрих Гамма (Eric Gamma) и Кент Бек (Kent Beck)

Впервые эта статья была опубликована в журнале Java Report под названием “Test Infected — Programmers Love Writing Tests”. Некоторые считают, что именно эта статья послужила стремительному росту популярности инфраструктуры JUnit. Статья может служить хорошим введением в способы и причины использования xUnit для автоматизации тестов.

[TPS]

Toyota Production System: Beyond Large-Scale Production

Издательство: Productivity Press (1995)

ISBN: 0-915-2991-4-3

Автор: Таichi Оно (Taiichi Ohno)

В этой книге, которая была написана отцом метода “just-in-time manufacturing”, показано, как в компании Тойота организовали систему, вызванную необходимостью производить небольшое количество автомобилей в соответствии с потребностями экономики. Среди описанных подходов можно выделить “Пять Почему”.

[UTF]

Unit Test Frameworks: Tools for High-Quality Software Development

Издательство: O'Reilly (2004)

ISBN: 0-596-00689-6

Автор: Пол Хэмилл (Paul Hamill)
Это краткое введение в наиболее популярные реализации xUnit.

[UTwHCM] Unit Testing with Hand-Crafted Mocks

<http://refactoring.be/articles/mocks/mocks.html>

Автор: Свен Гортс (Sven Gorts)

В этой статье перечислены идиомы, связанные с *созданными вручную тестовыми двойниками* (Hand-Built Test Double; см. *Настраиваемый тестовый двойник*, Configurable Test Double, с. 571) в частности, с *тестовыми заглушками* (Test Stub, с. 544) и *подставными объектами* (Mock Object, с. 559). Свен Гортс пишет следующее.

Многие из написанных в последние пару лет модульных тестов использовали подставные объекты для проверки поведения компонента в изоляции от остальной части системы. Несмотря на достаточное количество инструментов автоматической генерации подставных объектов, каждый используемый подставной класс был написан вручную. В этой статье приводится ретроспективное описание самых полезных идiom подставных объектов.

[UTwJ] Unit Testing in Java: How Tests Drive the Code

Издательство: Morgan Kaufmann (2003)
ISBN: 1-55860-868-0

Авторы: Йоханнес Линк (Johannes Link) с дополнениями от Петера Фролича (Peter Frohlich)

В этой книге приводится описание многих концепций и подходов модульного тестирования. К сожалению, формат книги не позволяет легко находить интересующую информацию.

[VCTP] The Virtual Clock Test Pattern

http://www.nusco.org/docs/virtual_clock.pdf

Автор: Паоло Перотта (Paolo Perrotta)

В этой статье описывается распространенный пример шаблона *генератор ответов* (Responder), который называется Virtual Clock Test Pattern (VCTP). Автор воспользовался им в качестве *декоратора* (Decorator) [GOF] для настоящих системных часов, что позволило “замораживать” и восстанавливать время. Для большинства тестов достаточно в этой роли использовать *фиксированную тестовую заглушку* (Hard-Coded Test Stub) или *настраиваемую тестовую заглушку* (Configurable Test Stub). Вот краткое описание статьи от самого Паоло.

Иногда очень сложно тестировать код, который зависит от системных часов. В этой статье рассматривается сама проблема, а также распространенные и пригодные для повторного использования решения.

[WEwLC]

Working Effectively with Legacy Code

Издательство: Prentice Hall (2005)

ISBN: 0-13-117705-2

Автор: Майкл Фезерс (Michael Feathers)

В этой книге рассматривается возврат контроля над унаследованной программной системой за счет интеграции автоматизированных модульных тестов. Ключевым элементом процесса является набор “способов разрыва зависимости” (состоящий по большей части из вариантов рефакторинга), который позволяет изолировать программное обеспечение для автоматизированного тестирования.

[Wp]

Wikipedia

Из Wikipedia:

Википедия (англ. Wikipedia) — многоязычная общедоступная свободно распространяемая энциклопедия, публикуемая в Интернете. Создается на многих языках мира коллективным трудом добровольных авторов, использующих технологию “вики”.

