

Экстремальное программирование

разработка через тестирование

- предсказуемый способ разработки программ
- шанс усвоить уроки, которые преподносит вам код
- жизнь пользователей разработанного вами программного обеспечения улучшится
- ваши коллеги смогут рассчитывать на вас, а вы сможете рассчитывать на них



Кент Бек

Экстремальное программирование

разработка через тестирование

- **предсказуемый способ разработки программ**
- **шанс усвоить уроки, которые преподносит вам код**
- **жизнь пользователей разработанного вами программного обеспечения улучшится**
- **ваши коллеги смогут рассчитывать на вас, а вы сможете рассчитывать на них**

Kent Beck

Test-driven development

by example



Addison-Wesley

An imprint of Addison Wesley Longman, Inc.
Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Кент Бек

БИБЛИОТЕКА ПРОГРАММИСТА

**Экстремальное
программирование**

**разработка через
тестирование**



**Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск**

2003

Кент Бек

Экстремальное программирование: разработка через тестирование Библиотека программиста

Перевел с английского П. Анджан

Главный редактор
Заведующий редакцией
Руководитель проекта
Научный редактор
Корректор
Верстка

*Е. Строганова
И. Корнеев
А. Пасечник
С. Орлов
В. Листова
П. Быстров*

ББК 32.973.2-018

УДК 681.3.06

Бек К.

**Б42 Экстремальное программирование: разработка через тестирование.
Библиотека программиста. — СПб.: Питер, 2003. — 224 с.: ил.**

ISBN 5-8046-0051-6

Изящный, гибкий и понятный код, который легко модифицировать, который корректно работает и который не подкидывает своим создателям неприятных сюрпризов. Неужели подобное возможно? Чтобы достичь цели, попробуйте тестировать программу еще до того, как она написана. Именно такая парадоксальная идея положена в основу методики TDD (Test-Driven-Development — разработка, основанная на тестировании). Бессмыслица? Не спешите делать скороспелых выводов. Рассматривая применение TDD на примере разработки реального программного кода, автор демонстрирует простоту и мощь этой новой методики. В книге рассматриваются два программных проекта, целиком и полностью реализованных с использованием TDD. За рассмотрением примеров следует обширный каталог приемов работы в стиле TDD, а также паттернов и рефакторингов, имеющих отношение к TDD. Книга будет полезна для любого программиста, желающего повысить производительность своей работы и получить удовольствие от программирования.

© Pearson Education, Inc., 2003

© Перевод на русский язык, ЗАО Издательский дом «Питер», 2003

© Издание на русском языке, оформление, ЗАО Издательский дом «Питер», 2003

Права на издание получены по соглашению с Addison-Wesley Longman.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 5-8046-0051-6

ISBN 0-321-14653-0 (англ.)

ООО «Питер Принт», 196105, Санкт-Петербург, ул. Благодатная, д. 67в.

Лицензия ИД № 05784 от 07.09.01.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано к печати 14.08.03. Формат 70×100/16. Усл. п. л. 18,06. Тираж 3000. Заказ 297

Отпечатано с готовых диапозитивов в ФГУП ордена Трудового Красного Знамени «Техническая книга»
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций
198005, Санкт-Петербург, Измайловский пр., 29

Содержание

Предисловие	10
Храбрость	11
Благодарности.	15
От издательства	16
Введение	17
Часть I. На примере денег	21
Глава 1. Мультивалютные деньги	22
Глава 2. Вырождающиеся объекты.	29
Глава 3. Равенство для всех	32
Глава 4. Данные должны быть закрытыми	36
Глава 5. Поговорим о франках.	39
Глава 6. Равенство для всех, вторая серия.	42
Глава 7. Яблоки и апельсины	47
Глава 8. Создание объектов	49
Глава 9. Потребность в валюте	53
Глава 10. Избавление от двух разных версий times()	58
Глава 11. Корень всего зла	63
Глава 12. Сложение, наконец-то	66
Глава 13. Делаем реализацию реальной.	71
Глава 14. Обмен валюты	76
Глава 15. Смешение валют	81
Глава 16. Абстракция, наконец-то!	85
Глава 17. Ретроспектива денежного примера	89
Что дальше?	89
Метафора	90

Использование JUnit	91
Метрики кода	92
Процесс	92
Качество тестов	93
Последний взгляд назад	95
Часть II. На примере xUnit	97
Глава 18. Первые шаги на пути к xUnit.	98
Глава 19. Сервируем стол (метод setUp)	104
Глава 20. Убираем со стола (метод tearDown).	108
Глава 21. Учет и контроль	112
Глава 22. Обработка несработавшего теста	115
Глава 23. Оформляем тесты в набор	118
Глава 24. Ретроспектива xUnit	123
Часть III. Паттерны для разработки через тестирование.	125
Глава 25. Паттерны разработки, основанной на тестах	126
Тест	126
Изолированный тест (Isolated Test)	128
Список тестов (Test List)	129
Вначале тест (Test First)	131
Вначале оператор assert (Assert First)	132
Тестовые данные (Test Data)	134
Понятные данные (Evident Data)	135
Глава 26. Паттерны красной полосы	136
One Step Test (Тест одного шага)	136
Starter Test (Начальный тест)	137
Explanation Test (Объясняющий тест)	139
Learning Test (Тест для изучения)	139
Another Test (Еще один тест)	140
Regression Test (Регрессионный тест)	141
Break (Перерыв)	141
Do over (Начать сначала)	143
Cheap Desk, Nice Chair (Дешевый стол, хорошие кресла)	144
Глава 27. Паттерны тестирования	145
Дочерний тест (Child Test)	145
Mock Object (Поддельный объект)	146
Self Shunt (Самошунтирование)	147
Log String (Строка-журнал)	149
Crush Test Dummy (Тестирование обработки ошибок)	149

Broken Test (Сломанный тест)	151
Clean Check-in (Чистый выпускаемый код)	151
Глава 28. Паттерны зеленой полосы	153
Fake It (Подделка).	153
Triangulate (Триангуляция)	155
Obvious Implementation (Очевидная реализация).	156
One to Many (От одного ко многим)	157
Глава 29. Паттерны xUnit	159
Assertion.	159
Fixture (Фикстура)	161
External Fixture (Внешняя фикстура)	163
Test Method (Тестовый метод)	164
Exception Test (Тест исключения)	166
All Tests (Все тесты)	166
Глава 30. Паттерны проектирования	168
Command (Команда).	170
Value Object (Объект-значение)	170
Null Object (Нуль-объект)	172
Template Method (Шаблонный метод)	173
Pluggable Object (Встраиваемый объект)	175
Pluggable Selector (Встраиваемый переключатель)	176
Factory Method (Фабричный метод)	178
Imposter (Самозванец).	179
Composite (Компоновщик)	180
Collecting Parameter (Накопление в параметре)	182
Singleton (Одиночка)	183
Глава 31. Рефакторинг	184
Reconcile Differences (Согласование различий)	184
Isolate Change (Изоляция изменений)	185
Migrate Data (Миграция данных)	186
Extract Method (Выделение метода)	188
Inline Method (Встраивание метода)	189
Extract Interface (Выделение интерфейса)	190
Move Method (Перемещение метода).	191
Method Object (Метод в объект)	192
Add Parameter (Добавление параметра)	193
Method Parameter to Constructor Parameter (Параметр метода в параметр конструктора)	193
Глава 32. Развитие навыков TDD	195
Насколько большими должны быть шаги?	195
Что не подлежит тестированию?	196
Как определить качество тестов?	196

Каким образом TDD ведет к созданию инфраструктур?	197
Сколько должно быть тестов?	198
Когда следует удалять тесты?	200
Каким образом язык программирования и среда разработки влияют на TDD?	201
Можно ли использовать TDD для разработки крупномасштабных систем?	201
Можно ли осуществлять разработку приложения исходя из тестов уровня приложения?	202
Как можно перейти к использованию TDD в середине работы над проектом?	203
Для кого предназначена методика TDD?	204
Зависит ли эффективность TDD от начальных условий?	205
Каким образом методика TDD связана с паттернами?	205
Почему TDD работает?	206
Что означает имя?	208
Как методика TDD связана с практиками экстремального программирования?	208
Нерешенные проблемы TDD	210
Послесловие	211
Приложение I. Диаграммы взаимовлияния	213
Обратная связь	214
Контроль над системой	215
Приложение II. Фибоначчи	217
Алфавитный указатель	220

Посвящается Синди: крыльям моей души

Предисловие

Чистый код, который работает (clean code that works), — в этой короткой, но содержательной фразе, придуманной Роном Джефффризом (Ron Jeffries), кроется весь смысл методики Test-Driven Development (TDD). Чистый код, который работает, — это цель, к которой стоит стремиться, и этому есть причины.

- Это предсказуемый способ разработки программ. Вы знаете, когда работу следует считать законченной, и можете не беспокоиться о длинной череде ошибок.
- У вас появляется шанс усвоить уроки, которые преподносит вам код. Если вы воспользуетесь первой же идеей, которая пришла вам в голову, у вас не будет шанса реализовать вторую, лучшую идею.
- Жизнь пользователей разработанного вами программного обеспечения улучшается.
- Ваши коллеги по команде могут на вас рассчитывать, а вы можете рассчитывать на них.
- Вам приятнее писать такой код.

Но как мы можем получить чистый код, который работает? Очень многие силы мешают нам получить чистый код, а иногда нам не удается получить даже код, который работает. Чтобы избавиться от множества проблем, мы будем разрабатывать код, исходя из автоматических тестов. Такой стиль программирования называется разработкой через тестирование, Test Driven Development (TDD). В рамках этой методики мы:

- пишем новый код только тогда, когда автоматический тест не сработал;
- удаляем дублирование.

Два простых правила, не правда ли? Однако они генерируют сложное индивидуальное и групповое поведение со множеством технических последствий:

- проектируя код, мы постоянно запускаем его и получаем представление о том, как он работает, это помогает нам принимать правильные решения;
- мы самостоятельно пишем свои собственные тесты, так как мы не можем ждать, что кто-то другой напишет тесты для нас;
- наша среда разработки должна быстро реагировать на небольшие модификации кода;
- дизайн программы должен базироваться на использовании множества сильно связанных компонентов, которые слабо сцеплены друг с другом, благодаря чему тестирование кода упрощается.

Два упомянутых нами правила TDD определяют порядок этапов программирования.

1. Красный — напишите небольшой тест, который не работает, а возможно, даже не компилируется.
2. Зеленый — заставьте тест работать как можно быстрее, при этом не думайте о правильности дизайна и чистоте кода. Напишите ровно столько кода, чтобы тест сработал.
3. Рефакторинг — удалите из написанного вами кода любое дублирование.

Красный—зеленый—рефакторинг — это мантра TDD.

Если допустить, что такой стиль программирования возможен, значит, можно предположить, что благодаря его использованию код будет содержать существенно меньше дефектов, кроме того, цель работы будет ясна всем, кто принимает в ней участие. Если так, тогда разработка только такого кода, который требуется, чтобы обеспечить срабатывание тестов, приводит также к социальным последствиям:

- если плотность дефектов будет достаточно низкой, тогда команда контроля качества (Quality Assurance, QA) может перейти от реактивной к превентивной (предупреждающей) работе;
- если количество неприятных сюрпризов будет достаточно небольшим, значит, менеджеры проекта смогут с высокой точностью оценивать трудозатраты и привлекать заказчиков к процессу ежедневной разработки проекта;
- если темы технических дискуссий будут четко определены, программисты смогут взаимодействовать друг с другом постоянно, а не раз в день или раз в неделю;
- и опять же, если плотность дефектов будет достаточно небольшой, мы сможем каждый день получать интегрированный рабочий продукт с добавленной в него новой функциональностью, благодаря чему мы сможем вступить с нашими заказчиками в деловые отношения совершенно нового типа.

Итак, концепция проста, но какова мотивация? Почему программист должен взять на себя дополнительную обязанность, связанную с написанием автоматических тестов? Зачем программисту двигаться вперед малюсенькими шажками, когда его мозг в состоянии продумать гораздо более сложную структуру дизайна? Храбрость.

Храбрость

TDD — это способ управления страхом в процессе программирования. Я не имею в виду страх перед падением со стула или страх перед начальником. Я имею в виду страх в смысле «это очень сложная задача, и, ознакомившись с условием, я понятия не имею, как ее решить». Боль — это когда природа говорит нам: «Стоп!», а страх — это когда природа говорит нам: «Будь осторожен!» Осторожность — это совсем неплохо, однако помимо пользы страх оказывает на нас некоторое негативное влияние:

- страх заставляет нас заблаговременно и тщательно обдумывать, к чему может привести то или иное действие;
- страх заставляет нас меньше общаться;
- страх заставляет нас пугаться результатов своей работы;
- страх делает нас раздражительным.

Ни один из этих эффектов нельзя назвать полезным в процессе программирования, особенно если вы работаете над сложной задачей. Итак, перед нами встает вопрос, каким образом мы можем выйти из сложной ситуации и

- вместо того чтобы пытаться предсказать будущее, немедленно приступить к практическому изучению проблемы;
- вместо того чтобы отгораживаться от остального мира, повысить уровень коммуникации;
- вместо того чтобы забыть об обратной связи, напротив, установить надежную обратную связь, при помощи которой тщательно контролировать результаты своих действий;
- (с раздражением вы должны справиться самостоятельно.)

Сравним программирование с подъемом ведра из колодца. Ведро наполнено водой, вы вращаете рычаг, наматывая тем самым цепь на барабан и поднимая ведро вверх. Если ведро небольшое, вполне подойдет обычный рычаг, напрямую соединенный с барабаном. Но если ведро большое и тяжелое, вы наверняка устанете, пока будете поднимать его на самый верх. Чтобы обеспечить возможность отдыхать между поворотами рычага, вам потребуется храповой механизм, который позволит фиксировать рычаг в случае, если вы устали. Чем тяжелее ведро, тем ближе друг к другу должны располагаться шипы на шестеренке храповика.

Тесты в TDD — это шипы на шестеренке храповика. Заставив тест работать, мы знаем, что теперь тест работает, отныне и навеки. Мы стали на шаг ближе к завершению работы, чем мы были до того, как тест сработал. После этого мы заставляем второй тест работать, затем третий, четвертый и т. д. Чем сложнее проблема, стоящая перед программистом, тем меньшую область функциональности должен покрывать каждый тест.

Читатели книги *Extreme Programming Explained* должно быть обратили внимание на разницу в тоне между XP (*Extreme Programming*) и TDD (*Test-Driven Development*). В отличие от XP методика TDD не является абсолютной. XP говорит: «Вот вещи, которые вы обязаны освоить, прежде чем сможете совершенствоваться дальше». TDD — это несколько менее конкретная методика. TDD исходит из того, что в процессе программирования вы знаете об интервале между принятием решения и контролем результатов, и предлагает вам инструменты для контроля над продолжительностью этого интервала. «Что, если я в течение недели буду проектировать дизайн на бумаге, а затем напишу код, используя подход „вначале тесты“? Будет ли это TDD?» Конечно, это будет TDD. Вы знаете величину интервала между принятием решения и оценкой результатов, и вы осознанно контролируете этот интервал.

Большинство людей, освоивших TDD, утверждают, что их практика программирования изменилась к лучшему. *Инфицированные тестами* (test infected) — такое определение придумал Эрих Гамма (Erich Gamma) для того, чтобы описать данное изменение. Освоив TDD, вы обнаруживаете, что пишете значительно больше тестов, чем раньше, и двигаетесь вперед малюсенькими шагами, которые раньше показались бы вам бессмысленными. С другой стороны, некоторые программисты, изучившие TDD, решили вернуться к использованию своих прежних практик, зарезервировав TDD для специальных случаев, когда обычное программирование не приводит к желаемому прогрессу.

Определенно, существуют задачи, которые невозможно (по крайней мере, на текущий момент) решить только при помощи тестов. В частности, TDD не позволяет механически продемонстрировать адекватность разработанного кода в области безопасности данных и взаимодействия между процессами. Безусловно, безопасность основана на коде, в котором не должно быть дефектов, однако она основана также на участии человека в процедурах защиты данных. Тонкие проблемы, возникающие в области взаимодействия между процессами, невозможно с уверенностью воспроизвести, просто запустив некоторый код.

После того как вы прочитаете эту книгу, вы сможете:

- начать с простого;
- написать автоматический тест;
- выполнить рефакторинг, чтобы реализовать решения в области дизайна по одному за один раз.

Книга разделена на три части.

Часть I. На примере денег. Пример типичного прикладного кода, написанного с использованием TDD. Этот пример позаимствован мною у Уорда Каннингэма (Ward Cunningham) много лет назад, и с тех пор я неоднократно использовал его для демонстрации TDD. В нем рассматривается мультивалютная арифметика: выполнение математических операций над денежными величинами, выраженными в различных валютах. Этот пример научит вас писать тесты до того, как написан тестируемый ими код, кроме того, вы научитесь органически развивать дизайн.

Часть II. На примере xUnit. Пример тестирования более сложной логики, включая Reflection API и исключения. В примере рассматривается разработка инфраструктуры автоматического тестирования. Этот пример познакомит вас также с архитектурой xUnit, которая лежит в основе множества инструментов тестирования. Во втором примере вы научитесь двигаться вперед еще меньшими шагами, а также разрабатывать систему с использованием механизмов самой этой системы.

Часть III. Паттерны для разработки через тестирование. Здесь рассматриваются паттерны, которые помогут вам найти ответы на множество вопросов, в частности, какие тесты писать и как писать тесты с использованием xUnit. Кроме того, здесь вы найдете описание некоторых избранных паттернов проектирования и рефакторингов, использовавшихся при рассмотрении примеров данной книги.

Я написал примеры так, как будто мы с вами принимаем участие в сеансе парного программирования. Если перед тем, как пойти гулять, вы предпочитаете вначале посмотреть на карту, можете сначала ознакомиться с паттернами в третьей части книги, а затем прочитать примеры и использовать их в качестве иллюстраций. Если вы предпочитаете сначала погулять, а потом взглянуть на карту и посмотреть, где вы побывали, тогда сначала прочитайте первые две части с примерами и обращайтесь к третьей части по мере необходимости, — для получения справочного материала. Некоторые из людей, просматривавших рукопись данной книги, указывали на то, что примеры усваиваются лучше, если во время чтения запустить среду разработки, набирать код и запускать тесты.

Касательно примеров хочу отметить следующее. Оба примера, мультивалютные вычисления и инфраструктура тестирования, могут показаться чрезвычайно простыми. Существуют более сложные, дефектные и уродливые решения этих же самых задач (мне лично неоднократно приходилось сталкиваться с подобными решениями). Чтобы сделать книгу более похожей на реальность, я мог бы продемонстрировать одно из таких решений. Однако моя цель и, я надеюсь, ваша цель — написать чистый код, который работает. Прежде чем пенять на то, что приведенные примеры слишком просты, на несколько секунд представьте себе мир программирования, в котором весь код выглядит также чисто и понятно, в котором нет слишком сложных решений, только проблемы, которые кажутся слишком сложными лишь с первого взгляда. Сложные проблемы нуждаются в тщательном обдумывании. TDD поможет вам добиться этого.

Благодарности

Спасибо всем тем, кто с необычайным усердием и самоотверженностью просматривал рукопись данной книги. Я беру на себя всю ответственность за представленный в книге материал, однако без посторонней помощи данная книга была бы куда менее читабельной и менее полезной. Перечислю всех, кто помогал мне, в произвольном порядке: Стив Фриман (Steve Freeman), Франк Вестфал (Frank Westphall), Рон Джеффрис (Ron Jeffries), Дирк Кёниг (Dirk Koning), Эдвард Хейят (Edward Heiatt), Таммо Фриис (Tammo Freese), Джим Ньюкирк (Jim Newkirk), Йоханнес Линк (Johannes Link), Манфред Ланж (Manfred Lange), Стив Хайес (Steve Hayes), Алан Френсис (Alan Francis), Джонатан Расмуссон (Jonathan Rasmusson), Шэйн Клаусон (Shane Clauson), Саймон Крэйз (Simon Crase), Кай Пентекост (Kay Pantecost), Мюррей Бишоп (Murrey Bishop), Райан Кинг (Ryan King), Билл Уэйк (Bill Wake), Эдмунд Швеппе (Edmund Schweppe), Кевин Лауренц (Kevin Lawrence), Джон Картер (John Carter), Флип (Phlip), Петер Хансен (Peter Hansen), Бен Шрёдер (Ben Schroeder), Алекс Чаффи (Alex Chaffee), Петер ван Руйен (Peter van Rooijen), Рик Кавала (Rick Kawala), Марк ван Хамерсвельд (Mark van Hamersveld), Дуг Шварц (Doug Swartz), Лурент Боссави (Laurent Bossavit), Илья Преуз (Ilia Preuz), Дэниэл Ле Берре (Daniel Le Berre), Франк Карвер (Frank Carver), Майк Кларк (Mike Clark), Кристиан Пекелер (Christian Pekeler), Карл Скотланд (Karl Scotland), Карл Манастер (Carl Manaster), Дж. Б. Рэйнсбергер (J. B. Rainsberger), Петер Линдберг (Peter Lindberg), Дарач Эннис (Darach Ennis), Кайл Кордес (Kyle Cordes), Джастин Сампсон (Justin Sampson), Патрик Логан (Patrik Logan), Даррен Хоббс (Darren Hobbs), Аарон Сансоне (Aaron Sansone), Сайвер Энстад (Syver Enstad), Шинобу Каваи (Shinobu Kawai), Эрик Мид (Erik Meade), Патрик Логан (Patrik Logan), Дан Равсторн (Dan Rawsthorne), Билл Рутисер (Bill Rutiser), Эрик Хэрман (Eric Herman), Пол Чишолм (Paul Chisholm), Азим Джалис (Asim Jalis), Айвэн Муур (Ivan Moor), Леви Пюрвилс (Levi Purvis), Рик Магридж (Rick Mugridge), Энтони Адаши (Antony Adachi), Найджел Торн (Nigel Thorne), Джон Блей (John Bley), Кари Хойджарви (Kari Hoijarvi), Мануэль Амаго (Manuel Amago), Каору Хосокава (Kaoru Hosokawa), Пэт Эйлер (Pat Eyler), Росс Шав (Ross Shaw), Сэм Джэнтл (Sam Gentle), Джин Райотт (Jean Rajotte), Филип Антрас (Phillipe Antras) и Джим Нино (Jaime Nino).

Я хотел бы выразить свою признательность всем тем программистам, с которыми я разрабатывал код в стиле «вначале тесты». Спасибо вам за терпение и внимание к идее, которая звучала полным сумасшествием, в особенности в самом начале развития TDD. Благодаря вам я научился значительно большему, чем если бы действовал самостоятельно. Мое обучение было наиболее успешным, когда

я сотрудничал с Массимо Арнольди (Massimo Arnoldi), Ральфом Битти (Ralph Beatti), Роном Джефффризом (Ron Jeffries), Мартином Фаулером (Martin Fowler) и (безусловно, не в последнюю очередь) Эрихом Гаммой (Erich Gamma), однако я хотел бы отметить, что помимо этих людей были и другие, благодаря которым я тоже научился очень многому.

Я хотел бы поблагодарить Мартина Фаулера (Martin Fowler) за помощь в области FrameMaker. Этот человек должен быть самым высокооплачиваемым на планете специалистом в области подготовки текста к печати (к счастью, он не против того, чтобы гонорар за эту книгу целиком достался мне).

Я начал свою жизнь настоящего программиста благодаря наставничеству и в рамках постоянного сотрудничества с Уордом Каннингэмом (Ward Cunningham). Иногда я рассматриваю разработку, основанную на тестах, как попытку предоставить каждому программисту, работающему в произвольной среде, ощущение комфорта и интимности, которое было у нас с Уордом, когда мы вместе разрабатывали программы Smalltalk в среде Smalltalk. Не существует способа определить первоначальный источник идей, если два человека обладают одним общим мозгом. Если вы предположите, что все хорошие идеи на самом деле изначально придумал Уорд, вы не будете далеки от истины.

В последнее время сформировалось клише: выразить глубочайшую признательность за те жертвы и лишения, которые вынуждена терпеть семья, один из членов которой заболел идеей написать книгу. Дело в том, что семейные жертвы и лишения также необходимы для написания книги, как и бумага. Я выражаю свою самую глубочайшую благодарность моим детям, которые не могли приступить к завтраку, пока я не закончу очередную главу, а также моей жене, которая в течение двух месяцев была вынуждена повторять каждую свою фразу три раза.

Спасибо Майку Хэндersonу (Mike Henderson) за воодушевление, а также Марси Барнс (Marcy Barns) за то, что она пришла на помощь в трудную минуту.

Наконец, спасибо неизвестному автору книги, которую я прочитал в 12-летнем возрасте. В этой книге было предложено сравнивать две ленты: с реальными результатами и ожидаемыми результатами, и кодировать до тех пор, пока реальные результаты не совпадут с ожидаемыми. Спасибо, спасибо, спасибо.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На web-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

Введение

Однажды рано утром в пятницу к Уорду Каннингэму (Ward Cunningham) зашел босс и представил его Питеру, перспективному заказчику системы WyCash. Эта система предназначалась для управления портфелем облигаций, ее разработкой и продажей занималась компания Уорда. «Возможности вашей системы впечатляют, — сказал Питер. — Но вот в чем проблема: я собираюсь открыть новый фонд облигаций. Как я понял, ваша система поддерживает только облигации, номинированные в долларах США. Мне же понадобится система, обеспечивающая поддержку различных валют». Босс повернулся к Уорду и спросил: «Сможем ли мы это сделать?»

Вот он, кошмарный сценарий для любого разработчика. Все шло хорошо, пока события развивались по намеченному плану, и вдруг все меняется. Надо сказать, это было кошмаром не только для Уорда — босс, съевший собаку на управлении программными проектами, тоже не знал, что ответить.

Система WyCash была разработана небольшой командой программистов за пару лет. Она позволяла работать с большинством ценных бумаг с фиксированным доходом, имеющих хождение на американском рынке. Более того, она поддерживала некоторые редкие инструменты рынка ценных бумаг, например гарантированные инвестиционные контракты (Guaranteed Investment Contracts), и этим выгодно отличалась от конкурентов.

В основу разработки WyCash легли объектно-ориентированные технологии, также была использована объектная база данных. Базовой абстракцией системы был класс Dollar, который отвечал за вычисления и форматирование результатов. В самом начале работы над системой его разработку поручили отдельной группе хороших программистов.

В течение последних шести месяцев Уорд и остальные члены команды постепенно уменьшали количество обязанностей класса Dollar. Оказалось, что числовые классы языка Smalltalk вполне подошли для реализации вычислений, а для округления до трех десятичных знаков был написан специальный код. Результаты вычислений становились все точнее и точнее, и, в конце концов, сложные алгоритмы тестирования, выполнявшие сравнение величин с учетом погрешности, были заменены простым сравнением реального и ожидаемого результатов.

За форматирование результатов в действительности отвечали классы пользовательского интерфейса, а не класс Dollar. Так как соответствующие тесты были написаны на уровне этих классов, в частности для подсистемы отчетов¹, поэтому

¹ Подробнее о подсистеме отчетов рассказано на c2.com/doc/oopsla91.html.

предполагаемые изменения не должны были их коснуться. В результате, спустя шесть месяцев, у объекта Dollar осталось не так уж много обязанностей...

Один из наиболее сложных алгоритмов, вычисление средневзвешенных величин, также постепенно менялся. Вначале существовало много различных реализаций этого алгоритма, разбросанных по всему коду. Однако позже, с возникновением подсистемы отчетов, стало очевидно, что существует только одно место, где этот алгоритм должен быть реализован, — класс `AveragedColumn`. Именно этим классом и занялся Уорд.

Если бы удалось внедрить в этот алгоритм поддержку работы с несколькими валютами, то система, в целом, смогла бы стать «мультивалютной». Центральная часть алгоритма отвечала бы за хранение количества денег «в столбце». При этом алгоритм должен быть достаточно абстрактным для вычисления средневзвешенных величин любых объектов, которые поддерживали арифметические операции. К примеру, с его помощью можно было бы вычислять средневзвешенное календарных дат.

Выходные прошли как обычно — за отдыхом, а в понедельник утром босс заинтересовался: «Ну как, сможем ли мы это сделать?» — «Дайте мне еще день, и я скажу точно», — ответил Уорд.

В вычислении средневзвешенной величины объект Dollar как бы являлся переменной. В случае наличия нескольких валют потребовалось бы по одной переменной на каждый тип валюты, нечто вроде многочлена. Только вместо $3x^2$ и $4y^3$ — 15 USD и 200 CHF¹.

Быстрый эксперимент показал, что при вычислениях можно работать не с объектом Dollar (доллар), а с более общим объектом — `Currency` (валюта). При этом, если выполнялась операция над двумя различными валютами, значение следовало возвращать в виде объекта `PolyCurrency` (мультивалютный). Сложность заключалась в том, чтобы добавить новую функциональность, не сломав при этом то, что уже работает. А что, если просто прогнать тесты?

После добавления к классу `Currency` нескольких (пока нереализованных) операций большинство тестов все еще успешно выполнялось; к концу дня проходили все тесты. Уорд интегрировал новый код в текущую версию и пошел к боссу. «Мы сможем это сделать», — уверенно сказал он.

Давайте задумаемся над этой историей. Через пару дней потенциальный рынок для системы `WyCash` увеличился в несколько раз, соответственно подскочила и ее ценность. Важно, что возможность создать значительную бизнес-ценность за такое короткое время не была случайной. Свою роль сыграли следующие факторы:

- Метод — Уорду и команде разработки `WyCash` потребовался опыт в пошаговом наращивании проектных возможностей системы, с хорошо отработанным механизмом внесения изменений.

¹ USD — доллары США, CHF — швейцарские франки. — *Примеч. перев.*

- **Мотив** — Уорду и его команде было необходимо четкое представление о значимости поддержки мультивалютности в WyCash, а также потребовалась смелость (взяться за такую, на первый взгляд, безнадежную задачу).
- **Возможность** — сочетание всеохватывающей, продуманной системы тестов и хорошо структурированной программы; язык программирования, обеспечивающий локализацию проектных решений и тем самым упрощающий идентификацию ошибок.

Мотив — это то, чем вы не можете управлять; сложно сказать, когда он у вас появится и заставит заняться техническим творчеством для решения бизнес-задач. Метод и возможность, с другой стороны, находятся под вашим полным контролем. Уорд и его команда создали метод и возможность благодаря таланту, опыту и дисциплине. Значит ли это, что если вы не входите в десятку лучших разработчиков планеты и у вас нет приличного счета в банке (настолько приличного, чтобы попросить босса погулять, пока вы занимаетесь делом), то такие подвиги не для вас?

Нет, вовсе нет. Всегда можно развернуть проект так, чтобы работа над ним стала творческой и интересной, даже если вы обычный разработчик и прогибаетесь под обстоятельствами, когда приходится туго. Test-Driven Development (разработка на основе тестирования), TDD — это набор способов, ведущих к простым программным решениям, которые может применять любой разработчик, а также тестов, придающих уверенность в работе. Если вы гений, эти способы вам не нужны. Если вы тугодум — они вам все равно не помогут. Для всех же нас, кто находится между этими крайностями, следование двум простым правилам поможет работать намного эффективнее:

- перед тем как написать любой фрагмент кода, создайте автоматизированный тест, который поначалу не будет срабатывать;
- устранили дублирование.

Как конкретно следовать этим правилам, какие существуют в данной области нюансы и какова область применимости этих способов — все это составляет тему книги, которую вы сейчас читаете. Вначале мы рассмотрим объект, созданный Уордом в момент вдохновения, — мультивалютные деньги (multi-currency money).



Часть I

На примере денег

Мы займемся реализацией примера, полностью разрабатывая код на основе тестирования (кроме случаев, когда в учебных целях мы будем допускать преднамеренные ошибки). Моя цель — дать вам почувствовать ритм разработки на основе тестирования (Test-Driven Development, TDD). Кратко можно сказать, что TDD заключается в следующем:

1. Быстро создать новый тест.
2. Запустить все тесты и обнаружить, что **новый тест не выполняется**.
3. Внести небольшие изменения.
4. Снова запустить все тесты и **на этот раз зафиксировать, что все они успешно срабатывают**.
5. Провести рефакторинг (refactoring) для устранения дублирования.

Кроме того, приходится находить ответы на следующие вопросы:

- Как добиться того, чтобы каждый тест покрывал небольшое приращение функциональности?
- За счет каких небольших и, наверное, неуклюжих изменений можно обеспечивать успешное прохождение новых тестов?
- Как часто следует запускать тесты?
- Из какого количества микроскопических шагов следует комплектовать рефакторинги?

Мультивалютные деньги

Вначале мы рассмотрим объект, созданный Уордом для системы WyCash, — мультивалютные деньги (см. «Введение»). Допустим, у нас есть отчет вроде этого.

Компания	Число акций	Цена	Всего
IBM	1000	25	25 000
GE	400	100	40 000
		Итого	65 000

Добавив различные валюты, мы получим мультивалютный отчет.

Компания	Число акций	Цена	Всего
IBM	1000	USD	25 000 USD
Novartis	400	150 CHF	60 000 CHF
		Итого	65 000 USD

Также необходимо указать курсы обмена.

Из	В	Курс
CHF	USD	1,5

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 * 2 = \10

Что нам понадобится, чтобы сгенерировать такой отчет? Или, другими словами, какой набор успешно проходящих тестов сможет гарантировать нам, что созданный код правильно генерирует отчет? Нам понадобится:

- выполнять сложение величин в двух различных валютах и конвертировать результат с учетом указанного курса обмена;
- выполнять умножение величин в валюте (стоимость одной акции) на число (количество акций), результатом этой операции должна быть величина в валюте.

Мы составим список задач, который будет напоминать нам о планах, не даст запутаться и покажет, когда все будет готово. В начале работы над задачей мы выделим ее полужирным шрифтом, **вот так**. Закончив работу над ней, мы ее вычеркнем, ~~вот так~~. Задумавшись о написании нового теста, мы добавим новую задачу в наш список.

Как видно из нашего списка задач, сначала мы займемся умножением. Итак, какой объект понадобится нам в первую очередь? Вопрос с подвохом. Мы начнем не с объектов, а с тестов. (Мне приходится постоянно напоминать себе об этом, так что я просто притворюсь, что вы также забывчивы, как и я.)

Попробуем снова. Итак, какой тест нужен нам в первую очередь? Если исходить из списка задач, первый тест представляется довольно сложным. Попробуем начать с малого — умножение, — сложно ли его реализовать? Займемся им для начала.

Когда мы пишем тест, мы воображаем, что у нашей операции идеальный интерфейс. Попробуем представить, как будет выглядеть наша операция снаружи. Конечно, наши представления не всегда будут находить воплощение, но в любом случае стоит начать с наилучшего возможного программного интерфейса (API) и при необходимости вернуться назад, чем сразу делать вещи сложными, уродливыми и «реалистичными».

Простой пример умножения¹:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
}
```

(Знаю, знаю: публичные поля, побочные эффекты, целые числа для денежных величин и все такое. Маленькие шаги — помните? Мы отметим, что где-то есть душок², и продолжим дальше. У нас есть тест, который не выполняется, и мы хотим как можно скорее увидеть зеленую полоску³.)

\$5 + 10 CHF = \$10, если курс обмена 2:1

\$5 * 2 = \$10

Сделать переменную amount закрытым членом класса

Побочные эффекты в классе Dollar?

Округление денежных величин?

Тест, который мы только что создали, даже не компилируется, но это легко исправить. (О том, когда и как создаются тесты, я расскажу позже — когда мы будем

¹ Название метода times() можно перевести на русский как «умножить на». — *Примеч. перев.*

² Код с душком (code that smells) — распространенная в XP метафора, означающая плохой код (содержащий дублирование). — *Примеч. перев.*

³ Имеется в виду индикатор срабатывания тестов в среде JUnit, имеющий форму полосы. Если все тесты успешно работали, полоса становится зеленой. Если хотя бы один тест не сработал, полоса становится красной. — *Примеч. перев.*

подробнее говорить о среде тестирования, JUnit.) Как проще всего заставить тест компилироваться (пусть он пока и не будет срабатывать)? У нас четыре ошибки компиляции:

- нет класса Dollar;
- нет конструктора;
- нет метода times(int);
- нет поля (переменной) amount.

Устраним их, одну за другой. (Я всегда ищу некоторую численную меру прогресса.) От одной ошибки мы избавимся, определив класс Dollar:

```
Dollar
class Dollar
```

Одной ошибкой меньше, осталось еще три. Теперь нам понадобится конструктор, причем совершенно не обязательно, чтобы он что-то делал — лишь бы компилировался.

```
Dollar
Dollar(int amount) {
}
```

Осталось две ошибки. Необходимо создать заглушку для метода times(). Снова мы выполним минимум работы, только чтобы заставить тест компилироваться:

```
Dollar
void times(int multiplier) {
}
```

Теперь осталась только одна ошибка. Чтобы от нее избавиться, нужно создать поле (переменную) amount.

```
Dollar
int amount;
```

Отлично! Теперь мы можем запустить тест и увидеть, что он не выполняется: ситуация продемонстрирована на рис. 1.1.

Загорается зловещий красный индикатор. Среда тестирования (JUnit в нашем случае) выполнила небольшой фрагмент кода, с которого мы начали, и выяснилось, что хотя мы ожидали результат «10» — вместо этого получили «0». Ужасно...

Вовсе нет! Неудача — это тоже прогресс. Теперь у нас есть конкретная мера неудачи. Это лучше, чем просто догадываться, что у нас что-то не так. Наша задача «реализовать мультивалютность» превратилась в «заставить работать этот тест, а потом заставить работать все остальные тесты». Так намного проще и намного меньше поводов для страха. Мы заставим этот тест работать.

Возможно, вам это и не понравится, но сейчас наша цель не получить идеальное решение, а заставить тест выполняться. Мы принесем свою жертву на алтарь истины и совершенства чуть позже.

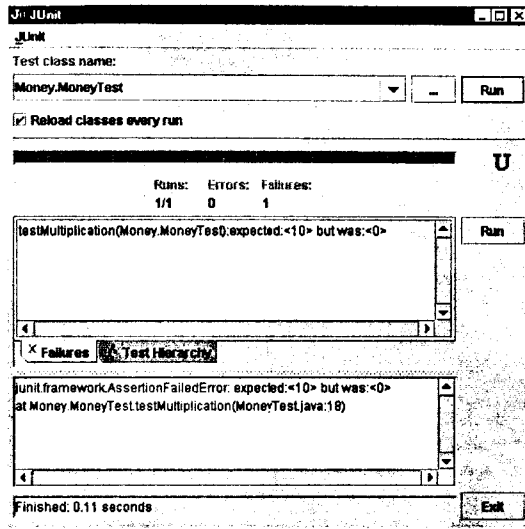


Рис. 1.1. Прогресс! Тест не проходит

Наименьшее изменение, которое заставит тест успешно выполняться, представляется мне таким:

```
Dollar
int amount= 10;
```

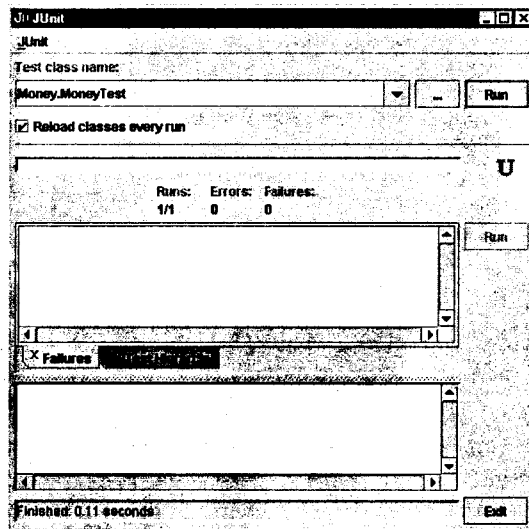


Рис. 1.2. Тест успешно выполняется

Рисунок 1.2 показывает результат повторного запуска теста. Теперь мы видим ту самую зеленую полоску, воспетую в поэмах и прославленную в веках.

Вот оно, счастье! Но радоваться рано, ведь цикл еще не завершен. Уж слишком мал набор входных данных, которые заставят такую странно пахнущую, такую наивную реализацию работать правильно. Перед тем как двигаться дальше, немного поразмышляем.

Вспомним, полный цикл TDD состоит из следующих этапов:

1. Добавить небольшой тест.
2. Запустить все тесты, при этом обнаружить, что что-то не срабатывает.
3. Внести небольшое изменение.
4. Снова запустить тесты и убедиться, что все они успешно выполняются.
5. Устранить дублирование с помощью рефакторинга.

ЗАВИСИМОСТЬ И ДУБЛИРОВАНИЕ

Стив Фримен (Steve Freeman) указал, что проблема с тестами и кодом заключается не в дублировании (на которое я еще не указал вам, но сделаю это, как только закончится отступление). Проблема заключается в зависимости между кодом и тестами — вы не можете изменить одно, не изменяя другого. Наша цель — иметь возможность писать новые осмысленные тесты, не меняя при этом код, что невозможно при нашей текущей реализации.

Зависимость является ключевой проблемой разработки программного обеспечения. Если фрагменты SQL, зависящие от производителя используемой вами базы данных, разбросаны по всему коду и вы хотите поменять производителя, то непременно окажется, что код зависит от этого производителя. Вы не сможете поменять производителя базы данных и при этом не изменить код.

Зависимость является проблемой, а дублирование — ее симптомом. Чаще всего дублирование проявляется в виде дублирования логики — одно и то же выражение появляется в различных частях кода. Объекты — отличный способ абстрагирования, позволяющий избежать данного вида дублирования.

В отличие от большинства проблем в реальной жизни, где устранение симптомов приводит только к тому, что проблема проявляется в более страшной форме где-то еще, устранение дублирования в программах устраняет и зависимость. Именно поэтому существует второе правило TDD. Устраняя дублирование перед тем, как заняться следующим тестом, мы максимизируем наши шансы сделать его успешным, внося всего одно изменение.

Мы выполнили первые четыре пункта цикла, и все готово к устранению дублирования. Но где же оно? Обычно мы замечаем дублирование в нескольких разных фрагментах кода, однако в нашем случае — друг друга дублируют тест и тестируемый код. Еще не видите? Как насчет того, чтобы написать так:

Dollar

```
int amount= 5 * 2;
```

Теперь ясно, откуда мы взяли число 10. Видимо, мы в уме произвели умножение, причем так быстро, что даже не заметили. Произведение «5 умножить на 2» присутствует как в тесте, так и в тестируемом коде. Только изначально в коде оно было представлено в виде константы 10. Сейчас же 5 и 2 отделены друг от друга, и мы должны безжалостно устранить дублирование перед тем, как двинуться дальше. Такие вот правила.

Действия, с помощью которого мы устранили бы 5 и 2 за один шаг, не существует. Но что если мы переместим установку поля (переменной) `amount` из инициализации объекта в метод `times()`?

Dollar

int amount;

```
void times(int multiplier) {
    amount= 5 * 2;
}
```

Тест все еще успешно выполняется, а индикатор остался зеленым. Успех нам пока сопутствует.

Такие шаги кажутся вам слишком мелкими? Помните, TDD не обязывает двигаться только микроскопическими шагами, речь идет о способности совершать эти микроскопические шаги. Буду ли я программировать день за днем такими маленькими шагами? Нет. Но когда дела совсем плохи, я рад возможности выполнять хоть такие шаги. Примените микроскопические шаги к любому собственному примеру. Если вы можете продвигаться достаточно маленькими шагами, вы сумеете делать шаги более крупного, точнее говоря, подходящего размера. Если же вы способны делать только огромные шаги, вы никогда не распознаете ситуацию, в которой более уместны меньшие шаги.

Оставим рассуждения. На чем мы остановились? Ну да, мы избавлялись от дублирования между кодом теста и рабочим кодом. Где мы можем взять 5? Это значение передавалось конструктору, поэтому мы можем сохранить его в переменной `amount`:

Dollar

```
Dollar(int amount) {
    this.amount= amount;
}
```

А если мы сохранили его в переменной `amount`, то сможем использовать это значение в методе `times()`:

Dollar

```
void times(int multiplier) {
    amount= amount * 2;
}
```

Значение параметра `multiplier` равно 2, так что мы подставим параметр вместо константы:

Dollar

```
void times(int multiplier) {
    amount= amount * multiplier;
}
```

Чтобы продемонстрировать, как хорошо мы знаем синтаксис языка Java, мы используем оператор `*=` (который, кстати, уменьшает дублирование):

Dollar

```
void times(int multiplier) {
    amount *= multiplier;
}
```

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

~~$\$5 * 2 = \10~~

Сделать переменную amount закрытым членом класса

Побочные эффекты в классе Dollar?

Округление денежных величин?

Теперь мы можем пометить первый тест как заверченный. Далее мы позаботимся о тех странных побочных эффектах; но сначала давайте подведем итоги. Мы сделали следующее:

- создали список тестов, которые — мы знаем — нам понадобятся;
- с помощью фрагмента кода описали, какой мы хотим видеть нашу операцию;
- временно проигнорировали особенности среды тестирования JUnit;
- заставили тесты компилироваться, написав соответствующие заглушки;
- заставили тесты работать, используя сомнительные приемы;
- слегка улучшили работающий код, заменив константы переменными;
- добавили пункты в наш список задач, вместо того чтобы заняться всеми этими задачами сразу.

Вырождающиеся объекты

Обычный цикл разработки на основе тестирования состоит из следующих этапов:

1. Напишите тест. Представьте, как будет реализована в коде воображаемая вами операция. Продумывая ее интерфейс, опишите все элементы, которые, как вам кажется, понадобятся.
2. Заставьте тест работать. Первоочередная задача — получить зеленую полосу. Если очевидно простое и элегантное решение, создайте его. Если же на реализацию такого решения потребуется время, отложите его. Просто отметьте, что к нему придется вернуться, когда будет решена основная задача — быстро получить зеленый индикатор. Такой подход довольно неприятен для опытных разработчиков (в эстетическом плане), ведь они следуют только правилам хорошей разработки. Но зеленая полоска прощает все грехи, правда, всего лишь на мгновение.
3. Улучшите решение. Теперь, когда система уже работает, избавьтесь от прошлых прегрешений и вернитесь на путь истинной разработки. Удалите дублирование, которое вы внесли, и быстро сделайте так, чтобы полоска снова стала зеленой.

Наша цель — *чистый код, который работает* (отдельное спасибо Рону Джефффризу (Ron Jeffries) за этот слоган). Иногда такой код не по силам даже самым лучшим программистам, и почти всегда он не достижим для большинства программистов (вроде меня). Разделяй и властвуй, приятель, — в этом весь смысл! Сначала мы напишем код, «который работает», после чего создадим «чистый код». Такой подход противоположен модели разработки на основе архитектуры, в котором вы сначала пишете «чистый код», а потом мучаетесь, пытаясь интегрировать в проект код, «который работает».

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

~~$\$5 * 2 = \10~~

Сделать переменную amount закрытым членом класса

Побочные эффекты в классе Dollar?

Округление денежных величин?

В нашем списке один тест, но мы замечаем нечто странное: при выполнении операции с объектом `Dollar` изменяется сам объект. Хотелось бы написать так:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
    five.times(3);
    assertEquals(15, five.amount);
}
```

Я не могу представить простого способа, который заставит этот тест выполняться. После первого вызова метода `times()` пять уже больше не пять — на самом деле это уже десять. Если же метод `times()` будет возвращать новый объект, тогда мы сможем умножать наши исходные пять баксов хоть целый день, и они не изменятся. Реализуя эту идею, нам потребуется изменить интерфейс объекта `Dollar`, поэтому придется изменить и тест. Это нормально, ведь вполне возможно, что наши догадки о правильном интерфейсе не более правдоподобны, чем догадки о правильной реализации.