[WWW]

World Wide Web

Ссылка вида [WWW] означает, что информация была найдена в Интернете. Для доступа к ней достаточно ввести в поисковую машину название интересующей статьи.

[XP123]

XP123

<http://xp123.com>

Управляющий сайта: Уильям Уэйк (William Wake)

На этом сайте расположены различные ресурсы для команд, практикующих экстремальное программирование.

[XPC]
XProgramming.com

<http://xprogramming.com>

Управляющий сайта: Рон Джейфриз (Ron Jeffries)

На этом сайте предлагаются различные ресурсы для команд, практикующих экстремальное программирование. Это один из лучших источников ссылок на программное обеспечение для автоматизации модульных тестов, включая реализации инфраструктуры xUnit.

[XPE]
eXtreme Programming Explained, Second Edition: Embrace Change

Издательство: Addison-Wesley (2005)

ISBN: 0-321-27865-8

Автор: Кент Бек (Kent Beck)

С этой книги началось распространение методологии экстремального программирования. В первом издании (ISBN: 0-201-61641-6) описывались рецепты на основе 12 процессов и лежащих в их основе принципов и ценностей. Во втором издании основное внимание уделяется ценностям и принципам. Процессы поделены на основной и результатирующий наборы. Последний набор должен применяться только после освоения основного. Среди остальных процессов в обоих изданиях рассматриваются парное программирования и разработка на основе тестов.

Предметный указатель

X

xUnit, 109; 110; 127; 133; 201
процедурная реализация, 134

A

Автоматизация тестов, 76

Б

Базы данных, 213; 215
тестирование, 217

В

Вставка зависимости, 192
Встроенное утверждение, 162
Встроенный автотест, 211

Д

Дельта-утверждение, 163

З

Зависимость тестов, 212
Запах кода, 73
 длинный тест, 230
 дублирование тестового кода, 254
 код теста со смещением, 243
 логика теста в продукте, 257
 непонятный тест, 230
 подробный тест, 230
 сложный в тестировании код, 251
 сложный тест, 230
 условная логика теста, 243
Запах поведения, 70
 “хрупкий” тест, 277
 медленные тесты, 289

неустойчивый тест, 267
отладка вручную, 285
roulette утверждений, 264
ручное вмешательство, 287
частая отладка, 285

Запах проекта, 69

высокая стоимость обслуживания тестов, 300
ошибки в продукте, 303
разработчики не пишут тесты, 298
тест с ошибками, 296

Запах теста, 67

Запуск тестов, 131

И

Инкрементная разработка, 87
Инкрементный тест, 353
Исследовательский тест, 106

К

Класс теста, 203
Команды тестов, 134
Контроль качества, 77
Контрольная точка, 119
Кроссфункциональные тесты, 106

М

Медленный тест, 351
Модульный тест, 63; 105

Н

Набор тестов, 130; 206
Наследование класса теста, 210

О

Объект набора тестов, 134
 Объект-заглушка, 183
 Ожидаемый объект, 166
 Опосредованный ввод, 175
 Опосредованный вывод, 175

П

Пакет тестов, 211
 Параметризованный тест, 169
 Повторное использование кода, 209
 Поддельный объект, 188
 Подкожный тест, 370
 Подставной объект, 187
 Поиск зависимости, 193
 Приемочный тест, 62; 104
 Принцип
 доносите намерение, 95
 изолируйте тестируемую систему, 97
 минимизируйте нетестируемый код, 98
 минимизируйте пересечения тестов, 98
 не вносите логику тестов в код продукта, 99
 не модифицируйте тестируемую систему, 95
 независимый тест, 96
 обеспечьте соответственные усилия и ответственность, 101
 понимание с одного взгляда, 95
 роверяйте одно условие за тест, 99
 проектируйте с учетом тестирования, 94
 разработка на основе тестов, 94
 сначала используйте “главный” вход, 94
 сначала пишите тесты, 94
 сохраняйте независимость тестов, 96
 тест одного условия, 99
 тестируйте аспекты по-отдельности, 101
 язык высокого уровня, 95
 Причина
 асинхронный код, 252
 асинхронный тест, 291
 взаимодействующие наборы тестов, 270