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product= five.times(2);
    assertEquals(10, product.amount);
    product= five.times(3);
    assertEquals(15, product.amount);
}
```

Новый тест не будет компилироваться, пока мы не изменим объявление метода `Dollar.times()`:

`Dollar`

```
Dollar times(int multiplier) {
    amount *= multiplier;
    return null;
}
```

Теперь тест компилируется, но не срабатывает. И это тоже прогресс! Чтобы заставить его работать, придется возвращать новый объект `Dollar` с правильным значением:

`Dollar`

```
Dollar times(int multiplier) {
    return new Dollar(amount * multiplier);
}
```

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 * 2 = \10

Сделать переменную `amount` закрытым членом класса

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

В главе 1, когда мы заставляли тест работать, мы начали с заглушки и постепенно улучшали код, пока он не стал полноценным кодом. Теперь мы кодировали сразу правильную реализацию и молились, пока выполнялись тесты (довольно короткие молитвы, честно говоря — выполнение тестов занимает миллисекунды). Нам повезло, тесты были успешны, и мы вычеркиваем еще один пункт.

Мне известны три способа для быстрого получения зеленого индикатора. Вот первые два из них:

- подделать реализацию, иначе говоря, создать заглушку (Fake It) — возвращать константу и постепенно заменять константы переменными до тех пор, пока не получим настоящий код;
- использовать очевидную реализацию (Obvious Implementation) — просто написать сразу настоящую реализацию.

При практическом использовании TDD я периодически переключаюсь между двумя этими способами. Когда все идет гладко и я знаю, что делать, — я просто создаю одну за другой очевидные реализации (запуская тесты каждый раз, чтобы убедиться, что то, что очевидно мне, также очевидно и компьютеру). Как только я наткнулась на красный индикатор, я возвращаюсь к методике «поддельная реализация», после чего провожу рефакторинг. Когда уверенность возвращается, я снова использую методику «очевидная реализация».

Есть еще одна, третья методика, «Триангуляция» (Triangulation), которую мы рассмотрим в главе 3. Подведем итоги. Мы выполнили следующее:

- сформулировали дефект проектирования (побочный эффект) в виде теста, который отказал (из-за дефекта);
- создали заглушку, обеспечившую быструю компиляцию кода;
- заставили тест успешно выполняться, написав вроде бы правильный код.

Преобразование чувства (например, отвращения, вызываемого побочными эффектами) в тест (например, двукратное перемножение одного и того же объекта Dollar) — обычная практика в TDD. Чем дольше я этим занимаюсь, тем легче эстетические суждения переводятся в тесты. В результате мои рассуждения о проектировании становятся более интересными. Сначала мы обсуждаем, должна ли система работать так или по-другому. После определения правильного поведения системы можно поговорить о наилучшем способе реализации этого поведения. Мы можем сколь угодно долго рассуждать об истине и совершенстве за пивом, но раз мы занимаемся программированием, у нас есть возможность оставить пустые разговоры и перейти к конкретике.

Равенство для всех

Если у меня есть целое число и я прибавляю к нему 1, то не предполагаю, что изменится исходное число, — в результате я ожидаю получить новое число. Объекты же обычно ведут себя другим образом. К примеру, если у меня есть контракт и я добавлю 1 к его сумме, то это будет означать, что сумма контракта должна измениться (да, несомненно, это пример для обсуждения многих интересных законов бизнеса, которые мы здесь рассматривать не будем).

Мы можем использовать объекты в качестве значений, так же как используем наш объект `Dollar`. Соответствующий паттерн называется `Value Object` (объект-значение). Одно из ограничений этого паттерна заключается в том, что значения атрибутов объекта устанавливаются в конструкторе и никогда в дальнейшем не изменяются.

Значительное преимущество использования паттерна `Value Object` состоит в том, что не нужно беспокоиться о проблеме наложения имен (`aliasing`). Скажем, у меня есть чек и я устанавливаю его сумму — \$5, а затем присваиваю эти же \$5 сумме другого чека. Одна из самых неприятных проблем на моей памяти заключалась в том, что изменение величины первого чека может приводить к непреднамеренному изменению величины второго. Это и есть проблема наложения имен (`aliasing`).

Когда вы используете `Value Objects`, не нужно беспокоиться о наложении имен (`aliasing`). Если у меня есть пять долларов (\$5), то они всегда гарантированно будут оставаться именно пятью долларами (\$5). Если вдруг кому-то понадобится \$7, тогда придется создавать новый объект.

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 * 2 = \10

Сделать переменную `amount` закрытым членом класса

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

Одно из следствий использования `Value Objects` заключается в том, что все операции должны возвращать результаты в виде новых объектов, о чем было расска-

зано в главе 2. Другое следствие заключается в том, что Value Objects должны реализовывать equals(), операцию проверки равенства, потому что одни \$5 ничем не отличаются от других.

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 * 2 = \10

Сделать переменную "amount" закрытым членом

Побочные эффекты в классе Dollar?

Округление денежных величин?

equals()

hashCode()

Кроме того, если использовать Dollar в качестве ключа хэш-таблицы, то, реализовывая equals(), придется реализовать и hashCode(). Добавим этот пункт в список задач и вернемся к нему, когда это будет необходимо.

Вы ведь не собираетесь немедленно приступить к реализации метода equals()? Отлично, я тоже об этом не думаю. Ударив себя линейкой по руке, я стал размышлять над тем, как протестировать равенство. Для начала \$5 должны быть равны \$5:

```
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
}
```

Полоска окрасилась красным. Для возврата значения true достаточно заглушки:

```
Dollar
public boolean equals(Object object) {
    return true;
}
```

Конечно, мы с вами знаем, что true на самом деле это "5==5", что, в свою очередь, означает "amount==5", что соответствует "amount==dollar.amount". Но если бы я сразу проследил все эти шаги, я не смог бы продемонстрировать третью и наиболее консервативную методику реализации — триангуляцию (Triangulation).

Если две станции слежения, находящиеся на известном расстоянии друг от друга, могут измерять азимут некоторого радиосигнала (брать пеленг), то этого вполне достаточно, чтобы вычислить удаление и азимут источника радиосигнала (как вы помните из курса тригонометрии, в отличие от меня). Это вычисление и называется триангуляцией.

По аналогии, когда мы выполняем триангуляцию, мы обобщаем код только в том случае, когда у нас два примера или больше. При этом мы ненадолго игнорируем дублирование между тестом и самим кодом (приложения). Когда второй пример потребует более общего решения, тогда и только тогда мы выполним обобщение.

Итак, для триангуляции нам понадобится второй пример. Как насчет того, чтобы проверить $\$5 \neq \6 ?

```
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
}
```

```
    assertFalse(new Dollar(5).equals(new Dollar(6)));
}
```

Теперь необходимо обобщить равенство (equality):

Dollar

```
public boolean equals(Object object) {
    Dollar dollar=(Dollar) object;
    return amount == dollar.amount;
}
```

\$5 + 10 CHF = \$10, если курс обмена 2:1

~~\$5 * 2 = \$10~~

Сделать переменную amount закрытым членом класса

Побочные эффекты в классе Dollar?

Округление денежных величин?

equals()

hashCode()

Мы могли бы использовать триангуляцию и для управления обобщением метода times(). Если бы у нас были примеры $\$5 * 2 = \10 и $\$5 * 3 = \15 , нам не удалось бы просто возвращать константу.

Думаю, триангуляция — довольно интересная вещь. Я использую ее в случае, если не знаю, как выполнять рефакторинг. Если же я представляю, как устранить дублирование между кодом и тестами и создать более общее решение, я просто создаю его. С какой стати я должен создавать еще один тест, если я сразу могу выполнить обобщение?

Однако когда мысли о проектировании не приходят в голову, триангуляция дает шанс посмотреть на проблему с другой стороны. Сколько степеней свободы вы хотите поддерживать в вашем приложении (какую степень универсальности, другими словами)? Просто попробуйте ввести некоторые из них, и, возможно, ответ станет очевиднее.

\$5 + 10 CHF = \$10, если курс обмена 2:1

~~\$5 * 2 = \$10~~

Сделать переменную amount закрытым членом класса

Побочные эффекты в классе Dollar?

Округление денежных величин?

equals()

hashCode()

Равенство значению null

Равенство объектов

Итак, сейчас операция «равенство» (equality) реализована полностью. Но как учесть сравнение со значением null и сравнение с другими объектами? Это часто используемые операции, пока они нам еще не нужны, поэтому мы просто добавим их в список задач.

Теперь, когда у нас есть операция равенства, можно напрямую сравнивать объекты Dollar. Это позволит нам сделать переменную amount закрытой, какой

и должна быть хорошая экземплярная переменная. Резюмируя все вышесказанное, мы:

- поняли, что для использования паттерна проектирования Value Object необходимо реализовать операцию проверки равенства;
- создали тест для этой операции;
- реализовали ее простейшим способом;
- продолжили тестирование (вместо того, чтобы сразу приступить к рефакторингу);
- выполнили рефакторинг так, чтобы охватить оба теста сразу.

Данные должны быть закрытыми

\$5 + 10 CHF = \$10, если курс обмена 2:1

~~\$5 * 2 = \$10~~

Сделать переменную amount закрытым членом класса

Побочные эффекты в классе Dollar?

Округление денежных величин?

equals()

hashCode()

Равенство значению null

Равенство объектов

Теперь, когда определено равенство (equality), с его помощью мы можем повысить наглядность наших тестов. По идее, метод `Dollar.times()` должен возвращать новый объект `Dollar`, величина которого равна величине исходного объекта (метод которого мы вызываем), умноженной на коэффициент. Однако наш тест не показывает этого явно:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product= five.times(2);
    assertEquals(10, product.amount);
    product= five.times(3);
    assertEquals(15, product.amount);
}
```

Мы можем переписать первую проверку (assert-операцию) так, чтобы сравнивать между собой объекты `Dollar`:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product=five.times(2);
    assertEquals(new Dollar(10), product);
    product= five.times(3);
    assertEquals(15, product.amount);
}
```

Выглядит неплохо, поэтому перепишем и вторую проверку:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product=five.times(2);
    assertEquals(new Dollar(10), product);
    product= five.times(3);
    assertEquals(new Dollar(15), product);
}
```

Теперь нам уже не нужна вспомогательная переменная `product`, поэтому удалим ее:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

Согласитесь, этот вариант теста значительно нагляднее.

Учтем внесенные изменения. Теперь только класс `Dollar` использует экземплярную переменную `amount`, поэтому мы можем сделать ее закрытой:

`Dollar`

```
private int amount;
```

\$5 + 10 CHF = \$10, если курс обмена 2:1

~~\$5 * 2 = \$10~~

Сделать переменную `amount` закрытым членом класса

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

`hashCode()`

Равенство значению `null`

Равенство объектов

Вычеркиваем еще один пункт из списка задач. Заметьте, мы подвергли себя риску: если тест для равенства не смог бы точно определить корректность операции сравнения, тогда и тест для умножения не смог бы проверить, правильно ли оно работает. В TDD принято активное управление риском. Мы не гонимся за совершенством. Выражая все двумя способами — тестами и кодом, — мы надеемся уменьшить дефекты настолько, чтобы уверенно идти дальше. Время от времени наши рассуждения будут нас подводить, позволяя появляться ошибкам. Когда это случится, мы вспомним урок о том, что надо написать тест и двигаться дальше. Все остальное время мы отважно продвигаемся вперед под победно развевающейся зеленой полоской нашего индикатора (вообще-то мой индикатор не развеивается, но я люблю помечтать).

Подводя итоги, мы:

- использовали только что разработанную функциональность для улучшения теста;

- заметили, что если два теста проваливаются одновременно, то наши дела плохи;
- продолжили, несмотря на риск;
- использовали новую функциональность тестируемого объекта для уменьшения зависимости между тестами и кодом.

Поговорим о франках

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

~~$\$5 * 2 = \10~~

Сделать переменную `amount` закрытым (`private`) членом

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

`hashCode()`

Равенство значению `null`

Равенство объектов

$5 \text{ CHF} * 2 = 10 \text{ CHF}$

Можем ли мы приступить к реализации первого, самого интересного теста в данном списке? Мне все еще кажется, что это будет слишком большой шаг. Я не представляю себе, каким образом можно написать этот тест за один маленький шагочок. Мне кажется, что вначале необходимо создать объект наподобие `Dollar`, который соответствовал бы не долларам, а франкам. Пусть это будет объект с названием `Franc`. Для начала объект `Franc` может функционировать в точности так же, как и объект `Dollar`, — если у нас будет такой объект, нам будет проще размышлять о реализации теста, связанного со смешанным сложением двух разных валют.

А если объект `Franc` работает так же, как и объект `Dollar`, значит, мы можем просто скопировать и слегка отредактировать тест для объекта `Dollar`:

```
public void testFrancMultiplication() {
    Franc five= new Franc(5);
    assertEquals(new Franc(10), five.times(2));
    assertEquals(new Franc(15), five.times(3));
}
```

(Хорошо, что в главе 4 нам удалось упростить разработанный нами тест для `Dollar`. Благодаря этому работа по редактированию теста существенно упростилась. Похоже, в данной книге дела идут довольно гладко, однако я не могу гарантировать вам, что в будущем все будет так же хорошо.)

Теперь нам надо получить зеленую полоску. Какой способ будет самым простым? Проще всего скопировать код класса Dollar и заменить Dollar на Franc.

Стоп. Подождите-ка. Я уже вижу, как некоторые наиболее яркие сторонники правильных подходов начинают морщиться и плевать. Повторное использование кода путем его дублирования через буфер обмена (copy-and-paste)? Пренебрежение абстракцией? А как же все эти разговоры об основополагающих принципах ООП и чистом дизайне?

Если вам не по себе, глубоко вздохните через нос, досчитайте до трех и медленно выдохните через рот. Вам лучше? Теперь вспомните, что наш цикл состоит из пяти этапов. Иногда последовательное выполнение всех этапов занимает у нас всего несколько секунд, однако в любом случае мы обязательно выполняем каждый из них:

1. Написать тест.
2. Сделать так, чтобы тест откомпилировался.
3. Запустить тест и убедиться в том, что тест не сработал.
4. Сделать так, чтобы тест сработал.
5. Удалить дублирование.

На разных этапах решаются разные задачи, преследуются разные цели. То, что совершенно недопустимо для одного из этапов, может быть вполне приемлемым для другого этапа. Однако в целом методика TTD работает только в случае, если ни один из этапов не упущен. Если вы пропустите хотя бы одно звено, развалится вся цепочка.

Первые три фазы цикла разработки TTD должны выполняться как можно быстрее. Определяющая характеристика этих этапов — скорость. На этих этапах в жертву скорости можно принести очень многое, в том числе чистоту дизайна.

Честно говоря, сейчас я несколько волнуюсь. Я только что разрешил вам забыть о принципах хорошего дизайна. Представляю себе, как вы приходите к своим коллегам, подчиненным и во всеуслышание объявляете: «Кент сказал, что все эти разговоры про хороший дизайн — полная ерунда!» Остановитесь. Цикл еще не закончен. Четырехногий уродец из благородного семейства пятиногих стульев вечно падает. Первые четыре шага нашего цикла не работают без пятого. Хороший дизайн в подходящее время! Сначала сделаем, чтобы код заработал, потом сделаем, чтобы код был правильным (make it run, make it right).

Теперь мне стало легче. Теперь я уверен, что до тех пор, пока вы не избавитесь от дублирования, вы не покажете свой код никому за исключением своего партнера по паре. На чем мы остановились? О, конечно. Забываем о принципах хорошего дизайна в угоду скорости (мы будем заниматься искуплением этого греха на протяжении нескольких следующих глав).

Franc

```
class Franc {
    private int amount;

    Franc(int amount) {
```

```

    this.amount= amount;
}

Franc times(int multiplier) {
    return new Franc(amount * multiplier);
}

public boolean equals(Object object) {
    Franc franc= (Franc) object;
    return amount == franc.amount;
}
}

```

\$5 + 10 CHF = \$10, если курс обмена 2:1

~~\$5 * 2 = \$10~~

Сделать переменную amount закрытым (private) членом

Побочные эффекты в классе Dollar?

Округление денежных величин?

equals()

hashCode()

Равенство значению null

Равенство объектов

~~5 CHF * 2 = 10 CHF~~

Дублирование Dollar/Franc

Общие операции equals()

Общие операции times()

Чтобы запустить код, нам не потребовалось прикладывать каких-либо усилий, поэтому мы смогли «перепрыгнуть» через этап «добейтесь, чтобы код откомпилировался» (Make it compile).

Зато теперь в нашем коде целое море дублирования. Прежде чем приступить к разработке следующего теста, мы должны избавиться от дублирования. Думаю, что следует начать с обобщения метода equals(). Однако об этом в следующей главе. На текущий момент мы можем вычеркнуть из нашего списка еще один пункт, однако вместе с этим нам придется добавить в него два дополнительных пункта.

В данной главе мы:

- решили отказаться от создания слишком большого теста и вместо этого создали маленький тест, чтобы обеспечить быстрый прогресс;
- создали код теста путем бесстыдного копирования и редактирования;
- хуже того, добились срабатывания теста путем копирования и редактирования разработанного ранее кода;
- дали себе обещание ни в коем случае не уходить домой до тех пор, пока не избавимся от дублирования.

Равенство для всех, вторая серия

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 * 2 = \10

Сделать переменную `amount` закрытым (`private`) членом

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

`hashCode()`

Равенство значению `null`

Равенство объектов

$5 \text{ CHF} * 2 = 10 \text{ CHF}$

Дублирование `Dollar/Franc`

Общие операции `equals()`

Общие операции `times()`

В книге *Crossing to Safety* автор Вэлленц Стегнер (Wallance Stegner) описывает рабочее место одного из персонажей. Каждый инструмент находится на предназначенном для него месте, пол чисто вымыт и подметен, повсюду превосходный порядок и чистота. Однако для того, чтобы добиться подобного положения вещей, персонаж не делал никаких специальных подготовительных процедур. «Подготовка была делом всей его жизни. Он подготавливался, затем убирался на рабочем месте.» (Конец книги заставил меня громко рассмеяться в бизнес-классе трансатлантического «Боинга-747». Так что если решите ее прочитать, читайте с осторожностью.)

В главе 5 мы добились срабатывания теста. Это значит, что требуемая функциональность реализована. Однако чтобы сделать это быстро, нам пришлось продублировать огромный объем кода. Теперь пришло время убрать за собой.

Мы можем сделать так, чтобы один из разработанных нами классов стал производным от другого. Я попробовал сделать это, однако понял, что в этом случае ничего не выиграю. Вместо этого удобнее создать суперкласс, который станет базовым классом для обоих разработанных нами классов. Ситуация проиллюстрирована на рис. 6.1. (Я уже пробовал так поступить и пришел к выводу, что это именно то, что нужно, однако придется приложить усилия.)

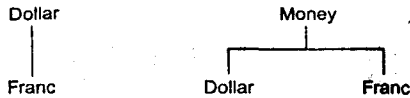


Рис. 6.1. Общий суперкласс для двух разработанных нами классов

Для начала попробуем реализовать в базовом классе `Money` общий для обоих производных классов код метода `equals()`. Начнем с малого:

```
Money
class Money
```

Запустим тесты, — они по-прежнему выполняются. Конечно же, мы пока не сделали ничего такого, что нарушило бы выполнение наших тестов, однако в любом случае лишний раз запустить тесты не помешает. Теперь попробуем сделать класс `Dollar` производным от класса `Money`:

```
Dollar
class Dollar extends Money {
    private int amount;
}
```

Работают ли тесты? Работают. Можем двигаться дальше. Перемещаем переменную `amount` в класс `Money`:

```
Money
class Money {
    protected int amount;
}
```

```
Dollar
class Dollar extends Money {
}
```

Режим видимости переменной `amount` потребовалось изменить: теперь вместо `private` используем модификатор доступа `protected`. В противном случае подкласс не сможет обратиться к этой переменной. (Если бы мы хотели двигаться еще медленнее, мы могли бы на первом шаге объявить переменную в классе `Money`, а на втором шаге удалить ее объявление из класса `Dollar`, однако я решил действовать смело и решительно.)

Теперь можно переместить код метода `equals()` вверх по иерархии классов, то есть в класс `Money`. Прежде всего мы изменим объявление временной переменной:

```
Dollar
public boolean equals(Object object) {
    Money dollar= (Dollar) object;
    return amount == dollar.amount;
}
```

Все тесты по-прежнему срабатывают. Теперь попробуем изменить приведение типа.

Dollar

```
public boolean equals(Object object) {
    Money dollar= (Money) object;
    return amount == dollar.amount;
}
```

Чтобы исходный код получился более осмысленным, изменим имя временной переменной:

Dollar

```
public boolean equals(Object object) {
    Money money= (Money) object;
    return amount == money.amount;
}
```

Теперь перемещаем метод из класса Dollar в класс Money:

Money

```
public boolean equals(Object object) {
    Money money= (Money) object;
    return amount == money.amount;
}
```

Теперь настало время удалить метод `Franc.equals()`. Прежде всего мы обнаруживаем, что у нас до сих пор нет теста, проверяющего равенство двух объектов класса `Franc`, — когда мы, особо не раздумывая, дублировали код класса `Dollar`, мы нагрели еще больше, чем думали. Таким образом, прежде чем модифицировать код, мы должны написать все необходимые тесты.

В ближайшем будущем, скорее всего, вам придется использовать подход TDD в отношении кода, который не обладает достаточным количеством тестов. Если код не обладает адекватным набором тестов, любой рефакторинг может привести к нарушению работоспособности кода. Иными словами, в ходе рефакторинга вы можете допустить ошибку, при этом все имеющиеся тесты будут срабатывать, как ни в чем не бывало. Ошибка может вскрыться слишком поздно, а ее устранение может стоить вам слишком дорого. Что же делать?

Прежде чем что-либо менять в коде, вы должны написать все тесты, которые кажутся вам необходимыми. Если этого не сделать, рано или поздно, выполняя рефакторинг, вы чего-нибудь поломаете. Код перестанет работать так, как должен. Вы потратите кучу времени на поиск ошибки и сформируете предубеждение против рефакторинга. Если подобный инцидент повториться, вы можете вообще перестать делать рефакторинг. Дизайн начнет деградировать. Вас уволят с работы. От вас уйдет ваша любимая собака. Вы перестанете мыться и чистить зубы. У вас начнется кариес. Чтобы сохранить здоровье вашим зубам, всегда сначала пишите тесты и только после этого выполняйте рефакторинг.

К счастью, в нашем случае написать тесты совсем не сложно. Для этого достаточно скопировать и немножко отредактировать тесты для класса `Dollar`:

```
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
}
```

```

    assertFalse(new Dollar(5).equals(new Dollar(6)));
    assertTrue(new Franc(5).equals(new Franc(5)));
    assertFalse(new Franc(5).equals(new Franc(6)));
}

```

Снова дублирование. Целых две строчки! Этот грех нам тоже придется искупить. Но чуть попозже.

Теперь, когда тесты на месте, мы можем сделать класс `Franc` производным от класса `Money`:

```

Franc
class Franc extends Money {
    private int amount;
}

```

Далее мы можем уничтожить поле `amount` класса `Franc`, так как это значение будет храниться в одноименном поле класса `Money`:

```

Franc
class Franc extends Money {
}

```

Метод `Franc.equals()` выглядит фактически так же, как и метод `Money.equals()`. Если мы сможем сделать эти два метода полностью совпадающими, мы сможем удалить реализацию этого метода в классе `Franc`. При этом смысл нашей программы не изменится. Для начала изменим объявление временной переменной:

```

Franc
public boolean equals(Object object) {
    Money franc= (Franc) object;
    return amount == franc.amount;
}

```

После этого изменим операцию преобразования типа:

```

Franc
public boolean equals(Object object) {
    Money franc= (Money) object;
    return amount == franc.amount;
}

```

Теперь, даже не меняя имя временной переменной, можно видеть, что смысл метода фактически такой же, как и смысл одноименного метода в классе `Money`. Однако для пушей уверенности мы можем все-таки переименовать временную переменную:

```

Franc
public boolean equals(Object object) {
    Money money= (Money) object;
    return amount == money.amount;
}

```

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

~~$\$5 * 2 = \10~~

Сделать переменную `amount` закрытым (`private`) членом

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

`hashCode()`

Равенство значению `null`

Равенство объектам

~~$5 \text{ CHF} * 2 = 10 \text{ CHF}$~~

Дублирование `Dollar/Franc`

Общие операции `equals()`

Общие операции `times()`

Сравнение франков (`Franc`) и долларов (`Dollar`)

Теперь нет никакой разницы между методами `Franc.equals()` и `Money.equals()`, и мы можем удалить избыточную реализацию этого метода в классе `Franc`. Запускаем тесты. Они срабатывают.

Что должно происходить при сравнении франков и долларов? Мы рассмотрим этот вопрос в главе 7.

В данной главе мы:

- поэтапно переместили общий код из одного класса (`Dollar`) в суперкласс (`Money`);
- сделали второй класс (`Franc`) подклассом общего суперкласса (`Money`);
- унифицировали две реализации метода `equals()` и удалили избыточную реализацию в классе `Franc`.

Яблоки и апельсины

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

~~$\$5 * 2 = \10~~

Сделать переменную `amount` закрытым (`private`) членом

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

`hashCode()`

Равенство значению `null`

Равенство объектов

~~$5 \text{ CHF} * 2 = 10 \text{ CHF}$~~

Дублирование `Dollar/Franc`

Общие операции `equals()`

Общие операции `times()`

Сравнение франков (`Franc`) и долларов (`Dollar`)

В конце главы 6 перед нами встал интересный вопрос: что будет, если мы сравним франки и доллары? Мы немедленно добавили соответствующий пункт в список предстоящих задач. Нам никак не избавиться от этой мысли. И в самом деле, что произойдет?

```
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
    assertFalse(new Dollar(5).equals(new Dollar(6)));
    assertTrue(new Franc(5).equals(new Franc(5)));
    assertFalse(new Franc(5).equals(new Franc(6)));
    assertFalse(new Franc(5).equals(new Dollar(5)));
}
```

Тест не срабатывает. С точки зрения написанного нами кода доллары — это франки. Прежде чем у наших швейцарских клиентов глаза на лоб полезут, давайте попробуем исправить код. Код сравнения двух денежных значений должен убедиться в том, что он не сравнивает доллары с франками. Чтобы убедиться в этом, мы должны проанализировать классы двух сравниваемых объектов, — два объекта класса `Money` считаются равными только в том случае, если у них равны значения `amount` и классы.

Money

```
public boolean equals(Object object) {
    Money money= (Money) object;
    return amount == money.amount
        && getClass().equals(money.getClass());
}
```

Использование классов подобным образом в разрабатываемом нами коде, по правде сказать, отдает неприятным запахом. Было бы лучше, если бы мы воспользовались критерием, который обладает смыслом в области финансов, а не в области объектов языка Java. Однако на текущий момент в нашей программе еще нет ничего, что соответствовало бы финансовому понятию «валюта», и пока я не вижу достаточно весомой причины для того, чтобы вводить в программу подобное понятие. Таким образом, пока что мы оставим код таким, какой он есть.

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 * 2 = \10

Сделать переменную `amount` закрытым (`private`) членом

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

`hashCode()`

Равенство значению `null`

Равенство объектов

$5 \text{ CHF} * 2 = 10 \text{ CHF}$

Дублирование `Dollar/Franc`

Общие операции `equals()`

Общие операции `times()`

Сравнение франков (`Franc`) и долларов (`Dollar`)

Валюта?

Теперь пришла пора избавиться от дублирующегося кода методов `times()`, после этого мы могли бы перейти к реализации смешанной валютной арифметики. Однако прежде чем продвинуться дальше, подведем итоги данной главы:

- мы превратили мучающее нас сомнение в тест;
- добились срабатывания теста приемлемым, но не идеальным способом, — `getClass()`;
- решили не добавлять в программу какой-либо **дополнительный дизайн** до тех пор, пока у нас не появится более весомая мотивация.

Создание объектов

\$5 + 10 CHF = \$10, если курс обмена 2:1

\$5 * 2 = \$10

Сделать переменную amount закрытым (private) членом

Побочные эффекты в классе Dollar?

Округление денежных величин?

equals()

hashCode()

Равенство значению null

Равенство объектов

5 CHF * 2 = 10 CHF

Дублирование Dollar/Franc

Общие операции equals()

Общие операции times()

Сравнение франков (Franc) и долларов (Dollar)

Валюта?

Две разные реализации метода times() выглядят на удивление похоже:

Franc

```
Franc times(int multiplier) {  
    return new Franc(amount * multiplier)  
}
```

Dollar

```
Dollar times(int multiplier) {  
    return new Dollar(amount * multiplier)  
}
```

Мы можем сделать их еще более похожими, для этого изменим тип возвращаемого значения на Money:

Franc

```
Money times(int multiplier) {  
    return new Franc(amount * multiplier)  
}
```

Dollar

```
Money times(int multiplier) {
    return new Dollar(amount * multiplier)
}
```

Следующий шаг вперед неочевиден. Два подкласса, являющихся производными по отношению к классу Money, мало чем отличаются друг от друга. Возникает желание избавиться от них. Однако мы не можем сделать это за один большой шаг, так как это нельзя будет назвать наглядной демонстрацией методики TDD.

Но что же делать? Полагаю, мы сможем приблизиться к решению задачи об уничтожении подклассов, если избавимся от прямых ссылок на подклассы. Для этого мы можем добавить в класс Money фабричный метод (Factory Method), который возвращал бы объект класса Dollar. Этот метод можно было бы использовать следующим образом:

```
public void testMultiplication() {
    Dollar five= Money.dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

Реализация этого метода создает объект класса Dollar и возвращает его:

Money

```
static Dollar dollar(int amount) {
    return new Dollar(amount);
}
```

Однако мы хотим избавиться от ссылок на Dollar, поэтому в коде теста мы должны изменить объявление объектной переменной:

```
public void testMultiplication() {
    Money five= Money.dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

Компилятор вежливо сообщает нам, что метод times() в классе Money не определен. На текущий момент мы не можем реализовать его, поэтому делаем класс Money абстрактным (может быть с этого стоило начать?) и объявляем абстрактный метод Money.times():

Money

```
abstract class Money
    abstract Money times(int multiplier);
```

Теперь мы можем изменить объявление фабричного метода:

Money

```
static Money dollar(int amount) {
    return new Dollar(amount);
}
```

Все тесты срабатывают, стало быть, по крайней мере, мы ничего не сломали. Теперь мы можем использовать фабричный метод повсюду в тестах:

```
public void testMultiplication() {
    Money five= Money.dollar(5);
    assertEquals(Money.dollar(10), five.times(2));
    assertEquals(Money.dollar(15), five.times(3));
}

public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(new Franc(5).equals(new Franc(5)));
    assertFalse(new Franc(5).equals(new Franc(6)));
    assertFalse(new Franc(5).equals(Money.dollar(5)));
}
```

Теперь мы находимся в несколько более выгодной позиции, чем раньше. Клиентский код ничего не знает о существовании подкласса **Dollar**. Освободив код тестов от ссылок на подклассы, мы получили возможность изменять структуру наследования, не внося при этом каких-либо изменений в клиентский код.

Прежде чем машинально редактировать код теста `testFrancMultiplication()`, обратите внимание, что теперь он не тестирует какой-либо иной логики за исключением той, которая уже протестирована функцией `testMultiplication()`. Напрашивается вопрос: нужна ли нам функция `testFrancMultiplication()`? Если мы уничтожим этот тест, потеряем ли мы уверенность в нашем коде? Похоже что нет, однако мы все же сохраним пока этот тест просто так, — на всякий случай.

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(Money.franc(5).equals(Money.franc(5)));
    assertFalse(Money.franc(5).equals(Money.franc(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}

public void testFrancMultiplication() {
    Money five= Money.franc(5);
    assertEquals(Money.franc(10), five.times(2));
    assertEquals(Money.franc(15), five.times(3));
}
```

Реализация метода `Money.franc()` почти такая же, как и реализация метода `Money.dollar()`:

Money

```
static Money franc(int amount) {
    return new Franc(amount);
}
```

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

~~$\$5 * 2 = \10~~

Сделать переменную amount закрытым (private) членом

Побочные эффекты в классе Dollar?

Округление денежных величин?

equals()

hashCode()

Равенство значению null

Равенство объектов

~~$5 \text{ CHF} * 2 = 10 \text{ CHF}$~~

Дублирование Dollar/Franc

~~Общие операции equals()~~

Общие операции times()

Сравнение франков (Franc) и долларов (Dollar)

Валюта?

Нужен ли test testFrancMultiplication()?

Далее мы планируем перейти к удалению дублирования методов times(). А сейчас вспомним, что в данной главе мы:

- сделали шаг на пути к удалению дублирования — сформировали общую сигнатуру для двух вариантов одного метода, — times();
- переместили сигнатуру в общий суперкласс;
- освободили тестовый код от ссылок на производные классы, для этого были созданы фабричные методы;
- обратили внимание на то, что когда подклассы исчезли, некоторые тесты стали избыточными, однако никаких действий предпринято не было.

Потребность в валюте

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 * 2 = \10

Сделать переменную amount закрытым (private) членом

Побочные эффекты в классе Dollar?

Округление денежных величин?

equals()

hashCode()

Равенство значению null

Равенство объектов

$5 \text{ CHF} * 2 = 10 \text{ CHF}$

Дублирование Dollar/Franc

Общие операции equals()

Общие операции times()

Сравнение франков (Franc) и долларов (Dollar)

Валюта?

Нужен ли тест testFrancMultiplication()?

Есть ли в нашем списке задач какой-либо пункт, который помог бы нам избавиться от этих надоедливых подклассов? Что произойдет, если мы попробуем ввести в нашу программу понятие валюты?

Каким образом мы можем реализовать понятие валюты в данный момент? Черт! Опять я говорю ерунду! Вместо того чтобы снова бить себя линейкой по рукам, попробую перефразировать: каким образом мы можем протестировать понятие валюты в данный момент? Слава богу, мои руки спасены.

Возможно, в будущем нам захочется создать специальный класс валюты, а также воспользоваться паттерном Flyweight Factory (Приспособленец) для того, чтобы избежать создания излишних объектов. Однако на текущий момент понятие валюты вполне можно реализовать в виде обычных строк:

```
public void testCurrency() {
    assertEquals("USD", Money.dollar(1).currency());
    assertEquals("CHF", Money.franc(1).currency());
}
```

Прежде всего объявим метод `currency()` в классе `Money`:

`Money`

```
abstract String currency();
```

Теперь реализуем этот метод в обоих подклассах:

`Franc`

```
String currency() {
    return "CHF";
}
```

`Dollar`

```
String currency() {
    return "USD";
}
```

Однако нам хотелось бы иметь одну и ту же реализацию для обоих подклассов, поэтому мы можем хранить идентификатор валюты в специальном поле класса и просто возвращать значение этой переменной в методе `currency()`. (Я начинаю делать рефакторинг быстрее, чтобы сэкономить время. Если вам покажется, что я рассказываю слишком быстро, не стесняйтесь попросить меня замедлиться. Постойте-ка, это же книга, — наверное, мне не стоит слишком уж ускоряться.)

`Franc`

```
private String currency;
Franc(int amount) {
    this.amount = amount;
    currency = "CHF";
}
String currency() {
    return currency;
}
```

То же самое можно сделать и с классом `Dollar`:

`Dollar`

```
private String currency;
Dollar(int amount) {
    this.amount = amount;
    currency = "USD";
}
String currency() {
    return currency;
}
```

Теперь мы можем переместить объявление переменной и реализацию метода `currency()` в базовый класс `Money`, так как в обоих подклассах они одинаковые:

`Money`

```
protected String currency;
String currency() {
```

```

return currency;
}

```

Если мы переместим константные строки "USD" и "CHF" в статические фабричные методы, два конструктора двух производных классов станут идентичными, в результате мы сможем создать общую реализацию.

Вначале добавим к конструктору дополнительный параметр:

Franc

```

Franc(int amount, String currency) {
    this.amount = amount;
    this.currency = "CHF";
}

```

При этом возникают ошибки в двух местах при обращении к конструктору:

Money

```

static Money franc(int amount) {
    return new Franc(amount, null);
}

```

Franc

```

Money times(int multiplier) {
    return new Franc(amount * multiplier, null);
}

```

Постойте-ка! Почему это метод `Franc.times()` обращается к конструктору вместо того, чтобы обращаться к фабричному методу? Будем ли мы заниматься этим сейчас или отложим любые связанные с этим модификации на более позднее время? В рамках догмы мы должны оставить посторонние дела на потом, — не следует прерывать то, чем мы сейчас занимаемся. Однако на практике я иногда позволяю себе отвлечься, но лишь ненадолго, и ни в коем случае я не разрешаю себе прерывать прерывание (этому правилу научил меня Джим Коплаен — Jim Coplien). В данном случае будет лучше, если мы подчистим метод `times()` прежде чем продолжить:

Franc

```

Money times(int multiplier) {
    return Money.franc(amount * multiplier);
}

```

Теперь фабричному методу можно передать значение "CHF":

Money

```

static Money franc(int amount) {
    return new Franc(amount, "CHF");
}

```

Наконец, мы можем присвоить значение параметра полю класса:

Franc

```

Franc(int amount, String currency) {
    this.amount = amount;
}

```



```

    this.currency = currency;
}

```

Может показаться, что я снова перемещаюсь вперед слишком маленькими шажками. Действительно ли я рекомендую вам работать в таком же темпе? Нет. Я рекомендую вам вначале научиться работать в таком темпе, а затем самостоятельно определять скорость работы, которая покажется вам наиболее эффективной. Я всего лишь попробовал двигаться вперед большими шагами и на половине дороги допустил глупую ошибку. Запутавшись, я вернулся назад на несколько минут, перешел на пониженную передачу и сделал работу заново, более мелкими шажками. Сейчас я чувствую себя уверенней, поэтому мы можем попробовать внести такие же изменения в класс Dollar за один большой шаг:

Money

```

static Money dollar(int amount) {
    return new Dollar(amount, "USD");
}

```

Dollar

```

Dollar(int amount, String currency) {
    this.amount = amount;
    this.currency = currency;
}
Money times(int multiplier) {
    return Money.dollar(amount * multiplier);
}

```

И это сработало с первого раза. Классно!

Подобная настройка скорости весьма характерна для TDD. Вам кажется, что слишком маленькие шажки ограничивают вас? Попробуйте двигаться быстрее. Почувствовали неуверенность? Переходите на короткий шаг. TDD — это процесс плавного управления, — немного в одну сторону, немного в другую сторону. Не существует одного-единственного наиболее правильного размера шага, ни сейчас, ни в будущем.

Теперь два конструктора выглядят абсолютно одинаково, и мы можем переместить реализацию в базовый класс:

Money

```

Money(int amount, String currency) {
    this.amount = amount;
    this.currency = currency;
}

```

Franc

```

Franc(int amount, String currency) {
    super(amount, currency);
}

```

Dollar

```

Dollar(int amount, String currency) {
    super(amount, currency);
}

```

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

~~$\$5 * 2 = \10~~

Сделать переменную `amount` закрытым (`private`) членом

Любознательные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

`hashCode()`

Равенство значению `null`

Равенство объектов

~~$5 \text{ CHF} * 2 = 10 \text{ CHF}$~~

Дублирование `Dollar/Franc`

Общие операции `equals()`

Общие операции `times()`

Сравнение франков (`Franc`) и долларов (`Dollar`)

Валюта?

Нужен ли тест `testFrancMultiplication()`?

Мы уже почти готовы переместить реализацию `times()` в базовый класс. Однако сейчас вспомним, что в данной главе мы:

- на некоторое время заблудились в крупномасштабных идеях дизайнера и, чтобы разобраться в проблеме, решили начать с решения небольшой задачи, на которую мы уже обратили внимание ранее;
- сделали одинаковыми два конструктора, для этого переместили отличающийся код в вызывающий метод (фабричный метод);
- на короткое время отвлеклись от рефакторинга для того, чтобы добавить в метод `times()` обращение к фабричному методу;
- выполнили аналогичный рефакторинг в отношении класса `Dollar` за один большой шаг;
- получили два абсолютно идентичных конструктора и переместили код в базовый класс.

Избавление от двух разных версий times()

\$5 + 10 CHF = \$10, если курс обмена 2:1

~~\$5 * 2 = \$10~~

Сделать переменную amount закрытым (private) членом

Побочные эффекты в классе Dollar?

Округление денежных величин?

equals()

hashCode()

Равенство значению null

Равенство объектов

~~5 CHF * 2 = 10 CHF~~

Дублирование Dollar/Franc

Общие операции equals()

Общие операции times()

Сравнение франков (Franc) и долларов (Dollar)

Валюта?

Нужен ли тест testFrancMultiplication()?

В конце данной главы мы должны получить единый класс Money, соответствующий понятию «деньги». Две реализации метода times() близки друг к другу, однако они не идентичны:

Franc

```
Money times(int multiplier) {  
    return Money.franc(amount * multiplier);  
}
```

Dollar

```
Money times(int multiplier) {  
    return Money.dollar(amount * multiplier);  
}
```

Увы, я не вижу простого способа добиться идентичности этих методов, однако в некоторых ситуациях, для того чтобы продвинуться дальше, требуется вернуться немного назад, — это напоминает кубик Рубика. Что будет, если мы заменим фаб-

ричные методы обращениями к оператору new? (Я отлично понимаю, что совсем недавно мы выполняли обратную процедуру, — заменяли new на обращение к фабричному методу. Однако что я могу поделать, — сейчас мы решаем несколько иную задачу. Понимаю, что это может показаться обескураживающим, однако потерпите немного.)

Franc

```
Money times(int multiplier) {
    return new Franc(amount * multiplier, "CHF");
}
```

Dollar

```
Money times(int multiplier) {
    return new Dollar(amount * multiplier, "USD");
}
```

Мы абсолютно уверены, что для класса Franc значение поля currency всегда будет равно "CHF", поэтому мы можем написать:

Franc

```
Money times(int multiplier) {
    return new Franc(amount * multiplier, currency);
}
```

Сработало! Теперь тот же трюк можно сделать и в отношении класса Dollar:

Dollar

```
Money times(int multiplier) {
    return new Dollar(amount * multiplier, currency);
}
```

Мы почти закончили. Имеет ли значение, используем мы в данном случае Franc или Money? Об этом можно рассуждать в течение некоторого времени, исходя из имеющихся знаний о внутреннем устройстве нашей системы, однако у нас есть чистый код и тесты, которые дают нам уверенность в том, что код работает так, как надо. Вместо того чтобы тратить несколько минут на рассуждения, мы можем спросить об этом компьютер. Для этого достаточно внести интересующие нас изменения в код и запустить тесты. Обучая методике TDD, я наблюдаю подобную ситуацию постоянно, — опытные умные программисты тратят от 5 до 10 минут на обсуждение вопроса, на который компьютер может дать ответ в течение 15 секунд. Если у вас нет тестов, вам остается только размышлять и предполагать. Если же у вас есть тесты, вместо того, чтобы напрасно тратить время, вы можете провести быстрый эксперимент. Как правило, если у вас есть тесты, быстрее спросить компьютер.

Чтобы провести интересующий нас эксперимент, мы модифицируем код так, чтобы метод Franc.times() возвращал значение типа Money:

Franc

```
Money times(int multiplier) {
    return new Money (amount * multiplier, currency);
}
```

Компилятор сообщает нам, что класс Money должен быть конкретным (не абстрактным) классом:

Money

```
class Money
Money times(int amount) {
    return null;
}
```

Получаем красную полоску. Сообщение об ошибке выглядит следующим образом: `expected:<Money.Franc@31aebf> but was:<Money.Money@478a43>`. Не очень-то информативно. Не так информативно, как нам хотелось бы. Чтобы получить более осмысленное сообщение об ошибке, добавляем в класс метод `toString()`:

Money

```
public String toString() {
    return amount + " " + currency;
}
```

О, ужас! Код без тестов?! Допустимо ли такое? Конечно же, прежде чем писать код метода `toString()`, мы могли бы написать соответствующий тест, однако:

- мы увидим результаты работы этого метода на экране;
- метод `toString()` используется только для отладки, поэтому риск, связанный с потенциальными ошибками, невелик;
- перед нами красная полоса, и мы предпочитаем не писать новых тестов до тех пор, пока не избавимся от красной полосы.

Обстоятельства приняты к сведению.

Теперь сообщение об ошибке выглядит следующим образом: `expected:<10 CHF> but was:<10 CHF>`. Выглядит осмысленней, однако сбивает с толку. В двух объектах хранятся одни и те же данные, однако при этом объекты не считаются равными. Причина проблемы кроется в реализации метода `equals()`:

Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount
        && getClass().equals(money.getClass());
}
```

В данном случае происходит сравнение имен классов, в то время как логичнее было бы сравнивать идентификаторы валют.

Лучше не писать никаких новых тестов, если перед вами красная полоса. Однако нам нужно внести изменения в разрабатываемый код, и мы не можем изменить код, не обладая соответствующим тестом. Консервативный подход заключается в том, чтобы отменить изменение, которое привело к появлению красной полосы. В этом случае мы вновь получим зеленую полоску. После этого мы сможем модифицировать тест для метода `equals()`, исправить его реализацию и вновь применить изначальное изменение.

В данном случае мы будем действовать консервативно. (Иногда я плюю на все и пишу тест, не обращая внимания на красную полосу, однако я поступаю так только убедившись в том, что дети уже спят.)

Franc

```
Money times(int multiplier) {
    return new Franc (amount * multiplier, currency);
}
```

Перед нами снова зеленая полоса. Мы попали в ситуацию, когда объект Franc(10,"CHF") не равен объекту Money(10,"CHF"), хотя нам бы хотелось, чтобы эти объекты были равны. Превращаем наше желание в тест:

```
public void testDifferentClassEquality() {
    assertTrue(new Money(10, "CHF").equals(new Franc(10, "CHF")));
}
```

Как и ожидалось, тест не сработал. Код метода equal() должен сравнивать идентификаторы валют, а не идентификаторы классов:

Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount
        && currency().equals(money.currency());
}
```

Теперь метод Franc.times() может возвращать значение Money, и при этом все тесты будут по-прежнему срабатывать:

Franc

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

Сработает ли этот трюк для метода Dollar.times()?

Dollar

```
Money times(int multiplier) {
    return new Money (amount * multiplier, currency);
}
```

Да! Теперь две реализации абсолютно идентичны, и мы можем переместить их в базовый класс.

Money

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 * 2 = \10

Сделать переменную `amount` закрытым (`private`) членом

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

`hashCode()`

Равенство значению `null`

Равенство объектов

$5 \text{ CHF} * 2 = 10 \text{ CHF}$

Дублирование `Dollar/Franc`

Общие операции `equals()`

Общие операции `times()`

Сравнение франков (`Franc`) и долларов (`Dollar`)

Валюта?

Нужен ли тест `testFrancMultiplication()`?

Метод умножения там, где ему следует быть, теперь мы готовы удалить ненужные нам производные классы.

В данной главе мы:

- сделали идентичными две реализации метода `times()`, для этого мы перенесли в тело метода код методов, к которым он обращался (`Inline Method`), и заменили константы переменными;
- добавили в класс отладочный метод `toString()` без теста;
- попробовали модифицировать код (заменяли тип возвращаемого значения на `Money` вместо `Franc`) и обратились к тестам для того, чтобы узнать, работает ли это;
- отменили модификацию и написали еще один тест, заставили тест работать и вновь применили модификацию.

Корень всего зла

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 * 2 = \10

Сделать переменную `amount` закрытым (`private`) членом

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

`hashCode()`

Равенство значению `null`

Равенство объектов

$5 \text{ CHF} * 2 = 10 \text{ CHF}$

Дублирование `Dollar/Franc`

Общие операции `equals()`

Общие операции `times()`

Сравнение франков (`Franc`) и долларов (`Dollar`)

Валюта?

Нужен ли тест `testFrancMultiplication()`?

Два производных класса, `Dollar` и `Franc`, обладают только конструкторами, однако конструктор — это недостаточная причина для того, чтобы создавать подкласс. Мы должны избавиться от бесполезных подклассов.

Мы можем изменить ссылки на подклассы ссылками на суперкласс, не меняя при этом смысл кода. Начнем с класса `Franc`:

`Franc`

```
static Money franc(int amount) {  
    return new Money (amount, "CHF");  
}
```

Затем перейдем к классу `Dollar`:

`Dollar`

```
static Money dollar(int amount) {  
    return new Money (amount, "USD");  
}
```


Ссылок на класс `Dollar` больше нет, поэтому мы можем удалить этот класс. Однако в только что написанном нами тесте есть одна ссылка на класс `Franc`:

```
public void testDifferentClassEquality() {
    assertTrue(new Money(10, "CHF").equals(new Franc(10, "CHF")));
}
```

Если равенство объектов достаточно хорошо протестировано другими тестами, значит, мы можем безбоязненно удалить этот тест. Давайте взглянем на другие тесты:

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(Money.franc(5).equals(Money.franc(5)));
    assertFalse(Money.franc(5).equals(Money.franc(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
```

Похоже на то, что все возможные случаи равенства достаточно хорошо протестированы другими тестами. Я бы даже сказал, что тестов слишком много. Мы можем удалить третье и четвертое выражение `assert`, так как они дублируют собой первое и второе выражение:

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
```

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 * 2 = \10

Сделать переменную `amount` закрытым (`private`) членом

Побочные эффекты в классе `Dollar`?

Округление денежных величин?

`equals()`

`hashCode()`

Равенство значению `null`

Равенство объектам

$5 \text{ CHF} * 2 = 10 \text{ CHF}$

Дублирование `Dollar/Franc`

Общие операции `equals()`

Общие операции `times()`

Сравнение франков (`Franc`) и долларов (`Dollar`)

Валюта?

Нужен ли тест `testFrancMultiplication()`?

Тест `testDifferentClassEquality()` служит доказательством того, что, сравнивая объекты, мы сравниваем различные валюты, но не различные классы. Этот тест имеет смысл только в случае, если в программе существует несколько различных классов. Однако мы уже избавились от класса `Dollar` и намерены точно так же из-

бавиться от класса `Franc`. Иными словами, в нашем распоряжении останется только один денежный класс: `Money`. Принимая во внимание наши намерения, тест `testDifferentClassEquality()` оказывается для нас излишней обузой. Мы удаляем его, а затем избавляемся от класса `Franc`.

Мы обращаем внимание также на то, что в программе присутствуют отдельные тесты для тестирования умножения франков и долларов. Если взглянуть в код, можно увидеть, что на текущий момент логика метода, реализующего умножение, не зависит от типа валюты (зависимость была бы только в случае, если бы мы использовали два различных класса). Таким образом, мы можем уничтожить функцию `testFrancMultiplication()`, не опасаясь того, что мы потеряем какую-либо уверенность в правильности работы системы.

Итак, в нашем распоряжении единый денежный класс и мы готовы приступить к реализации сложения.

Подведем итог. В этой главе мы:

- закончили потрошить производные классы и избавились от них;
- удалили тесты, которые имели смысл только при использовании старой структуры кода, однако оказались избыточными при использовании новой структуры кода.

Сложение, наконец-то

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

Наступил новый день, и я заметил, что список задач переполнен вычеркнутыми пунктами. Лучше всего переписать оставшиеся не зачеркнутыми пункты в новый свежий список. (Я люблю физически копировать пункты из старого списка в новый список. Если в старом списке много мелких недоделанных задач, вместо того, чтобы копировать их в новый список, я просто добавляю в программу соответствующий код. В результате из-за моей лени куча мелочей, которая могла бы расти со временем, просто исчезает. Используйте свои слабости.)

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1
 $\$5 + \$5 = \$10$

Пока что я не представляю себе, каким образом можно реализовать смешанное сложение долларов и франков, поэтому предлагаю начать с более простой задачи: $\$5 + \$5 = \$10$.

```
public void testSimpleAddition() {  
    Money sum= Money.dollar(5).plus(Money.dollar(5));  
    assertEquals(Money.dollar(10), sum);  
}
```

Мы могли бы подделать реализацию, просто вернув значение `Money.dollar(10)`, однако в данном случае реализация кажется очевидной. Давайте попробуем:

Money

```
Money plus(Money addend) {  
    return new Money(amount + addend.amount, currency);  
}
```

(Далее я буду ускорять процесс разработки для того, чтобы сэкономить бумагу и сохранить ваш интерес. Там, где дизайн не очевиден, я буду подделывать реализацию и выполнять рефакторинг. Я надеюсь, что благодаря этому вы увидите, каким образом в TDD выполняется контроль над величиной шагов.)

Сказав, что планирую увеличить скорость, я немедленно замедляю процесс разработки. Однако я не планирую замедлять процесс написания кода, который заставляет тесты срывать. Я планирую замедлить процесс написания самих тестов. Некоторые ситуации и некоторые тесты требуют тщательного обдумывания. Каким образом мы планируем представить арифметику со смешанными валютами? Это как раз тот случай, когда требуется тщательное обдумывание.

Наиболее важное и сложное ограничение, с которым нам приходится иметь дело, заключается в том, что мы не хотим, чтобы код нашей системы знал о существовании каких-либо валют. Нам хотелось бы, чтобы система имела дело с деньгами и не зависела от того, в какой валюте представлены эти деньги. Возможная стратегия состоит в том, чтобы немедленно преобразовывать любые денежные значения в некоторую единую стандартную валюту (попробуйте угадать, какая валюта является самой любимой у американских программистов). Однако подобное решение не позволит нам с легкостью варьировать соотношения (курсы обмена) между различными валютами.

Вместо этого мы хотели бы найти решение, которое позволило бы нам в удобной форме реализовать механизм обменных курсов и при этом обеспечить запись арифметических выражений в форме, близкой к стандартной арифметической записи.

Решение основано на объектах. Если имеющийся объект ведет себя не так, как нам бы хотелось, мы создаем еще один объект, обладающий точно таким же внешним протоколом, но отличающейся внутренней реализацией. Такой паттерн называется *Imposter* (Самозванец).

Возможно, многим это покажется хиромантией. Каким образом в данной ситуации можно использовать паттерн *Imposter* (Мошенник)? Однако я не собираюсь шутить над вами — не существует формулы, позволяющей генерировать гениальные дизайнерские решения. Решение проблемы было придумано Уордом Каннигемом десятилетие назад. Я еще не встречал человека, который независимо от Уорда придумал бы нечто подобное. К сожалению, методика TDD не гарантирует вам генерацию гениальных идей. Вместе с тем, благодаря TDD вы имеете тесты, формирующие вашу уверенность в коде, а также тщательно вылизанный код, — все это является хорошей почвой для возникновения идеи и для воплощения этой идеи в реальность.

Итак, что же является решением в нашем случае? Предлагается создать объект, который ведет себя как объект *Money*, однако соответствует сумме двух объектов *Money*. Чтобы объяснить эту идею, я пробовал использовать несколько разных метафор. Например, можно рассматривать сумму различных денежных величин как *бумажник*. В один бумажник можно положить несколько разных банкнот разных валют и разных достоинств.

Еще одна метафора: *выражение* (expression). Имеется в виду математическое выражение, например: $(2 + 3) * 5$. В нашем случае мы имеем дело с денежными величинами, поэтому выражение может быть таким: $(\$2 + 3 CHF) * 5$. Класс *Money* — это атомарная форма выражения. В результате выполнения любых операций над денежными величинами получается объект класса *Expression*. Одним

из таких объектов может быть объект `Sum`¹. После того как операция (например, сложение нескольких значений в разных валютах) выполнена, полученный объект `Expression` может быть приведен к некоторой заданной валюте. Преобразование к некоторой валюте осуществляется на основании набора заданных курсов обмена.

Как выразить эту метафору в виде набора тестов? Прежде всего, мы знаем, к чему мы должны прийти:

```
public void testSimpleAddition() {
    ...
    assertEquals(Money.dollar(10), reduced);
}
```

Переменная `reduced` — это объект класса `Expression`, который создан путем применения обменных курсов в отношении объекта `Expression`, полученного в результате выполнения математической операции. Кто в реальном мире отвечает за применение обменных курсов? *Банк*. Стало быть, было бы неплохо, если бы мы могли написать:

```
public void testSimpleAddition() {
    ...
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

(Плохо, когда в одной программе смешиваются две разные метафоры: *банк* и *математическое выражение*. Однако сейчас предлагаю не заострять на этом внимания. Сначала воплотим в жизнь то, что запланировали, а затем посмотрим, можно ли улучшить нашу систему с литературно-художественной точки зрения.)

Обратите внимание на важное дизайнерское решение: метод `reduce()` принадлежит объекту `bank`. С такой же легкостью мы могли бы написать:

```
...reduced= sum.reduce("USD", bank).
```

Почему ответственным за выполнение операции `reduce()` сделан именно объект `bank`? На самом деле ответ следующий: «Это первое, что пришло мне в голову», однако такой ответ нельзя считать удовлетворительным. Почему мне в голову пришло сделать ответственным за выполнение операции `reduce()` именно объект класса `Bank`, а не объект класса `Expression`? Вот что мне известно на текущий момент:

- Объекты класса `Expression` по всей видимости лежат в самом сердце того, что мы делаем. Я стараюсь делать объекты, являющиеся сердцем системы, как можно менее зависимыми от всего остального мира. Благодаря этому они остаются гибкими в течение длительного времени («гибкие» в данном случае означает «простые для понимания, тестирования и повторного использования»).

¹ В переводе на русский язык *sum* — это *сумма*. — *Примеч. перев.*

- Я могу предположить, что класс `Expression` будет нести ответственность за множество операций. Значит, мы должны по возможности освободить этот класс от лишней ответственности и переложить часть ответственности на другие классы там, где это допустимо. В противном случае класс `Expression` разрастется до неконтролируемых размеров.

Конечно, это всего лишь догадки — этого не достаточно для того, чтобы принимать какие-либо окончательные решения, однако этого вполне достаточно для того, чтобы я начал двигаться в избранном направлении. Безусловно, если выяснится, что наша система вполне может обойтись без класса `Bank`, я переложу ответственность за выполнение метода `reduce()` на класс `Expression`.

Если мы используем объект `bank`, значит, его необходимо создать:

```
public void testSimpleAddition() {
    ...
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

Сумма двух объектов `Money` — это объект класса `Expression`:

```
public void testSimpleAddition() {
    ...
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

Наконец, операция, в которой мы абсолютно уверены, — создание пяти долларов:

```
public void testSimpleAddition() {
    Money five= Money.dollar(5);
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

Что надо сделать для того, чтобы данный код откомпилировался? Для начала создадим интерфейс `Expression` (мы могли бы создать класс, однако интерфейс обладает существенно меньшим весом):

`Expression`

```
interface Expression
```

Метод `Money.plus()` должен возвращать значение типа `Expression`:

`Money`

```
Expression plus(Money addend) {
    return new Money(amount + addend.amount, currency);
}
```

Это означает, что класс `Money` должен реализовать интерфейс `Expression` (это очень просто, так как в этом интерфейсе пока что нет ни одной операции):

`Money`

```
class Money implements Expression
```

Кроме того, нам потребуется пустой класс `Bank`:

`Bank`

```
class Bank
```

Добавим в этот класс заглушку для метода `reduce()`:

`Bank`

```
Money reduce(Expression source, String to) {
    return null;
}
```

Теперь код компилируется и выдает нам красную полоску. Ура! У нас прогресс! Теперь можем легко подделать реализацию:

`Bank`

```
Money reduce(Expression source, String to) {
    return Money.dollar(10);
}
```

Зеленая полоса! Теперь мы готовы к выполнению рефакторинга.

Но сначала подведем итоги главы. В этой главе мы:

- вместо большого теста решили реализовать меньший тест, чтобы добиться быстрого прогресса (вместо `$5 + 10 CHF` решили реализовать выполнение более простой операции `$5 + $5`);
- основательно обдумали возможные метафоры для нашего предполагаемого дизайна;
- переписали изначальный тест в свете новой избранной нами метафоры;
- как можно быстрее добились компиляции теста;
- добились того, чтобы тест срабатывал;
- с трепетом посмотрели вперед, оценивая объем рефакторинга, который необходим для того, чтобы сделать реализацию реальной.

Делаем реализацию реальной

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1
 $\$5 + \$5 = \$10$

Мы не можем вычеркнуть пункт $\$5 + \5 до тех пор, пока не удалим из кода все присутствующее в нем дублирование. Внимательно взглянем на код. Дублирования кода нет, однако присутствует дублирование данных — $\$10$ в созданной нами «поддельной» реализации:

Bank

```
Money reduce(Expression source, String to) {  
    return Money.dollar(10);  
}
```

Это выражение по своей сути дублирует выражение $\$5 + \5 в коде теста:

```
public void testSimpleAddition() {  
    Money five= Money.dollar(5);  
    Expression sum= five.plus(five);  
    Bank bank= new Bank();  
    Money reduced= bank.reduce(sum, "USD");  
    assertEquals(Money.dollar(10), reduced);  
}
```

Раньше, если у нас была «поддельная» реализация, для нас было очевидным, как можно вернуться назад и сформировать реальную реализацию. Для этого достаточно было заменить константы переменными. Однако в данном случае для меня не очевидно, каким образом я могу вернуться назад. Поэтому, несмотря на некоторый риск, я решаю двигаться вперед.

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1
 $\$5 + \$5 = \$10$

Операция $\$5 + \5 возвращает объект Money

Прежде всего, метод `Money.plus()` должен возвращать не просто объект `Money`, а реальное выражение (`Expression`), то есть сумму (`Sum`). (Возможно, в будущем мы оптимизируем специальный случай сложения двух одинаковых валют, однако это произойдет позже.)

Итак, в результате сложения двух объектов `Money` должен получиться объект класса `Sum`:

```
public void testPlusReturnsSum() {
    Money five= Money.dollar(5);
    Expression result= five.plus(five);
    Sum sum= (Sum) result;
    assertEquals(five, sum.augend);
    assertEquals(five, sum.addend);
}
```

(Вы когда-нибудь слышали о том, что в английском языке первое слагаемое обозначается термином *augend*, а второе слагаемое — термином *addend*? Об этом не слышал даже автор до тех пор, пока не приступил к написанию данной книги.)

Написанный только что тест скорее всего не проживет слишком долго. Дело в том, что он сильно связан с конкретной реализацией разрабатываемой нами операции и мало связан с видимым внешним поведением этой операции. Однако если мы заставим его работать, мы подозреваем, что окажемся на один шаг ближе к поставленной цели. Чтобы откомпилировать тест, нам потребуется класс `Sum` с двумя полями: `augend` и `addend`:

`Sum`

```
class Sum {
    Money augend;
    Money addend;
}
```

В результате получаем исключение преобразования классов (`ClassCastException`) — метод `Money.plus()` возвращает объект `Money`, но не объект `Sum`:

`Money`

```
Expression plus(Money addend) {
    return new Sum(this, addend);
}
```

Класс `Sum` должен обладать конструктором:

`Sum`

```
Sum(Money augend, Money addend) {
}
```

Кроме того, класс `Sum` должен поддерживать интерфейс `Expression`:

`Sum`

```
class Sum implements Expression
```

Наша система компилируется, однако тесты не срабатывают — это из-за того, что конструктор класса `Sum` не присваивает значения полям (мы могли бы соз-

дать «поддельную» реализацию, проинициализировав поля константами, однако я обещал двигаться быстрее):

```
Sum
    Sum(Money augend, Money addend) {
        this.augend= augend;
        this.addend= addend;
    }
```

Теперь в метод `Bank.reduce()` передается объект класса `Sum`. Если суммируются две одинаковые валюты и если целевая валюта совпадает с валютой обоих слагаемых, значит, результатом будет объект класса `Money`, чье значение будет равно сумме значений двух слагаемых.

```
public void testReduceSum() {
    Expression sum= new Sum(Money.dollar(3), Money.dollar(4));
    Bank bank= new Bank();
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(7), result);
}
```

Я тщательно выбираю значения параметров таким образом, чтобы нарушить работу существующего теста. Когда мы приводим (метод `reduce()`) объект класса `Sum` к некоторой валюте, в результате (с учетом упомянутых упрощенных условий) должен получиться объект класса `Money`, чье значение (`amount`) совпадает с суммой значений двух объектов `Money`, переданных конструктору объекта `Sum`, а валюта (`currency`) совпадает с валютой обоих этих объектов:

```
Bank
    Money reduce(Expression source, String to) {
        Sum sum= (Sum) source;
        int amount= sum.augend.amount + sum.addend.amount;
        return new Money(amount, to);
    }
```

Код выглядит уродливо по двум причинам:

- мы выполняем приведение типов (к типу `Sum`), в то время как код должен работать с любым объектом типа `Expression`;
- мы используем открытые поля и два уровня ссылок на поля объектов.

Это достаточно легко исправить. Вначале переместим тело метода в класс `Sum` и благодаря этому избавимся от лишнего уровня ссылок.

```
Bank
    Money reduce(Expression source, String to) {
        Sum sum= (Sum) source;
        return sum.reduce(to);
    }
```

```
Sum
    public Money reduce(String to) {
        int amount= augend.amount + addend.amount;
```

```
return new Money(amount, to);
}
```

На секундочку заглянем в будущее. Приведение (**reduce**) суммы к некоторой валюте не может быть выполнено, если объект **Sum** не знает об обменном курсе. Однако обменный курс хранится в классе **Bank**, значит, скорее всего, в будущем нам потребуется передавать в метод **Sum.reduce()** еще один параметр типа **Bank**. Однако сейчас наш код не требует этого. Поэтому мы не добавляем никаких лишних параметров, чтобы лишний раз в них не путаться. (Что касается меня, то искушение было столь велико, что я все-таки добавил этот параметр, когда в первый раз писал данный код, — мне очень, очень стыдно.)

\$5 + 10 CHF = \$10, если курс обмена 2:1

\$5 + \$5 = \$10

Операция \$5 + \$5 возвращает объект **Money**
Bank.reduce(Money)

Так, а что же происходит в случае, если аргументом метода **Bank.reduce()** является объект **Money**?

Давайте напишем тест, слава богу, перед нами зеленая полоса и мы не видим каких-либо других очевидных способов модификации кода:

```
public void testReduceMoney() {
    Bank bank= new Bank();
    Money result= bank.reduce(Money.dollar(1), "USD");
    assertEquals(Money.dollar(1), result);
}
```

Bank

```
Money reduce(Expression source, String to) {
    if (source instanceof Money) return (Money) source;
    Sum sum= (Sum) source;
    return sum.reduce(to);
}
```

Какой кошмар! Отвратительно! Однако снова мы получили зеленую полоску и можем приступить к рефакторингу. Каждый раз, когда мы напрямую проверяем имя класса, вместо этого мы можем использовать полиморфизм. Класс **Sum** реализует метод **reduce(String)**, если класс **Money** также будет реализовывать этот метод, мы сможем добавить метод **reduce(String)** в состав интерфейса **Expression**.

Bank

```
Money reduce(Expression source, String to) {
    if (source instanceof Money)
        return (Money) source.reduce(to);
    Sum sum= (Sum) source;
    return sum.reduce(to);
}
```

Money

```
public Money reduce(String to) {
    return this;
}
```

Включаем метод `reduce(String)` в состав интерфейса `Expression`:

Expression

```
Money reduce(String to);
```

Теперь мы можем избавиться от этих уродливых операций приведения типа и проверок имени класса:

Bank

```
Money reduce(Expression source, String to) {
    return source.reduce(to);
}
```

Я не вполне доволен ситуацией, когда в интерфейсе `Expression` и классе `Bank` присутствуют методы с одинаковыми именами, но с разным набором параметров. Я так и не смог найти приемлемого решения этой проблемы для языка Java. В языках, в которых параметры идентифицируются при помощи ключевых слов, разница между методами `Bank.reduce(Expression, String)` и `Expression.reduce(String)` делается очевидной благодаря синтаксису языка. Однако в языках, в которых различие параметров определяется различием их позиций в списке параметров, разница между двумя подобными методами становится менее очевидной.

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 + \$5 = \$10$

Операция $\$5 + \5 возвращает объект `Money`

`Bank.reduce(Money)`

Приведение объекта `Money` с одновременной конверсией валют

`Reduce(Bank,String)`

Теперь можно приступить к задаче реального обмена одной валюты на другую.

В данной главе мы:

- не отметили тест как завершенный, так как не избавились от дублирования;
- чтобы прояснить реализацию, решили двигаться вперед вместо того, чтобы двигаться назад;
- написали тест для того, чтобы форсировать создание объекта, который, как нам кажется, потребуется нам в будущем (объект класса `Sum`);
- ускорили процесс реализации (конструктор класса `Sum`);
- реализовали код с приведением типов в одном месте, добились срабатывания тестов, а затем переместили код туда, где он должен находиться;
- воспользовались полиморфизмом для того, чтобы избавиться от оператора явной проверки типа (класса).

Обмен валюты

\$5 + 10 CHF = \$10, если курс обмена 2:1

\$5 + \$5 = \$10

Операция \$5 + \$5 возвращает объект Money

Bank.reduce(Money)

Приведение объекта Money с одновременной конверсией валют

Reduce(Bank,String)

Изменения, перемены, обмены — их объятия заслуживают внимания (особенно если у вас есть книга с фразой в заголовке «в объятиях изменений» (embrace change))¹. Впрочем, нас заботит простейшая форма обмена — у нас есть два франка и мы хотим получить один доллар. Это звучит как готовый тест:

```
public void testReduceMoneyDifferentCurrency() {
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(Money.franc(2), "USD");
    assertEquals(Money.dollar(1), result);
}
```

Когда я конвертирую франки в доллары, я просто делю значение на два (мы по-прежнему игнорируем все эти неприятные проблемы, связанные с дробными числами). Чтобы сделать полоску зеленой, мы добавляем в код еще одну уродливую конструкцию:

Money

```
public Money reduce(String to) {
    int rate = (currency.equals("CHF") && to.equals("USD"))
        ? 2
        : 1;
    return new Money(amount / rate, to);
}
```

¹ Используя игру слов (английское change означает как «изменение», так и «обмен»), автор намекает на свою знаменитую книгу-бестселлер *Extreme Programming Explained: Embrace Change*. Русский перевод: Бек К. Экстремальное программирование. СПб.: Питер, 2002. 224 с. — Примеч. ред.

Получается, что класс `Money` знает о курсе обмена. Это неправильно. Единственным местом, в котором выполняются любые операции, связанные с курсом обмена, должен быть класс `Bank`. Мы должны передать параметр типа `Bank` в метод `Expression.reduce()`. (Вот видите? Мы так и думали, что нам это потребуется. И мы оказались правы.) Вначале меняем вызывающий код:

`Bank`

```
Money reduce(Expression source, String to) {
    return source.reduce(this, to);
}
```

Затем меняем код реализаций:

`Expression`

```
Money reduce(Bank bank, String to);
```

`Sum`

```
public Money reduce(Bank bank, String to) {
    int amount = augend.amount + addend.amount;
    return new Money(amount, to);
}
```

`Money`

```
public Money reduce(Bank bank, String to) {
    int rate = (currency.equals("CHF") && to.equals("USD"))
        ? 2
        : 1;
    return new Money(amount / rate, to);
}
```

Методы должны быть открытыми (`public`), так как все методы интерфейсов должны быть открытыми (я надеюсь, можно не объяснять, почему).

Теперь мы можем вычислить курс обмена внутри класса `Bank`:

`Bank`

```
int rate(String from, String to) {
    return (from.equals("CHF") && to.equals("USD"))
        ? 2
        : 1;
}
```

И обратиться к объекту `bank` с просьбой предоставить значение курса обмена:

`Money`

```
public Money reduce(Bank bank, String to) {
    int rate = bank.rate(currency, to);
    return new Money(amount / rate, to);
}
```

Эта надоедливая цифра 2 снова отвсечивает как в разрабатываемом коде, так и в теле теста. Чтобы избавиться от нее, мы должны создать таблицу обменных курсов в классе `Bank` и при необходимости обращаться к этой таблице для полу-

чения значения обменного курса. Для этой цели мы могли бы воспользоваться хэш-таблицей, которая ставит в соответствие паре валют соответствующий обменный курс. Можем ли мы в качестве ключа использовать двухэлементный массив, содержащий в себе две валюты? Проверяет ли метод `Array.equals()` эквивалентность элементов массива?

```
public void testArrayEquals() {
    assertEquals(new Object[] {"abc"}, new Object[] {"abc"});
}
```

Нет. Тест не сработал. Придется создавать специальный объект, который будет использоваться в качестве ключа хэш-таблицы:

Pair

```
private class Pair {
    private String from;
    private String to;

    Pair(String from, String to) {
        this.from = from;
        this.to = to;
    }
}
```

Мы планируем использовать объекты `Pair` в качестве ключей, поэтому нам необходимо реализовать методы `equals()` и `hashCode()`. Я не собираюсь писать для этого тесты, так как мы разрабатываем код в контексте рефакторинга. Дело в том, что от работоспособности этого кода жестко зависит срабатывание существующих тестов. Если код работает неправильно, существующие тесты не сработают. Однако в случае, если бы я программировал в паре с кем-то, кто плохо представлял бы себе направление дальнейшего движения, или в случае, если бы логика кода была бы несколько более сложной, я несомненно приступил бы к разработке специальных тестов.

Pair

```
public boolean equals(Object object) {
    Pair pair = (Pair) object;
    return from.equals(pair.from) && to.equals(pair.to);
}
public int hashCode() {
    return 0;
}
```

0 — ужасное хэш-значение, однако такой метод хэширования легко реализовать, стало быть мы быстрее получим работающий код. Поиск валюты будет выполняться в соответствии с алгоритмом простого линейного поиска. Позже, когда у нас будет множество валют, мы сможем тщательнее проработать этот вопрос, используя реальные данные.

Теперь нам нужно место, в котором мы могли бы хранить значения обменных курсов:

Bank

```
private Hashtable rates= new Hashtable();
```

Нам также потребуется метод добавления нового курса обмена:

Bank

```
void addRate(String from, String to, int rate) {
    rates.put(new Pair(from, to), new Integer(rate));
}
```

А также метод возврата значения обменного курса:

Bank

```
int rate(String from, String to) {
    Integer rate= (Integer) rates.get(new Pair(from, to));
    return rate.intValue();
}
```

Подождите-ка минутку! Перед нами красная полоса. Что случилось? Взглянув на код, мы обнаруживаем, что проблема в неправильном значении курса при обмене доллара на доллары. Мы ожидаем, что при обмене USD на USD курс обмена будет равен 1, однако на текущий момент это не так. Открытие было для нас сюрпризом, поэтому мы оформляем его в виде дополнительного теста:

```
public void testIdentityRate() {
    assertEquals(1, new Bank().rate("USD", "USD"));
}
```

Теперь у нас три ошибки, однако все они могут быть исправлены при помощи одного небольшого изменения:

Bank

```
int rate(String from, String to) {
    if (from.equals(to)) return 1;
    Integer rate= (Integer) rates.get(new Pair(from, to));
    return rate.intValue();
}
```

Зеленая полоска!

\$5 + 10 CHF = \$10, если курс обмена 2:1

\$5 + \$5 = \$10

Операция \$5 + \$5 возвращает объект Money

Bank.reduce(Money)

Приведение объекта Money с одновременной конверсией валют

Reduce(Bank,String)

Далее мы переходим к нашему последнему, самому большому тесту, \$5 + 10 CHF. В данной главе мы применили несколько важных технологий:

- добавили параметр, который, как мы ожидаем, нам понадобится;
- удалили дублирование между кодом и тестами;

- написали тест (`testArrayEquals`), чтобы проверить порядок функционирования встроенной операции Java;
- создали вспомогательный закрытый (`private`) класс, не обладающий собственными тестами;
- допустили ошибку при рефакторинге, написали еще один тест для того, чтобы изолировать проблему.

Смешение валют

\$5 + 10 CHF = \$10, если курс обмена 2:1

~~\$5 + \$5 = \$10~~

Операция \$5 + \$5 возвращает объект Money

Bank.reduce(Money)

Приведение объекта Money с одновременной конверсией валют

Reduce(Bank,String)

Теперь мы готовы написать тест, с которого все началось, — \$5 + 10 CHF:

```
public void testMixedAddition() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

Именно такой код нам хотелось бы написать. К сожалению, для начала мы получаем кучу ошибок компиляции. Когда мы обобщали код, в процессе перехода от Money к Expression мы оставили много висящих хвостов, на которые я, конечно же, обратил внимание, но решил вас не беспокоить. Теперь настало время заняться устранением дефектов.

Мы не сможем достаточно быстро обеспечить компиляцию предыдущего теста. Как только мы внесем в код первое изменение, нам потребуется внести в код еще изменения и так далее. Теперь мы можем двигаться дальше одним из двух путей. Мы можем заставить тест работать быстро, для этого надо написать более специфичный тест и затем выполнить обобщение. Второй путь: довериться компилятору, компилятор поможет нам найти все ошибки. Давайте попробуем действовать медленно (на практике я бы внес в код все необходимые изменения за один раз).

```
public void testMixedAddition() {
    Money fiveBucks= Money.dollar(5);
```

```

Money tenFrancs= Money.franc(10);
Bank bank= new Bank();
bank.addRate("CHF", "USD", 2);
Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
assertEquals(Money.dollar(10), result);
}

```

Тест не срабатывает. Мы получаем 15 USD вместо 10 USD. Дело в том, что метод `Sum.reduce()` не выполняет приведение аргументов:

Sum

```

public Money reduce(Bank bank, String to) {
    int amount= augend.amount + addend.amount;
    return new Money(amount, to);
}

```

Если мы выполним приведение обоих аргументов, тест должен сработать:

Sum

```

public Money reduce(Bank bank, String to) {
    int amount= augend.reduce(bank, to).amount
        + addend.reduce(bank, to).amount;
    return new Money(amount, to);
}

```

И действительно, тест срабатывает. Теперь мы можем заменить тип `Money` на тип `Expression`. Чтобы избежать взаимовлияний, мы начнем издалека и будем двигаться в направлении тестирующего кода. Итак, поля `augend` и `addend` теперь могут обладать типом `Expression`:

Sum

```

Expression augend;
Expression addend;

```

Аргументы конструктора тоже могут обладать типом `Expression`:

Sum

```

Sum(Expression augend, Expression addend) {
    this.augend= augend;
    this.addend= addend;
}

```

(Класс `Sum` начинает напоминать мне `Composite`, однако еще не настолько, чтобы я захотел обобщить его.) С классом `Sum`, пожалуй, закончили, а что насчет `Money`?

Аргумент метода `plus()` может обладать типом `Expression`:

Money

```

Expression plus(Expression addend) {
    return new Sum(this, addend);
}

```

Метод `times()` может возвращать значение типа `Expression`:

`Money`

```
Expression times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

Это означает, что в состав интерфейса `Expression` должны входить операции `plus()` и `times()`. С классом `Money` закончили. Теперь мы можем изменить аргументы метода `plus()` в нашем тестирующем коде:

```
public void testMixedAddition() {
    Money fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

Объект `tenFrancs` теперь принадлежит типу `Expression`, а это значит, что мы должны внести в код некоторые изменения. К счастью, компилятор подсказывает нам, что именно мы должны сделать. Прежде всего вносим изменение:

```
public void testMixedAddition() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

Компилятор вежливо сообщает, что метод `plus()` не входит в состав интерфейса `Expression`. Добавим этот метод в интерфейс:

`Expression`

```
Expression plus(Expression addend);
```

После этого мы должны добавить этот метод в классы `Money` и `Sum`. `Money`? Да, этот метод должен быть открытым (`public`) в классе `Money`:

`Money`

```
public Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

Что касается класса `Sum`, просто добавим заглушку и отметим необходимость реализации этого метода в списке задач:

`Sum`

```
public Expression plus(Expression addend) {
    return null;
}
```

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 + \$5 = \$10$

Операция $\$5 + \5 возвращает объект Money

`Bank.reduce(Money)`

Приведение объекта Money с одновременной конверсией валют

`Reduce(Bank,String)`

`Sum.plus`

`Expression.times`

Теперь программа компилируется и все тесты срабатывают.

Мы готовы выполнить обобщение класса Money на основе Expression.

Как всегда, подведем краткий итог. В этой главе мы:

- написали желательный для нас тест, затем модифицировали его, чтобы добиться его срабатывания за один шаг;
- выполнили обобщение (использовали более абстрактное объявление);
- воспользовались подсказками компилятора для того, чтобы внести изменения (`Expression fiveBucks`), которые привели к необходимости внесения в код дополнительных изменений (добавление метода `plus()` в интерфейс `Expression` и т. п.).

Абстракция, наконец-то!

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 + \$5 = \$10$

Операция $\$5 + \5 возвращает объект Money

`Bank.reduce(Money)`

Приведение объекта Money с одновременной конверсией валют

`Reduce(Bank,String)`

Sum.plus

`Expression.times`

Чтобы завершить добавление метода `Expression.plus`, мы должны реализовать метод `Sum.plus()`. Затем нам потребуется метод `Expression.times()` и мы можем считать пример завершенным. Вот тест для метода `Sum.plus()`:

```
public void testSumPlusMoney() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Expression sum= new Sum(fiveBucks, tenFrancs).plus(fiveBucks);
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(15), result);
}
```

Мы могли бы создать объект `Sum` путем сложения `fiveBucks` и `tenFrancs`, однако приведенный код, который явно создает объект `Sum`, является более понятным. Ведь мы пишем эти тесты не только для того, чтобы получить удовольствие от программирования, но также для того, чтобы будущие поколения программистов могли оценить нашу гениальность. Однако они не смогут сделать этого, если код будет непонятным. Таким образом, разрабатывая любой код, думайте о тех, кто будет его читать.

В данном случае код теста длиннее, чем сам тестируемый код. Код точно такой же, как код в классе `Money` (кажется, я уже предвижу необходимость создания абстрактного класса):

Sum

```
public Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

\$5 + 10 CHF = \$10, если курс обмена 2:1

\$5 + \$5 = \$10

Операция \$5 + \$5 возвращает объект Money

Bank.reduce(Money)

Приведение объекта Money с одновременной конверсией валют

Reduce(Bank,String)

Sum.plus

Expression.times

При использовании TDD вы часто будете сталкиваться с тем, что количество строк в тестовом коде будет приблизительно таким же, как и количество строк в тестируемом коде. Чтобы методика TDD обладала экономическим смыслом, вы должны либо записывать в два раза большее количество строк кода, чем обычно, либо реализовывать ту же самую функциональность при помощи количества строк, в два раза меньшего, чем обычно. Эти показатели рекомендуется оценить самостоятельно на примере собственной практики. Однако, выполняя оценку, вы должны принять во внимание время, которое тратится на отладку, интеграцию и объяснение внутреннего устройства другим людям.

\$5 + 10 CHF = \$10, если курс обмена 2:1

\$5 + \$5 = \$10

Операция \$5 + \$5 возвращает объект Money

Bank.reduce(Money)

Приведение объекта Money с одновременной конверсией валют

Reduce(Bank,String)

Sum.plus

Expression.times

Если мы получили работающий метод Sum.times(), значит, объявление Expression.times() не составит для нас труда. Вот соответствующий тест:

```
public void testSumTimes() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Expression sum= new Sum(fiveBucks, tenFrancs).times(2);
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(20), result);
}
```

И снова тест длиннее, чем тестируемый код. (Те, кто достаточно много работал с JUnit, должно быть уже догадались, как можно решить эту проблему. Остальные должны прочитать раздел «Fixture (Фикстура)» в главе, посвященной паттернам xUnit.)

Sum

```
Expression times(int multiplier) {
    return new Sum(augend.times(multiplier).addend.times(multiplier));
}
```

В предыдущей главе мы изменили тип переменных `augend` и `addend` на `Expression`, поэтому теперь для того, чтобы код можно было откомпилировать, нам необходимо добавить в интерфейс `Expression` метод `times()`:

Expression

```
Expression times(int multiplier);
```

При этом нам необходимо изменить режим видимости методов `Money.times()` и `Sum.times()` (они должны стать открытыми):

Sum

```
public Expression times(int multiplier) {
    return new Sum(augend.times(multiplier).addend.times(multiplier));
}
```

Money

```
public Expression times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 + \$5 = \$10$

Операция $\$5 + \5 возвращает объект `Money`

`Bank.reduce(Money)`

Приведение объекта `Money` с одновременной конверсией валют

`Reduce(Bank,String)`

`Sum.plus`

`Expression.times`

Все заработало.