взаимодействующие тесты, 268
 война запуска тестов, 274
 всегда успешный тест, 308
 гибкий тест, 245
 зависимость продукта от теста, 260
 зарегулированная программа, 283
 засорение равенства, 260
 использование медленного компонента, 290
 ловушка для теста, 257
 логика продукта внутри теста, 247
 недостаточно времени, 298
 неопределенный тест, 275
 неповторяющийся тест, 272
 непонятный тест, 297; 301
 неправильная стратегия автоматизации тестов, 299
 несколько тестовых условий, 249
 нетестированное требование, 307
 нетестированный код, 306
 нетестируемый код теста, 253
 неуместная информация, 235
 нечасто запускаемые тесты, 303
 одинокий тест, 271
 опосредованное тестирование, 239
 оптимизм по отношению к ресурсу, 272
 отсутствующее сообщение для утверждения, 265
 отсутствующий модульный тест, 305
 повторное изобретение колеса, 256
 повторное использование через копирование, 255
 потерянный тест, 304
 ручная генерация события, 288
 ручная настройка тестовой конфигурации, 287
 ручная проверка результата, 288
 слишком много тестов, 292
 слишком связанный тест, 283
 сложная очистка, 248
 сложный в тестировании код, 297; 298; 301
 таинственный гость, 232
 тесно связанный код, 252
 тестовая конфигурация общего характера, 234; 290
 только для тестов, 259
 условная логика проверки, 246
 утечка ресурсов, 271

фиксированная зависимость, 252
 фиксированные данные теста, 238
 хрупкая тестовая конфигурация, 284
 хрупкий тест, 296; 301
 чувствительное сравнение, 283
 чувствительность к данным, 280
 чувствительность к интерфейсу, 278
 чувствительность к контексту, 282
 чувствительность к поведению, 279
 энергичный тест, 231; 264
Проверка поведения, 164
Проверка состояния, 161

P

Разработка на основе поведения, 89
Разработка на основе примеров, 87
Рефакторинг
 встраивание ресурса, 738
 выделение тестируемого компонента, 737
 замена зависимости тестовым двойником, 740
 минимизация данных, 739
 настройка внешнего ресурса, 741
 отращиваемый класс, 737
 создание уникального ресурса, 739
Рефакторинг записанных тестов, 314

C

Самопроверяющийся тест, 159
Соглашение об именовании, 206
Специальное утверждение, 168

T

Тест, 75
 инициализации зависимости, 381
 компонента, 105; 371
 конструктора, 381
 презентационного уровня, 369
 с внедрением ошибок, 106
 свойств, 106
 служебного уровня, 370
 уровня хранения, 370
 функциональности, 104
 эргономики, 106
Тестовая заглушка, 185

Тестовая конфигурация, 111; 112; 130; 137
 настройка, 346
 встроенная, 140
 гибридная, 145
 делегированная, 141
 неявная, 143
 новая, 139
 временная, 114; 139
 постоянная, 115
 общая, 116; 154
 отказ от очистки, 152
 очистка, 145
 автоматизированная, 151
 постоянная, 147
 создание, 140; 156

Тестовые файлы, 211
Тестовый двойник, 183; 189; 197; 743
 настройка, 190
 тестовый агент, 186
Тестовый метод, 203
Точка наблюдения, 119

Y

Управляемый данными тест, 169
Условная логика теста, 170

X

Хранимые процедуры, 217

П

Цель
 выполняемая спецификация, 77
 выразительные тесты, 83
 локализация дефектов, 78
 не вносить рисков с помощью тестов, 79
 не навреди, 79
 повторяемые тесты, 81
 полностью автоматизированные тесты, 81
 простые тесты, 83
 разделение интереса, 83
 репеллент для ошибок, 78
 самопроверяющийся тест, 81
 страховочная сеть, 79
 тесты как документация, 79