Осталось провести эксперимент для случая, когда в результате выполнения операции $\$5 + \5 получается объект `Money`. Вот соответствующий тест:

```
public void testPlusSameCurrencyReturnsMoney() {
    Expression sum= Money.dollar(1).plus(Money.dollar(1));
    assertTrue(sum instanceof Money);
}
```

Тест выглядит несколько неопрятно, так как тестирует внутреннюю реализацию, а не внешнее поведение объектов. Однако он принуждает нас внести в программу изменения, которые нам необходимы, и, в конце концов, это всего лишь эксперимент. Вот код, который мы должны модифицировать для того, чтобы заставить тест работать:

Money

```
public Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```


$\$5 + 10 \text{ CHF} = \10 , если курс обмена 2:1

$\$5 + \$5 = \$10$

Операция $\$5 + \5 возвращает объект Money

`Bank.reduce(Money)`

Приведение объекта Money с одновременной конверсией валют

`Reduce(Bank,String)`

`Sum.plus`

`Expression.times`

Не существует очевидного и ясного способа проверить валюту аргумента, в случае если этот аргумент является объектом класса Money (по крайней мере, я не могу найти такого способа, однако вы можете над этим подумать). Эксперимент окончился неудачей, мы удаляем тест (который нам все равно не нравился).

Подводим итог. Мы:

- написали тест так, чтобы его смысл легко был понят другими программистами, которые в будущем будут читать разработанный нами код;
- наметили эксперимент, призванный сравнить эффективность TDD по отношению к обычному стилю программирования, используемому вами на текущий момент;
- снова столкнулись с необходимостью изменения множества объявлений в разрабатываемом коде и снова воспользовались услугами компилятора для того, чтобы исправить все неточности;
- попробовали провести быстрый эксперимент, однако отказались от идеи, так как она не работала, и уничтожили соответствующий тест.

Ретроспектива денежного примера

Давайте попробуем дать общую развернутую оценку рассмотренному нами примеру реализации мультивалютных вычислений. Попробуем оценить использованный нами процесс и полученные результаты. Вот несколько тезисов, которых нам хотелось бы коснуться:

- *Что дальше?* Как определить дальнейшее направление разработки?
- *Метафора.* Впечатляющий эффект, который метафора оказывает на структуру дизайна.
- *Использование JUnit.* Как часто мы запускали тесты и как мы использовали JUnit?
- *Метрики кода.* Численные характеристики результирующего кода.
- *Процесс.* Мы говорим: «красный—зеленый—рефакторинг». Как много усилий прикладывается на каждом из этих этапов?
- *Качество тестов.* Каким образом характеристики тестов TDD соотносятся с характеристиками обычных методик тестирования?

Что дальше?

Можно ли считать код завершенным? Нет. Существует кошмарное дублирование между методами `Sum.plus()` и `Money.plus()`. Если мы преобразуем интерфейс `Expression` в класс (не совсем обычное преобразование, — чаще классы становятся интерфейсами), мы получим возможность переместить общий код в единый общий метод.

На самом деле мне сложно представить себе код, который полностью завершен. Методику TDD можно использовать как способ приближения к идеалу, однако это будет не самое эффективное использование. Если вы имеете дело с крупной системой, тогда части системы, с которыми вы работаете каждый день, должны быть тщательно «вылизаны». Иными словами, дизайн должен быть чистым и понятным, а код должен быть хорошо протестированным. В этом случае вы изо дня в день сможете вносить в систему необходимые изменения, не теряя при

этом уверенности в работоспособности кода. Однако на периферии системы, где располагаются части, к которым вы обращаетесь относительно редко, количество тестов может быть меньше, а дизайн — уродливее.

Когда я завершаю решение всех очевидных задач, я люблю запускать инструмент критики кода (например, SmallLint for Smalltalk). Многие полученные в результате этого предложения мне и без того известны. Со многими предложениями я не согласен. Однако автоматизированный инструмент критики кода ни о чем не забывает, поэтому иногда он обнаруживает то, что было упущено мною из виду.

Еще один полезный вопрос: «Какие дополнительные тесты необходимо написать для системы?» Иногда вы думаете, что некоторый тест не должен срабатывать, однако, добавив его в тестовый набор, вы обнаруживаете, что он срабатывает. В этом случае необходимо определить, почему так происходит. Иногда тест, который не должен сработать, действительно не срабатывает, и вы добавляете его в набор, как признак известного вам ограничения разрабатываемой системы или как напоминание о работе, которую необходимо выполнить позднее.

Наконец, когда список задач пуст, неплохо еще раз проверить дизайн. Удовлетворяет ли данная реализация всем предъявляемым требованиям? Существует ли дублирование, которое сложно удалить при использовании данного дизайна? (Сохранившееся дублирование — признак нереализованного дизайна.)

Метафора

Лично для меня самым большим сюрпризом, связанным с кодированием данного денежного примера, явилось впечатляющее отличие результирующего дизайна от тех разработок, с которыми мне приходилось иметь дело до написания этой книги. Я выполнял разработку аналогичного мультивалютного кода для различных, реально используемых в производстве программных систем по меньшей мере три раза (насколько я могу припомнить). Кроме того, я использовал эту же задачу для разного рода публикаций еще раз шесть или семь. Помимо публикаций я пятнадцать раз программировал этот пример перед аудиторией на различных конференциях (программировал со сцены, — звучит здорово, но выглядит менее впечатляюще). Наконец, прежде чем написать окончательный вариант первой части данной книги, я перебрал три или четыре различных направления разработки кода (я менял направление своих мыслей в соответствии с поступающими ранними рецензиями и отзывами о написанном материале). И вот, пока я работал над текстом первой части, мне в голову пришла мысль использовать в качестве метафоры *математические выражения* (expressions). В результате дизайн стал развиваться по совершенно иному, не известному мне ранее пути.

Я никогда не думал, что метафора — это настолько мощный инструмент. Многие думают, что метафора — это всего лишь источник имен. Разве не так? Похоже, что нет.

Для представления «комбинации нескольких денежных величин, которые могут быть выражены в разных валютах», Уорд Каннингэм использовал метафору *вектора*. Имеется в виду математический вектор, — набор коэффициентов, каждому из которых соответствует некоторая валюта. Лично я некоторое время

использовал метафору *суммы денег* (MoneySum), затем придумал *денежный мешок* (MoneyBag) — звучит понятно и близко к реальности, — наконец, остановился на метафоре *бумажника* (Wallet). Что такое бумажник и как он функционирует, известно абсолютно всем. Все эти метафоры подразумевают, что набор денежных значений (объектов Money) является плоским. Иначе говоря, выражение 2 USD + 5 CHF + 3 USD эквивалентно выражению 5 USD + 5 CHF. Два значения в одной и той же валюте автоматически сливаются в одно.

Метафора *математического выражения* избавила меня от множества неприятных проблем, связанных со слиянием дублирующихся валют. Результирующий код получился чище, чем я когда-либо видел. Конечно же, я несколько обеспокоен производительностью кода, основанного на подобной метафоре, однако, прежде чем приступать к оптимизации, я намерен проанализировать статистику обращений к различным участкам кода.

Почему я был вынужден переписать заново то, что я уже писал до этого не меньше 20-ти раз? Буду ли я и дальше сталкиваться с подобными сюрпризами? Существует ли способ, который позволит мне найти правильное решение, по крайней мере, в течение первых трех попыток? А может быть этот способ позволит мне найти правильное решение с первой попытки?

Использование JUnit

Я поручил инфраструктуре JUnit вести журнал в процессе кодирования мультивалютного примера. Выяснилось, что за все время кодирования я нажал на клавишу Enter ровно 125 раз. Оценку интервала между запусками тестов нельзя считать достоверной, так как в ходе работы я не только программировал, но и писал текст книги. Однако когда я занимался только программированием, я запускал тесты приблизительно раз в минуту.

На рис. 17.1 представлена гистограмма интервалов между запусками тестов. Большое количество длительных интервалов, скорее всего, обусловлено тем, что я тратил значительное время на написание текста книги.



Рис. 17.1. Гистограмма интервалов времени между запусками тестов

Метрики кода

В табл. 17.1 сведены некоторые статистические данные о написанном коде.

Таблица 17.1. Метрики кода

	Функциональный код	Тесты
Количество классов	5	1
Количество функций ⁽¹⁾	22	15
Количество строк кода ⁽²⁾	91	89
Цикломатическая сложность кода ⁽³⁾	1,04	1
Количество строк/Количество функций	4,1 ⁽⁴⁾	5,9 ⁽⁵⁾

Вот некоторые примечания к данной таблице:

1. Мы не реализовали весь программный интерфейс (API) целиком, поэтому не можем достоверно оценить полное количество функций или количество функций на один класс или количество строк кода на один класс. Однако соотношения этих параметров можно считать поучительными. Количество функций и количество строк в тестах приблизительно такое же, как и в функциональном коде.
2. Количество строк кода в тестах можно сократить, если извлечь из кода общие *фикстуры* (fixtures). Однако общее соотношение между строками функционального кода и строками тестирующего кода при этом сохранится.
3. Цикломатическая сложность (cyclomatic complexity) — это величина, характеризующая сложность обычного потока управления в программе. Цикломатическая сложность тестов равна 1, так как в тестирующем коде нет ни ветвлений, ни циклов. Цикломатическая сложность функционального кода близка к единице, так как вместо явных ветвлений для передачи управления чаще используется полиморфизм.
4. Оценка количества строк в функции дана с учетом заголовка функции и замыкающей скобки.
5. Количество строк на функцию для тестирующего кода в нашем случае больше чем могло бы быть, так как мы не выделили общего кода в виде *фикстур*. О фикстурах рассказывается в главе 29, которая посвящена методам работы с xUnit.

Процесс

Цикл TDD выглядит следующим образом:

- добавить небольшой тест;
- запустить все тесты и обнаружить, что добавленный тест не сработал;

- внести в код изменения;
- запустить тесты и убедиться в том, что они сработали;
- выполнить рефакторинг для того, чтобы удалить дублирование.

Если исходить из того, что разработка теста — это один шаг, какое количество изменений требуется сделать для того, чтобы выполнить компиляцию, запуск и рефакторинг? (Под изменением я подразумеваю изменение определения метода или класса.) На рис. 17.2 показана гистограмма количества изменений для каждого из тестов «денежного» примера, над которым мы работали в первой части книги.

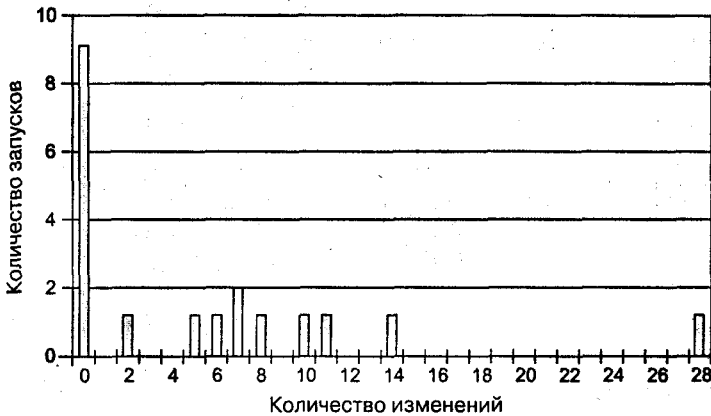


Рис. 17.2. Гистограмма количества изменений, приходящихся на каждый рефакторинг

Я полагаю, что если бы мы собирали статистику для достаточно крупного проекта, мы обнаружили бы, что количество изменений, необходимых для компиляции и запуска кода, очень невелико (это количество можно уменьшить, если среда разработки будет понимать то, что пытаются ей сказать тесты, и, например, будет автоматически добавлять в функциональный код необходимые заглушки). Однако количество изменений, вносимых в код во время рефакторинга, должно соответствовать (вот главный тезис) кривой распределения с эксцессом больше нормального, то есть с большим числом изменений, чем предсказывается стандартной кривой нормального распределения. Подобный профиль характерен для многих других естественных процессов, например для изменения стоимости акций на рынке ценных бумаг¹.

Качество тестов

Тесты являются неотъемлемой частью методики TDD. Тесты TDD могут быть запущены в любое время работы над программой, а также после того, как про-

¹ Mandelbrot, Benoit, ed. 1997. *Fractals and Scaling in Finance*. New York: Springer-Verlag.

грамма будет завершена. Однако не стоит путать их с другими важными типами тестирования:

- тестированием производительности;
- стрессовым тестированием;
- тестированием практичности.

Тем не менее если плотность вероятности дефектов в коде, разработанном с использованием TDD, невелика, роль профессионального тестирования меняется. Если обычно профессиональное тестирование используется для постоянного надзора за работой программистов, то при использовании TDD профессиональное тестирование больше напоминает вспомогательный инструмент, облегчающий коммуникацию между теми, кто знает, как должна работать система, и теми, кто создает систему.

Как можно оценить качество разработанных нами тестов? Вот два широко распространенных метода:

- *Покрытие операторов кода (statement coverage)*. Для оценки качества тестов этой характеристики недостаточно, однако ее можно использовать как отправную точку. Если программист ревностно следует всем требованиям TDD, покрытие функционального кода тестами должно составлять 100 %. Для оценки этой характеристики можно использовать специальные программные средства. Например, программа JProbe (www.sitaka.com/software/jprobe) сообщает нам, что в нашем примере не покрытой тестами осталась всего одна строка в одном методе — `Money.toString()`. Напомню, что эта строка была добавлена в отладочных целях, фактически она не является функциональным кодом.
- *Намеренное добавление дефекта (defect insertion)*. Это еще один способ проверки качества тестов. Идея проста: необходимо изменить значение строки кода и убедиться в том, что тест перестал срабатывать. Делать это можно как вручную, так и при помощи специального инструмента, такого как Jester (jester.sourceforge.net). Этот инструмент сообщает нам, что в нашей программе существует всего одна строка, которую можно изменить, не нарушив срабатывания тестов. Вот эта строка: `Pair.hash.Code()`. Здесь мы просто подделали реализацию — вместо хэш-кода метод возвращает константное значение: 0. Если одно константное значение заменить другим, смысл программы не изменится (одна подделка ничем не лучше другой), таким образом, подобную модификацию кода нельзя считать дефектом.

Флип (Phlip), один из рецензентов моей книги, сообщил мне некоторые дополнительные соображения относительно покрытия тестами. Абсолютный показатель покрытия вычисляется следующим образом: количество тестов, предназначенных для тестирования различных аспектов программы, необходимо разделить на количество аспектов, которые нуждаются в тестировании (сложность логики программы). Существует два способа улучшить покрытие тестами. Во-первых, можно написать больше тестов. Отсюда разница в количестве тестов, которые пишутся разработчиком, использующим TDD, и профессиональным

тестером. (В главе 32 приводится пример задачи, для решения которой я написал 6 тестов, а человек, профессионально занимающийся тестированием, — 65 тестов.) Однако существует и другой способ улучшить покрытие — можно ограничиться фиксированным набором тестов и упростить логику программы. Подобный эффект зачастую достигается в процессе рефакторинга — условные операторы заменяются сообщениями классов или вовсе удаляются из программы. Флип выражает эту мысль так: «Вместо того чтобы увеличить количество тестов и тем самым охватить всевозможные комбинации входных данных (говоря точнее, эффективное подмножество всех комбинаций), мы оставляем количество тестов неизменным и меняем количество внутренних структурных комбинаций кода».

Последний взгляд назад

Существуют три важных навыка, которые необходимо освоить тем, кто впервые изучает TDD:

- три основных подхода, которые используются для того, чтобы заставить тест срабатывать: подделка реализации (Fake It), триангуляция (Triangulation) и очевидная реализация (Obvious Implementation);
- удаление дублирования между функциональным кодом и тестами — важный способ формирования дизайна;
- способность контролировать расстояние между тестами: когда дорога становится скользкой, необходимо двигаться маленькими шажками; когда дальнейший путь ясен, можно увеличить скорость.



Часть II

На примере xUnit

Какой подход использовать для разработки инструмента для программирования, основанного на тестах? Естественно, программирование, основанное на тестах.

Архитектура xUnit хорошо реализуется на языке Python, поэтому во второй части книги я перейду на использование Python. Не беспокойтесь, для тех, кто никогда раньше не имел дела с Python, я добавлю в текст необходимые пояснения. Когда вы прочитаете вторую часть, вы, во-первых, освоите базовые навыки программирования на Python, во-вторых, узнаете, как самому разработать свою собственную инфраструктуру для автоматического тестирования, и, в-третьих, ознакомитесь с более сложным примером использования методики TDD — три по цене одного!

Первые шаги на пути к xUnit

Разработка инструмента тестирования с использованием самого этого инструмента для тестирования многим может показаться чем-то, напоминающим хирургическую операцию на своем собственном мозге. («Только не вздумай трогать центры моторики! О! Слишком поздно! Игра окончена.») Сначала эта идея может показаться жутковатой. Однако инфраструктура тестирования обладает более сложной внутренней логикой, если сравнивать с относительно несложным денежным примером, рассмотренным в первой части книги. Часть II можно рассматривать, как шаг в сторону разработки «реального» программного обеспечения. Кроме того, вы можете рассматривать этот материал, как упражнение в самодокументируемом программировании.

Прежде всего, у нас должна быть возможность создать тест и запустить тестовый метод. Например: `TestCase("testMethod").run()`. Возникает проблема: мы собираемся написать тест для программного кода, который мы будем использовать для написания тестов. Так как у нас пока еще нет даже намека на инфраструктуру тестирования, мы вынуждены проверить правильность нашего самого первого шага вручную. К счастью, мы достаточно хорошо отдохнули, а значит, вероятность того, что мы допустим ошибку, относительно невелика. Однако, чтобы сделать ее еще меньше, мы планируем двигаться маленькими-маленькими шажками, тщательно проверяя все, что мы делаем. Вот список задач, который приходит на ум, когда начинаешь размышлять о разработке собственной инфраструктуры тестирования:

- Вызов тестового метода

- Вызов метода `setUp` перед обращением к методу

- Вызов метода `tearDown` после обращения к методу

- Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

- Выполнение нескольких тестов

- Отчет о результатах

Конечно же, мы по-прежнему работаем в стиле «вначале тесты». Для нашего прото-теста нам потребуется небольшая программа, которая должна отображать на экране значение «истина» в случае, если произошло обращение к тестовому

методу, и значение «ложь» в противном случае. Теперь представим, что у нас есть тест, который будет устанавливать флаг внутри тестового метода, в этом случае мы могли бы после выполнения теста отобразить состояние флага на экране и самостоятельно убедиться в том, что флаг установлен правильно. После того как мы сможем проверить это вручную, мы можем попробовать автоматизировать процесс.

Итак, у нас намечилась следующая стратегия. Мы создаем объект, который соответствует нашему тесту. В объекте содержится флаг. Перед выполнением тестового метода флаг должен быть установлен в состояние «ложь». Тестовый метод устанавливает флаг в состояние «истина». После выполнения тестового метода мы должны проверить состояние флага. Назовем наш тестовый класс именем `WasRun`¹, так как объект этого класса будет сигнализировать нам о том, был ли выполнен тестовый метод. Флаг внутри этого класса также будет называться `wasRun` (это несколько сбивает с толку, однако `wasRun` — такое подходящее имя). Собственно объект (экземпляр класса `WasRun`) будет называться просто `test`. Таким образом, мы можем написать `assert test.wasRun` (ключевое слово `assert` является встроенным механизмом языка Python).

Язык программирования Python является интерпретируемым — команды исполняются по мере того, как они читаются из файла с исходным кодом. Таким образом, чтобы выполнить тестирование, мы можем написать следующий короткий файл и попробовать запустить его на исполнение:

```
test= WasRun("testMethod")
print test.wasRun
test.testMethod()
print test.wasRun
```

Мы ожидаем, что эта миниатюрная программа напечатает `None` до выполнения тестового метода и `1` после выполнения метода. (В языке Python значение `None` является аналогом `null` или `nil` и наряду с числом `0` соответствует значению «ложь».) Однако программа не делает того, что мы от нее ждем. И немудрено, — мы еще не определили класс `WasRun` (вначале тесты!).

`WasRun`

```
class WasRun:
    pass
```

(Ключевое слово `pass` используется в случае, если реализация класса или метода отсутствует.) Теперь интерпретатор сообщает нам, что в классе `WasRun` нет атрибута с именем `wasRun`. Создание атрибута выполняется в момент создания объекта (экземпляра класса), то есть в процессе выполнения конструктора (для удобства конструктор любого класса называется `__init__`). Внутри конструктора мы присваиваем флагу `wasRun` значение `None` (ложь):

`WasRun`

```
class WasRun:
```

¹ В переводе с английского языка *was run* означает *был выполнен*. — Примеч. перев.

```
def __init__(self, name):
    self.wasRun= None
```

Теперь программа действительно отображает на экране значение `None`, однако после этого интерпретатор сообщает нам, что мы должны определить в классе `WasRun` метод `testMethod`. (Было бы неплохо, если бы среда разработки автоматически реагировала на это: самостоятельно создавала бы функцию-заглушку и открывала редактор с курсором, установленным в теле этой функции. Не правда ли, это было бы просто здорово? Но, похоже, производители IDE еще не скоро до этого додумаются.)

WasRun

```
def testMethod(self):
    pass
```

Запускаем файл и видим на экране два значения: `None` и `None!`. Нам хотелось бы видеть `None` и `1`. Чтобы получить желаемый результат, в теле метода `testMethod` присваиваем флагу `wasRun` желаемое значение:

WasRun

```
def testMethod(self):
    self.wasRun= 1
```

Запускаем программу, — то, что нужно! Мы получили желаемый результат. Зеленая полоса — ур-р-ра! Нам предстоит немалый рефакторинг, однако если мы видим перед собой зеленую полосу, значит, мы добились прогресса.

Теперь, вместо того, чтобы напрямую обращаться к нашему тестовому методу, мы должны использовать наш реальный интерфейс — метод `run()`. Изменяем тест следующим образом:

```
test= WasRun("testMethod")
print test.wasRun
test.run()
print test.wasRun
```

Чтобы заставить тест работать, достаточно воспользоваться следующей несложной реализацией:

WasRun

```
def run(self):
    self.testMethod()
```

Наша тестовая программа снова печатает на экране то, что нам нужно. Зачастую при выполнении рефакторинга у вас возникает ощущение, что вам необходимо разделить код, с которым вы работаете, на две части, чтобы работать с ними по отдельности. Если в конце вашей работы они снова сольются воедино, — замечательно. Если же нет, значит, вы можете оставить их отдельно друг от друга.

¹ В языке Python заголовок определения метода (функции) начинается со служебного слова `def`, а завершается двоеточием. Операторы тела метода записываются ниже, в отдельных строчках. Группировка операторов тела отображается отступами (вместо фигурных скобок). — *Примеч. ред.*

В данном случае, со временем мы планируем создать класс `TestCase`, однако вначале мы должны обособить части нашего примера.

Следующий этап — динамический вызов метода `testMethod`. Одной из приятных отличительных характеристик языка Python является возможность использования имен классов и методов в качестве функций (см. создание экземпляра класса `WasRun`). Получив атрибут, соответствующий имени теста, мы можем обратиться к нему, как к функции. В результате будет выполнено обращение к методу с соответствующим именем¹.

`WasRun`

```
class WasRun:
    def __init__(self, name):
        self.wasRun= None
        self.name= name
    def run(self):
        method = getattr(self, self.name)
        method()
```

Это еще один паттерн рефакторинга: разработать код, который работает с некоторым конкретным экземпляром, и обобщить этот код так, чтобы он мог работать со всеми остальными экземплярами, для этого константы заменяются переменными. В данном случае роль константы играет не некоторое значение, а жестко фиксированный код (имя конкретного метода). Однако принцип остается одним и тем же. В рамках TDD эта проблема решается очень легко: методика TDD снабжает вас конкретными работающими примерами, исходя из которых вы можете выполнить обобщение. Это значительно проще, чем выполнять обобщение исходя только из собственных умозаключений.

Теперь наш маленький класс `WasRun` занят решением двух разных задач: во-первых, он следит за тем, был ли выполнен метод или нет; во-вторых, он динамически обращается к методу. Пришло время разделить полномочия (разделить нашу работу на две разные части). Прежде всего, создадим пустой суперкласс `TestCase` и сделаем класс `WasRun` производным классом:

`TestCase`

```
class TestCase:
    pass
```

`WasRun`

```
class WasRun(TestCase): . . .
```

Теперь мы можем переместить атрибут `name` из подкласса в суперкласс:

`TestCase`

```
def __init__(self, name):
    self.name= name
```

¹ Спасибо Дункану Бусу (Duncan Booth) за то, что он исправил допущенную мной ошибку, типичную для малоопытных программистов Python, и подсказавшему мне решение, в большей степени соответствующее этому языку.

WasRun

```
def __init__(self, name):
    self.wasRun= None
    TestCase.__init__(self, name)
```

Наконец, замечаем, что метод `run()` использует только атрибуты суперкласса, значит, скорее всего, он должен располагаться в суперклассе. (Я всегда стараюсь размещать операции рядом с данными.)

TestCase

```
def __init__(self, name):
    self.name= name
def run(self):
    method = getattr(self, self.name)
    method()
```

Естественно, между выполнениями этих модификаций я каждый раз запускаю тесты для того, чтобы убедиться в том, что все работает как надо.

Нам надоело смотреть на то, как наша программа каждый раз печатает одно и то же: `None` и `1`. Используя разработанный нами механизм, мы можем теперь написать:

TestCaseTest

```
class TestCaseTest(TestCase):
    def testRunning(self):
        test= WasRun("testMethod")
        assert(not test.wasRun)
        test.run()
        assert(test.wasRun)
TestCaseTest("testRunning").run()
```

Вызов тестового метода

Вызов метода `setUp` перед обращением к методу

Вызов метода `tearDown` после обращения к методу

Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Сердцем этого теста являются операторы `print`, превращенные в выражения `assert`, таким образом, вы можете видеть, что выполненная нами процедура — это усложненная форма рефакторинга `Extract Method` (Выделение метода).

Я открою вам маленький секрет. Шажки, которыми мы двигались от теста к тесту в данной главе, выглядят смехотворно маленькими. Однако до того, как получить представленный результат, я пытался выполнить разработку более крупными шагами и потратил на это около шести часов (конечно, мне пришлось тратить дополнительное время на изучение тонкостей языка Python). Я два раза начинал с нуля и каждый раз думал, что мой код сработает, однако этого не происходило. Я понял, что попытка пропустить самый начальный, самый примитивный этап разработки, — это грубейшее нарушение принципов TDD.

Конечно же, вы не обязаны постоянно перемещаться такими лилипутскими шажками. После того как вы освоите TDD, вы сможете двигаться вперед более уверенно, реализуя между тестами значительно больший объем функциональности. Однако для того, чтобы в совершенстве освоить TDD, вы должны научиться перемещаться маленькими шажками тогда, когда это необходимо.

Далее мы планируем перейти к решению задачи обращения к методу `setUp()`. Однако в начале подведем итог.

В данной главе мы:

- поняли, как начать работу со смехотворно-малюсенького шага;
- реализовали функциональность путем создания «константного» фиксированного кода, а затем обобщения этого кода путем замены констант на переменные;
- использовали паттерн **Pluggable Selector** (Встраиваемый селектор) и дали себе обещание не использовать его вновь в течение как минимум четырех месяцев, так как он существенно усложняет анализ кода;
- начали работу над инфраструктурой тестирования маленькими шажками.

Сервируем стол (метод setUp)

Когда вы начинаете писать тесты, вы обнаруживаете, что действуете в рамках некоторой общей последовательности (Билл Уэйк (Bill Wake) придумал сокращение **3A** — *Arrange, Act, Assert*):

- вначале вы создаете некоторые тестовые объекты — *Arrange*;
- затем вы заставляете эти объекты действовать — *Act*;
- затем вы проверяете результаты их работы — *Assert*.

Вызов тестового метода

Вызов метода setUp перед обращением к методу

Вызов метода tearDown после обращения к методу

Метод tearDown должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Первый этап — *Arrange* — зачастую совпадает для нескольких разных тестов, в то время как второй и третий этапы для разных тестов отличаются. У меня есть два числа: 7 и 9. Если я сложу их, я должен получить 16; если я вычту второе из первого, я ожидаю получить -2; наконец, если я умножу их, я полагаю, должно получиться 63. Стимуляция и ожидаемые результаты отличаются, однако исходные данные одни и те же: два числа 7 и 9.

Если подобное повторяется раз за разом в разных масштабах (а ведь так оно и есть), мы должны подумать о том, каким образом можно оптимизировать создание тестовых объектов. Возникает конфликт между следующими двумя ограничениями:

- *Производительность*. Мы хотим, чтобы тесты срабатывали как можно быстрее. Отсюда следует, что если одни и те же объекты используются в нескольких тестах, мы хотели бы, чтобы создание этих объектов выполнялось всего один раз.
- *Изоляция*. Мы хотели бы, чтобы успех или неудача одного теста никак не влияли на срабатывание других тестов. Если несколько тестов используют

одни и те же объекты и если один из тестов меняет внутреннее состояние какого-либо объекта, результаты работы остальных тестов, скорее всего, изменятся.

Взаимозависимость между тестами приводит к одному весьма неприятному эффекту: если один тест перестает срабатывать, остальные десять тестов тоже перестают срабатывать, несмотря на то, что соответствующий им тестируемый код корректен. Характерен также и другой еще более неприятный эффект, когда порядок срабатывания тестов имеет значение: если я запускаю тест А перед тестом Б, оба теста срабатывают, если же я запускаю тест Б перед тестом А, тест А не выполняется. Или, еще кошмарнее, код, тестируемый тестом Б, некорректен, однако из-за того, что тест А запускается перед тестом Б, тест Б срабатывает.

Итак, мы хотим избежать взаимозависимости между тестами. Предположим, что мы можем сделать процедуру создания объектов достаточно быстрой. В этом случае мы могли бы создавать объекты для теста каждый раз перед выполнением очередного теста. Этот подход в замаскированном виде уже использовался нами в классе `WasRun`, в котором мы хотели, чтобы перед запуском теста флаг `wasRun` был установлен в положение «ложь». Напишем тест:

`TestCaseTest`

```
def testSetUp(self):
    test= WasRun("testMethod")
    test.run()
    assert(test.wasSetUp)
```

Чтобы запустить этот код, необходимо добавить в конец нашего файла строку `TestCaseTest("testSetUp").run()`. Интерпретатор вежливо сообщает нам, что атрибут с именем `wasSetUp` отсутствует. И немудрено, ведь мы пока еще не определили значение этого атрибута. Вот необходимый для этого код:

`WasRun`

```
def setUp(self):
    self.wasSetUp= 1
```

Однако метод `setUp()` должен быть откуда-то вызван. Обращение к методу `setUp()` — это работа класса `TestCase`. Добавим соответствующий код:

`TestCase`

```
def setUp(self):
    pass
def run(self):
    self.setUp()
    method = getattr(self, self.name)
    method()
```

Чтобы заставить тест работать, мы сделали целых два шага — это слишком много, особенно если учесть, что мы движемся почти вслепую. Сработает ли тест? Да, он сработал. Однако если вы хотите чему-то научиться, попробуйте придумать, каким образом мы можем заставить тест срабатывать, изменяя лишь по одному методу за один шаг.

Немедленно воспользуемся новым механизмом для того, чтобы сократить длину наших тестов. Прежде всего упростим класс `WasRun`, для этого перенесем процедуру установки флага `wasRun` в метод `setUp()`:

`WasRun`

```
def setUp(self):
    self.wasRun= None
    self.wasSetUp= 1
```

Теперь мы должны упростить метод `testRunning()` — его можно освободить от обязанности проверять состояние флага перед тем, как будет выполнен тестовый метод. Можем ли мы быть настолько уверенными в правильности нашего кода? Только при условии, что в наборе тестов присутствует тестовый метод `testSetUp()`. Это часто встречающийся паттерн — один тест может быть простым только в случае, если в системе присутствует другой тест, который корректно срабатывает:

`TestCaseTest`

```
def testRunning(self):
    test= WasRun("testMethod")
    test.run()
    assert(test.wasRun)
```

Мы также можем упростить сами тесты. В обоих случаях мы создаем экземпляр класса `WasRun`, а ведь задача создания тестовых объектов возлагается на фикстуру — именно об этом мы с вами говорили. Стало быть мы можем создать объект `WasRun` в методе `setUp()`, а затем использовать его в тестовых методах. Каждый тестовый метод выполняется в отдельном экземпляре класса `TestCaseTest`, поэтому два разных теста не могут быть взаимозависимы. (Мы исходим из того, что объект не будет взаимодействовать с внешним миром некоторым непредусмотренным уродливым способом, например путем изменения значений глобальных переменных.)

`TestCaseTest`

```
def setUp(self):
    self.test= WasRun("testMethod")
def testRunning(self):
    self.test.run()
    assert(self.test.wasRun)
def testSetUp(self):
    self.test.run()
    assert(self.test.wasSetUp)
```

~~Вызов тестового метода~~

Вызов метода `setUp` перед обращением к методу

Вызов метода `tearDown` после обращения к методу

Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Теперь сделаем так, чтобы после выполнения тестового метода обязательно выполнялся метод `tearDown()`.

В данной главе мы:

- решили, что на текущий момент простота тестов важнее, чем их производительность;
- написали тест для метода setUp() и реализовали этот метод;
- использовали метод setUp() для того, чтобы упростить тестируемый объект-контейнер теста;
- использовали метод setUp() для того, чтобы упростить тесты, тестирующие созданный нами тестовый объект (я же говорил, что временами это напоминает нейрохирургическую операцию на собственном мозге).

Убираем со стола (метод tearDown)

Вызов тестового метода

Вызов метода setUp перед обращением к методу

Вызов метода tearDown после обращения к методу

Метод tearDown должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Иногда для выполнения теста требуется выделить некоторые внешние ресурсы. Очевидно, что связанные с этим операции должны выполняться в теле метода setUp(). Если мы хотим, чтобы тесты были независимыми друг от друга, мы должны позаботиться об освобождении этих ресурсов. Для выполнения связанных с этим операций предлагаю использовать специальный метод tearDown(), который будет автоматически выполняться после завершения теста.

Каким образом можно протестировать выполнение метода tearDown()? Упрощенный способ — использовать еще один флаг. Однако все эти флаги начинают сбивать меня с толку. Если мы будем использовать флаги, мы упустим один очень важный аспект: метод setUp() должен быть выполнен непосредственно перед обращением к тестовому методу, а метод tearDown() — непосредственно после обращения к тестовому методу. Чтобы проверить это обстоятельство, я намерен изменить стратегию тестирования. Я предлагаю создать миниатюрный журнал, в котором будет отмечаться последовательность выполнения методов. Каждый метод будет добавлять в конец журнала соответствующую запись. Таким образом, просмотрев журнал, мы сможем установить порядок выполнения методов.

Вызов тестового метода

Вызов метода setUp перед обращением к методу

Вызов метода tearDown после обращения к методу

Метод tearDown должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Строка журнала в классе WasRun

WasRun

```
def setUp(self):
    self.wasRun= None
    self.wasSetUp= 1
    self.log= "setUp "
```

Теперь мы можем изменить метод `testSetUp()`, чтобы вместо флага он проверял содержимое журнала:

TestCaseTest

```
def testSetUp(self):
    self.test.run()
    assert("setUp " == self.test.log)
```

После этого мы можем удалить флаг `wasSetUp`. Мы также можем добавить в журнал запись о выполнении метода:

WasRun

```
def testMethod(self):
    self.wasRun= 1
    self.log= self.log + "testMethod "
```

В результате нарушается работа теста `testSetUp()`, так как в момент выполнения этого метода журнал содержит строку `"setUp testMethod "`. Изменяем ожидаемое значение:

TestCaseTest

```
def testSetUp(self):
    self.test.run()
    assert("setUp testMethod " == self.test.log)
```

Теперь этот тест выполняет работу обоих тестов, поэтому мы можем удалить `testRunning` и переименовать `testSetUp`:

TestCaseTest

```
def setUp(self):
    self.test = WasRun("testMethod")
def testTemplateMethod(self):
    self.test.run()
    assert("setUp testMethod " == test.log)
```

Мы используем экземпляр класса `WasRun` всего в одном месте, поэтому необходимо отменить добавленный ранее хитрый трюк, связанный с `setUp()`:

TestCaseTest

```
def testTemplateMethod(self):
    test= WasRun("testMethod")
    test.run()
    assert("setUp testMethod " == test.log)
```

Нет ничего страшного в том, что мы сделали рефакторинг, исходя из нескольких ранних соображений, а чуть позже отменили его — подобная ситуация складывается достаточно часто. Некоторые предпочитают подождать, пока у них на-

копиться достаточное количество оснований, указывающих на необходимость рефакторинга, иными словами, они оттягивают выполнение рефакторинга, чтобы быть полностью уверенными в его необходимости. Они поступают так потому, что не любят аннулировать результаты проделанной работы. Однако я предпочитаю не отвлекаться на рассуждения о том, не придется ли в будущем отменять тот или иной рефакторинг, который необходим мне в настоящем. Вместо этого я предпочитаю сосредоточиться на дизайне. По этой причине я рефлексивно делаю рефакторинг тогда, когда считаю нужным, ни капли не опасаясь того, что сразу же после этого мне, возможно, придется отменить его.

Вызов тестового метода

Вызов метода setUp перед обращением к методу

Вызов метода tearDown после обращения к методу

Метод tearDown должен быть вызван даже в случае, если тестовый метод не сработал.

Выполнение нескольких тестов

Отчет о результатах

Строка журнала в классе WasRun

Теперь мы готовы к реализации метода tearDown(). Ага! Опять я вас поймал! Теперь мы готовы к тестированию метода tearDown():

TestCaseTest

```
def testTemplateMethod(self):
    test = WasRun("testMethod")
    test.run()
    assert("setUp testMethod tearDown " == test.log)
```

Он не срабатывает. Чтобы заставить его работать, выполняем несложные доработки:

TestCase

```
def run(self, result):
    result.testStarted()
    self.setUp()
    method = getattr(self, self.name)
    method()
    self.tearDown()
```

WasRun

```
def setUp(self):
    self.log = "setUp "
def testMethod(self):
    self.log = self.log + "testMethod "
def tearDown(self):
    self.log = self.log + "tearDown "
```

Неожиданно мы получаем ошибку не в классе WasRun, а в классе TestCaseTest. У нас нет «пустой» реализации метода tearDown() в классе TestCase:

TestCase

```
def tearDown(self):
    pass
```

Мы начинаем получать пользу от инфраструктуры, которую мы разрабатываем. Замечательно! Никакого рефакторинга не требуется. Очевидная реализация, созданная нами после обнаружения ошибки, сработала, и код получился чистым.

Вызов тестового метода

Вызов метода `setUp` перед обращением к методу

Вызов метода `tearDown` после обращения к методу

Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Строка журнала в классе `WasRun`

Далее мы перейдем к формированию отчета о результатах выполнения тестов. Вместо использования встроенного в Python механизма обработки ошибок мы планируем реализовать и использовать собственный механизм наблюдения за срабатыванием тестов.

В данной главе мы:

- перешли от использования флагов к использованию журнала;
- создали тесты для метода `tearDown()` и реализовали этот метод с использованием нового механизма протоколирования;
- обнаружили проблему и, вместо того, чтобы возвращаться назад, смело исправили ошибку.

Учет и контроль

Вызов тестового метода

Вызов метода `setUp` перед обращением к методу

Вызов метода `tearDown` после обращения к методу

Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Строка журнала в классе `WasRun`

Метод `tearDown()` должен выполняться даже тогда, когда в процессе выполнения теста возникло исключение. Однако для того, чтобы добиться этого, мы должны перехватывать исключения. (Честно говоря, я уже пытался реализовать это, но у меня не получилось, и я вернул все на свои места). Если при реализации перехвата исключений мы допустим ошибку, мы можем не заметить этого, так как стандартный механизм доклада об исключениях не будет функционировать.

Работая в стиле TDD важно понимать, что особое значение имеет порядок, в котором вы реализуете тесты. Выбирая тест, над которым я буду работать дальше, я стараюсь выбрать тест, который, во-первых, послужит для меня источником новых знаний, а во-вторых, достаточно прост, чтобы я был уверен в том, что могу заставить его работать. Если я добиваюсь срабатывания этого теста, однако захожу в тупик при реализации следующего теста, я вполне могу выполнить откат назад на два шага. Было бы неплохо, если бы среда разработки оказывала мне в этом помощь. Например, было бы неплохо, если бы в момент срабатывания всех тестов автоматически создавалась резервная копия всего исходного кода, с которым я работаю.

После выполнения всех тестов мы хотели бы получить информацию о том, как они сработали, например: «запущено 5, не сработало 2: `TestCaseTest.testFooBar — ZeroDivideException`, `MoneyTest.testNegation — AssertionError`». Если тесты перестают выполняться или результаты перестают отображаться на экране, по крайней мере мы сможем обнаружить ошибку. Однако наша инфраструктура не обязана знать обо всех разработанных тестах.

Пусть метод `TestCase.run()` возвращает объект класса `TestResult`, который будет хранить в себе результаты выполнения теста (в начале тест будет только один, однако позже мы усовершенствуем этот объект).

TestCaseTest

```
def testResult(self):
    test= WasRun("testMethod")
    result= test.run()
    assert("1 run, 0 failed" == result.summary())
```

Начнем с поддельной реализации:

TestResult

```
class TestResult:
    def summary(self):
        return "1 run, 0 failed"
```

Теперь сделаем так, что объект класса `TestResult` возвращается в результате выполнения метода `TestCase.run()`:

TestCase

```
def run(self):
    self.setUp()
    method = getattr(self, self.name)
    method()
    self.tearDown()
    return TestResult()
```

Теперь, когда все тесты сработали, мы можем сделать реализацию метода `summary()` реальной. Как и раньше, будем двигаться маленькими шажками. Для начала заменим количество сработавших тестов символьной константой:

TestResult

```
def __init__(self):
    self.runCount= 1
def summary(self):
    return "%d run, 0 failed" % self.runCount
```

(Оператор `%` в языке Python является аналогом функции `sprintf` языка C.) Однако `runCount` не может быть константой, это должна быть переменная, значение которой вычисляется, исходя из количества выполненных тестов. Мы можем инициализировать эту переменную значением 0, а затем увеличивать ее значение на единицу каждый раз при выполнении очередного теста.

TestResult

```
def __init__(self):
    self.runCount= 0
def testStarted(self):
    self.runCount= self.runCount + 1
def summary(self):
    return "%d run, 0 failed" % self.runCount
```

Теперь мы должны позаботиться о вызове этого нового метода:

TestCase

```
def run(self):
    result= TestResult()
    result.testStarted()
    self.setUp()
    method = getattr(self, self.name)
    method()
    self.tearDown()
    return result
```

Мы могли бы преобразовать константную строку "0", обозначающую количество несработавших тестов, в переменную точно таким же способом, каким мы сделали это с переменной `runCount`, однако существующие тесты этого не требуют. Поэтому мы пишем новый тест:

TestCaseTest

```
def testFailedResult(self):
    test= WasRun("testBrokenMethod")
    result= test.run()
    assert("1 run, 1 failed", result.summary)
```

Здесь:

WasRun

```
def testBrokenMethod(self):
    raise Exception
```

Вызов тестового метода

Вызов метода `setUp` перед обращением к методу

Вызов метода `tearDown` после обращения к методу

Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Строка журнала в классе `WasRun`

Отчет о не сработавших тестах

Мы немедленно обращаем внимание на то, что исключение, генерируемое в рамках метода `WasRun.testBrokenMethod()`, не перехватывается. Нам хотелось бы перехватить это исключение и в отчете о результатах тестирования отметить, что тест не сработал. Мы добавляем соответствующий пункт в список задач.

Подведем итог. Мы:

- разработали поддельную реализацию и начали поэтапно делать ее реальной путем замены констант переменными;
- написали еще один тест;
- когда тест не сработал, написали еще один тест меньшего масштаба для того, чтобы обеспечить срабатывание сбойного теста.

Обработка несработавшего теста

Вызов тестового метода

Вызов метода `setUp` перед обращением к методу

Вызов метода `tearDown` после обращения к методу

Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Строка журнала в классе `WasRun`

Отчет о не сработавших тестах

Напишем еще один тест меньшего масштаба, демонстрирующий, что при обнаружении несработавшего теста наша инфраструктура распечатывает на экране корректный результат:

`TestCaseTest`

```
def testFailedResultFormatting(self):
    result= TestResult()
    result.testStarted()
    result.testFailed()
    assert("1 run, 1 failed" == result.summary())
```

В данном тесте фигурируют два новых метода: `testStarted()` и `testFailed()`. Первый из них должен вызываться в начале работы теста, а второй — в случае, если тест не срабатывает. Тестовый метод `testFailedResultFormatting()` предполагает, что если эти два метода вызываются в указанном порядке и в указанных случаях, отчет о результате тестирования должен выглядеть корректно. Если мы заставим этот тестовый метод работать, наша проблема сведется к тому, чтобы обеспечить обращение к методу `testStarted()` в начале выполнения теста и к методу `testFailed()` в случае, если тест не сработал.

Чтобы реализовать функциональность методов `testStarted()` и `testFailed()`, воспользуемся двумя счетчиками: счетчиком запущенных тестов и счетчиком несработавших тестов:

TestResult