тесты как спецификация, 77
тесты как страховка, 79
устойчивый к изменениям тест, 84

III

Шаблон

абстрактный тест, 646
автоматизированный модульный тест, 319
автоматическая очистка, 521
агент, 552
брокер компонентов, 692
вспомогательный класс теста, 651
вспомогательный метод теста, 610
вставка зависимости, 684
встроенная настройка, 434
встроенная очистка, 527
выбор тестов, 429
вызываемая инфраструктурой
очистка, 533
вычисляемое значение, 722
генератор объекта набора тестов, 326
декоратор настройки, 471
делегированная настройка, 437
дельта-утверждение, 505
заглушка, 544
заменитель, 730
записанный тест, 312
записывающая тестовая заглушка, 552
именованный набор тестов, 604
имитатор объекта набора тестов, 327
инфраструктура автоматизации
тестов, 332
класс теста, 401
класс теста для каждого класса, 627
класс теста для каждой тестовой кон-
фигурации, 639
класс теста для каждой функции, 633
константное значение, 718
кукла, 565
ленивая настройка, 460
ловушка для теста, 713
локатор служб, 692
манипуляция через “черный ход”, 359
метод очистки, 533
метод с утверждением, 390
метод создания, 441
минимальная тестовая
конфигурация, 336

минимальный контекст, 336
минимальный объект, 700
написанный вручную тест, 319
настраиваемый тестовый двойник, 571
настройка тестовой конфигурации на-
бора, 465
неявная настройка, 449
неявная очистка, 533
новая тестовая конфигурация, 344
новый контекст, 344
обнаружение тестов, 420
общая тестовая конфигурация, 350
общий контекст, 350
объект набора тестов, 414
объект теста, 410
объект-заглушка, 730
очистка откатом транзакции, 675
очистка со сборкой мусора, 518
очистка усечением таблиц, 668
очистка через “ловушки”, 533
параметризованный тест, 618
перечисление тестов, 425
песочница с базой данных, 658
поддельный объект, 565
подставной объект, 558
поиск зависимости, 692
предварительно созданная
тестовая конфигурация, 454
проверка поведения, 489
проверка состояния, 484
программа запуска тестов, 405
программный тест, 319
простейший интерпретатор
тестов xUnit, 326
реестр компонентов, 692
реестр тестовых объектов, 521
самозванец, 538
связанный с тестом подкласс, 591
сгенерированное значение, 726
сообщение для утверждения, 398
специальное утверждение, 495
стандартная тестовая
конфигурация, 338
стандартный контекст, 338
сторожевое утверждение, 510
суперкласс теста, 646
тест “роботом”-пользователем, 312
тест на основе записи
и воспроизведения, 312
тест на основе сценария, 319

- тест уровня, 368
тест хранимой процедуры, 662
тестирование взаимодействия, 489
тестовая заглушка, 544
тестовая конфигурация, 401
тестовый агент, 552
тестовый двойник, 538
тестовый метод, 378
тестовый стенд, 454
точное значение, 718
управляемый данными тест, 322
утверждение незаконченного
теста, 514
- фабрика наборов тестов, 425
фабрика объектов, 692
фиксированное значение, 718
фиксированный тестовый
двойник, 581
цепочки тестов, 477
четырехфазный тест, 387

Э

Эндоскопическое тестирование, 197

Научно-популярное издание

Джерард Месарош

**Шаблоны тестирования xUnit:
рефакторинг кода тестов**

Литературный редактор *Л.Н. Красножон*

Верстка *О.В. Романенко*

Художественный редактор *С.А. Чернокозинский*

Корректоры *Л.А. Гордиенко, Л.В. Чернокозинская*

ООО “И. Д. Вильямс”
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 20.08.2008. Формат 70x100/16

Гарнитура NewtonC . Печать офсетная

Усл. печ. л. 67,08. Уч.-изд. л. 49,1

Тираж 3000 экз. Заказ № 0000

Отпечатано по технологии СтР
в ОАО “Печатный двор” им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15