```
def __init__(self):
    self.runCount= 0
    self.errorCount= 0
def testFailed(self):
    self.errorCount= self.errorCount + 1
```

Если счетчик будет работать корректно (мы должны были бы это протестировать, однако кофе, похоже, ударил мне в голову), значит, отчет будет распечатан корректно:

TestResult

```
def summary(self):
    return "%d run, %d failed" % (self.runCount, self.failureCount)
```

Теперь можно предположить, что если обращение к методу testFailed() будет выполнено корректно, мы получим на экране желаемый результат. Но где следует осуществить обращение к этому методу? В обработчике исключения, перехваченного в процессе выполнения тестового метода:

TestCase

```
def run(self):
    result= TestResult()
    result.testStarted()
    self.setUp()
    try:
        method = getattr(self, self.name)
        method()
    except:
        result.testFailed()
    self.tearDown()
    return result
```

В этом коде присутствует одно тонкое место: если исключение возникает во время исполнения метода setUp(), оно не будет перехвачено. Это означает, что в определенной степени тесты зависят друг от друга. Однако прежде чем мы сможем изменить код, мы должны написать еще один тест. (Я научил мою старшую дочку Бетани программировать в стиле TDD, когда ей было 12 лет. Это самый первый стиль программирования, с которым она познакомилась. Она не умеет программировать иначе и абсолютно уверена в том, что добавлять в программу новый функциональный код можно только в случае, если существует не срабатывающий тест. Однако более опытные программисты, как правило, вынуждены напоминать себе о том, что сначала необходимо писать тесты, а потом — код, заставляющий их выполняться.) Я оставляю этот следующий тест и его реализацию вам в качестве самостоятельного упражнения (мои пальцы опять устали).

Вызов тестового метода

Вызов метода `setUp` перед обращением к методу

Вызов метода `tearDown` после обращения к методу

Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Строка журнала в классе `WasRun`

Отчет о не сработавших тестах

Перехват и отчет об ошибках `setUp`

Далее мы сделаем так, чтобы наша инфраструктура смогла запустить несколько тестов подряд.

В данной главе мы:

- заставили наш тест меньшего масштаба сработать;
- заново приступили к реализации более крупного теста;
- заставили крупный тест сработать, воспользовавшись механизмом, реализованным благодаря маленькому тесту;
- обратили внимание на потенциальную проблему, но вместо того, чтобы немедленно браться за ее решение, добавили соответствующую пометку в список задач.

Оформляем тесты в набор

Вызов тестового метода

Вызов метода `setUp` перед обращением к методу

Вызов метода `tearDown` после обращения к методу

Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Строка журнала в классе `WasRun`

Отчет о не сработавших тестах

Перехват и отчет об ошибках `setUp`

Мы не можем оставить работу над `xUnit`, не реализовав класс `TestSuite`, представляющий собой набор тестов. Конец нашего файла, где мы запускаем все наши тесты, выглядит весьма неопрятно:

```
print TestCaseTest("testTemplateMethod").run().summary()
print TestCaseTest("testResult").run().summary()
print TestCaseTest("testFailedResultFormatting").run().summary()
print TestCaseTest("testFailedResult").run().summary()
```

Дублирование — это всегда плохо, за исключением случаев, когда вы используете его в качестве мотивации для поиска недостающего элемента дизайна. В данном случае нам хотелось бы сгруппировать тесты и запустить их при помощи одной команды. (Мы приложили массу усилий для того, чтобы обеспечить изоляцию тестов, однако все эти усилия не окупят себя, если мы будем запускать тесты по одному.) Еще одна хорошая причина, по которой было бы неплохо реализовать `TestSuite`, заключается в том, что этот класс хорошо демонстрирует использование паттерна `Composite` (Компоновщик), — мы хотим, чтобы набор тестов вел себя в точности так же, как единичный тест.

Мы хотим обладать возможностью создать набор тестов (объект `TestSuite`), добавить в него несколько тестов и получить общий результат исполнения всех этих тестов:

TestCaseTest

```
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.add(WasRun("testBrokenMethod"))
    result= suite.run()
    assert("2 run, 1 failed" == result.summary())
```

Чтобы заставить тест работать, создадим в объекте `TestSuite` список тестов и реализуем метод `add()`, который просто добавляет тест, переданный в качестве аргумента, в список:

TestSuite

```
class TestSuite:
    def __init__(self):
        self.tests= []
    def add(self, test):
        self.tests.append(test)
```

(В языке Python оператор `[]` создает пустую коллекцию.)

Однако с реализацией метода `run()` возникают проблемы. Мы хотим, чтобы результаты срабатывания всех тестов накапливались в едином объекте класса `TestResult`. Таким образом, мы можем написать следующий код:

TestSuite

```
def run(self):
    result= TestResult()
    for test in tests:
        test.run(result)
    return result
```

Здесь оператор цикла «`for test in tests`» выполняет итерации по всем элементам последовательности `tests`, иницилируя соответствующий тестовый вариант. Однако паттерн `Composite` (Компоновщик) подразумевает, что набор объектов должен обладать точно таким же интерфейсом, каким обладает отдельный объект. Если мы передаем параметр методу `TestCase.run()`, значит, мы должны передавать точно такой же параметр методу `TestSuite.run()`. Можно использовать одну из трех альтернатив:

- Воспользоваться встроенным в язык Python механизмом значения параметра по умолчанию. К сожалению, значение параметра по умолчанию вычисляется во время компиляции, но не во время исполнения, а мы не хотим повторно использовать один и тот же объект `TestResult`.
- Разделить метод на две части — одна создает объект `TestResult`, а вторая выполняет тест, используя переданный ей объект `TestResult`. Я не могу придумать хороших имен для двух частей метода, а это означает, что данная стратегия не является самой лучшей.
- Создавать объекты `TestResult` в вызывающем коде.

Мы будем создавать объекты `TestResult` в вызывающем коде. Этот паттерн называется `Collecting Parameter` (Накапливающий параметр).

TestCaseTest

```
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.add(WasRun("testBrokenMethod"))
    result= TestResult()
    suite.run(result)
    assert("2 run, 1 failed" == result.summary())
```

При использовании данного подхода метод `run()` не возвращает никакого явного значения:

TestSuite

```
def run(self, result):
    for test in tests:
        test.run(result)
```

TestCase

```
def run(self, result):
    result.testStarted()
    self.setUp()
    try:
        method = getattr(self, self.name)
        method()
    except:
        result.testFailed()
    self.tearDown()
```

Теперь мы можем облагородить обращение к тестовым методам в конце файла:

```
suite= TestSuite()
suite.add(TestCaseTest("testTemplateMethod"))
suite.add(TestCaseTest("testResult"))
suite.add(TestCaseTest("testFailedResultFormatting"))
suite.add(TestCaseTest("testFailedResult"))
suite.add(TestCaseTest("testSuite"))
result= TestResult()
suite.run(result)
print result.summary()
```

Вызов тестового метода

Вызов метода `setUp` перед обращением к методу

Вызов метода `tearDown` после обращения к методу

Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Строка журнала в классе `WasRun`

Отчет о не сработавших тестах

Перехват и отчет об ошибках `setUp`

Создать объект `TestSuite` автоматически на основе класса `TestCase`

Здесь слишком много дублирования, от которого можно избавиться, если обеспечить способ конструирования набора тестов автоматически, исходя из предоставленного класса `TestCase`.

Однако вначале восстановим корректное выполнение четырех не сработавших тестов (эти тесты используют старый интерфейс функции `run()` без аргументов):

TestCaseTest

```
def testTemplateMethod(self):
    test= WasRun("testMethod")
    result= TestResult()
    test.run(result)
    assert("setUp testMethod tearDown " == test.log)
def testResult(self):
    test= WasRun("testMethod")
    result= TestResult()
    test.run(result)
    assert("1 run, 0 failed" == result.summary())
def testFailedResult(self):
    test= WasRun("testBrokenMethod")
    result= TestResult()
    test.run(result)
    assert("1 run, 1 failed" == result.summary())
def testFailedResultFormatting(self):
    result= TestResult()
    result.testStarted()
    result.testFailed()
    assert("1 run, 1 failed" == result.summary())
```

Обратите внимание, что каждый из тестов создает экземпляр класса `TestResult` — эту операцию можно выполнить однократно внутри метода `setUp()`. Благодаря реализации этой идеи мы упростим тесты, однако сделаем их несколько более сложными в прочтении:

TestCaseTest

```
def setUp(self):
    self.result= TestResult()
def testTemplateMethod(self):
    test= WasRun("testMethod")
    test.run(self.result)
    assert("setUp testMethod tearDown " == test.log)
def testResult(self):
    test= WasRun("testMethod")
    test.run(self.result)
    assert("1 run, 0 failed" == self.result.summary())
def testFailedResult(self):
    test= WasRun("testBrokenMethod")
    test.run(self.result)
    assert("1 run, 1 failed" == self.result.summary())
def testFailedResultFormatting(self):
    self.result.testStarted()
```

```

self.result.testFailed()
assert("1 run, 1 failed" == self.result.summary())
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.add(WasRun("testBrokenMethod"))
    suite.run(self.result)
    assert("2 run, 1 failed" == self.result.summary())

```

Вызов тестового метода

Вызов метода `setUp` перед обращением к методу

Вызов метода `tearDown` после обращения к методу

Метод `tearDown` должен быть вызван даже в случае, если тестовый метод не сработал

Выполнение нескольких тестов

Отчет о результатах

Строка журнала в классе `WasRun`

Отчет о не сработавших тестах

Перехват и отчет об ошибках `setUp`

Создать объект `TestSuite` автоматически на основе класса `TestCase`

Все эти бесчисленные ссылки `self` выглядят ужасно, однако без этого в языке Python никак не обойтись. Если бы этот язык изначально был объектно-ориентированным, наверное, в этих ссылках не было бы надобности, а ссылки на глобальные переменные требовали бы квалификации. Однако язык Python изначально является интерпретируемым языком с добавленной в него поддержкой объектов (надо отметить, что поддержка объектов в этом языке реализована великолепно). В результате по умолчанию переменные считаются глобальными, а явные ссылки на `self` — необходимыми.

Я оставляю реализацию оставшихся пунктов вам в качестве упражнения. Надеюсь, обретенные навыки работы в стиле TDD помогут вам.

Чтобы подвести итог, напомню, что в данной главе мы:

- написали тест для класса `TestSuite`;
- написали часть реализации, однако не добились срабатывания тестов — это нарушение правил (я уверен, что существует простая поддельная реализация, которая заставила бы тесты срабатывать, благодаря чему мы могли бы выполнять рефакторинг, имея перед глазами зеленую полоску, однако сейчас я не хочу думать на эту тему);
- изменили интерфейс метода `run()`, благодаря чему набор тестов можно использовать точно так же, как и отдельный тест, — в результате тесты, наконец, сработали;
- выполнили рефакторинг имеющихся тестов — переместили общий код создания объекта результатов в метод `setUp()`.

Ретроспектива xUnit

Если перед вами встала задача разработки своей собственной инфраструктуры тестирования, методика, описанная в части II данной книги, послужит вам руководством. Не следует слишком много внимания уделять деталям реализации, — значительно больший интерес представляют тесты. Если вы напишете код, обеспечивающий срабатывание представленных здесь тестов, значит, в вашем распоряжении окажется минимальная инфраструктура тестирования, пригодная для запуска тестов в условиях изоляции и обеспечивающая композицию тестов. Вы сможете приступить к разработке программного кода в стиле TDD.

На момент написания данной книги инфраструктура тестирования xUnit адаптирована для более чем 30 языков программирования. Язык, на котором вы программируете, скорее всего, уже обладает своей собственной реализацией xUnit. Однако даже если кто-то уже делал это до вас, возможно, будет лучше, если вы попытаетесь разработать свою собственную новую версию xUnit самостоятельно. Существует две важных причины:

- *Контроль над реализацией.* основополагающая характеристика xUnit — это простота. Мартин Фаулер (Martin Fowler) сказал: «Никогда в истории программной индустрии еще не было случая, чтобы столь многие разработчики были обязаны столь немногочисленному количеству строк программного кода». На мой взгляд, некоторые реализации xUnit к настоящему времени стали слишком большими и сложными. Если вы разработаете собственную версию xUnit, то получите инструмент, который вы будете контролировать в полном объеме.
- *Обучение.* Когда я сталкиваюсь с необходимостью изучить новый язык программирования, я приступаю к реализации xUnit. Когда я добиваюсь срабатывания первых восьми-десяти тестов, я овладеваю навыками работы с основными конструкциями и возможностями нового для меня языка.

Когда вы начнете работать с xUnit, вы обнаружите, что существует значительная разница между не сработавшими выражениями `assert` и ошибками других типов, возникающими в процессе выполнения тестов. В отличие от остальных ошибок выражения `assert` требуют больше времени для отладки. Из-за этого боль-

шинство реализаций xUnit отличает свои операторов assert от всех остальных ошибок: в рамках GUI зачастую информация об ошибках отображается в начале списка.

Инфраструктура JUnit объявляет простой интерфейс Test, который реализуется классами TestCase и TestSuite. Если вы хотите создать тестовый класс, который мог бы взаимодействовать со стандартными средствами тестирования, встроенными в JUnit, вы можете реализовать функции интерфейса Test самостоятельно:

```
public interface Test {  
    public abstract int countTestCases();  
    public abstract void run(TestResult result);  
}
```

В языках с оптимистическим (динамическим) приведением типов можно даже не объявлять о поддержке этого интерфейса — достаточно реализовать входящие в его состав операции. При использовании языка сценариев Сценарий может ограничивать реализацию countTestCases() возвратом единицы и выполнять проверку TestResult на отказ, а вы можете выполнять ваши сценарии с обычными TestCases.

Часть III

Паттерны для разработки через тестирование

Далее следуют «величайшие хиты» — паттерны для разработки, основанной на тестировании. Некоторые из них являются эффективными приемами работы в стиле TDD, другие являются паттернами проектирования, наконец, третьи — это рефакторинги. Третья часть книги является коллекцией справочного материала, необходимого как для освоения представленных в книге примеров, так и для самостоятельного совершенствования навыков работы в стиле TDD. Здесь представлены сведения, которые помогут вам лучше понять смысл примеров, рассмотренных в первых двух частях книги, а также подогреют ваш интерес и стимулируют обратиться к дополнительной информации, которую следует искать в других источниках.

Паттерны разработки, основанной на тестах

Прежде чем приступить к обсуждению эффективных методов тестирования, давайте попробуем ответить на несколько стратегических вопросов:

- Что такое тестирование?
- Когда мы выполняем тестирование?
- Какая логика нуждается в тестировании?
- Какие данные нуждаются в тестировании?

Тест

Каким образом следует тестировать программное обеспечение? При помощи автоматических тестов.

Тестировать означает *проверять*. Ни один программист не считает работу над некоторым фрагментом кода завершённой, не проверив перед этим его работоспособность (исключение составляют либо слишком самоуверенные, либо слишком небрежные программисты, но я надеюсь, что среди читателей данной книги таких нет). Однако, если вы тестируете свой код, это не означает, что у вас *есть* тесты. *Тест* — это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода. Когда программист проверяет работоспособность разработанного им кода, он выполняет тестирование вручную: нажимает кнопки на клавиатуре и смотрит на результат работы программы, отображаемый на экране. В данном контексте тест состоит из двух этапов: стимулирование кода и проверка результатов его работы. *Автоматический тест* выполняется автоматически: вместо программиста стимулированием кода и проверкой результатов

занимается компьютер, который отображает на экране результат выполнения теста: *код работоспособен* или *код неработоспособен*. В чем состоит принципиальное отличие автоматического теста от тестирования кода вручную?

На рис. 25.1 представлена диаграмма взаимовлияния между стрессом и тестированием (она напоминает диаграммы Герри Вейнберга (Gerry Weinberg) в его книге *Quality Software Management*). Стрелка между узлами диаграммы означает, что увеличение первого показателя влечет за собой увеличение второго показателя. Стрелка с кружком означает, что увеличение первого показателя влечет за собой уменьшение второго показателя.

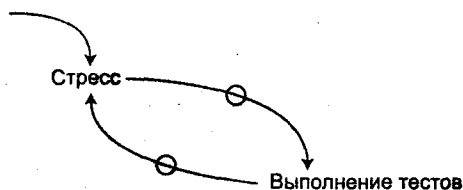


Рис. 25.1. Зловещая спираль «нет времени для тестирования»

Что происходит, когда уровень стресса возрастает?

Чем больший стресс вы ощущаете, тем меньше вы тестируете разрабатываемый код. Чем меньше вы тестируете разрабатываемый код, тем больше ошибок вы допускаете. Чем больше ошибок вы допускаете, тем выше уровень стресса, который вы ощущаете. Получается замкнутый круг с позитивной обратной связью: рост стресса приводит к росту стресса.

Что надо сделать, чтобы разорвать этот зловещий цикл? Необходимо либо добавить новый элемент, либо заменить один из элементов, либо изменить стрелки. Попробуем заменить тестирование на автоматическое тестирование.

«Я только что внес в код изменение. Нарушил ли я тем самым его работоспособность?» Рисунок 25.1 показывает динамику в действии. При использовании автоматического тестирования, когда я начинаю ощущать стресс, я запускаю тесты. Тесты превращают страх в скуку. «Нет, я ничего не сломал. Тесты по-прежнему показывают зеленый цвет.» Чем больший стресс я ощущаю, тем чаще я запускаю тесты. Выполнив тесты, я успокаиваюсь. Когда я спокоен, я допускаю меньше ошибок, а это ведет к снижению уровня стресса.

«Да поймите же вы, что у нас нет времени на тестирование!» — теперь эта жалоба перестает быть актуальной, так как выполнение автоматического тестирования почти не требует времени. Компьютер выполняет тестирование значительно быстрее, чем человек. Если вы не выполняете тестирования, вы опасаетесь за корректность кода. Используя автоматическое тестирование, вы можете выбирать удобный для вас уровень страха.

Должны ли вы запустить тест сразу же после его написания, несмотря на то что вы полностью уверены в том, что он не сработает? Конечно, вы можете этого не делать. Но... Приведу поучительный пример. Некоторое время назад я работал вместе с двумя очень умными молодыми программистами над реализацией транзакций, выполняемых внутри оперативной памяти (это чрезвычайно мощная тех-

нология, поддержка которой должна быть добавлена во все современные языки программирования). Перед нами встал вопрос: каким образом мы можем реализовать откат транзакции, если начали выполнение транзакции, затем изменили значение нескольких переменных, а затем нарушили ее выполнение (транзакция была уничтожена сборщиком мусора)? Достаточно просто, чтобы проверить способности малоопытных разработчиков. Отойдите в сторону и смотрите, как работает мастер. Вот тест. Теперь подумаем над тем, как заставить его работать. Мы приступили к написанию кода.

Прошло два часа. Два часа, заполненных мучениями и разочарованиями (в большинстве случаев при возникновении ошибки среда разработки давала фатальный сбой и ее приходилось перезапускать). Испробовав множество методов решения проблемы, мы отменили все внесенные нами в код модификации, восстановили изначальное состояние системы и вернулись к тому, с чего начали: заново написали тот самый тест. На удачу запустили его. Он сработал. Это было потрясение... Оказалось, что механизм поддержки транзакций на самом деле не менял значения переменных до тех пор, пока транзакция не считалась полностью выполненной. Надеюсь, теперь вы сами решите для себя, нужно ли вам запускать тесты сразу же после их написания.

Изолированный тест (Isolated Test)

Каким образом исполнение одного теста может повлиять на выполнение другого теста? Никаким.

Я впервые столкнулся с автоматическим тестированием, когда был еще молодым программистом. В то время в компании с другими программистами (привет, Джоси, привет Джон!) я занимался разработкой отладчика с графическим интерфейсом. Для контроля корректности его работы использовалась длинная серия автоматических тестов. Это был набор автоматически выполняемых тестов, основанных на взаимодействии с графическим интерфейсом (специальная программа перехватывала нажатия на клавиши и события, связанные с мышью, а затем автоматически воспроизводила их, имитируя тем самым работу пользователя с программой). Для исполнения всей серии этих тестов требовалось длительное время, поэтому обычно тесты запускались вечером, перед уходом с работы, и выполнялись в течение почти всей ночи. Каждое утро, когда я приходил на работу, я видел на своем стуле аккуратно сложенную пачку листов, на которых были распечатаны результаты ночного тестирования. (Привет, Эл!) В хорошие дни это мог быть всего один лист, на котором было написано, что ничего не поломалось. В плохие дни на стуле могла лежать огромная кипа бумаги — по одному листу на каждый «сломанный» тест. Постепенно я стал пугаться вида листов бумаги на моем стуле, — если я приходил на работу и видел на своем стуле кипу бумажных листов, меня немедленно бросало в дрожь.

Работая в таком стиле, я пришел к двум важным выводам. Во-первых, тесты должны выполняться достаточно быстро, чтобы я мог запускать их самостоятельно и делать это достаточно часто. В этом случае я мог бы обнаруживать ошибки

раньше, чем кто-либо другой. Во-вторых, спустя некоторое время я заметил, что огромная кипа бумаги далеко не всегда означает огромную кучу проблем. Чаще оказывалось, что в самом начале выполнения тестов один из них не срабатывал, оставляя систему в непредсказуемом состоянии, в результате следующий тест тоже не срабатывал, а за ним и многие другие — по цепочке.

В то время мы пытались решить эту проблему, автоматически перезапуская систему перед выполнением каждого теста, однако для этого требовалось слишком большое время. Именно тогда у меня возникла еще одна хорошая мысль: тестирование можно выполнять на более низком уровне: вовсе не обязательно, чтобы каждый из тестов выполнялся в отношении всего приложения в целом. Чтобы убедиться в работоспособности всего приложения, достаточно протестировать каждую из его составных частей. Тестирование части приложения можно выполнить быстрее, чем тестирование всего приложения. Однако самый важный вывод состоял в том, что выполнение одного теста никоим образом не должно влиять на выполнение другого теста. Тесты должны полностью игнорировать друг друга. Если один из тестов не срабатывает, это значит, что в программе присутствует одна проблема. Если не срабатывают два теста, значит, в программе присутствуют две проблемы.

Если тесты изолированы друг от друга, значит, порядок их выполнения не имеет значения. Если я хочу выполнить не все, а некоторое подмножество тестов, я не должен беспокоиться о том, что некоторый тест не сработает только потому, что некоторый другой тест не был предварительно запущен.

Производительность является основной причиной, по которой предлагается делать данные общими для нескольких тестов. Требование изоляции тестов принуждает вас разделить проблему на несколько ортогональных измерений, благодаря чему формирование среды для каждого из тестов выполняется достаточно просто и быстро. Иногда, чтобы выполнить подобное разделение, вам приходится прикладывать значительные усилия. Если вы хотите, чтобы разрабатываемое вами приложение можно было протестировать при помощи набора изолированных друг от друга тестов, вы должны «собрать» это приложение из множества относительно небольших взаимодействующих между собой объектов. Я всегда знал о том, что это неплохая идея, и всегда радовался, когда мне удавалось реализовать ее на деле, однако я не был знаком ни с одной методикой, которая позволяла бы мне *регулярно* воплощать эту идею в жизнь. Ситуация изменилась в лучшую сторону после того, как я стал писать **изолированные тесты**.

Список тестов (Test List)

Что необходимо тестировать? Прежде чем начать, запишите на листке бумаги список всех тестов, которые вам потребуются. Чтобы успешно справляться с программистским стрессом, вы должны постоянно соблюдать важное правило: никогда не делайте шага вперед до тех пор, пока не узнаете, в каком месте ваша нога должна коснуться земли. Приступая к сеансу программирования, определите, **какие задачи вы намерены решить в ходе этого сеанса**.

В рамках весьма распространенной стратегии предлагается держать все в голове. Я пробовал использовать этот подход в течение нескольких лет, однако постоянно сталкивался с одной и той же проблемой. По мере того как я работаю, передо мной возникают все новые и новые задачи, которые необходимо решить. Чем больше задач предстоит решить, тем меньше внимания я уделяю тому, над чем я работаю. Чем меньше внимания я уделяю тому, над чем я работаю, тем меньше задач мне удастся решить. Чем меньше задач мне удастся решить, тем больше вещей, о которых мне придется помнить в процессе работы. Замкнутый круг.

Я пытался игнорировать случайные элементы списка и программировать по прихоти, однако это не позволяет разорвать замкнутый круг.

Я выработал привычку записывать на листик бумаги все задачи, которые я планирую решить в течение нескольких следующих часов. Этот листик постоянно лежит рядом с моим компьютером. Похожий список задач, которые я планирую решить в течение ближайшей недели или ближайшего месяца, приколот к стене над моим компьютером. Если я записал все эти задачи на бумагу, я уверен в том, что я ничего не забуду. Если передо мной возникает новая задача, я быстро и осознанно решаю, к какому списку («сейчас» или «позднее») она принадлежит и нужно ли вообще ей заниматься.

В контексте разработки, основанной на тестировании, список задач — это список тестов, которые мы планируем реализовать. Прежде всего включите в список примеры всех операций, которые требуется реализовать. Далее, для каждой из операций, которые еще не существуют, внесите в список нуль-версию этой операции. Наконец, перечислите в списке все рефакторинги, которые потребуются выполнить для того, чтобы в конце сеанса программирования получить чистый код.

Но зачем записывать тесты на бумагу, когда можно записать их один за другим в виде готового тестирующего кода? Существует пара причин, по которым я не рекомендую заниматься массовым созданием тестов. Во-первых, каждый из тестов создает некоторую инерцию, мешающую выполнению рефакторинга. Чем больше тестов, тем больше эта инерция. Согласитесь, что выполнить рефакторинг кода, для тестирования которого написаны два теста, сложнее, чем выполнить рефакторинг кода, для тестирования которого написан всего один тест. Конечно, существуют инструменты автоматизированного рефакторинга, которые упрощают эту задачу (например, специальный пункт в меню осуществляет модификацию имени переменной в строке, где она объявляется, и во всех местах, в которых эта переменная используется.) Однако представьте, что вы написали десять тестов для некоторого метода и *после* этого обнаружили, что порядок аргументов метода следует изменить на обратный. В подобной ситуации вам придется приложить существенные усилия для того, чтобы заставить себя сделать рефакторинг. Во-вторых, чем больше не работающих тестов, тем дольше путь к зеленой полосе. Если перед вами десять «сломанных» тестов, зеленую полосу вы увидите еще не скоро. Если вы хотите быстро получить перед собой зеленую полосу, вы должны выкинуть все десять тестов. Если же вы хотите добиться срабатывания всех этих тестов, вы будете вынуждены долгое время смотреть на красную полосу. Если вы настолько приучены к опрятности и аккуратности

кодирования, что не можете позволить себе даже дойти до туалета, если полоса красная, значит, вам предстоит серьезное испытание.

Консервативные скалолазы придерживаются одного важного правила. У человека есть две руки и две ноги, всего четыре конечности, которыми он может цепляться за скалу. В любой момент времени по крайней мере три конечности должны быть надежно сцеплены со скалой. Динамические перемещения, когда скалолаз перемещает с места на место одновременно две конечности, считаются чрезвычайно опасными. Методика TDD в чистом виде подразумевает использование похожего принципа: в любой момент времени вы должны быть не дальше одного изменения от зеленой полосы.

По мере того как вы заставляете тесты срабатывать, перед вами будет возникать необходимость реализации новых тестов. Заносите эти новые тесты в список задач. То же самое относится и к рефакторингам.

«Это выглядит ужасно.»

«<вдох> Добавьте это в список. Мы вернемся к этому перед тем, как завершить работу над задачей.»

Необходимо позаботиться о пунктах, оставшихся в списке на момент завершения сеанса программирования. Если на самом деле вы находитесь в середине процесса реализации некоторой функциональности, воспользуйтесь этим же списком позднее. Если вы обнаружили необходимость выполнения более крупномасштабных рефакторингов, выполнить которые в настоящий момент не представляется возможным, внесите их в список «позднее». Я не могу припомнить ситуации, когда мне приходилось переносить реализацию теста в список «позднее». Если я могу придумать тест, который может не сработать, реализация этого теста важнее, чем выпуск кода, над которым я работаю.

Вначале тест (Test First)

Когда нужно писать тесты? Перед тем как вы приступите к написанию тестируемого кода.

Вы не должны выполнять тестирование после. Конечно, вашей основной целью является работающая функциональность. Однако вам необходима методика формирования дизайна, вам нужен метод контроля над объемом работ.

Рассмотрим обычную диаграмму взаимовлияния между стрессом и тестированием (не путать со стресс-тестированием — это совершенно другая вещь): верхний узел — это стресс; он соединяется с тестированием (нижний узел) при помощи негативной связи; тестирование, в свою очередь, при помощи негативной связи соединяется со стрессом. Эта диаграмма представлена в первом разделе данной главы. Чем больший стресс вы ощущаете, тем меньше вы выполняете тестирование. Когда вы знаете, что выполняемого тестирования недостаточно, у вас повышается уровень стресса. Замкнутый цикл с позитивной обратной связью. Что можно сделать, чтобы разорвать его?

Что если мы всегда будем выполнять тестирование вначале? В этом случае мы можем инвертировать диаграмму: вверху будет располагаться узел «Предвари-

тельное тестирование», который при помощи негативной связи будет соединяться с расположенным внизу узлом «Стресс», который, в свою очередь, также при помощи негативной связи будет соединяться с узлом «Предварительное тестирование».

Когда мы начинаем работу с написания тестов, мы снижаем стресс, а значит, тестирование может быть выполнено более тщательно. Конечно, уровень стресса зависит от множества других факторов, стало быть можно допустить, что возникнет ситуация, в которой из-за высокого уровня стресса нам все-таки придется отказаться от тестирования. Однако помимо прочего предварительное тестирование является мощным инструментом формирования дизайна и средством контроля над объемом работы. Значит, скорее всего, мы будем выполнять тестирование даже при среднем уровне стресса.

Вначале оператор assert (Assert First)

Когда следует писать оператор assert¹? Попробуйте писать их в первую очередь. Неужели вам не нравится самоподobie?

- С чего следует начать построение системы? С формулировки пожеланий² о том, как должна работать система, полученная в результате вашей работы.
- С чего следует начать разработку некоторой функциональности? С написания тестов, которые должны сработать тогда, когда код будет полностью завершен.
- С чего начать написание теста? С операторов assert, которые должны быть выполнены в рамках теста.

С этой методикой познакомил меня Джим Ньюкирк (Jim Newkirk). Когда я начинаю разработку теста с операторов assert, я ощущаю мощный упрощающий эффект. Когда вы пишете тест, вы решаете несколько проблем в одно и то же время, даже несмотря на то, что при этом вам не нужно думать о реализации.

- Частью чего является новая функциональность? Является ли она модификацией существующего метода? Является ли она новым методом существующего класса? Является ли она методом с известным именем, но реализованным в другом месте? А может быть новая функциональность — это новый класс?
- Какие имена присвоить используемым элементам?
- Каким образом можно проверить правильность результата работы функциональности?

¹ В переводе с английского языка *assert* означает *утверждать, предполагать*. Иначе говоря, оператор *assert* фиксирует предположение о прогнозируемом результате теста. — *Примеч. научн. ред.*

² Имеются в виду *пожелания пользователей* (user stories). Другой вариант перевода: *пользовательские истории*. — *Примеч. перев.*

- Что считать правильным результатом работы функциональности?
- Какие другие тесты можно придумать, исходя из данного теста?

Малюсенький мозг, такой как у меня, не сможет хорошо поработать над решением всех этих проблем, если они будут решаться одновременно. Две проблемы из приведенного списка можно легко отделить от всех остальных: «Что считать правильным результатом?» и «Каким образом можно проверить правильность результата?»

Вот пример. Представьте, что нам надо реализовать обмен данными с другой системой через сокет. После завершения операции сокет должен быть закрыт, а в буфер должна быть прочитана строка `abc`.

```
testCompleteTransaction() {
    ...
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
```

Откуда должен быть прочитан объект `reply`? Конечно же, из сокета:

```
testCompleteTransaction() {
    ...
    Buffer reply= reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
```

А откуда берется сокет? Мы создаем его, подключаясь к серверу:

```
testCompleteTransaction() {
    ...
    Socket reader= Socket("localhost", defaultPort());
    Buffer reply= reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
```

Однако перед этим мы должны открыть сервер:

```
testCompleteTransaction() {
    Server writer= Server(defaultPort(), "abc");
    Socket reader= Socket("localhost", defaultPort());
    Buffer reply= reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
```

Теперь мы можем изменить имена в соответствии с используемым контекстом, однако в данном случае мы малюсенькими шажками сформировали набросок теста, генерируя каждое решение в течение пары секунд. Мы начали с написания оператора `assert`.

Тестовые данные (Test Data)

Какие данные следует использовать для предварительных тестов? Используйте данные, которые делают тест простым в прочтении и понимании. Помните о том, что вы пишете тесты для аудитории. Не разбрасывайте данные в изобилии по всему тесту только потому, что вам хочется добавить в тест как можно больше разнообразных данных. Если в разных местах теста используются разные данные, разница должна быть осмысленной. Если не существует концептуальной разницы между 1 и 2, используйте 1.

Вместе с тем, если ваша система должна поддерживать несколько разновидностей ввода, значит, все эти разновидности должны быть отражены в тестах. Однако не следует использовать в качестве входных данных список из десяти элементов, если при использовании списка из трех элементов будет получен точно такой же дизайн и реализация.

Попытайтесь не использовать одну и ту же константу в нескольких местах для обозначения более чем одного понятия. Например, если вы намерены тестировать операцию `plus()`, вам наверняка захочется в качестве теста использовать операцию $2 + 2$ — ведь это классический пример сложения. Возможно, вам захочется использовать другую операцию: $1 + 1$ — ведь она самая простая из всех возможных. Однако не забывайте, что в данном случае речь идет о двух разных слагаемых, которые могут быть разными объектами. При использовании выражения $2 + 2$ слагаемые оказываются одинаковыми, а значит, тест не является достаточно общим. Представьте, что в ходе дальнейшей разработки вы пришли к выводу, что результат выполнения операции `plus()` по тем или иным причинам должен зависеть от порядка слагаемых (сложно представить себе ситуацию, в которой результат сложения зависит от порядка слагаемых, однако может случиться, что операция `plus()` может перестать быть просто сложением, таким образом, общая идея должна быть вам понятной). Чтобы обеспечить более полноценное тестирование, попробуйте использовать 2 в качестве первого аргумента и 3 в качестве второго аргумента (в свое время тест $3 + 4$ был классическим начальным тестом при запуске новой виртуальной машины `Smalltalk`).

Альтернативой паттерна `Test Data` (Тестовые данные) является паттерн `Realistic Data` (Реалистичные данные), в рамках которого для тестирования используются данные из реального мира. Реалистичные данные удобно применять в следующих ситуациях:

- вы занимаетесь тестированием системы реального времени, используя цепочки внешних событий, которые возникают в реальных условиях эксплуатации этой системы;
- вы сравниваете вывод текущей системы с выводом предыдущей системы (параллельное тестирование);
- вы выполняете рефакторинг кода, имитирующего некоторый реальный процесс, и ожидаете, что после выполнения рефакторинга результирующие данные будут в точности такими же, как до рефакторинга, в особенности если речь идет о точности операций с плавающей точкой.

Понятные данные (Evident Data)

Каким образом в тесте можно отразить предназначение **тех или иных данных**? Добавьте в тест ожидаемый и реально полученный результат и попытайтесь сделать отношение между ними понятным. Вы пишете тесты не только для компьютера, но и для читателя. Через несколько дней, месяцев или лет кто-нибудь будет смотреть на ваш код и спрашивать себя: «Что имел в виду этот шутник, когда писал этот запутанный код?» Попробуйте оставить своему читателю как можно больше подсказок, **имейте в виду, что этим разочарованным читателем можете оказаться вы сами.**

Вот пример. Если мы конвертируем одну валюту в другую, мы берем комиссию 1,5 за выполнение операции. Представьте, что мы обмениваем американские доллары (USD) на британские фунты стерлингов (GBP). Пусть курс обмена будет составлять 2:1. Если мы хотим обменять \$100, в результате мы должны получить 50 GBP – $1,5\% = 49,25$ GBP. Мы могли бы написать следующий тест:

```
Bank bank= new Bank();
bank.addRate("USD". "GBP", STANDARD_RATE);
bank.commission(STANDARD_COMMISSION);
Money result= bank.convert(new Note(100, "USD"), "GBP");
assertEquals(new Note(49.25, "GBP"), result);
```

Однако **вместо этого мы можем сделать порядок вычислений более очевидным:**

```
Bank bank= new Bank();
bank.addRate("USD". "GBP", 2);
bank.commission(0.015);
Money result= bank.convert(new Note(100, "USD"), "GBP");
assertEquals(new Note(100 / 2 * (1 - 0.015), "GBP"), result);
```

Прочитав этот тест, я вижу взаимосвязь между входными значениями и значениями, используемыми в составе формулы.

Паттерн Evident Data (Понятные данные) обладает **побочным эффектом**: он в некоторой степени облегчает программирование. После того как мы в понятной форме записали выражение `assert`, мы получаем представление о том, что именно нам необходимо запрограммировать. В данном случае мы видим, что тестируемый код должен содержать в себе операции деления и умножения. Мы даже можем воспользоваться паттерном Fake It (Поддельная реализация) для того, чтобы узнать, где должна располагаться та или иная операция.

Паттерн Evident Data (Понятные данные) выглядит, как исключение из правила о том, что в коде не должно быть «магических» чисел. Дело в том, что в рамках одного метода легко понять предназначение того или иного числа. Однако если в программе уже имеются объявленные символьные константы, я предпочитаю использовать их вместо конкретных численных значений.

Паттерны красной полосы

В данной главе речь пойдет о паттернах, которые подскажут вам, когда писать тесты, где писать тесты и когда завершить написание тестов.

One Step Test (Тест одного шага)

Какой следующий тест лучше всего выбрать из списка задач для реализации? Выбирайте тест, который, во-первых, научит вас чему-либо, а во-вторых, который вы сможете реализовать.

Каждый тест должен соответствовать одному шагу в направлении к вашей основной цели. Взгляните на этот список тестов и попробуйте определить, какой тест лучше всего выбрать в качестве следующего для реализации:

- плюс;
- минус;
- умножение;
- деление;
- сложение с такой же валютой;
- равенство;
- равенство нулю;
- нулевой обмен;
- обмен одной и той же валюты;
- обмен двух валют;
- курс кросс-обмена.

Не существует единственно правильного ответа. То, что для меня, ни разу не занимавшегося реализацией этих объектов, будет выглядеть, как один шаг, для вас, обладающих достаточным опытом, может оказаться одной десятой шага. Если вы не можете найти в списке тест, соответствующий одному шагу, добавьте в список дополнительные тесты, реализация которых поможет вам приблизиться к реализации тестов, уже присутствующих в списке.

Когда я смотрю на список тестов, я рассуждаю: «Это очевидно, это очевидно, об этом я не имею ни малейшего представления, это очевидно, здесь — никаких

идей, о чем я думал, когда писал это? А! Я вспомнил! Я думаю, что мог бы это сделать». Этот последний тест я буду реализовывать следующим. С одной стороны, он не кажется мне очевидным, с другой стороны, я уверен в том, что смогу заставить его работать.

Программа, выросшая из подобных тестов, может быть написана в рамках нисходящего подхода (сверху вниз), так как вы можете начать с теста, который ориентирован на вариант полного вычисления. Программа, выросшая из тестов, может быть написана и в рамках восходящего подхода (снизу вверх), так как вы начинаете с небольших кусочков и собираете их в конструкцию постепенно увеличивающегося размера.

И нисходящий, и восходящий подходы не представляют реального описания процесса. Во-первых, вертикальная метафора — это упрощенная визуализация процесса изменения программы в течение разработки. Для описания процесса разработки, основанной на тестировании, лучше подходит метафора *Развития* (Growth) или *Эволюции*: внешняя среда влияет на программу, а программа влияет на внешнюю среду. Во-вторых, если мы хотим, чтобы в нашей метафоре присутствовало направление, лучшим описанием будет «от известного к неизвестному». Подразумевается, что мы обладаем некоторыми знаниями и опытом и ожидаем, что в процессе разработки мы будем узнавать нечто новое. Объединим эти две метафоры вместе и получим, что программа эволюционирует от известного к неизвестному.

Starter Test (Начальный тест)

С какого теста следует начать разработку? Начните с тестирования варианта операции, который не подразумевает выполнения каких-либо осмысленных действий, то есть ничего не делает.

Приступая к реализации операции, вы прежде всего должны ответить на вопрос: «Где она должна располагаться?» Пока вы не ответите на этот вопрос, вы не будете знать, какой код необходимо написать для того, чтобы протестировать эту операцию. Как уже неоднократно рекомендовалось, не следует решать несколько проблем одновременно. Значит, вы должны выбрать такой тест, который позволил бы вам искать ответ только на один этот вопрос и позволил бы на время забыть обо всех остальных вопросах.

Если вы с самого начала приступите к реализации реалистичного теста, вам придется искать ответы на несколько вопросов одновременно:

- Где должна располагаться операция?
- Какие входные данные считать корректными?
- Каким должен быть корректный результат выполнения операции при использовании выбранных входных данных?

Если вы начнете с реалистичного теста, вы слишком долгое время будете вынуждены действовать без обратной связи. Красный—зеленый—рефакторинг, красный—зеленый—рефакторинг. На выполнение этого цикла должно уходить всего несколько минут.

Но как сократить время исполнения цикла? Для этого вы можете воспользоваться тривиальными входными и выходными данными. Вот простой пример: если функция должна складывать многозначные вещественные числа с точностью до тысячного знака после запятой, вовсе не обязательно начинать ее реализацию с написания теста, проверяющего результат сложения таких огромных чисел. Вполне можно начать с тривиального теста $3 + 4 = 7$. Вот еще один пример. В группе электронных новостей, посвященной экстремальному программированию, один из участников поинтересовался, каким образом можно написать программу минимизации количества полигонов (многоугольников), составляющих собой некоторую поверхность. На входе набор полигонов, комбинация которых представляет собой некоторый трехмерный объект. На выходе должна получиться комбинация полигонов, которая описывает точно такой же объект (поверхность), но в состав которой входит минимальное возможное количество полигонов. «Каким образом я могу разработать подобную программу, если для того, чтобы заставить тест работать, я должен быть как минимум доктором наук?»

Используя паттерн **Starter Test** (Начальный тест), мы получаем ответ:

- Вывод должен быть точно таким же, как ввод. Некоторые комбинации полигонов изначально являются минимальными.
- Ввод должен быть как можно меньшего размера. Например, единственный полигон или даже пустой список полигонов.

Мой начальный тест выглядел следующим образом:

```
Reducer r= new Reducer(new Polygon());
assertEquals(0, reducer.result().npoints);
```

Отлично! Первый тест сработал. Теперь можно перейти к остальным тестам в списке...

К начальному тесту следует применять рассмотренное ранее правило **One Step Test** (Тест одного шага): самый первый тест должен научить вас чему-то новому, кроме того, вы должны обладать возможностью достаточно быстро заставить его работать. Если вы реализуете подобный код уже не в первый раз, вы можете выбрать начальный тест для одной или даже двух операций. Вы должны быть уверены в том, что сможете быстро заставить тест работать. Если вы приступаете к реализации чего-либо достаточно сложного и делаете это впервые, начните с самого простого теста, который вы только можете представить.

Я часто замечаю, что мой начальный тест работает на достаточно высоком уровне и скорее напоминает тест всего приложения. Например, простой сокетный сервер. Самый первый тест выглядит следующим образом:

```
StartServer
Socket= new Socket
Message= "hello"
Socket.write(message)
AssertEquals(message, socket.read)
```

Остальные тесты пишутся только на стороне сервера: «Предположим, что мы получаем строки наподобие этой...»

Explanation Test (Объясняющий тест)

Каким образом вы можете распространить в своей команде использование автоматического тестирования? Для любых объяснений используйте тесты и спрашивайте тесты у тех, кто пытается вам что-либо объяснить.

Если вы единственный член команды, работающий в стиле TDD, вы можете почувствовать себя неуютно и одиноко. Однако вскоре после того, как вы начнете работать в стиле TDD, вы обратите внимание на уменьшение количества проблем, связанных с интеграцией, и снижение количества дефектов, обнаруженных в оттестированном коде. Дизайн вашего кода будет проще и его проще будет объяснять. Может случиться так, что ваши коллеги проявят интерес к тестированию и предварительному тестированию разрабатываемого кода.

Опасайтесь оголтелого энтузиазма со стороны новичков. Подобный энтузиазм может оттолкнуть тех, кто еще не до конца понял преимущества и необходимость предварительного тестирования. Если внедрение TDD производить насильственными методами, это может привести к негативным результатам. Если вы менеджер или лидер, вы не должны насильно заставлять людей менять стиль, в рамках которого они работают.

Но что можно сделать? Лучше всего предлагать вашим коллегам объяснять работу кода в форме тестов: «Подожди-ка, если я правильно понял, объект Foo будет таким, а объект Bar будет таким, значит, в результате получится 76?» Кроме того, вы можете объяснять работу кода в виде тестов: «Вот как это работает. Если объект Foo будет таким, а объект Bar будет таким, в результате получится 76. Однако если объект Foo будет таким, а объект Bar будет таким, в результате получится 67».

Вы можете делать это на более высоком уровне абстракции. Если кто-то пытается объяснить вам работу кода при помощи диаграммы последовательности обмена сообщениями, вы можете предложить ему преобразовать эту диаграмму в более понятную форму. После этого вы пишете тест, содержащий в себе все видимые на диаграмме объекты и сообщения.

Learning Test (Тест для изучения)¹

Когда необходимо писать тесты для программного обеспечения, разработанного сторонними разработчиками? Перед тем как вы впервые воспользуетесь новыми возможностями этого программного обеспечения.

Предположим, что вы приступаете к разработке программы, основанной на использовании библиотеки Mobile Information Device Profile (MIDP) для языка Java. Вы собираетесь сохранить некоторые данные в объекте RecordStore и затем извлечь их оттуда. Должны ли вы просто написать код в надежде на то, что он работает? Это один из возможных методов разработки.

¹ Спасибо Джиму Ньюкирку (Jim Newkirk) и Лауренту Боссави (Laurent Bossavit) за то, что они независимо друг от друга предложили мне этот паттерн.

Есть альтернативный метод. Обратите внимание на то, что вы собираетесь использовать новый метод нового класса. Вместо того чтобы просто воспользоваться им внутри разрабатываемого вами кода, вы пишете небольшой тест, который позволяет вам убедиться в том, что API работает так, как вы того ожидаете. Таким образом, вы можете написать:

```
RecordStore store;

public void setUp() {
    store= RecordStore.openRecordStore("testing", true);
}

public void tearDown() {
    RecordStore.deleteRecordStore("testing");
}

public void testStore() {
    int id= store.addRecord(new byte[] {5, 6}, 0, 2);
    assertEquals(2, store.getRecordSize(id));
    byte[] buffer= new byte[2];
    assertEquals(2, store.getRecord(id, buffer, 0));
    assertEquals(5, buffer[0]);
    assertEquals(6, buffer[1]);
}
```

Если ваше понимание API совпадает с действительностью, значит, тест работает с первого раза.

Джим Ньюкирк рассказал мне о проекте, в котором разработка тестов для обучения выполнялась на регулярной основе. Как только от сторонних разработчиков поступала новая версия пакета, в отношении этого пакета немедленно запускались имеющиеся тесты. В случае необходимости в тесты вносились исправления. Если тесты не срабатывали, не было никакого смысла запускать приложение, так как оно определено не заработает. Если же все тесты срабатывают, значит и приложение работает.

Another Test (Еще один тест)

Каким образом предотвратить уход дискуссии от основной темы? Когда возникает посторонняя, но интересная мысль, добавьте в список еще один тест и вернитесь к основной теме.

Я люблю пространные дискуссии (вы уже прочитали большую часть книги, поэтому, скорее всего, пришли к такому же выводу самостоятельно). Если постоянно жестко следить за соблюдением основной темы обсуждения, можно потерять множество бриллиантовых идей. Вы перескакиваете с одной темы на другую, затем на третью и, наконец, не успеваете заметить, что ушли далеко от того, с чего начали. А кого это волнует, ведь обсуждать вещи в таком стиле — это круто!

Иногда программирование — это прорыв, генерация гениальной идеи и вдохновение. Однако в большинстве случаев программирование — весьма рутинная работа. У меня есть десять вещей, которые я должен реализовать. Я постоянно откладываю на потом задачу номер четыре. Один из способов избежать работы (и, возможно, сопутствующего страха) — вступить в длинные пространственные рассуждения на самые разные темы.

Потеряв огромное количество времени впустую, я пришел к выводу, что иногда лучше сосредоточиться на конкретной проблеме и не отвлекаться на побочные мысли. Когда я работаю в подобном стиле, я приветствую любые новые идеи, однако не позволяю им слишком сильно отвлекать мое внимание. Я записываю новые идеи в список и затем возвращаюсь к тому, над чем я работаю.

Regression Test (Регрессионный тест)

Что необходимо сделать в первую очередь в случае, если обнаружен дефект? Написать самый маленький из всех возможных тестов, который не срабатывает, и восстановить его работоспособность.

Регрессионные тесты — это тесты, которые вы наверняка написали бы в процессе обычной разработки, если бы своевременно обнаружили проблему. Каждый раз, когда вы сталкиваетесь с необходимостью написать регрессионный тест, подумайте о том, каким образом вы могли бы узнать о необходимости написания этого теста ранее, то есть тогда, когда выполняли разработку.

Полезным может оказаться тестирование на уровне всего приложения. Регрессионные тесты для всего приложения дают пользователям возможность сообщить вам, что неправильно и чего они на самом деле ожидают. Регрессионные тесты меньшего масштаба являются для вас способом улучшить качество тестирования. Если вы получили доклад о дефекте, в котором описывается появление большого отрицательного целого числа в отчете, значит, в будущем вам необходимо уделить дополнительное внимание тестированию граничных значений для целых чисел.

Возможно, для того чтобы изолировать дефект, вам потребуется выполнить рефакторинг системы. В этом случае, демонстрируя дефект, система как бы говорит вам: «Ты еще не вполне закончил проектировать меня».

Break (Перерыв)

Что делать, если вы почувствовали усталость или зашли в тупик? Прервите работу и отдохните.

Выпейте кофе, пройдитеесь или даже вздремните. Стряхните с себя эмоциональное напряжение, связанное с решениями, которые вы принимаете, и символами, которые вы набираете на клавиатуре.

Иногда самого короткого перерыва достаточно для того, чтобы недостающая идея возникла в вашей голове. Возможно, вставая из-за компьютера, вы неожиданно нащупаете нужную нить: «Да я же не попробовал этот метод с пересмот-

ренными параметрами!» В любом случае прервитесь. Дайте себе пару минут. Идея никуда не убежит.

Если, несмотря на отдых, идея не приходит вам в голову, пересмотрите цели, которые вы поставили перед собой для текущего сеанса программирования. Можно ли считать эти цели по-прежнему реалистичными или вы должны выбрать новые цели? Является ли то, чего вы пытаетесь достичь, невозможным? Если так, то каким образом это повлияет на всю вашу команду?

Дэйв Унгар (Dave Ungar) называет это Методологией душа (Shower Methodology). Если вы знаете, чего писать, пишите. Если вы не знаете, что писать, примите душ и стойте под ним до тех пор, пока не поймете, что нужно писать. Очень многие команды были бы более счастливыми, более продуктивными и пахли бы существенно лучше, если бы они воспользовались этим советом. TDD — это усовершенствование предложенной Унгаром методологии душа. Если вы знаете, что писать, пишите очевидную реализацию (Obvious Implementation). Если вы не знаете, что писать, создайте поддельную реализацию (Fake It). Если правильный дизайн по-прежнему не ясен, триангулируйте (Triangulate). Если вы по-прежнему не знаете, что писать, можете, наконец, принять душ.

На рис. 26.1 показана динамика процессов, связанных с перерывом. В процессе работы вы устаете. В результате внимание рассеивается, и вам становится сложнее заметить то, что вы устали. Поэтому вы продолжаете работать и устаете еще больше.

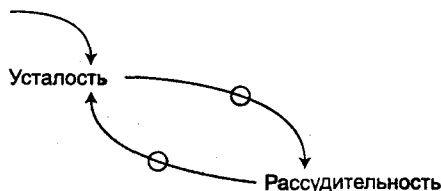


Рис. 26.1. Усталость негативно влияет на рассудительность, которая негативно влияет на усталость

Чтобы разорвать этот негативный замкнутый круг, необходимо добавить дополнительный внешний элемент.

- В масштабе нескольких часов держите бутылку с водой рядом с вашей клавиатурой и время от времени прихлебывайте из нее. Благодаря этому естественная физиология будет подсказывать вам, когда и зачем необходимо сделать короткий перерыв в работе.
- В масштабе дня вы должны хорошо отдохнуть после завершения рабочего времени.
- В масштабе недели вы отдыхаете в уик-энд. И это наполняет вас силами и идеями, благодаря чему вы можете приступить к новой рабочей неделе. (Моя жена утверждает, что самые лучшие идеи возникают у меня вечером в пятницу.)
- В масштабе года вы получаете отпуск, что позволяет вам полностью освежиться. Французы подходят к этому вопросу очень правильно — двух по-

следовательных недель отпуска недостаточно. В течение первой недели вы сбрасываете с себя рабочее напряжение, а в течение второй недели вы подсознательно готовите себя к работе. Поэтому для того, чтобы хорошо отдохнуть и эффективно работать в течение всего следующего года, требуется три, а лучше четыре недели отдыха.

Существует обратная сторона данного паттерна. Иногда в случае, если перед вами стоит сложная проблема, вам требуется, наоборот, поднажать, поднапрячься и потратить дополнительное время и усилия для того, чтобы решить ее. Однако большинство программистов инфицировано духом саморазрушения: «Я угроблю свое здоровье, отрекусь от своей семьи и даже выпрыгну из окна, лишь бы этот код заработал». Поэтому я не буду давать здесь каких-либо советов. Если вы чувствуете, что у вас развивается болезненное пристрастие к кофе, наверное, вам не стоит делать слишком частых перерывов. В крайнем случае, просто пройдитесь.

Do over (Начать сначала)

Что делать, если вы зашли в тупик? Выкиньте код и начните работу сначала.

Вы заблудились. Вы решили передохнуть. Вымыли руки. Еще раз попытались вспомнить дальнейший путь. И все равно вы заблудились. Код, который выглядел так неплохо всего час назад, теперь выглядит запутанно и непонятно, одним словом, отвратительно. Вы не можете представить себе, как заставить работать следующий тест, а впереди у вас еще 20 тестов, которые необходимо реализовать.

Подобное случалось со мной несколько раз, пока я писал эту книгу. Код получался слишком кривым. «Но я должен закончить книгу. Мои дети хотят есть, а сборщики налогов стучаться в мою дверь». У меня возникает желание выпрямить код настолько, чтобы можно было продолжать двигаться вперед. Однако на самом деле в большинстве случаев продуктивнее отдохнуть немного и начать все заново. Однажды я был вынужден выкинуть 25 страниц рукописи потому, что она была основана на очевидно глупом программистском решении.

Хорошую историю на эту тему рассказал мне Тим Макиннон (Tim Maskinnon). Однажды он проводил интервью с потенциальным новым сотрудником. Чтобы оценить уровень его мастерства, он предложил ему программировать в паре в течение часа. К концу этого часа они реализовали несколько новых тестов и выполнили несколько рефакторингов. Однако это был конец рабочего дня, они оба чувствовали себя усталыми, поэтому решили полностью убрать из системы результаты своей работы.

Если вы программируете в паре, смена партнера — это хороший повод отказаться от плохого кода и начать решение задачи с начала. Вы пытаетесь объяснить смысл запутанного кода, над которым работали до этого, и вдруг ваш партнер, совершенно не связанный с ошибками, которые вы допустили, берет у вас клавиатуру и говорит: «Я ужасно извиняюсь за мою тупость, но что, если мы попробуем начать по-другому?»

Cheap Desk, Nice Chair (Дешевый стол, хорошие кресла)

В какой физической обстановке следует использовать TDD? Используйте удобное, комфортное кресло. На всей остальной мебели можно сэкономить.

Вы не сможете хорошо программировать, если ваша спина будет болеть. К сожалению, организации, которые выкладывают по \$100 000 в месяц за работу команды программистов, как правило, отказываются тратить \$10 000 на покупку хороших кресел.

Я предлагаю использовать дешевые, самые примитивные и ужасные на вид столы для установки компьютеров, но зато купить самые лучшие кресла, которые я только смогу найти. Дешевых столов можно купить столько, сколько нужно, значит, я получаю в свое распоряжение достаточное количество рабочего места и могу легко его увеличить. При этом я чувствую себя комфортно за компьютером, моя спина не устает.

Если вы программируете в паре, позаботьтесь о том, чтобы вам было удобно. Расчистите пространство на столе таким образом, чтобы вам было удобно передавать клавиатуру из рук в руки. Когда я работаю наставником, я люблю выполнять один простой прием: незаметно подходить со спины к программирующей паре и ненавязчиво поправлять клавиатуру так, чтобы она располагалась удобно по отношению к человеку, который с ней работает.

Манфред Лэндж (Manfred Lange) считает, что аккуратное распределение ресурсов необходимо выполнить также в отношении компьютерного аппаратного обеспечения. Рекомендуется использовать дешевые/медленные/старые компьютеры для индивидуальной электронной почты и работы с Интернетом, но зато приобрести самые современные и самые быстрые компьютеры для разработки.

Паттерны тестирования

В данной главе описываются более детальные методики разработки тестов.

Дочерний тест (Child Test)

Как заставить работать тест, который оказался слишком большим? Напишите тест меньшего размера, который представляет собой сломанную часть большого теста. Добейтесь срабатывания маленького теста. Заново напишите большой тест.

Ритм красный—зеленый—рефакторинг чрезвычайно важен для достижения успеха. Не бойтесь потратить дополнительные усилия для того, чтобы поддерживать этот ритм, — дополнительные усилия с лихвой окупят себя. Я достаточно часто попадаю в подобную ситуацию: сначала записываю тест, а потом оказывается, что для его реализации требуется выполнить не одно, а несколько изменений. Я неожиданно оказываюсь на большом расстоянии от зеленой полосы. Даже десять минут с красной полосой заставляют меня нервничать.

Когда я записываю тест, который оказывается слишком большим, я, прежде всего, пытаюсь усвоить урок. Почему тест оказался слишком большим? Что надо было бы сделать иначе, чтобы тест получился меньше по размеру?

Покончив с размышлениями, я удаляю изначальный тест и начинаю заново. «Похоже, заставить все эти три вещи работать одновременно — это слишком сложная задача. Однако, если я вначале добьюсь срабатывания А, В и С, мне не составит труда заставить работать всю эту штуку вместе.» Иногда я действительно удаляю тест, однако в некоторых случаях я просто изменяю его имя таким образом, чтобы оно начиналось на х, — в этом случае тестовый метод не будет выполнен. (Скажу вам по секрету, что иногда я вообще не трогаю изначальный тест. Да, да! Только т-с-с-с! Никому об этом не рассказывайте! Слава богу, в большинстве подобных случаев мне удается быстро заставить работать дочерний тест. Однако получается, что я в течение пары минут живу вместе с двумя сломанными тестами. Возможно, когда я так поступаю, я совершаю ошибку. Этот пережиток сохранился у меня с тех самых старых времен, когда я выполнял тестирование после завершения разработки или вообще не тестировал свой код.)

Попробуйте оба варианта. Прислушайтесь к своим ощущениям. Если у вас есть два сломанных теста, вы, как правило, начинаете программировать иначе. Делайте выводы.

Mock Object (Поддельный объект)

Как выполнять тестирование объекта, который базируется на сложном и тяжеловесном ресурсе? Создайте поддельную версию ресурса, которая будет возвращать константы.

Использование поддельных объектов — это тема для отдельной книги. Существует огромное количество материала, посвященного поддельным объектам¹. Здесь я попытаюсь очень коротко познакомить читателей с этой концепцией. Классическим примером является база данных. Чтобы запустить базу данных, требуется значительное время, поддержка чистоты базы данных требует дополнительных затрат, кроме того, если база данных располагается на удаленном сетевом сервере, ваши тесты будут связаны с конкретным физическим местоположением в сети. Наконец, база данных является емким источником ошибок разработки.

Чтобы уменьшить количество проблем, рекомендуется в процессе тестирования отказаться от работы непосредственно с базой данных. Большинство тестов пишутся в отношении объекта, который функционирует подобно базе данных, однако располагается в оперативной памяти.

```
public void testOrderLookup() {
    Database db= new MockDatabase();
    db.expectQuery("select order_no from Order where cust_no is 123");
    db.returnResult(new String[]{"Order 2" ,"Order 3"});
}
```

Если объект `MockDatabase` не принимает ожидаемого запроса, он генерирует исключение. Если запрос корректен, объект возвращает нечто, напоминающее результирующий набор данных (result set), состоящий из нескольких константных строк.

Помимо высокой производительности и надежности поддельные объекты обладают также еще одним преимуществом: читабельностью. Если вы работаете с реальной базой данных, заполненной реальными данными, в результате обработки запроса вы можете получить ответ, состоящий из 14 строк. Возможно, вам будет нелегко понять, откуда взялось число 14 и в чем, собственно, состоит смысл теста.

Если вы хотите воспользоваться поддельными объектами, не следует хранить тяжеловесные ресурсы в глобальных переменных (даже если они замаскированы при помощи паттерна Singleton (Одиночка)). Если вы так поступите, вам придется

¹ Например, www.mockobjects.com.

вначале настроить глобальный поддельный объект, затем выполнить тест, а затем позаботиться о том, чтобы вернуть поддельный объект в исходное состояние.

В свое время я очень строго следил за выполнением этого правила. Мы вместе с Массимо Арнольди (Massimo Arnoldi) разрабатывали код, который взаимодействовал с набором курсов обмена валют, хранящимся в глобальной переменной. Для разных тестов требовалось использовать разные наборы данных, и в некоторых случаях курсы обмена валют должны были быть разными для разных тестов. Вначале мы пытались использовать для тестирования глобальную переменную, однако в конце концов нас это утомило, и однажды утром (смелые решения, как правило, приходят ко мне по утрам) мы решили передавать объект Exchange (в котором хранились курсы обмена) в качестве параметра везде, где это было необходимо. Мы думали, что нам придется модифицировать сотни методов. Однако дело кончилось тем, что мы добавили дополнительный параметр в десять или пятнадцать методов и по ходу дела подчистили другие аспекты дизайна.

Паттерн поддельных объектов заставляет вас тщательно следить за видимостью объектов, снижая взаимозависимости между ними. Поддельные объекты добавляют в ваш проект некоторый риск, — что если поддельный объект ведет себя не так, как реальный объект? Чтобы снизить этот риск, вы можете разработать специальный набор тестов для поддельных объектов, которые должны быть выполнены в отношении реального объекта, чтобы убедиться в том, что имитация достаточно близка к оригиналу.

Self Shunt (Самошунтирование)

Каким образом можно убедиться в том, что один объект корректно взаимодействует с другим? Заставьте тестируемый объект взаимодействовать не с целевым объектом, а с вашим тестом.

Предположим, что вы хотите динамически обновлять зеленую полосу, отображаемую в рамках тестируемого пользовательского интерфейса. Если мы сможем подключить наш объект к объекту `TestResult`, значит, мы сможем получать оповещения о запуске теста, о том, что тест не сработал, а также о том, что весь набор тестов начал работу или, наоборот, завершил работу. Каждый раз, получив оповещение о запуске теста, мы можем выполнить обновление интерфейса. Вот соответствующий тест:

ResultListenerTest

```
def testNotification(self):
    result= TestResult()
    listener= ResultListener()
    result.addListener(listener)
    wasRun("testMethod").run(result)
    assert 1 == listener.count
```

Тест нуждается в объекте, который подсчитывал бы количество оповещений:

ResultListener

```
class ResultListener:
```

```
def __init__(self):
    self.count= 0
def startTest(self):
    self.count= self.count + 1
```

Подождите-ка! Зачем нам нужен отдельный объект `Listener`? Все необходимые функции мы можем возложить на объект `TestCase`. В этом случае объект `TestCase` становится подобием поддельного объекта (`Mock Object`).

`ResultListenerTest`

```
def testNotification(self):
    self.count= 0
    result= TestResult()
    result.addListener(self)
    WasRun("testMethod").run(result)
    assert 1 == self.count
def startTest(self):
    self.count= self.count + 1
```

Тесты, написанные с использованием паттерна `Self Shunt` (Самошунтирование), как правило, читаются лучше, чем тесты, написанные без него. Предыдущий тест является неплохим примером. Счетчик был равен 0, а затем стал равен 1. Вы можете проследить за последовательностью действий прямо в коде теста. Каким образом счетчик стал равен 1? Очевидно, кто-то обратился к методу `startTest()`. Каким образом произошло обращение к методу `startTest()`? Это произошло в начале выполнения теста. Вторая версия теста использует два разных значения переменной `count` в одном месте, в то время как первая версия теста присваивает переменной `count` значение 0 в одном классе и проверяет эту переменную на равенство значению 1 в другом классе.

Возможно, при использовании `Self Shunt` (Самошунтирование) вам потребуется применить рефакторинг `Extract Interface` (Выделение интерфейса), для того чтобы получить интерфейс, который должен быть реализован вашим тестом. Вы должны сами определить, что проще: выделение интерфейса или тестирование существующего объекта в рамках концепции «черный ящик». Однако я часто замечал, что интерфейсы, выделенные при выполнении самошунтирования в дальнейшем, как правило, оказываются полезными для решения других задач.

В результате использования `Self Shunt` (Самошунтирование) вы можете наблюдать, как тесты в языке Java обрастают разнообразными причудливыми интерфейсами. В языках с оптимистической типизацией класс теста обязан реализовать только те операции интерфейса, которые действительно используются в процессе исполнения теста. Однако в Java вы обязаны реализовать абсолютно все операции интерфейса, несмотря на то что некоторые из них будут пустыми. По этой причине интерфейсы следует делать как можно менее емкими. Реализация каждой операции должна либо возвращать значение, либо генерировать исключение — это зависит от того, каким образом вы хотите быть оповещены о том, что произошло нечто неожиданное.

Log String (Строка-журнал)

Каким образом можно убедиться в том, что обращение к методам осуществляется в правильном порядке? Создайте строку, используйте ее в качестве журнала, каждый раз при обращении к методу добавляйте в строку-журнал некоторое символическое сообщение.

Данный прием продемонстрирован ранее, в главе 20, в которой мы тестировали порядок обращения к методам класса `TestCase`. В нашем распоряжении шаблонный метод (`Template Method`), который, как мы предполагаем, обращается к методу `setUp()`, затем к тестовому методу, а затем к методу `tearDown()`. Нам хотелось бы убедиться в том, что обращение к методам осуществляется в указанном порядке. Реализуем методы таким образом, чтобы каждый из них добавлял свое имя в строку-журнал. Исходя из этого, можно написать следующий тест:

```
def testTemplateMethod(self):
    test= WasRun("testMethod")
    result= TestResult()
    test.run(result)
    assert("setUp testMethod tearDown " == test.log)
```

Реализация тоже очень проста:

`WasRun`

```
def setUp(self):
    self.log= "setUp "
def testMethod(self):
    self.log= self.log + "testMethod "
def tearDown(self):
    self.log= self.log + "tearDown "
```

Паттерн `Log String` (Строка-журнал) особенно полезен в случае, когда вы реализуете `Observer` (Наблюдатель) и желаете протестировать порядок поступления оповещений. Если вас прежде всего интересует, какие именно оповещения генерируются, однако порядок их поступления для вас не важен, вы можете создать множество (`Set`) строк, при обращении к методам добавлять строки в множество и в выражении `assert` использовать операцию сравнения множеств.

Паттерн `Log String` (Строка-журнал) хорошо сочетается с паттерном `Self Shunt` (Самозанутирование). Объект-тест реализует методы занутируемого интерфейса таким образом, что каждый из них добавляет запись в строку-журнал, затем проверяется корректность этих записей.

Crush Test Dummy (Тестирование обработки ошибок)

Каким образом можно протестировать работу кода, обращение к которому происходит в случае ошибки, возникновение которой маловероятно? Создайте специальный объект, который вместо реальной работы генерирует исключение.

Непротестированный код считается *неработающим*. Но что же тогда делать с кодом обработки ошибок? Надо ли его тестировать? Только в случае, если вы хотите, чтобы он работал.

Предположим, что мы хотим проверить, что происходит с нашим приложением в случае, если на диске не остается свободного места. Неужели для этого необходимо вручную создавать огромное количество файлов? Есть альтернатива. Мы можем *имитировать* заполнение диска. Фактически мы используем паттерн Fake It (Подделка).

Вот наш тест для класса File:

```
private class FullFile extends File {
    public FullFile(String path) {
        super(path);
    }
    public boolean createNewFile() throws IOException {
        throw new IOException();
    }
}
```

Теперь мы можем написать тест *ожидаемого* исключения:

```
public void testFileSystemError() {
    File f = new FullFile("foo");
    try {
        saveAs(f);
        fail();
    } catch (IOException e) {
    }
}
```

Тест кода обработки ошибки напоминает Mock Object (Поддельный объект), однако в данном случае нам не надо подделывать весь объект. Для реализации этой методики удобно использовать анонимные внутренние (inner) классы языка Java. При этом вы можете переопределить только один необходимый вам метод, сделать это можно прямо внутри вашего теста, благодаря чему код теста станет более удобным в прочтении:

```
public void testFileSystemError() {
    File f = new File("foo") {
        public boolean createNewFile() throws IOException {
            throw new IOException();
        }
    };
    try {
        saveAs(f);
        fail();
    } catch (IOException e) {
    }
}
```

Broken Test (Сломанный тест)

Как следует завершить сеанс программирования, если вы программируете в одиночку? Оставьте последний тест неработающим.

Этому приему научил меня Ричард Гэбриел (Richard Gabriel). Вы заканчиваете сеанс на середине предложения. Когда вы возвращаетесь к работе, вы смотрите на начало предложения и вспоминаете, о чем вы думали, когда писали его. Вспомнив ход своих мыслей, вы завершаете предложение и продолжаете работу. Если, по возвращении к работе вам не надо завершать никакого предложения, вы вынуждены потратить несколько минут для того, чтобы сначала просмотреть уже написанный код, изучить список задач, вспомнить о том, над чем вы собирались работать, затем попытаться восстановить прежнее состояние ваших мыслей и, наконец, приступить к работе.

Я использовал подобный трюк при работе над моими одиночными проектами и мне очень понравился эффект. В конце рабочего сеанса следует написать тест и запустить его для того, чтобы убедиться, что он не работает. Когда вы позже возвращаетесь к коду, перед вами очевидное место, с которого можно продолжить. Вы получаете хорошо заметную закладку, которая помогает вам вспомнить, о чем вы думали. Оставленный неработающим тест должен быть прост в реализации, благодаря чему вы сможете быстро выйти на прежний маршрут и продолжить движение к победе.

Сначала я думал, что оставленный недоделанным тест будет действовать мне на нервы. Однако нет. Ведь вся программа в целом еще далека от завершения, один сломанный тест не сделает ее менее завершенной, и я отлично знаю об этом. Возможность быстро восстановить общий ход разработки спустя несколько недель простоя стоит того, чтобы немножко поступиться принципами и оставить работу, имея перед глазами красную полосу.

Clean Check-in (Чистый выпускаемый код)

Как следует завершить сеанс программирования, если вы программируете в составе команды? Все ваши тесты должны работать.

Я противоречу сам себе? Да.

Бабба Уитман (Bubba Whitman), Walt's stevedore brother

Когда вы работаете над кодом не один, а вместе с вашими коллегами, ситуация полностью меняется. Когда вы возвращаетесь к работе над кодом, над которым помимо вас работает еще несколько человек, вы не можете сказать точно, что именно произошло с кодом с того момента, как вы видели его последний раз. По этой причине, прежде чем выпускать разрабатываемый вами код, убедитесь в том, что все тесты срабатывают. (Это напоминает правило изоляции тестов, в соответствии с которым каждый тест гарантированно оставляет систему в хорошем известном состоянии. Это правило можно применить и в отношении программистов, которые поочередно интегрируют свой код в систему, каждый раз проверяя, остается ли система в хорошем состоянии или нет.)

Набор тестов, которые вы запускаете в ходе интеграции, может быть существенно объемнее набора тестов, который вы запускаете каждую минуту в процессе разработки функциональности. (До того времени, пока это не станет слишком утомительным, я рекомендую вам постоянно запускать в отношении вашего кода полный набор тестов.) Что делать, если при попытке интеграции вы обнаружили, что один из тестов полного набора не срабатывает?

Самое простое правило рекомендует отбросить проделанную работу и начать все заново. Сломанный тест является признаком того, что вы не обладаете достаточным запасом знаний для того, чтобы запрограммировать то, над чем вы работаете. Если команда будет действовать в соответствии с этим правилом, у ее членов появится тенденция выполнять интеграцию более часто, так как тот, кто успеет приступить к интеграции раньше других, не рискует потерять проделанную работу. По всей видимости частая интеграция и проверка — это неплохая практика.

Существует другой, менее строгий подход: вы можете исправить дефект и попробовать снова выполнить интеграцию. Однако не забывайте о том, что вы не должны слишком долго занимать интеграционные ресурсы: если вы не можете решить проблему в течение нескольких минут, откажитесь от идеи интеграции и начните работу заново. Об этом можно не говорить, но я все-таки скажу, что закомментирование теста с целью заставить работать весь тестовый набор строго запрещается и должно приводить к самым серьезным карательным санкциям.

Паттерны зеленой полосы

Когда у вас есть сломанный тест, вы должны заставить его работать. Если вы рассматриваете красную полосу как состояние, из которого следует выйти как можно быстрее, вы должны овладеть приемами быстрого получения зеленой полосы. Используйте следующие паттерны для того, чтобы заставить ваш тест сработать (даже если полученный в результате этого код не просуществует и часа).

Fake It (Подделка)

Если у вас есть сломанный тест, какой должна быть самая первая реализация? Сделайте так, чтобы тестируемый метод возвращал константу. После того как тест начал срабатывать, постепенно трансформируйте константу в выражение с использованием переменных.

Пример использования этого подхода продемонстрирован в ходе разработки нашей реализации xUnit. Вначале мы использовали строковую константу:

```
return "1 run, 0 failed"
```

Затем эта строка была преобразована в выражение:

```
return "%d run, 0 failed" % self.runCount
```

Однако этим дело не кончилось. В конце мы получили выражение:

```
return "%d run, %d failed" % (self.runCount , self.failureCount)
```

Паттерн Fake It (Подделка) напоминает страховочную веревку, которая соединяет вас с верхней точкой маршрута, когда вы карабкаетесь по скале. Пока что вы еще не забрались на самый верх (тест на месте и работает, но тестируемый код некорректен). Однако в любой точке маршрута вы держитесь за веревку и знаете, что когда достигнете самого верха, то будете в безопасности (тест срабатывает в ходе рефакторинга, а также после того, как вы получили окончательный результирующий код).

Паттерн Fake It (Подделка) многим может показаться совершенно бесполезным. Зачем писать код, который абсолютно точно придется заменять другим?

Дело в том, что иметь хоть какой-то работающий код — это лучше, чем вообще не иметь работающего кода, в особенности если у вас есть тесты, которые могут доказать работоспособность кода. Петер Хансен (Peter Hansen) рассказал мне следующую историю:

Буквально вчера два новичка в области TDD — мой партнер и я — решили в точности следовать букве закона. То есть мы написали тест, а затем написали самый простой, но совершенно бесполезный код, который обеспечивал срабатывание теста. Пока мы писали этот код, мы обнаружили, что тест написан неправильно.

Каким образом поддельная реализация подсказала им, что написанный ими тест некорректен? Я понятия не имею, однако я счастлив, что они вовремя обнаружили это. Быть может, если они не воспользовались бы поддельной реализацией, они пошли бы по ложному пути. Возможно, исправление связанных с этим ошибок обошлось бы им дороже.

При использовании паттерна **Fake It** (Подделка) возникает как минимум два положительных эффекта:

- *Психологический.* Если перед вами зеленая полоса, вы чувствуете себя совершенно иначе, чем если перед вами красная полоса. Когда полоса зеленая, вы знаете, на чем стоите. Вы можете смело и уверенно приступить к рефакторингу.
- *Контроль над объемом работы.* Программисты привыкли пытаться предвидеть возникновение в будущем самых разнообразных проблем. Если вы начинаете с конкретного примера и затем осуществляете обобщение кода, это помогает вам избавиться от излишних опасений. Вы можете сконцентрироваться на решении конкретной проблемы и поэтому выполнить работу лучше. При переходе к следующему тесту вы опять же концентрируетесь на нем, так как знаете, что предыдущий тест гарантированно работает.

Нарушает ли паттерн Fake It (Подделка) правило о том, что не следует писать код, который вам не потребуется? Я так не думаю, ведь на этапе рефакторинга вы удаляете дублирование данных между тестом и тестируемым кодом. Допустим, я написал¹:

```
assertEquals(new MyDate("28.2.02"), new MyDate("1.3.02").yesterday());
```

MyDate

```
public MyDate yesterday() {
    return new MyDate("28.2.02");
}
```

Между тестом и кодом существует дублирование. Попробуем исправить ситуацию:

MyDate

```
public MyDate yesterday() {
```

¹ Спасибо Дирку Кенигу (Dierk König) за пример.

```
return new MyDate(new MyDate("1.3.02").days()-1);
}
```

Однако дублирование по-прежнему присутствует. Чтобы избавиться от него, заменяем `MyDate("1.3.02")` на `this` (в моем тесте эти значения равны). Получается:

`MyDate`

```
public MyDate yesterday() {
    return new MyDate(this.days()-1);
}
```

Однако увидеть возможность подобных подстановок с первого взгляда удастся далеко не всегда и далеко не всем, поэтому для пущей ясности вы можете использовать триангуляцию (**Triangulation**) по крайней мере до тех пор, пока вам не надоест. Когда вам надоест, вы чаще будете пользоваться **Fake It** (Подделка) или **Obvious Implementation** (Очевидная реализация).

Triangulate (Триангуляция)

Какой самый консервативный способ позволяет формировать абстракцию при помощи тестов? Делайте код абстрактным только в случае, если у вас есть два или большее количество примеров.

Рассмотрим пример. Предположим, мы хотим написать функцию, которая возвращает сумму двух целых чисел. Мы пишем:

```
public void testSum() {
    assertEquals(4, plus(3, 1));
}

private int plus(int augend, int addend) {
    return 4;
}
```

Чтобы получить представление о правильном дизайне, мы добавляем еще один пример:

```
public void testSum() {
    assertEquals(4, plus(3, 1));
    assertEquals(7, plus(3,4));
}
```

Теперь, когда у нас есть еще один пример, мы можем сделать реализацию метода `plus()` абстрактной:

```
private int plus(int augend, int addend) {
    return augend + addend;
}
```

Триангуляция выглядит привлекательно, так как правила ее выполнения вполне понятны. Правила для паттерна **Fake It** (Подделка) основаны на ощущении дублирования кода между тестом и поддельным кодом. Это ощущение может быть субъективным, поэтому правила выглядят несколько туманными. Несмотря

на то что они кажутся простыми, правила триангуляции создают замкнутый цикл. После того как мы написали два выражения `assert` и сформировали абстрактную корректную реализацию метода `plus()`, мы можем уничтожить одно из выражений `assert`, так как теперь оно является избыточным. Однако, как только мы это сделаем, мы можем упростить реализацию `plus()` таким образом, чтобы этот метод возвращал константу. После этого нам надо будет снова добавить выражение `assert`.

Я использую триангуляцию только в случае, если я действительно не уверен, какая абстракция является корректной. В других ситуациях я предпочитаю использовать `Fake It` (Подделка) или `Obvious Implementation` (Очевидная реализация).

Obvious Implementation (Очевидная реализация)

Какой способ подходит для реализации простых операций? Просто реализуйте их.

Паттерны `Fake It` (Подделка) и `Triangulate` (Триангуляция) позволяют вам двигаться маленькими шажками. Но иногда вы абсолютно уверены в том, как можно корректно реализовать операцию. Вперед! Пишите то, что вы думаете. Например, должен ли я использовать `Fake It` (Подделка) для реализации чего-либо столь же простого, как метод `plus()`? Как правило, нет. Обычно для таких простых методов я просто пишу очевидную реализацию. Если при этом передо мной неожиданно появляется красная полоса за красной полосой, я перехожу на более короткий шаг.

В паттернах `Fake It` (Подделка) и `Triangulate` (Триангуляция) не существует никакой особенной добродетели. Если вы знаете, что писать, и если это получится достаточно быстро, то смело пишите готовый код. Однако помните о том, что используя только очевидную реализацию, вы требуете от себя совершенства¹. С психологической точки зрения это может быть разрушительный ход. Что если то, что вы написали, на самом деле не является самым простым изменением, которое заставляет тест работать? Что если ваш партнер покажет вам еще более простой вариант кода? Вы проиграли! Весь мир вокруг вас рассыпается на мелкие кусочки.

Известный слоган гласит: «чистый код, который работает» (`clean code that works`). Если вы будете решать проблему «чистый код» одновременно с проблемой «который работает», для вас это может оказаться слишком много. Как только вы поймете это, вернитесь обратно к решению проблемы «который работает» и только после этого принимайтесь за решение проблемы «чистый код».

При использовании паттерна `Obvious Implementation` (Очевидная реализация) следите за тем, насколько часто вы сталкиваетесь с красной полосой. Часто приходится попадать в ловушку: я записываю очевидную реализацию, но она не работает. Но теперь я точно знаю, что именно я должен написать. Поэтому я вношу в код изменения. Однако тест по-прежнему не работает. Но теперь-то я уж точно

¹ Спасибо Лауренту Боссави (Laurent Bossavit) за дискуссию.

знаю... Это часто случается при возникновении ошибок типа «индекс отличается на единицу» и «положительные/отрицательные числа».

Прежде всего вы должны следить за соблюдением ритма красный—зеленый—рефакторинг. Очевидная реализация — это вторая передача. Будьте готовы снизить скорость, если ваш мозг начинает выписывать чеки, которые не могут быть оплачены вашими пальцами.

One to Many (От одного ко многим)

Каким образом реализовать операцию, которая работает с коллекцией объектов? Сначала реализуйте эту операцию в отношении одного объекта, затем модернизируйте ее для работы с коллекцией таких объектов.

Например, предположим, что мы разрабатываем функцию, которая суммирует массив чисел. Мы можем начать с одного числа:

```
public void testSum() {
    assertEquals(5, sum(5));
}
private int sum(int value) {
    return value;
}
```

(Я добавил метод `sum()` в класс `TestCase`, чтобы не создавать новый класс для всего одного метода.)

Теперь мы хотим протестировать `sum(new int[] {5, 7})`. Для начала добавим в метод `sum()` параметр, соответствующий массиву целых чисел:

```
public void testSum() {
    assertEquals(5, sum(5, new int[] {5}));
}
private int sum(int value, int[] values) {
    return value;
}
```

Этот этап можно рассматривать как пример применения паттерна `Isolate Change` (Изоляция изменения). После того как мы добавили параметр, мы можем менять реализацию, не затрагивая код теста.

Теперь мы можем использовать коллекцию вместо единственного значения:

```
private int sum(int value, int[] values) {
    int sum = 0;
    for (int i = 0; i < values.length; i++)
        sum += values[i];
    return sum;
}
```

Теперь мы можем удалить неиспользуемый параметр:

```
public void testSum() {
    assertEquals(5, sum(new int[] {5}));
}
```

```
private int sum(int[] values) {  
    int sum= 0;  
    for (int i= 0; i<values.length; i++)  
        sum += values[i];  
    return sum;  
}
```

Предыдущий шаг — это тоже демонстрация Isolate Change (Изоляция изменения). Мы изменили код и в результате можем менять тест, не затрагивая код. Теперь мы можем расширить тест так, как планировали:

```
public void testSum() {  
    assertEquals(12, sum(new int[] {5, 7}));  
}
```

Паттерны JUnit

В этой главе рассматриваются паттерны, предназначенные для использования при работе с JUnit.

Assertion

Каким образом убедиться в том, что тест работает корректно? Напишите булевское выражение, которое автоматически подтверждает ваше мнение о том, что код работает.

Если мы хотим сделать тесты полностью автоматическими, значит, абсолютно все предположения о работе тестируемого кода необходимо превратить в тесты, при этом результат выполнения этих тестов должен говорить нам, работает ли код корректно или нет. Проще говоря, мы должны обладать возможностью щелкнуть на кнопке и через короткое время узнать, работает ли код корректно или нет. Отсюда следует, что:

- в результате выполнения теста должно получиться булевское значение: «истина» (True) указывает на то, что все в порядке, а «ложь» указывает на то, что произошло нечто непредвиденное;
- проверка результата срабатывания каждого теста выполняется компьютером автоматически при помощи какой-либо разновидности оператора `assert()`.

Мне приходилось видеть выражения наподобие `assertTrue(rectangle.area() != 0)`. Для того чтобы тест сработал, метод `area()` должен вернуть любое ненулевое значение — это не очень полезный тест. Рекомендуется делать тесты более конкретными. Если площадь прямоугольника должна быть равна 50, так и пишите: `assertTrue(rectangle.area() == 50)`. Во многих реализациях JUnit присутствует специальное выражение `assert()` для тестирования равенства (эквивалентности). Отличительная черта состоит в том, что вместо одного булевского параметра выражение `assertEquals()` принимает два произвольных объекта и пытается определить, являются ли они эквивалентными. Преимущество состоит в том, что в случае, если тест не срабатывает, выражение `assertEquals()` генерирует более информативное сообщение с указанием двух несовпадающих значений. Ожидаемое значение,

как правило, указывается первым. Например, предыдущее выражение в среде JUnit можно переписать следующим образом: `assertEquals(50, rectangle.area())`.

Думать об объектах, как о черных ящиках, достаточно тяжело. Представим, что у нас есть объект `Contract`, состояние которого содержится в поле `status`, которое может принадлежать либо классу `Offered`, либо классу `Running`. В этом случае можно написать тест, исходя из предполагаемой реализации:

```
Contract contract= new Contract(); // по умолчанию состояние Offered
contract.begin(); // состояние меняется на Running
assertEquals(Running.class, contract.status.class);
```

Этот тест слишком сильно зависит от текущей реализации объекта `status`. Однако тест должен срабатывать даже в случае, если поле `status` станет булевым значением. Может быть, когда `status` меняется на `Running`, можно протестировать дату начала работы над контрактом:

```
assertEquals(. . . contract.startDate()); // генерирует исключение, если
// status равен Offered
```

Я признаю, что пытаюсь плыть против течения, когда настаиваю на том, что все тесты должны быть написаны только с использованием публичного (`public`) протокола. Существует специальный пакет `JXUnit`, который является расширением `JUnit` и позволяет тестировать значения переменных, даже `tex`, которые объявлены как закрытые.

Желание протестировать объект в рамках концепции белого ящика — это не проблема тестирования, это проблема проектирования. Каждый раз, когда у меня возникает желание протестировать значение переменной-члена для того, чтобы убедиться в работоспособности кода, я получаю возможность улучшить дизайн системы. Если я забываю о своих опасениях и просто проверяю значение переменной, я теряю такую возможность. Иначе говоря, если идея об улучшении дизайна не приходит мне в голову, ничего не поделаешь. Я проверяю значение переменной, смахиваю непрошеную слезу, вношу соответствующую отметку в список задач и продолжаю двигаться вперед, надеясь, что наступит день, когда смогу найти подходящее решение.

Самая первая версия `xUnit` для `Smalltalk` (под названием `SUnit`) обладала очень простыми выражениями `assert`. Если одно из выражений не срабатывало, автоматически открывалось окно отладчика, вы исправляли код и продолжали работу. Среда разработки `Java` не настолько совершенна, к тому же построение приложений `Java` часто выполняется в пакетном режиме, поэтому имеет смысл добавлять в выражение `assert()` дополнительную информацию о проверяемом условии. Таким образом, если `assert()` не срабатывает, на экране отображается дополнительное информационное сообщение.

В `JUnit` это реализуется при помощи необязательного первого параметра¹. Например, если вы напишете `assertTrue("Должно быть True", false)` и тест не работает,

¹ Обычно необязательные параметры располагаются в конце списка аргументов, однако в данном случае необязательная информационная строка размещается в начале списка, так как благодаря этому тесты удобнее читать.

то вы увидите на экране приблизительно следующее сообщение: `Assertion failed: Должно быть True`. Обычно подобного сообщения достаточно, чтобы направить вас напрямую к источнику ошибки в коде. В некоторых группах разработчиков действует жесткое правило, что все выражения `assert()` должны снабжаться подобными информационными сообщениями. Попробуйте оба варианта и самостоятельно определите, окупаются ли для вас затраты, связанные с информационными сообщениями.

Fixture¹ (Фикстура)

Каким образом осуществляется создание общих объектов, которые используются в нескольких тестах? Конвертируйте локальные переменные из тестов в переменные-члены класса `TestCase`. Переопределите метод `setUp()` и инициализируйте в нем эти переменные (то есть выполните создание всех необходимых объектов).

Если мы привыкли удалять дублирование из функционального (тестируемого) кода, должны ли мы удалять его из тестирующего кода? Может быть.

Существует проблема: зачастую вам приходится писать больше кода для того, чтобы установить объекты, используемые тестируемым методом, в интересующее вас состояние. Код, инициализирующий объекты, часто оказывается одинаковым для нескольких тестов. Такие объекты называются фикстурой теста (используется также английский термин *scaffolding* — *строительные леса, подмости*). Дублирование подобного кода — это плохо. Вот две основные причины:

- написание подобного кода требует дополнительного времени, даже если для этого требуется просто скопировать блок текста через буфер обмена, а мы хотим добиться того, чтобы написание тестов занимало как можно меньше времени;
- если нам приходится вручную менять интерфейс, перед нами встает необходимость изменять его в нескольких разных тестах (именно этого всегда следует ожидать от дублирования).

Однако дублирование кода инициализации объектов обладает также некоторыми преимуществами. Если код инициализации располагается непосредственно рядом с тестирующими выражениями `assert()`, весь код теста можно прочесть от начала и до конца. Если мы выделили инициализационный код в отдельный метод, нам приходится помнить о том, что этот метод вызывается, нам приходится вспоминать, как именно выглядят объекты, и только вспомнив все это, мы можем написать остальную часть теста.

Среда xUnit поддерживает оба стиля написания тестов. Если вы думаете, что читателям будет сложно вспомнить объекты фикстуры, вы можете разместить код создания фикстуры непосредственно в теле теста. Однако вы также можете переместить этот код в метод с названием `setUp()`. В этом методе вы можете создать все объекты, которые будут использоваться в тестовых методах.

¹ В переводе с английского *fixture* означает *арматура* или *фиксированная принадлежность некоторого движущегося механизма*. — Прим. перев.

Далее приводится пример, который слишком прост для того, чтобы мотивировать выделение общего кода фикстуры, но зато достаточно короток, чтобы поместиться в данной книге. Мы можем написать:

EmptyRectangleTest

```
public void testEmpty() {
    Rectangle empty= new Rectangle(0.0,0.0,0);
    assertTrue(empty.isEmpty());
}

public void testWidth() {
    Rectangle empty= new Rectangle(0.0,0.0,0);
    assertEquals(0.0, empty.getWidth(), 0.0);
}
```

(Помимо прочего здесь также демонстрируется версия `assertEquals()` для чисел с плавающей точкой, которая принимает третий параметр — точность сравнения.) Мы можем избавиться от дублирования, написав:

EmptyRectangleTest

```
private Rectangle empty;

public void setUp() {
    empty= new Rectangle(0.0,0.0,0);
}

public void testEmpty() {
    assertTrue(empty.isEmpty());
}
public void testWidth() {
    assertEquals(0.0, empty.getWidth(), 0.0);
}
```

Общий код выделен в виде отдельного метода. Среда `xUnit` гарантирует, что метод `setUp()` объекта `TestCase` будет обязательно вызван перед обращением к любому тестовому методу этого объекта. Теперь тестовые методы выглядят проще, однако, прежде чем понять их смысл, мы должны вспомнить о существовании метода `setUp()` и уточнить, что происходит внутри этого метода.

Какой из этих двух стилей предпочтительней? Попробуйте использовать каждый из них. Я фактически всегда выделяю общий код фикстуры и перемещаю его в метод `setUp()`, однако у меня хорошая память. Те, кто читает мои тесты, часто жалуются на то, что им приходится вспоминать слишком о многом. Значит, возможно, мне следует выделять меньший объем кода, чтобы сделать тесты более понятными.

Взаимоотношения между подклассами класса `TestCase` и экземплярами этих подклассов являются наиболее запутанной стороной инфраструктуры `xUnit`. Каждый новый тип фикстуры требует создания нового подкласса класса `TestCase`. Каждая новая фикстура создается внутри экземпляра подкласса, используется один раз, а затем уничтожается.

В предыдущем примере, если мы хотим написать тесты для непустого прямоугольника (Rectangle), нам придется создать новый класс, который можно назвать, например, NormalRectangleTest. У этого класса будет свой собственный метод setUp(), в котором будет создан новый экземпляр объекта Rectangle, необходимый ему для тестирования. Этот Rectangle будет соответствовать непустому прямоугольнику. В общем случае, если я хочу использовать несколько отличающуюся фикстуру, я создаю новый подкласс класса TestCase.

Это означает, что не существует прямого простого соответствия между классами тестов и функциональными (тестируемыми) классами. Иногда одна фикстура используется для тестирования нескольких классов (подобное случается не часто). Иногда для тестирования единственного функционального класса требуется создать две или три фикстуры. На практике в большинстве случаев получается, что количество классов тестов приблизительно совпадает с количеством функциональных классов. Однако это происходит вовсе не потому, что для каждого функционального класса вы создаете один-единственный класс теста.

External Fixture (Внешняя фикстура)

Каким образом осуществляется освобождение внешних ресурсов в фикстуре? Переопределите метод tearDown() и освободите в нем ресурсы, выделенные в рамках создания фикстуры.

Помните о том, что каждый тест должен оставить рабочую среду в том же состоянии, в котором она была до того, как тест начал работу. Например, если внутри теста вы открываете файл, вы должны позаботиться о том, чтобы закрыть этот файл перед тем, как тест завершит работу. Вы можете написать:

```
testMethod(self):
    file= File("foobar").open()
    try:
        ...выполнить тест...
    finally:
        file.close()
```

Если файл используется в нескольких тестах, вы можете сделать его частью общей фикстуры:

```
setUp(self):
    self.file= File("foobar").open()
testMethod(self):
    try:
        ...выполнить тест...
    finally:
        self.file.close()
```

Во-первых, возникает неприятное дублирование выражений finally — это означает, что мы упустили что-то в дизайне. Во-вторых, при написании подобного метода можно легко допустить ошибку, например забыть добавить ключевое слово finally или вообще забыть о необходимости закрытия файла. Наконец, в этом тесте

существует три сбивающих с толку строки — `try`, `finally` и сама команда `close`, — эти выражения не относятся непосредственно к процедуре тестирования.

Инфраструктура `xUnit` гарантирует, что метод под названием `tearDown()` будет обязательно вызван после выполнения тестового метода. Метод `tearDown()` будет вызван вне зависимости от того, что случится внутри тестового метода (однако следует иметь в виду, что если сбой произойдет в ходе выполнения метода `setUp()`, метод `tearDown()` вызываться не будет). Мы можем преобразовать предыдущий тест следующим образом:

```
setUp(self):
    self.file= File("foobar").open()
testMethod(self):
    ...выполнить тест...
tearDown(self):
    self.file.close()
```

Test Method (Тестовый метод)

Что такое единичный тест? Это метод, имя которого начинается с префикса `test`.

В процессе разработки вам придется иметь дело с сотнями, а может быть и тысячами тестов, каким образом можно уследить за всеми этими тестами?

Языки объектно-ориентированного программирования обеспечивают трехуровневую организацию исходного кода:

- модуль (в языке Java — *пакет*, по-английски, *package*);
- класс;
- метод.

Если мы пишем тесты, как обычный исходный код, мы должны найти способ структурирования тестов с использованием элементов этой структуры. Если мы используем классы для представления фикстур, значит, методы этих классов являются естественным местом размещения тестирующего кода. Все тесты, использующие некоторую фикстуру, должны быть методами одного класса. Тесты, работающие с другой фиксурой, должны располагаться в другом классе.

В `xUnit` используется соглашение, в соответствии с которым имя тестового метода должно начинаться с префикса `test`. Специальные инструменты могут автоматически производить поиск таких методов и создавать из них наборы тестов (`TestSuite`). Остальная часть имени теста должна информировать будущего ни о чем не ведающего читателя, зачем написан данный тест. Например, в наборе тестов, созданных при разработке инфраструктуры `JUnit`, можно обнаружить тест с именем `testAssertPosInfinityNotEqualsNegInfinity`. Я не помню, чтобы я писал этот тест, однако, исходя из имени, я предполагаю, что в какой то момент разработки было обнаружено, что код метода `assert()` инфраструктуры `JUnit` для чисел с плавающей точкой не делал различия между положительной и отрицательной бесконечностью. Используя тест, я могу быстро найти код `JUnit`, осуществляющий сравнение чисел с плавающей точкой, и посмотреть, как осуществляется обработка

положительной и отрицательной бесконечности. (На самом деле код выглядит не идеально — для поддержки бесконечности используется условный оператор).

Код тестового метода должен быть очень прост в прочтении и очень прямолинеен. Если вы разрабатываете тест и видите, что его код становится слишком длинным, попробуйте поиграть в «детские шашки». Цель игры — написать самый маленький тестовый метод, который представляет собой реальный прогресс в направлении вашей конечной цели. Размер в три строки, судя по всему, является минимальным размером (если, конечно, вы не хотите делать тест намеренно бессмысленным). И постоянно помните о том, что вы пишете тесты для людей, а не только для компьютера и себя самого.

Патрик Логан (Patrick Logan) рассказал об идее, с которой я намерен поэкспериментировать. Эта идея также описана Макконнеллом (McConnell)¹, а также Кэйном (Caine) и Гордоном (Gordon)²:

В последнее время я фактически постоянно применяю методику «основных тезисов» (outlines) в любой моей работе. Тестирование не является исключением. Когда я пишу тесты, я прежде всего записываю план из нескольких пунктов — тезисов, — которые я хотел бы реализовать в этом тесте. Например:

```
/* Добавить в пространство кортежей3 */
/* Извлечь из пространства кортежей */
/* Читать из пространства кортежей */
```

Это самые основные тезисы, однако я добавляю в каждую из этих категорий конкретные тесты. Когда я добавляю тесты, я добавляю в мой список тезисов еще один уровень комментариев:

```
/* Добавить в пространство кортежей */
/* Извлечь из пространства кортежей */
/** Извлечение несуществующего элемента **/
/** Извлечение существующего элемента **/
/** Извлечение нескольких элементов **/
/* Читать из пространства кортежей */
```

Как правило, мне хватает двух-трех уровней комментариев. Я не могу представить себе ситуацию, в которой мне могло бы потребоваться больше уровней. Список тезисов становится документацией контракта для тестируемого класса. Приведенные здесь примеры, конечно же, сокращены, однако в языках программирования, поддерживающих контракты, тезисы могли бы быть более конкретными. (Я не использую какие-либо добавления к Java, обеспечивающие автоматизацию в стиле Eiffel.)

Сразу же после самого низкого уровня комментариев располагается исходный код теста.

¹ McConnell, Steve. 1993. *Code Complete*, chapter 4. Seattle, Washington: Microsoft Press.

² Caine, S. H., Gordon, E. K. 1975. PDL: A Tool for Software Design, *AFIPS Proceedings of the 1975 National Computer Conference*.

³ В среде англоязычных программистов запись базы данных иногда обозначается термином *tuple* — кортеж. — Примеч. перев.

Exception Test (Тест исключения)

Каким образом можно протестировать ожидаемое исключение? Перехватите исключение и игнорируйте его, сбой теста происходит только в случае, если исключение не сгенерировано.

Предположим, что мы пишем код, осуществляющий поиск значения. Если значение не обнаружено, мы хотим сгенерировать исключение. Тестирование механизма поиска выполняется относительно просто:

```
public void testRate() {
    exchange.addRate("USD", "GBP", 2);
    int rate= exchange.findRate("USD", "GBP");
    assertEquals(2, rate);
}
```

Тестирование исключения может оказаться неочевидным. Вот как мы это делаем:

```
public void testMissingRate() {
    try {
        exchange.findRate("USD", "GBP");
        fail();
    } catch (IllegalArgumentException expected) {
    }
}
```

Если метод `findRate()` не генерирует исключения, произойдет обращение к методу `fail()` — это метод `xUnit`, который докладывает о том, что тест не сработал. Обратите внимание, что мы перехватываем только то исключение, которое должно быть сгенерировано методом `findRate()`. Благодаря этому, если будет сгенерировано какое-либо другое (неожиданное для нас) исключение (включая сбой метода `assert`), мы узнаем об этом.

All Tests (Все тесты)

Каким образом можно запустить все тесты вместе? Создайте тестовый набор, включающий в себя все имеющиеся тестовые наборы, — один для каждого пакета (`package`) и один, объединяющий в себе все тесты пакетов для всего приложения.

Предположим, вы добавили подкласс класса `TestCase` и в этот подкласс вы добавили тестовый метод. В следующий раз, когда будут выполняться все тесты, добавленный вами тестовый метод также должен быть выполнен. (Во мне опять проснулась привычка действовать в стиле TDD — должно быть вы заметили, что предыдущее предложение — это эскиз теста, который я, наверное, написал бы, если бы не был занят работой над данной книгой.) К сожалению, в большинстве реализаций `xUnit`, равно как и в большинстве IDE, не поддерживается стандартный механизм запуска абсолютно всех тестов, поэтому в каждом пакете необходимо определить класс `AllTests`, который реализует статический метод `suite()`, возвращающий объект класса `TestSuite`. Вот класс `AllTests` для «денежного» примера:

```
public class AllTests {
    public static void main(String[] args) {
        junit.swingui.TestRunner.run(AllTests.class);
    }
    public static Test suite() {
        TestSuite result= new TestSuite("TFD tests");
        result.addTestSuite(MoneyTest.class);
        result.addTestSuite(ExchangeTest.class);
        result.addTestSuite(IdentityRateTest.class);
        return result;
    }
}
```

Вы также должны включить в класс `AllTests()` метод `main()`, благодаря чему класс можно будет запустить напрямую из IDE или из командной строки.

Паттерны проектирования

В чем заключается основная идея паттернов? Нам кажется, что мы постоянно сталкиваемся с разнообразными, не повторяющимися проблемами, однако на деле оказывается, что большая часть проблем, которые нам приходится решать, обусловлена используемыми нами инструментами, но не основной задачей, которая перед нами стоит¹. Если исходить из этого предположения, то можно найти (и мы действительно находим) общие проблемы со стандартными решениями, несмотря на все разнообразие контекстов, в рамках которых нам приходится работать.

Использование объектов для организации вычислений — это один из лучших примеров стандартного решения, направленного на устранение множества общих проблем, с которыми программистам приходится сталкиваться при разработке самого разнообразного программного обеспечения. Колоссальный успех паттернов проектирования (design patterns) является доказательством общности проблем, с которыми сталкиваются объектно-ориентированные программисты². Книга *Design Patterns* (Паттерны проектирования) имела большой успех, однако популярность этой книги стала причиной сужения взгляда на паттерны проектирования. Что я имею в виду? Книга рассматривает дизайн как фазу разработки программы, однако авторы совершенно не учитывают, что рефакторинг — это мощный инструмент формирования дизайна. Дизайн в рамках TDD требует несколько иного взгляда на паттерны проектирования.

В данной главе я расскажу о нескольких полезных паттернах проектирования. Безусловно, мое изложение не претендует на то, чтобы быть полным. Представленная здесь информация о паттернах может оказаться полезной при изучении рассматриваемых в книге примеров. Вот краткое перечисление рассмотренных здесь паттернов:

¹ Alexander Christopher. 1970. *Notes on the Synthesis of Form*. Cambridge, MA: Harvard University Press.

² Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. 1995. *Design Patterns: Elements of Reusable Object Oriented Software*. Boston: Addison-Wesley. Русское издание: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. *Приемы объектно-ориентированного проектирования*. Паттерны проектирования. СПб.: Питер, 2001.

- **Command (Команда)** — обращение к некоторому коду представляется в виде объекта, а не в виде простого сообщения;
- **Value Object (Объект-значение)** — после создания объекта его значение никогда не меняется, благодаря этому удается избежать проблем, связанных с наложением имен (aliasing);
- **Null Object (Ноль-объект)** — соответствует базовому случаю вычислений объекта;
- **Template Method (Шаблонный метод)** — представляет инвариантную последовательность операций, определяемую при помощи абстрактных методов, которые можно переопределить с помощью наследования.
- **Pluggable Object (Встраиваемый объект)** — представляет вариацию при помощи обращения к другому объекту с двумя или большим количеством реализаций;
- **Pluggable Selector (Встраиваемый переключатель)** — позволяет избежать создания многочисленных подклассов путем динамического обращения к различным методам для различных экземпляров класса;
- **Factory Method (Фабричный метод)** — вместо конструктора для создания объекта используется специальный метод;
- **Imposter (Самозванец)** — представляет вариацию при помощи создания новой реализации уже существующего протокола;
- **Composite (Компоновщик)** — композиция объектов ведет себя так же, как один объект;
- **Collecting Parameter (Накопление в параметре)** — результаты вычислений, выполняемых в разных объектах, накапливаются в специальном объекте, который передается объектам, выполняющим вычисления, в качестве параметра.

В табл. 30.1 описывается, на каких этапах TDD используется тот или иной паттерн проектирования.

Таблица 30.1. Использование паттернов проектирования при разработке, основанной на тестировании (TDD)

Паттерн	Написание тестов	Рефакторинг
Command (Команда)	✓	
Value Object (Объект-значение)	✓	
Null Object (Ноль-объект)		✓
Template Method (Шаблонный метод)		✓
Pluggable Object (Встраиваемый объект)		✓
Pluggable Selector (Встраиваемый переключатель)		✓
Factory Method (Фабричный метод)	✓	✓
Imposter (Самозванец)	✓	✓
Composite (Компоновщик)	✓	✓
Collecting Parameter (Накопление в параметре)	✓	✓

Command (Команда)

Что делать, если выполнение некоторой операции представляет собой нечто более сложное, чем простое обращение к методу? Создайте объект, соответствующий этой операции, и обратитесь к этому объекту.

Передача сообщений — это отличный механизм. Языки программирования делают передачу сообщений синтаксически простым действием, а среды разработки позволяют с легкостью манипулировать сообщениями (например, автоматическое выполнение рефакторинга по переименованию метода). Однако в некоторых случаях простой передачи сообщения не достаточно.

Например, представьте, что вы хотели бы занести в журнал запись о том, что сообщение было передано. Для этой цели можно воспользоваться средствами языка (например, методы-оболочки), однако простые операции протоколирования — это далеко не все, в чем вы можете нуждаться. Представьте, что мы хотим вызвать некоторую процедуру, но несколько позднее. Для этой цели можно создать новый программный поток, сразу же приостановить его работу, а затем, когда это потребуются, запустить его. Однако в подобной ситуации нам придется иметь дело с параллельными потоками, а это может оказаться слишком тяжеловесным подходом.

Для выполнения операций с подобными дополнительными условиями зачастую требуются сложные затратные механизмы. Однако в большинстве случаев мы можем избежать излишней сложности и лишних затрат. Проблему вызова можно решить с помощью более конкретной и гибкой формы, чем сообщение. Для этого достаточно создать специальный объект. Создайте объект, представляющий собой вызов операции. Занесите в этот объект все необходимые параметры операции. Когда операция готова к выполнению, используйте для этого универсальный протокол, например метод `run()`.

Отличным примером использования данного подхода является интерфейс `Runnable` языка Java:

`Runnable`

```
interface Runnable
    public abstract void run();
```

В рамках реализации метода `run()` вы можете делать все, что вам нравится. К сожалению, Java не поддерживает синтаксически легковесного способа создания объектов `Runnable` и обращения к этим объектам, поэтому они не используются так часто, как их эквиваленты в других языках (блоки или лямбда-выражения в Smalltalk/Ruby или LISP).

Value Object (Объект-значение)

Каким образом следует спроектировать объект, который будет широко использоваться, но для которого идентификация не имеет особого значения? Настройте состояние объекта в момент его создания и никогда не меняйте его. В результате

выполнения любых операций в отношении данного объекта должен получаться новый объект.

Объектно-ориентированный подход — это великолепная вещь. Надеюсь, я имею право написать эту фразу в данной книге. Объекты являются отличным способом организовать логику для последующего понимания и роста. Однако существует одна маленькая проблема (хорошо, хорошо, вообще-то проблем больше, однако сейчас мы коснемся только одной из них).

Представьте, что я — объект, и у меня есть прямоугольник (Rectangle). Я вычисляю некоторое значение, зависящее от этого прямоугольника, например его площадь. Чуть позже некто (например, другой объект) вежливо просит меня предоставить ему мой прямоугольник для выполнения некоторой операции. Чтобы не показаться невежливым, я предоставляю ему мой прямоугольник. А через пару мгновений, вы только посмотрите, прямоугольник был модифицирован у меня за спиной! Значение площади, которое я вычислил ранее, теперь не соответствует действительности и не существует способа известить меня об этом.

Это классический пример проблемы наложения имен (aliasing). Если два объекта ссылаются на один и тот же третий объект и если один из первых двух тем или иным образом изменяет третий, общий для них, объект, второму объекту лучше не полагаться на текущее состояние общего объекта.

Существует несколько способов решения проблемы наложения имен. Во-первых, вы можете никому не отдавать объект, от состояния которого вы зависите. Вместо этого в случае необходимости вы можете создавать копии этого объекта. Такой подход может потребовать слишком много времени и слишком много пространства, кроме того, игнорируется ситуация, когда вы хотите сделать изменения некоторого объекта общими для нескольких других объектов, зависящих от его состояния. Еще одно решение — паттерн Observer (Обозреватель). В этом случае, если вы зависите от состояния некоторого объекта, вы должны предварительно сообщить ему об этом, иначе говоря, зарегистрироваться. Объект, за состоянием которого следят, оповещает все зарегистрированные им объекты-наблюдатели о своем изменении. Паттерн Observer (Обозреватель) может затруднить понимание последовательности выполнения операций, кроме того, логика формирования и удаления зависимостей между объектами выглядит далеко не идеальной.

Еще одно решение предлагает несколько ограничить возможности, которыми обладает типичный объект в рамках ООП. Образно говоря, объект становится «менее чем объектом». Что это значит? Обычные объекты обладают состоянием, которое изменяется с течением времени. Если мы захотим, мы можем запретить им меняться. Если у меня есть объект и я знаю, что он не может измениться, я могу передавать ссылки на этот объект любому другому объекту, не беспокоясь при этом о проблеме наложения имен. Если объект не поддерживает возможности своего изменения, никаких модификаций у меня за спиной не может произойти.

Я помню, как похожая ситуация возникла с целыми числами, когда я впервые изучал язык Smalltalk. Если я изменяю бит 2 на 1, почему все двойки не становятся шестерками?

```

a := 2.
b := a.
a := a bitAt: 2 put: 1.
a => 6
b => 2

```

Целые числа — это значения, которые маскируются под объекты. В языке Smalltalk это утверждение является истиной для небольших целых чисел и имитируется в случае, если целое число не уместается в рамках машинного слова. Когда я устанавливаю бит, то получаю в свое распоряжение новый объект с установленным битом. Старый объект остается неизменным.

В рамках Value Object (Объект-значение) каждая операция должна возвращать новый объект, изначальный объект должен оставаться неизменным. Пользователи должны знать, что они используют объект-значение. В этом случае полученный объект следует сохранить (как в предыдущем примере). Конечно же, из-за необходимости создания новых объектов полученный в результате код может оказаться медленным. Однако в данном случае любые проблемы с производительностью должны решаться в точности так же, как и любые другие проблемы с производительностью: вы должны оценить производительность при помощи тестов с реальными данными, определить, насколько часто производится обращение к медленному коду, выполнить профилирование и определить, какой именно код должен быть оптимизирован и как лучше всего этого достичь.

Я предпочитаю использовать Value Object (Объект-значение) в ситуациях, когда операции, выполняемые над объектами, напоминают алгебру. Например, пересечение и объединение геометрических фигур, операции над значениями, с каждым из которых хранится единица измерения, а также операции символической арифметики. Каждый раз, когда использование Value Object (Объект-значение) имеет хоть какой-то смысл, я пытаюсь его использовать, так как результирующий код проще читать и отлаживать.

Все объекты-значения должны реализовать операцию сравнения (а во многих языках подразумевается, что они должны реализовать также операцию хеширования). Если я имею один контракт и другой контракт и они не являются одним и тем же объектом, значит, они не равны. Однако если у меня есть одни пять франков и другие пять франков, для меня не имеет значения тот факт, что это два разных объекта — пять франков и в Африке пять франков — они равны.

Null Object (Ноль-объект)

Каким образом реализовать специальные случаи использования объектов? Создать специальный объект, представляющий собой специальный случай. Специальный объект должен обладать точно таким же протоколом, что и обычный объект, но он должен вести себя специальным образом.

В качестве примера рассмотрим код, который я позаимствовал из `java.io.File`:

```
java.io.File
```

```

public boolean setReadOnly() {
    SecurityManager guard = System.getSecurityManager();

```

```

    if (guard != null) {
        guard.canWrite(path);
    }
    return fileSystem.setReadOnly(this);
}

```

В классе `java.io.File` можно обнаружить 18 проверок `guard != null`. Я преклоняюсь пред усердием, с которым разработчики библиотек Java стараются сделать файлы безопасными для всего остального мира, однако я также начинаю немножко нервничать. Будут ли программисты Sun и в будущем столь же аккуратны, чтобы не забыть проверить результат выполнения метода `getSecurityManager()` на равенство значению `null`?

В рамках альтернативного решения можно создать новый класс `LaxSecurity`, который вообще не генерирует исключений:

`LaxSecurity`

```

public void canWrite(String path) {
}

```

Если кто-то пытается получить `SecurityManager`, однако предоставить такой объект нет возможности, вместо него мы возвращаем `LaxSecurity`:

`SecurityManager`

```

public static SecurityManager getSecurityManager() {
    return security == null ? new LaxSecurity() : security;
}

```

Теперь мы можем не беспокоиться о том, что кто-то забудет проверить результат выполнения метода на равенство значению `null`. Изначальный код становится существенно более чистым:

`File`

```

public boolean setReadOnly() {
    SecurityManager security = System.getSecurityManager();
    security.canWrite(path);
    return fileSystem.setReadOnly(this);
}

```

Однажды во время выступления на конференции OOPSLA нас с Эрихом Гаммой (Erich Gamma) спросили, можно ли использовать Null Object (Нуль-объект) в рамках одного из классов `JHotDraw`. Я принялся рассуждать о преимуществах такой модернизации, в то время как Эрих посчитал, что для этого нам придется увеличить код на десять строк, при этом мы избавимся от одного условного оператора — преимущество сомнительно. (К тому же аудитория была весьма недовольна нашей несогласованностью.)

Template Method (Шаблонный метод)

Каким образом можно запрограммировать инвариантную последовательность операций, обеспечив при этом возможность модификации или замены отдельных действий в будущем? Напишите метод, который реализуется исключительно в терминах других методов.

В программировании существует огромное количество классических последовательностей:

- ввод—обработка—вывод;
- отправить сообщение—принять ответ;
- прочитать команду—вернуть результат.

Нам хотелось бы четко и понятно обозначить универсальность этих последовательностей и при этом обеспечить возможность варьирования реализаций каждого из отдельных этапов.

Поддерживаемый любым объектно-ориентированным языком механизм наследования обеспечивает простой способ определения универсальных последовательностей. В суперклассе создается метод, целиком и полностью написанный в терминах других методов. Каждый из подклассов может реализовать эти методы так, как ему удобнее. Например, базовая последовательность выполнения теста определяется в инфраструктуре JUnit следующим образом:

TestCase

```
public void runBare() throws Throwable {
    setUp();
    try {
        runTest();
    }
    finally {
        tearDown();
    }
}
```

Классы, производные от `TestCase`, могут реализовать `setUp()`, `runTest()` и `tearDown()` так, как им этого хочется.

При использовании `Template Method` (Шаблонный метод) возникает вопрос: надо ли создавать для подметодов реализации по умолчанию? В `TestCase.runBare()` все три подметода обладают реализациями по умолчанию:

- методы `setUp()` и `tearDown()` не выполняют никаких операций;
- метод `runTest()` динамически обнаруживает и запускает все тестовые методы, исходя из имени класса-теста.

Если общая последовательность не имеет смысла тогда, когда не определен один из ее этапов, вы должны отметить это, воспользовавшись любой подходящей возможностью используемого вами языка программирования.

- в Java можно объявить под-метод абстрактным;
- в Smalltalk создайте реализацию метода, которая генерирует ошибку `SubclassResponsibility`.

Я не рекомендую изначально проектировать код так, чтобы в нем использовался шаблонный метод. Лучше всего формировать шаблонные методы, исходя из накопленного опыта. Каждый раз, когда я говорю себе: «Ага, вот последовательность, а вот — детали реализации», — позднее я всегда обнаруживаю, что мне приходится переделывать созданный мною шаблонный метод, заново перетасовывая код между общим и частным.

Если вы обнаружили два варианта последовательности в двух подклассах, вы должны попытаться постепенно приблизить их друг к другу. После того как вы отделите отличающиеся части и выделите общую часть, то, что останется, и есть шаблонный метод. После этого вы можете переместить шаблонный метод в суперкласс и избавиться от дублирования.

Pluggable Object (Встраиваемый объект)

Каким образом можно выразить несколько разных вариантов поведения кода? Проще всего использовать явный условный оператор:

```
if (circle) then {
    . . . код, относящийся к circle . . .
} else {
    . . . код, не относящийся к circle
}
```

Однако подобный корявый код имеет тенденцию распространяться по всей программе. Если для определения разницы между окружностями (circle) и не окружностями вы будете использовать условный оператор хотя бы в одном месте вашего кода, с большой долей уверенности можно сказать, что позднее подобный оператор придется добавлять также и в другом месте кода, затем в третьем и т. д.

Вторая по важности задача TDD — удаление дублирования, поэтому вы должны подавить угрозу распространения явных условных операторов в зародыше. Если вы видите, что одно и то же условие проверяется в двух разных местах вашего кода, значит, настало время выполнить базовое объектно-ориентированное преобразование: PluggableObject (Встраиваемый объект).

Иногда обнаружить необходимость применения этого паттерна не так уж и просто. Один из самых любимых мною примеров использования встраиваемого объекта был придуман мною и Эрихом Гаммой. Представьте, что мы занимаемся разработкой графического редактора. Если вы когда-нибудь занимались чем-либо подобным, должно быть, вы знаете, что операция выделения объектов обладает несколько усложненной логикой. Если курсор мыши находится над графической фигурой и пользователь нажимает на кнопку мыши, значит, последующие перемещения мыши приводят к перемещению фигуры, а при отпускании кнопки мыши выбранная фигура остается на новом месте. Если курсор мыши не находится над какой-либо фигурой, значит, нажав на кнопку, пользователь выделяет несколько фигур, последующие перемещения мыши приводят к изменению размера прямоугольника выделения, а при отпускании кнопки мыши фигуры внутри прямоугольника выделения становятся выделенными. Изначальный код выглядит примерно так:

SelectionTool

```
Figure selected;
public void mouseDown() {
    selected= findFigure();
    if (selected != null)
```



```

        select(selected);
    }
    public void mouseMove() {
        if (selected != null)
            move(selected);
        else
            mqveSelectionRectangle();
    }
    public void mouseUp() {
        if (selected == null)
            selectAll();
    }
}

```

В глаза бросаются три похожих условных оператора (я же говорил вам, что они плодятся, как мухи). Что делать, чтобы избавиться от них? Создаем встраиваемый объект, `SelectionMode`, обладающий двумя реализациями: `SingleSelection` и `MultipleSelection`.

SelectionTool

```

SelectionMode mode;
public void mouseDown() {
    selected= findFigure();
    if (selected != null)
        mode= SingleSelection(selected);
    else
        mode= MultipleSelection();
}
public void mouseMove() {
    mode.mouseMove();
}
public void mouseUp() {
    mode.mouseUp();
}
}

```

В языках с явными интерфейсами вы обязаны реализовать интерфейс с двумя (или больше) встраиваемыми объектами.

Pluggable Selector¹ (Встраиваемый переключатель)

Как обеспечить отличающееся поведение для разных экземпляров одного и того же класса? Сохраните имя метода в переменной и динамически обращайтесь к этому методу.

¹ Подробнее об этом паттерне рассказывается в книге Beck, К. 1997. *The Smalltalk Best Practice Patterns*, с. 70–73. Englewood-Cliffs, NJ: Prentice-Hall. Ссылаться на свои собственные работы — это не самая лучшая идея, однако, как говорил философ Филлис Диллер (Phyllis Diller): «Конечно же, я смеюсь над собственными шутками, просто я не доверяю никому, кроме себя».

Что делать, если у вас есть десять подклассов одного базового класса и в каждом из них реализован только один метод? Может оказаться, что создание подклассов — это слишком тяжеловесный механизм для реализации столь небольших отличий в поведении объектов.

```
abstract class Report {
    abstract void print();
}
class HTMLReport extends Report {
    void print() { ...
}
}
class XMLReport extends Report {
    void print() { ...
}
}
```

Альтернативное решение: создать единственный класс с оператором `switch`. В зависимости от значения поля происходит обращение к разным методам. Однако в этом случае имя метода упоминается в трех местах:

- при создании экземпляра;
- в операторе `switch`;
- в самом методе.

```
abstract class Report {
    String printMessage;

    Report(String printMessage) {
        this.printMessage= printMessage;
    }
    void print() {
        switch (printMessage) {
            case "printHTML" :
                printHTML();
                break;
            case "printXML" :
                printXML();
                break;
        }
    };

    void printHTML() {
    }

    void printXML() {
    }
}
```

Каждый раз, когда вы добавляете новую разновидность печати, вы должны позаботиться о добавлении нового метода печати и редактировании оператора `switch`.

Паттерн Pluggable Selector (Встраиваемый переключатель) предлагает динамически обращаться к методу с использованием Reflection API:

```
void print() {
    Method runMethod= getClass().getMethod(printMessage, null);
    runMethod.invoke(this, new Class[0]);
}
```

По-прежнему существует весьма неприятная зависимость между создателями отчетов и именами методов печати, однако, по крайней мере, мы избавились от оператора switch.

Естественно, этим паттерном не следует злоупотреблять. Самая большая связанная с ним проблема состоит в отслеживании вызываемого кода. Используйте встраиваемый переключатель только в случае, когда вы оказались в стандартной ситуации: каждый из подклассов обладает всего одним методом, и у вас есть желание сделать этот код более чистым.

Factory Method (Фабричный метод)

Как лучше всего создавать объекты в случае, если вы хотите обеспечить гибкость при создании объектов? Вместо того чтобы использовать конструктор, создайте объект внутри специального метода.

Безусловно, конструкторы являются выразительным инструментом. Если вы используете конструктор, всем, кто читает код, однозначно становится ясно, что вы создаете объект. Однако конструкторы, в особенности в Java, не обеспечивают достаточной гибкости.

В рассмотренном нами ранее «денежном» примере при создании объекта мы хотели бы возвращать объект иного класса. У нас есть следующий тест:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

Мы хотели бы добавить в программу новый класс Money, однако мы не можем этого сделать, так как для тестирования нам нужен экземпляр класса Dollar. Чтобы решить проблему, достаточно добавить в программу дополнительный уровень перенаправления — метод, который будет возвращать объект иного класса. В этом случае мы сможем оставить выражения assert без изменений:

```
public void testMultiplication() {
    Dollar five = Money.dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

Money

```
static Dollar dollar(int amount) {
```

```

    return new Dollar(amount);
}

```

Такой метод называется *фабричным методом (Factory Method)*, так как он предназначен для создания объектов.

Недостаток этого паттерна заключается в том, что предназначение фабричного метода не очевидно: вы должны помнить о том, что этот метод создает объекты, вместе с тем это обычный метод, а не конструктор. Фабричный метод следует использовать только тогда, когда вы действительно нуждаетесь в гибкости, которую он обеспечивает. В противном случае для создания объектов вполне можно использовать обычные конструкторы.

Imposter (Самозванец)

Каким образом можно добавить в программу новую вариацию некоторой функциональности? Создайте новый объект с точно таким же протоколом, как и существующий объект, но с отличающейся реализацией.

При использовании процедурно-ориентированного подхода для решения подобной задачи в программу требуется добавить как минимум один условный оператор. Как было продемонстрировано ранее, при обсуждении паттерна *Pluggable Selector* (Встраиваемый переключатель), такие условные операторы имеют тенденцию плодиться подобно саранче. Чтобы избавиться от дублирования, требуется полиморфизм.

Представьте, что у вас уже есть необходимая инфраструктура. У вас уже есть объект, который реализует необходимую функциональность. Теперь вы хотите, чтобы ваша система делала нечто отличающееся. Если вы обнаружили очевидное место для добавления оператора *if* и при этом не возникает дублирования какой-либо существующей логики, действуйте смело и решительно. Однако, зачастую, для добавления вариации требуется внести изменения в код нескольких методов.

Если вы работаете в стиле TDD, решение об использовании *Imposter* (Самозванец) может возникнуть, исходя из разных предпосылок. Иногда вы пишете тест и у вас возникает желание реализовать новый сценарий. Однако ни один из существующих объектов не выражает того, что вы хотите выразить. Представьте, что мы тестируем графический редактор и нам уже удалось реализовать корректное рисование прямоугольников:

```

testRectangle() {
    Drawing d= new Drawing();
    d.addFigure(new RectangleFigure(0, 10, 50, 100));
    RecordingMedium brush= new RecordingMedium();
    d.display(brush);
    assertEquals("rectangle 0 10 50 100\n", brush.log());
}

```

Теперь мы хотим реализовать рисование овалов. В данном случае необходимость применения *Imposter* (Самозванец) очевидна: заменяем *RectangleFigure* на *OvalFigure*.

```
testOval() {
    Drawing d= new Drawing();
    d.addFigure(new OvalFigure(0, 10, 50, 100));
    RecordingMedium brush= new RecordingMedium();
    d.display(brush);
    assertEquals("oval 0 10 50 100\n". brush.log());
}
```

Как правило, чтобы увидеть необходимость использования Imposter (Самозванец) еще до начала разработки кода, требуется озарение. Именно озарением можно назвать момент, когда Уорд Каннингэм решил, что вектор объектов Money может вести себя так же, как одиночный объект Money. Сначала можно подумать, что они отличаются, однако после вы понимаете, что они одинаковы.

Вот два примера использования Imposter (Самозванец) в процессе рефакторинга:

- **Null Object** (Нуль-объект) — вы можете рассматривать отсутствие данных в точности так же, как и присутствие данных;
- **Composite** (Компоновщик) — вы можете рассматривать коллекцию объектов как одиночный объект.

Решение об использовании Imposter (Самозванец) в процессе рефакторинга принимается для устранения дублирования, впрочем, целью любого рефакторинга является устранение дублирования.

Composite (Компоновщик)

Каким образом лучше всего реализовать объект, чье поведение является композицией функций некоторого набора других объектов? Примените паттерн Imposter (Самозванец) — заставьте этот объект вести себя подобно тому, как ведут себя отдельные объекты, входящие в набор.

Мой любимый пример основан на двух объектах: Account (счет) и Transaction (транзакция). Этот пример помимо прочего демонстрирует некоторую противоречивость паттерна Composite (Компоновщик), но об этом позже. В объекте Transaction хранится изменение величины счета (безусловно, транзакция — это более сложный и интересный объект, однако на данный момент мы ограничимся лишь мизерной долей его возможностей):

Transaction

```
Transaction(Money value) {
    this.value= value;
}
```

Объект Account вычисляет баланс счета путем суммирования значений относящихся к нему объектов Transaction:

Account

```
Transaction transactions[];
Money balance() {
    Money sum= Money.zero();
```

```

for (int i= 0; i < transactions.length; i++)
    sum= sum.plus(transactions[i].value);
return sum;
}

```

Все выглядит достаточно просто:

- в объектах Transaction хранятся значения;
- в объекте Account хранится баланс.

Теперь самое интересное. У клиента есть несколько счетов, и он хочет узнать общий баланс по всем этим счетам. Первая мысль, которая приходит в голову: создать новый класс OverallAccount, который суммирует балансы для некоторого набора объектов Account. Дублирование! Дублирование!

А что, если классы Account и Transaction будут поддерживать один и тот же интерфейс? Давайте назовем его Holding (сбережения), потому что сейчас мне не удается придумать что-либо лучшее:

Holding

```

interface Holding
    Money balance();

```

Чтобы реализовать метод balance() в классе Transaction, достаточно вернуть хранящееся в этом классе значение:

Transaction

```

Money balance() {
    return value;
}

```

Теперь в классе Account можно хранить не транзакции, а объекты Holding:

Account

```

Holding holdings[];
Money balance() {
    Money sum= Money.zero();
    for (int i= 0; i < holdings.length; i++)
        sum= sum.plus(holdings[i].balance());
    return sum;
}

```

Проблема, связанная с созданием класса OverallAccount, испарилась в воздухе. Объект OverallAccount — это просто еще один объект Account, в котором хранятся не транзакции, а другие объекты Account.

Теперь о противоречивости. В приведенном примере хорошо чувствуется запах паттерна Composite (Компоновщик). В реальном мире транзакция не может содержать в себе баланс. В данном случае программист идет на уловку, которая совершенно не логична с точки зрения всего остального мира. Вместе с тем, преимущества подобного дизайна неоспоримы, и ради этих преимуществ можно пожертвовать некоторым концептуальным несоответствием. Если присмотреться, подобные несоответствия встречаются нам на каждом шагу: папки (Folders), в которых содержатся другие папки (Folders), наборы тестов (TestSuites), в которых со-

держатся другие наборы тестов (TestSuites), рисунки (Drawings), в которых содержатся другие рисунки (Drawings). Любая из этих метафор не достаточно хорошо соответствует взаимосвязи между вещами в реальном мире, однако все они существенно упрощают код.

Я вынужден был длительное время экспериментировать с паттерном Composite (Компоновщик), прежде чем научился понимать, когда его следует использовать, а когда — нет. Наверное, вы уже поняли, что я не могу предоставить вам однозначных рекомендаций относительно решения проблемы, в каких ситуациях коллекция объектов является просто коллекцией объектов, а в каких это — объект-компоновщик. Хорошая новость состоит в том, что когда вы достаточно хорошо освоите рефакторинг, вы наверняка сможете обнаружить возникновение дублирования, воспользоваться паттерном Composite (Компоновщик) и обнаружить, что код существенно упростился.

Collecting Parameter (Накопление в параметре)

Каким образом вы можете сформировать результат операции, если эта операция распределена между несколькими объектами? Используйте параметр, в котором будут накапливаться результаты операции.

Простым примером является интерфейс `java.io.Externalizable`. Метод `writeExternal` этого интерфейса осуществляет запись объекта и всех объектов, на которые ссылается данный объект. Чтобы обеспечить общую запись, все записываемые объекты должны взаимодействовать друг с другом, поэтому методу передается параметр — объект класса `ObjectOutput`, — в котором осуществляется накопление:

```
java.io.Externalizable
public interface Externalizable extends java.io.Serializable {
    void writeExternal(ObjectOutput out) throws IOException;
}
```

Добавление параметра-накопителя зачастую является следствием использования паттерна Composite (Компоновщик). В начале разработки JUnit не было необходимости накапливать результаты исполнения нескольких тестов в объекте `TestResult` до тех пор, пока в инфраструктуру не была добавлена возможность создания и запуска нескольких тестов.

Необходимость использования параметра-накопителя возникает в ситуации, когда возрастает сложность объекта, получаемого в результате комплексной операции. Например, представьте, что нам необходимо реализовать вывод объекта `Expression` на экран в виде строки символов. Если обычная, не структурированная строка — это все, что нам нужно, значит, конкатенации будет вполне достаточно:

```
testSumPrinting() {
    Sum sum= new Sum(Money.dollar(5), Money.franc(7));
    assertEquals("5 USD + 7 CHF", sum.toString());
}
String toString() {
    return augend + " + " + addend;
}
```

Однако, если мы хотим отобразить объект Expression в виде древовидной структуры, код может выглядеть следующим образом:

```
testSumPrinting() {
    Sum sum= new Sum(Money.dollar(5), Money.franc(7));
    assertEquals("+\n\t5 USD\n\t7 CHF", sum.toString());
}
```

В этом случае нам придется воспользоваться параметром-накопителем, это может выглядеть примерно так:

```
String toString() {
    IndentingStream writer= new IndentingStream();
    toString(writer);
    return writer.contents();
}
```

```
void toString(IndentingWriter writer) {
    writer.println("+");
    writer.indent();
    augend.toString(writer);
    writer.println();
    addend.toString(writer);
    writer.exdent();
}
```

Singleton (Одиночка)

Каким образом можно реализовать глобальную переменную в языке, в котором не поддерживаются глобальные переменные? Не следует этим заниматься. Ваша программа скажет вам большое спасибо, если вместо этого вы еще раз хорошенько обдумаете дизайн и откажетесь от мысли использовать глобальные переменные.

Рефакторинг

Рассматриваемые здесь паттерны помогут вам изменить дизайн системы маленькими шажками.

В рамках TDD рефакторинг¹ используется интересным образом. Обычно рефакторинг не может изменить семантику программы ни при каких условиях. В рамках TDD условия семантики формулируются при помощи тестов, которые уже срабатывают. Таким образом, в рамках TDD мы можем, например, заменить константы переменными и с чистой совестью назвать эту процедуру рефакторингом, потому что набор срабатывающих тестов при этом не изменился. Однако набор срабатывающих тестов может состоять всего из одного теста. Возможно, семантика программы должна описываться большим количеством тестов. Возможно также, что некоторые из этих потенциальных тестов в результате выполнения рефакторинга перестали бы срабатывать, если бы они существовали. Однако их нет, поэтому мы о них не беспокоимся.

Отсюда следует, что на программиста, работающего в стиле TDD, возлагается важная обязанность: он должен иметь достаточное количество тестов, описывающих семантику программы. Достаточное по крайней мере настолько, настолько он может судить на момент завершения работы над кодом. Необходимо понимать, что рефакторинг выполняется не с учетом всех существующих тестов, а с учетом всех возможных тестов. Фраза наподобие: «Я знаю, что там была проблема, но все тесты сработали, поэтому я посчитал код законченным и интегрировал его в систему», — не может считаться оправданием. Пишите больше тестов.

Reconcile Differences (Согласование различий)

Каким образом вы можете унифицировать два схожих фрагмента кода? Постепенно делайте их все более похожими друг на друга. Унифицируйте их только в случае, если они абсолютно идентичны.

¹ Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley. Русское издание: *Файлер, М. Рефакторинг: улучшение существующего кода*. СПб.: Символ-Плюс, 2003.

Подчас выполнение рефакторинга — это весьма нервная работа. Простые рефакторинги очевидны. Если я извлекаю метод и делаю это механически корректно, вероятность того, что поведение системы изменится, чрезвычайно мала. Однако некоторые из рефакторингов заставляют вас внимательно анализировать последовательность выполнения операций и порядок модификации данных. Построив длинную цепочку умозаключений, вы приходите к выводу, что запланированное вами изменение кода, скорее всего, не приведет к изменению поведения системы. Однако любой подобный рефакторинг уменьшает количество волос на вашей голове.

Сложные рефакторинги — это именно то, чего мы пытаемся избежать, когда придерживаемся стратегии маленьких шажков и конкретной обратной связи. Полностью избежать сложных рефакторингов невозможно, однако можно уменьшить сферу их влияния на остальной код.

Подобные рефакторинги возникают на разных уровнях:

- Два цикла выглядят похоже. Если вы сделаете их идентичными, вы можете объединить их в единый цикл.
- Две ветви условного оператора выглядят похоже. Сделав их идентичными, вы можете избавиться от условного оператора.
- Два метода выглядят похоже. Сделав их идентичными, вы можете избавиться от одного из них.
- Два класса выглядят похоже. Сделав их идентичными, вы можете избавиться от одного из них.

Иногда задачу согласования различий удобнее решать в обратном порядке. Иными словами, вы представляете себе самый тривиальный последний этап этой процедуры, а затем двигаетесь в обратном направлении. Например, если вы хотите избавиться от нескольких подклассов, наиболее тривиальный последний шаг можно будет выполнить в случае, если подкласс ничего не содержит. В этом случае везде, где используется подкласс, можно будет использовать суперкласс, при этом поведение системы не изменится. Что надо сделать, чтобы очистить подкласс от методов и данных? Для начала метод можно сделать полностью идентичным одному из методов суперкласса. Постепенно переместив все методы и все данные в суперкласс, вы сможете заменить ссылки на подкласс ссылками на суперкласс. После этого подкласс можно уничтожить.

Isolate Change (Изоляция изменений)

Каким образом можно модифицировать одну часть метода или объекта, состоящего из нескольких частей? Вначале изолируйте изменяемую часть.

Мне приходит в голову аналогия с хирургической операцией: фактически все тело оперируемого пациента покрыто специальной простыней за исключением места, в котором, собственно, осуществляется операция. Благодаря такому покрытию хирург имеет дело с фиксированным набором переменных. Перед выполнением операции врачи сколь угодно долго могут обсуждать, какое влияние на

здоровье пациента оказывает тот или иной орган, однако во время операции внимание хирурга должно быть сфокусировано.

Вы можете обнаружить, что после того, как вы изолировали изменение, а затем внесли это изменение в код, результат получился настолько тривиальным, что вы можете отменить изоляцию. Например, если мы обнаружили, что внутри метода `findRate()` должно присутствовать всего одно действие — возврат значения поля, мы можем вместо обращений к методу `findRate()` напрямую обратиться к полю. В результате метод `findRate()` можно будет удалить. Однако подобные изменения не следует выполнять автоматически. Постарайтесь найти баланс между затратами, связанными с использованием дополнительного метода, и пользой, которую приносит дополнительная концепция, добавленная в код.

Для изоляции изменений можно использовать несколько разных способов. Наиболее часто используется Extract Method (Выделение метода), помимо него также используются Extract Object (Выделение объекта) и Method Object (Метод в объект).

Migrate Data (Миграция данных)

Каким образом можно перейти от одного представления к другому? Временно дублируйте данные.

Как

Вначале рассмотрим версию «от внутреннего к внешнему». В рамках этого подхода вы вначале изменяете внутреннее представление, а затем меняете внешний интерфейс.

1. Создайте экземплярную переменную в новом формате.
2. Инициализируйте переменную нового формата везде, где инициализируется переменная старого формата.
3. Используйте переменную нового формата везде, где используется переменная старого формата.
4. Удалите старый формат.
5. Измените внешний интерфейс таким образом, чтобы использовать новый формат.

Однако в некоторых ситуациях вам удобнее вначале изменить API. В этом случае вы должны выполнить рефакторинг следующим образом:

1. Добавьте параметр в новом формате.
2. Обеспечьте преобразование параметра в новом формате во внутреннее представление, обладающее старым форматом.
3. Удалите параметр в старом формате.
4. Замените использование старого формата на использование нового формата.
5. Удалите старый формат.

Зачем

Проблема миграции данных возникает каждый раз, когда вы используете паттерн **One to Many** (От одного ко многим). Предположим, что мы хотим реализовать объект `TestSuite`, используя паттерн **One to Many** (От одного ко многим). Мы можем начать так:

```
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.run(self.result)
    assert("1 run. 0 failed" == self.result.summary())
```

Чтобы реализовать этот тест, начнем с одного элемента `test`:

```
class TestSuite:
    def add(self, test):
        self.test= test
    def run(self, result):
        self.test.run(result)
```

Теперь мы приступаем к дублированию данных. Вначале инициализируем коллекцию тестов:

`TestSuite`

```
def __init__(self):
    self.tests= []
```

В каждом месте, где инициализируется поле `test`, мы добавляем новый тест в коллекцию:

`TestSuite`

```
def add(self, test):
    self.test= test
    self.tests.append(test)
```

Теперь мы используем коллекцию тестов вместо единичного теста. Исходя из существующего набора тестов, данное преобразование можно считать рефакторингом (оно не нарушает семантику), так как в составе коллекции присутствует только один элемент.

`TestSuite`

```
def run(self, result):
    for test in self.tests:
        test.run(result)
```

Теперь можно удалить не используемую экземплярную переменную `test`:

`TestSuite`

```
def add(self, test):
    self.tests.append(test)
```

Поэтапную миграцию данных можно использовать также при переходе между эквивалентными форматами, использующими различные протоколы, например, если речь идет о Java, при переходе от `Vector/Enumerator` к `Collection/Iterator`.

Extract Method (Выделение метода)

Каким образом длинный сложный метод можно сделать простым в прочтении? Выделите небольшую часть длинного метода в отдельный метод и обратитесь к этому методу из длинного метода.

Как

Выделение метода на самом деле является несколько более сложным атомарным рефакторингом. Здесь я опишу самый типичный случай. К счастью, многие среды разработки поддерживают автоматическое выполнение этого рефакторинга. Итак, чтобы выделить метод:

1. Определите фрагмент кода, который можно выделить в отдельный метод. Хорошими кандидатами являются тела циклов, сами циклы, а также ветви условных операторов.
2. Убедитесь в том, что внутри этого фрагмента не происходит присвоения значений временным переменным, объявленным вне области видимости, соответствующей этому фрагменту.
3. Скопируйте код из старого метода в новый метод. Откомпилируйте его.
4. Для каждой временной переменной или параметра изначального метода, используемого в новом методе, добавьте параметр в новый метод.
5. Сделайте так, чтобы в нужном месте старый метод обращался к новому методу.

Зачем

Я использую Extract Method (Выделение метода), когда пытаюсь понять сложный код. «Значит так, этот кусок кода делает вот это. А этот кусок делает это. К чему мы там дальше обращаемся?» После получаса код выглядит гораздо лучше, ваш партнер начинает понимать, что вы действительно оказываете ему помощь, а вы существенно лучше понимаете, что же все-таки происходит внутри кода.

Я использую выделение метода для того, чтобы избавиться от дублирования, когда вижу, что два метода обладают сходными участками кода. В этом случае я выделяю схожие участки в отдельный метод. (Браузер рефакторинга Smalltalk Refactoring Browser выполняет еще более полезную задачу: он просматривает код в поисках метода, аналогичного коду, который вы намерены выделить, и в случае, если такой метод уже есть, предлагает вам использовать уже существующий метод вместо того, чтобы создавать новый.)

Разделение методов на множество мелких кусочков может пойти слишком далеко. Если я не вижу, куда идти дальше, я часто использую Inline Method (Встраивание метода), чтобы собрать код в одном месте и увидеть новый, более удобный способ разделения.

Inline Method (Встраивание метода)

Каким образом можно упростить код в случае, если становится сложно уследить за последовательностью передачи управления от метода к методу? Замените обращение к методу кодом этого метода.

Как

1. Скопируйте код метода в буфер обмена.
2. Вставьте код метода вместо обращения к методу.
3. Замените все формальные параметры фактическими параметрами. Если, например, вы передаете `reader.getNext()`, то есть выражение, обладающее побочным эффектом, будьте осторожны и присвойте полученное значение временной переменной.

Зачем

Один из тех, кто рецензировал мою рукопись, пожаловался мне на сложность кода в первой части книги, когда объекту `Bank` предписывается преобразовать объект `Expression` к объекту `Money`.

```
public void testSimpleAddition() {
    Money five= Money.dollar(5);
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

«Это слишком сложно. Почему бы не попросить выполнить преобразование у самого объекта `Money`?» Ну что же, поставим эксперимент. Как это сделать? Давайте построим метод `Bank.reduce()` и посмотрим, как это будет выглядеть:

```
public void testSimpleAddition() {
    Money five= Money.dollar(5);
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= sum.reduce(bank, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

Возможно, вторая версия понравится вам больше, возможно, нет. Важно понимать, что при помощи `Inline Method` (Встраивание метода) вы можете экспериментировать с последовательностью выполнения действий. Когда я выполняю рефакторинг, я формирую у себя в голове мысленную картину системы с кусками логики и потоком выполнения программы, перетекающим от одного объекта к другому объекту. Когда мне кажется, что я вижу нечто многообещающее, я использую рефакторинг для того, чтобы попробовать это и увидеть результат.

В разгаре битвы я могу вдруг обнаружить, что попался в ловушку собственной гениальности. (Я не буду говорить, насколько часто это происходит.) Когда это происходит, я использую Inline Method (Встраивание метода) для того, чтобы разобраться в той путанице, которую я создал: «Так, этот объект обращается к этому, этот к этому... не могу понять, что же здесь происходит?» Я встраиваю несколько уровней абстракции и смотрю, что же на самом деле происходит. После этого я могу заново выделить абстракцию, используя более удобный способ.

Extract Interface (Выделение интерфейса)

Каким образом создать альтернативные реализации операций в языке Java? Создайте интерфейс, в котором будут содержаться общие операции.

Как

1. Напишите объявление интерфейса. Иногда в качестве имени интерфейса используется имя существующего класса. В этом случае вы должны предварительно переименовать класс.
2. Сделайте так, чтобы существующий класс реализовывал объявленный вами интерфейс.
3. Добавьте в интерфейс все необходимые методы. В случае необходимости измените режим видимости методов класса.
4. Там, где это возможно, измените объявления с класса на интерфейс.

Зачем

Иногда необходимость выделения интерфейса возникает в случае, когда вы переходите от одной реализации к другой. Например, у вас есть класс `Rectangle` (прямоугольник), и вы хотите создать класс `Oval` (овал) — в этом случае вы создаете интерфейс `Shape` (форма). В подобных ситуациях подобрать имя для интерфейса, как правило, несложно. Однако иногда вы вынуждены изрядно помучиться, прежде чем сможете обнаружить подходящую метафору.

Иногда в ситуации, когда вам нужно выделить интерфейс, вы используете `Crash Test Dummy` (Тестирование обработки ошибок) или другой поддельный объект (`Mock Object`). В этом случае подбор подходящего имени выполняется сложнее, так как в вашем распоряжении лишь один пример использования интерфейса. В подобных случаях у меня возникает соблазн наплевать на информативность и назвать интерфейс `IFile`, а реализующий его класс — `File`. Однако я приучил себя останавливаться на мгновение и размышлять о том, достаточно ли хорошо я понимаю то, над чем работаю? Возможно, интерфейс лучше назвать `File`, а реализующий его класс — `DiskFile`, так как соответствующая реализация основана на том, что данные, содержащиеся в файле, хранятся на жестком диске.

Move Method (Перемещение метода)

Каким образом метод можно переместить в новое место, где он должен находиться? Добавьте его в класс, к которому он должен принадлежать, затем обратитесь к нему.

Как

1. Скопируйте метод в буфер обмена.
2. Вставьте метод в целевой класс. Присвойте ему подходящее имя. Откомпилируйте его.
3. Если внутри метода происходит обращение к изначальному объекту, добавьте в метод параметр, при помощи которого внутрь метода будет передаваться изначальный объект. Если внутри метода происходит обращение к переменным-членам изначального объекта, передавайте их в виде параметров. Если внутри метода переменным-членам изначального объекта присваиваются значения, вы должны отказаться от идеи переноса метода в новый объект.
4. Замените тело изначального метода обращением к новому методу.

Зачем

Это один из моих самых любимых рефакторингов, выполняемых в процессе консультирования. Дело в том, что он наиболее эффективно демонстрирует неправильные предположения относительно дизайна кода. Вычисление площади — это обязанность объекта **Shape** (форма):

Shape

```
...
int width= bounds.right() - bounds.left();
int height= bounds.bottom() - bounds.top();
int area= width * height;
...
```

Каждый раз, когда я вижу, что внутри метода, принадлежащего одному объекту, происходит обращение к нескольким методам другого объекта, я начинаю смотреть на код с подозрением. В данном случае я вижу, что в методе, принадлежащем объекту **Shape**, происходит обращение к четырем методам объекта **bounds** (класс **Rectangle**). Пришло время переместить эту часть метода в класс **Rectangle**:

Rectangle

```
public int area() {
    int width= this.right() - this.left();
    int height= this.bottom() - this.top();
    return width * height;
}
```


Shape

```
...
int area= bounds.area();
...
```

Рефакторинг *Move Method* (Перемещение метода) обладает тремя важными преимуществами:

- Очень легко увидеть необходимость применения этого рефакторинга, при этом не требуется глубокое понимание смысла кода. Как только вы увидите два или больше сообщения, адресованных другому объекту, значит, можно смело приступать.
- Механика выполнения рефакторинга быстра и безопасна.
- Результаты зачастую приводят к просветлению. «Но класс *Rectangle* не выполняет никаких вычислений... О! Теперь я вижу. Так действительно лучше.»

Иногда у вас возникает желание переместить только часть метода. Вы можете вначале выделить метод, переместить весь метод, а затем встроить метод в исходный класс. Или вы можете придумать способ сделать все это за один шаг.

Method Object (Метод в объект)

Каким образом лучше всего реализовать сложный метод, использующий несколько параметров и локальных переменных? Преобразуйте метод в отдельный объект.

Как

1. Создайте класс с таким же количеством параметров, как и оригинальный метод.
2. Сделайте локальные переменные метода экземплярными переменными нового класса.
3. Определите в новом классе метод с именем *run()*. Тело этого метода будет таким же, как и тело оригинального метода.
4. В оригинальном методе создайте новый объект и обратитесь к методу *run()* этого объекта.

Зачем

Объекты-методы полезны в качестве подготовительного этапа перед добавлением в систему абсолютно нового вида внутренней логики. Например, представьте, что для вычисления общего денежного потока используется несколько разных методов, позволяющих учесть в вычислении несколько разных компонентов общего денежного потока. Вначале вы можете создать объект-метод, вычисляющий

общий денежный поток первым способом. Затем вы можете описать следующий способ вычислений при помощи тестов меньшего масштаба. После этого добавление в программу нового способа вычислений будет несложным делом.

Объекты-методы также позволяют упростить код, в отношении которого неудобно использовать Extract Method (Выделение метода). В некоторых ситуациях вы вынуждены иметь дело с блоком кода, который работает с обширным набором временных переменных и параметров, и каждый раз, когда вы пытаетесь выделить хотя бы часть этого кода в отдельный метод, вы вынуждены переносить в новый метод пять или шесть временных переменных и параметров. Результирующий выделенный метод выглядит ничем не лучше, чем изначальный код, так как его сигнатура слишком длинна. В результате создания объекта-метода вы получаете в свое распоряжение новое пространство имен, в рамках которого вы можете извлекать методы, без необходимости передачи в них каких-либо параметров.

Add Parameter (Добавление параметра)

Каким образом можно добавить в метод новый параметр?

Как

1. Если метод входит в состав интерфейса, вначале добавьте параметр в интерфейс.
2. Добавьте параметр.
3. Воспользуйтесь сообщениями об ошибках, выдаваемыми вашим компилятором, для того чтобы узнать, в каких местах происходит обращение к данному методу. В каждом из этих мест внесите необходимые модификации в вызывающий код.

Зачем

Добавление параметра зачастую связано с расширением функциональности. Чтобы обеспечить срабатывание первого теста, вы написали код без параметра, однако далее условия изменились, и для корректного выполнения вычислений вам необходимо принять во внимание дополнительные данные.

Добавление параметра также может быть вызвано необходимостью миграции от одного представления данных к другому. Вначале вы добавляете параметр, затем вы удаляете из кода все ссылки на старый параметр, затем вы удаляете сам старый параметр.

Method Parameter to Constructor Parameter (Параметр метода в параметр конструктора)

Каким образом переместить параметр из метода или методов в конструктор?

Как

1. Добавьте параметр в конструктор.
2. Добавьте в класс экземплярную переменную с тем же именем, что и параметр.
3. Установите значение переменной в конструкторе.
4. Одну за другой преобразуйте ссылки `parameter` в ссылки `this.parameter`.
5. Когда в коде не останется ни одной ссылки на параметр, удалите параметр из метода.
6. После этого удалите ненужный теперь префикс `this`.
7. Присвойте переменной подходящее имя.

Зачем

Если вы передаете один и тот же параметр нескольким разным методам одного и того же объекта, вы можете упростить API, передав параметр только один раз (удалив дублирование). Напротив, если вы обнаружили, что некоторая экземплярная переменная используется только в одном методе объекта, вы можете выполнить обратный рефакторинг.

Развитие навыков TDD

В данной главе я намерен сформулировать несколько вопросов, над которыми полезно подумать, если вы намерены интегрировать TDD в используемый вами процесс разработки. Некоторые из вопросов просты, другие требуют тщательного обдумывания. Ответы на некоторые из этих вопросов вы найдете здесь же, однако иногда, чтобы ответить на некоторый вопрос, вам придется провести собственные исследования.

Насколько большими должны быть шаги?

В этом вопросе на самом деле скрыто два вопроса:

- Какой объем функциональности должен покрывать каждый тест?
- Как много промежуточных стадий должно быть преодолено в процессе каждого рефакторинга?

Вы можете писать тесты таким образом, что каждый из них будет требовать добавления в функциональный код единственной строки и выполнения небольшого рефакторинга. Вы также можете писать тесты таким образом, что каждый из них будет требовать добавления в функциональный код сотен строк, при этом у вас будут уходить часы на рефакторинг. Какой из этих путей лучше?

Часть ответа состоит в том, что вы должны уметь работать и так и этак. Общая тенденция TDD очевидна — чем меньше шаги, тем лучше. В данной книге мы занимались разработкой маленьких тестов на уровне отдельных фрагментов программы. Однако некоторые программисты экспериментируют в области разработки программ, исходя из тестов на уровне всего приложения.

Если вы только приступаете к освоению рефакторинга, вы должны двигаться маленькими шажками. Процесс ручного рефакторинга чреват ошибками. Чем больше ошибок вы наделаете, тем меньшим будет ваше желание выполнять рефакторинг в дальнейшем. После того как вы 20 раз проделаете рефакторинг маленькими шажками, можете приступить к экспериментам по удлинению этих шажков.

Автоматизация существенно ускоряет процессы, связанные с рефакторингом. То, что раньше требовало от вас выполнения 20 маленьких шагов вручную, теперь становится единственным пунктом в меню. Количество выполняемых вами рефакторингов увеличивается на порядок, и это неизменно сказывается на качестве. Когда вы знаете, что в вашем распоряжении великолепный инструмент, вы становитесь более агрессивным при выполнении рефакторинга. Вы пытаетесь ставить больше экспериментов, и смотреть, какой способ структурирования кода лучше.

На момент написания данной книги браузер рефакторинга Refactoring Browser for Smalltalk по-прежнему является наилучшим инструментом из этой категории. В настоящее время многие среды разработки для Java начинают поддерживать рефакторинг. Кроме того, я уверен, что в скором будущем поддержка рефакторинга появится и в других языках и средах разработки.

Что не подлежит тестированию?

Флип (Phlip) предложил высказывание, которое может служить ответом на этот вопрос: «Пишите тесты до тех пор, пока страх не превратится в скуку». Высказывание подразумевает, что вы должны найти ответ сами. Однако вы читаете эту книгу для того, чтобы найти в ней ответы на вопросы, поэтому попробуйте воспользоваться следующим списком. Тестировать следует:

- условные операторы;
- циклы;
- операции;
- полиморфизм.

Однако только те из них, которые вы написали сами. Не тестируйте чужой код, если только у вас нет причин не доверять ему. В некоторых ситуациях недостатки (можно сказать жестче: «ошибки») во внешнем коде заставляют вас добавлять дополнительную логику в разрабатываемый вами код. Надо ли тестировать подобное поведение внешнего кода? Иногда я документирую непредсказуемое поведение (ошибку) внешнего кода при помощи теста, который перестанет срабатывать в случае, если в следующей версии внешнего кода ошибка будет исправлена.

Как определить качество тестов?

Тесты — это канарейка, которую берут в угольную шахту для того, чтобы по ее поведению определить присутствие запаха плохого дизайна. Далее перечисляются некоторые атрибуты тестов, которые указывают на то, что дизайн тестируемого кода начинает плохо пахнуть:

- Длинный код инициализации. Если вы вынуждены написать сотни строк кода, создавая объекты для одного простого оператора `assert()`, значит, что-то не так. Это означает, что ваши объекты слишком большие и их требуется разделить.

- Дублирование кода инициализации. Если вы не можете быстро найти общее место для общего кода инициализации, это означает, что у вас слишком много объектов, которые слишком тесно взаимодействуют друг с другом.
- Тесты выполняются слишком медленно. Если тесты TDD работают слишком медленно, значит, они не будут запускаться достаточно часто. Значит, программист будет в течение некоторого времени работать, вообще не запуская тестов. Значит, когда он их все-таки запустит, скорее всего, многие из них не сработают. На самом деле здесь кроется серьезная проблема: если тесты работают медленно, значит, тестирование частей и компонентов разрабатываемого приложения связано с проблемами. Сложности при тестировании частей и фрагментов приложения указывают на существование недостатков дизайна. Иными словами, улучшив дизайн, вы можете увеличить скорость работы тестов. (Стандартом принято считать десятиминутный набор тестов, по аналогии с ускорением силы тяжести в $9,8 \text{ м/с}^2$. Если для выполнения набора тестов требуется более 10 минут, этот набор обязательно надо сократить или тестируемое приложение должно быть оптимизировано так, чтобы для выполнения набора тестов требовалось не более 10 минут.)
- Хрупкие тесты. Если ваши тесты неожиданно начинают ломаться в самых непредсказуемых местах, это означает, что одна часть разрабатываемого приложения непредсказуемым образом влияет на работу другой части. В этом случае необходимо улучшить дизайн таким образом, чтобы данный эффект исчез. Для этого можно либо устранить связь между частями приложения, либо объединить две части воедино.

Каким образом TDD ведет к созданию инфраструктур?

Инфраструктура (framework) — набор обобщенного кода, который можно использовать в качестве базы при разработке разнообразных прикладных программ. На самом деле TDD является неплохим инструментом разработки инфраструктур. Парадокс: если вы перестаете думать о будущем вашего кода, вы делаете код значительно более адаптируемым для повторного использования в будущем.

Очень многие умные книги говорят об обратном: «кодируйте для сегодняшнего дня, но проектируйте для завтрашнего дня» (code for today, design for tomorrow). Похоже, что TDD переворачивает этот совет с ног на голову: «кодируйте для завтрашнего дня, проектируйте для сегодняшнего дня» (code for tomorrow, design for today). Вот что происходит на практике:

- В программу добавляется первая функциональность. Она реализуется просто и прямолинейно, поэтому реализация выполняется быстро и с наименьшим количеством дефектов.
- В программу добавляется вторая функциональность, которая является вариацией первой. Дублирование между двумя функциональностями объ-

единяется и размещается в одном месте. Различия оказываются в разных местах (как правило, в разных методах или в разных классах).

- В программу добавляется третья функциональность, которая является вариацией первых двух. Уже имеющаяся общая логика, как правило, может использоваться в том виде, в котором она уже присутствует в программе, возможно, потребуется внести незначительные изменения. Отличающаяся логика должна располагаться в отдельном месте — в другом методе или в другом классе.

В процессе разработки мы постепенно приводим код в соответствие с принципом открытости/закрытости (Open/Closed Principle¹), утверждающим, что объекты должны быть открыты для использования, но закрыты для модификации. Изюминкой является то, что при использовании TDD этот принцип выполняется именно для тех вариаций, с которыми действительно приходится иметь дело на практике. Таким образом, TDD позволяет формировать инфраструктуры, удобные для представления таких вариаций, с необходимостью реализации которых программист реально сталкивается. Однако эти инфраструктуры могут оказаться неэффективными в случае, если потребуется реализовать вариацию, которая редко встречается в реальности (или которая не была еще реализована ранее).

Что же произойдет, если необходимость реализации непредвиденной вариации возникнет спустя три года после разработки инфраструктуры? Дизайн быстро эволюционирует таким образом, чтобы сделать вариацию возможной. Принцип открытости/закрытости (Open/Closed Principle) на короткое время будет нарушен, однако это нарушение обойдется относительно недорого, так как имеющиеся тесты сообщат вам о том, что, изменяя код, вы ничего не поломали.

В пределе, когда вариации возникают достаточно быстро, стиль TDD невозможно отличить от заблаговременного проектирования. Однажды я всего за несколько часов с нуля разработал инфраструктуру составления отчетов. Те, кто следил за этим, были абсолютно уверены в том, что это трюк. Они думали, что я сел за разработку, уже имея в голове готовую инфраструктуру. Однако это не так. Просто я долгое время практиковал TDD, благодаря этому я исправляю допущенные мною многочисленные ошибки быстрее, чем вы успеваете заметить, что я их допустил.

Сколько должно быть тестов?

Насколько емкой должна быть обратная связь? Рассмотрим простую задачу: дано три целых числа, обозначающих длины сторон треугольника. Метод должен возвращать:

¹ Один из основных принципов объектно-ориентированного программирования, утверждающий, что разрабатываемый код должен быть достаточно гибким для того, чтобы его можно было повторно использовать, без внесения в него каких-либо модификаций (то есть он должен быть открыт для использования, но закрыт для модификаций). — *Примеч. перек.*

- 1 — в случае, если треугольник равносторонний;
- 2 — в случае, если треугольник равнобедренный;
- 3 — в случае, если треугольник не равносторонний и не равнобедренный.

Если длины сторон заданы некорректно (невозможно построить треугольник со сторонами заданной длины), метод должен генерировать исключение.

Вперед! Попробуйте решить задачу (мое решение, написанное на языке Smalltalk, приведено в конце данного подраздела).

Это отчасти напоминает игру «Угадай мелодию» («Я могу закодировать задачу за четыре теста!» — «А я — за три!» — «О'кей попробуйте»). Для решения задачи я написал шесть тестов, а Боб Биндер (Bob Binder) в своей книге *Testing Object-Oriented Systems*¹ (Тестирование объектно-ориентированных систем) для этой же самой задачи написал 65 тестов. Сколько на самом деле нужно тестов? Вы должны решить это сами, исходя из собственного опыта и рассуждений.

Когда я думаю о необходимом количестве тестов, я пытаюсь оценить приемлемое *среднее время между сбоями* (MTBF, Mean Time Between Failures). Например, в языке Smalltalk целые числа ведут себя как целые числа, а не как значения 32-битного счетчика. Иными словами, значение максимального возможного целого числа ограничивается не тридцатью двумя битами, а объемом памяти. Это означает, что вы можете обойтись без тестирования MAXINT. Безусловно, определенный предел существует, ведь теоретически можно создать целое число, для хранения которого не хватит имеющейся памяти. Но должен ли я тратить время на написание и реализацию теста, пытающегося заполнить память невероятно огромным целым числом? Как это повлияет на MTBF моей программы? Если я в обозримом будущем не собираюсь иметь дело с треугольниками, размер сторон которых измеряется такими числами, значит, моя программа не станет существенно менее надежной, если я не реализую такой тест.

Имеет ли смысл писать тот или иной тест? Это зависит от того, насколько аккуратно вы оцените MTBF. Если обстоятельства требуют, чтобы вы увеличили MTBF от 10 лет до 100 лет, значит, имеет смысл уделить время для разработки самых маловероятных и чрезвычайно редко возникающих ситуаций (если, конечно, вы не можете каким-либо иным образом доказать, что подобные ситуации никогда не могут возникнуть).

Взгляд на тестирование в рамках TDD прагматичен. В TDD тесты являются средством достижения цели. Целью является код, в корректности которого мы в достаточной степени уверены. Если знание особенностей реализации без какого-либо теста дает нам уверенность в том, что код работает правильно, мы не будем писать тест. Тестирование черного ящика (когда мы намеренно игнорируем реализацию) обладает рядом преимуществ. Если мы игнорируем код, мы наблюдаем другую систему ценностей: тесты сами по себе представляют для нас ценность. В некоторых ситуациях это вполне оправданный подход, однако он отличается от TDD.

¹ Binder, Bob. 1999. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Boston: Addison-Wesley. Это действительно *исчерпывающее* руководство по тестированию.

TriangleTest

```

testEquilateral
  self assert: (self evaluate: 2 side: 2 side: 2) = 1

testIsosceles
  self assert: (self evaluate: 1 side: 2 side: 2) = 2

testScalene
  self assert: (self evaluate: 2 side: 3 side: 4) = 3

testIrrational
  [self evaluate: 1 side: 2 side: 3]
    on: Exception
    do: [:ex | ^self].
  self fail

testNegative
  [self evaluate: -1 side: 2 side: 2]
    on: Exception
    do: [:ex | ^self].
  self fail

testStrings
  [self evaluate: 'a' side: 'b' side: 'c']
    on: Exception
    do: [:ex | ^self].
  self fail

evaluate: aNumber1 side: aNumber2 side: aNumber3
  | sides |
  sides := SortedCollection
    with: aNumber1
    with: aNumber2
    with: aNumber3.
  sides first <= 0 ifTrue: [self fail].
  (sides at: 1) + (sides at: 2) <= (sides at: 3) ifTrue: [self fail].
  ^sides asSet size

```

Когда следует удалять тесты?

Чем больше тестов, тем лучше, однако если два теста являются избыточными по отношению друг к другу, должны ли вы сохранить оба этих теста в наборе? Это зависит от двух критериев.

- Первый критерий — это уверенность. Никогда не удаляйте тест, если в результате этого снизится ваша уверенность в корректности поведения системы.
- Второй критерий — это коммуникация. Если у вас есть два теста, которые тестируют один и тот же участок кода, однако читателем эти тесты рассматриваются как два различных сценария, сохраните оба этих теста.

Отсюда следует, что если у вас есть два теста, которые можно считать избыточными как в отношении уверенности, так и в отношении коммуникации, удалите наименее полезный из этих тестов.

Каким образом язык программирования и среда разработки влияют на TDD?

Попробуйте использовать подход TDD в среде Smalltalk с браузером Refactoring Browser. Теперь попробуйте работать в среде C++ с редактором vi. Почувствуйте разницу.

В языках программирования и средах разработки, в которых цикл TDD выполняется сложнее (тест—компиляция—запуск—рефакторинг), возникает тенденция двигаться вперед более длинными шагами:

- каждый тест покрывает собой больший объем функциональности;
- рефакторинг выполняется с меньшим количеством промежуточных шагов.

Приводит ли это к замедлению разработки или, наоборот, разработка ускоряется?

В языках программирования и средах, в которых проще выполнить цикл TDD, у вас будет возникать желание больше экспериментировать с кодом. Позволит ли это двигаться быстрее и формировать лучшие решения, или вам кажется, что лучше тратить время на дополнительные размышления о дизайне?

Можно ли использовать TDD для разработки крупномасштабных систем?

Позволяет ли методика TDD разрабатывать крупномасштабные программные проекты? Какие новые типы тестов вам потребуется написать? Какие новые типы рефакторингов могут потребоваться?

Самой крупной программной системой, целиком и полностью разработанной в стиле TDD, в разработке которой я принимал участие, является система LifeWare (www.lifeware.ch). Работа над системой велась в течение 4 лет. Объем работ оценивается в 40 человеко-лет. На текущий момент система включает в себя 250 000 строк функционального кода и 250 000 строк тестирующего кода (разработка ведется на языке Smalltalk). Набор тестов системы включает в себя 4000 тестов, для исполнения которых требуется 20 минут. Полный набор тестов запускается несколько раз каждый день. Реализованный в системе огромный объем функциональности, похоже, никак не снижает эффективности TDD. Избавляясь от дублирования, вы стараетесь создать большое количество маленьких объектов, которые можно тестировать изолированно друг от друга независимо от размера приложения.

Можно ли осуществлять разработку приложения исходя из тестов уровня приложения?

Если мы будем выполнять разработку, используя только внутренние программистские тесты (их называют тестами модулей — *unit tests*, — хотя они не вполне соответствуют этому определению), мы рискуем столкнуться с проблемой: полученная в результате этого система может оказаться не совсем тем или, что хуже, совсем не тем, чего хочет получить пользователь. Программист будет работать над программой, которая, *по его мнению*, должна быть полезна для пользователя, однако у пользователя может оказаться совершенно другое мнение. Чтобы решить проблему, можно разработать набор тестов на уровне приложения (*application-level tests*). Разработкой этих тестов должны заниматься сами пользователи (при поддержке программистов). Написанные пользователями тесты должны точно определять, что именно должна делать разрабатываемая система. Такой стиль можно назвать разработкой, основанной на тестах уровня приложения (ATDD, *Application Test-Driven Development*).

Встает техническая проблема. Каким образом можно написать и запустить тест для функциональности, которая еще не существует? Мне кажется, что всегда можно найти способ решения этой проблемы. Например, можно разработать интерпретатор, который будет вежливо сигнализировать о том, что обнаружен тест, выполнить который на данный момент невозможно по причине отсутствия в системе необходимых возможностей.

Существует также социальная проблема. У пользователей (на самом деле я имею в виду команду, в состав которой входят пользователи) появляется новая обязанность: разработка тестов. Процедура разработки тестов уровня приложения требует добавления дополнительного этапа в цикл работы над продуктом, — а именно, разработка пользовательских тестов выполняется перед началом реализации очередного объема функциональности. Организации часто сопротивляются подобному сдвигу ответственности. Новый этап требует координированных усилий множества членов команды, то есть перед тем, как приступить непосредственно к разработке кода, члены команды вынуждены потратить время на разработку пользовательских тестов.

Описанная в данной книге методика TDD целиком и полностью находится под вашим контролем. Иначе говоря, исполнение TDD зависит только от одного человека — от вас. Если у вас возникло желание, вы можете начать использовать ее с сегодняшнего дня. Однако если вы будете смешивать ритм красный—зеленый—рефакторинг с техническими, социальными и организационными проблемами разработки пользовательских тестов, вы вряд ли сможете добиться успеха. В данном случае следует воспользоваться правилом *One Step Test* (Тест одного шага). Сначала добейтесь равномерности ритма красный—зеленый—рефакторинг в собственной практике, затем расширьте область применения TDD.

Еще один аспект ATDD: определение длины цикла между разработкой теста и получением результатов его работы. Если заказчик написал тест, а потом в течение десяти дней ждет его срабатывания, это значит, что он большую часть вре-

мени смотрит на красную полосу. Если я работаю в стиле TDD на уровне программиста, я:

- немедленно получаю зеленую полосу;
- упрощаю внутренний дизайн.

Как можно перейти к использованию TDD в середине работы над проектом?

У вас есть некоторый объем кода, про который можно сказать, что он корректно работает в большей или меньшей степени. Теперь вы хотите разрабатывать весь новый код в рамках концепции TDD. Что делать?

О проблеме перехода на использование TDD в середине работы над проектом можно написать целую книгу (или даже несколько книг). В данном небольшом разделе я очень поверхностно затрону несколько связанных с этим вопросов.

Самая большая проблема заключается в том, что код, изначально написанный без учета тестов, как правило, сложен в тестировании. Интерфейсы и взаимосвязи между объектами не достаточно хорошо спроектированы, поэтому вам сложно изолировать некоторый кусок логики, запустить его и проверить результаты.

«Надо это исправить» — скажете вы. Да, однако любой рефакторинг (без применения средств автоматизации), скорее всего, приведет к возникновению ошибок, и эти ошибки сложно будет обнаружить, так как у вас нет тестов. Проблема яйца и курицы. Змея кусает сама себя за хвост. Замкнутый цикл саморазрушения. Что же делать?

Прежде всего, скажу, чего делать не надо: не надо писать тесты для всего кода и выполнять рефакторинг всего кода. Для этого может потребоваться несколько месяцев, в течение которых у вас не будет времени добавить в систему новой функциональности. Если вы тратите имеющиеся у вас деньги и при этом не зарабатываете новых, долго вы так не протянете.

Поэтому прежде всего мы должны ограничить область планируемых нами изменений. Если мы видим часть системы, которую можно существенно улучшить, но которая вполне может быть оставлена без изменений на текущий момент, мы оставляем их без изменений. Возможно, взглянув на код и вспомнив грехи прошлого, вы не сможете удержаться от слез, однако возьмите себя в руки, — если код не требует немедленного вмешательства, лучше не изменять его.

Во-вторых, мы должны разорвать замкнутый круг между тестами и рефакторингом. Мы можем использовать в качестве обратной связи не только тесты, но и другие способы, например чрезвычайно осторожная работа в паре с партнером. Мы можем использовать обратную связь более высокого уровня, например тесты на уровне всей системы. Мы знаем, что такие тесты не являются адекватными, однако они прибавляют нам уверенности. Благодаря такой обратной связи мы можем сделать части кода, нуждающиеся в изменении, более удобными для внесения изменений.

Через некоторое время части системы, которые постоянно меняются, станут выглядеть так, как будто их разработали с использованием TDD. Мы медленно углубимся в дремучий лес наших старых ошибок.

Для кого предназначена методика TDD?

Каждая практика программирования явно или не явно базируется на системе ценностей. TDD — не исключение. Если вам нравится лепить вместе куски кода, которые более-менее работают, и вы счастливы думать о том, что вам не придется возвращаться к полученному в результате этого коду в дальнейшем, значит, TDD — не для вас. Методика TDD базируется на очаровательно-наивном предположении программиста о том, что чем красивее код, тем вероятнее успех. TDD помогает вам обращать внимание на правильные вопросы в подходящие для этого моменты времени. Благодаря этому вы можете делать дизайн чище и модифицировать его по мере того, как перед вами встают новые обстоятельства.

Я сказал, что предположение наивное, однако, скорее всего, я преувеличил. На самом деле наивно предполагать, что чистый код — это все, что необходимо для успеха. Мне кажется, что хорошее проектирование — это лишь 20 % успеха. Безусловно, если проектирование будет плохим, вы можете быть на 100 % уверены в том, что проект провалится. Однако приемлемый дизайн сможет обеспечить успех проекта только в случае, если остальные 80 % будут там, где им полагается быть.

С этой точки зрения TDD — это чрезвычайно мощный инструмент. Если сравнивать со средним уровнем индустрии разработки программного обеспечения, методика TDD позволяет вам писать код, содержащий значительно меньше дефектов и формировать значительно более чистый дизайн. Те, кто стремится к изяществу, могут найти в TDD средство для достижения цели.

Методика TDD также подходит для тех, у кого формируется эмоциональная привязанность к коду. Когда я был молодым программистом, самым большим разочарованием для меня была ситуация, когда проект начинался с огромным воодушевлением, а затем, с течением времени, код становился все более отвратительным. Год спустя у меня, как правило, формировалось устойчивое желание уйти из проекта, чтобы никогда в жизни не иметь дела с этим гнусно пахнущим кодом. TDD позволяет вам с течением времени поддерживать уверенность в вашем коде. По мере того как тестов становится все больше (а ваше мастерство тестирования улучшается), вы обретаете все большую уверенность в том, что система ведет себя именно так, как вам надо. По мере того как вы улучшаете дизайн, становится возможным все большее количество изменений. Моя цель заключается в том, чтобы через год работы мне было бы интереснее и приятнее работать над проектом, чем в самом начале проекта, и TDD помогает мне достигнуть этой цели.

Зависит ли эффективность TDD от начальных условий?

Складывается впечатление, что разработка идет гладко только в случае, если тесты выполняются в определенном порядке. В этом случае мы можем наблюдать классическую последовательность красный—зеленый—рефакторинг—красный—зеленый—рефакторинг. Вы можете попробовать взять те же самые тесты, но реализовать их в другом порядке, и у вас возникнет ощущение, что вы не сможете, как прежде, выполнять разработку маленькими шажками. Действительно ли одна последовательность тестов на порядок быстрее/проще в реализации, чем другая последовательность? Существуют ли какие-либо признаки тестов, которые могут подсказать, в какой последовательности их следует реализовать? Если методика TDD чувствительна к начальным условиям в малом масштабе, можно ли считать ее предсказуемой в более крупном масштабе? (Вот аналогия: отдельные потоки реки Миссисипи непредсказуемы, однако вы можете с уверенностью сказать, что через устье реки протекает приблизительно 2 000 000 кубических футов воды в секунду.)

Каким образом методика TDD связана с паттернами?

Все мои технические публикации — это поиск фундаментальных правил, которые позволяют обычным людям действовать так, как действуют эксперты. Это связано с тем, как я сам осваиваю то или иное ремесло, — я нахожу эксперта, которому можно подражать, и постепенно выясняю, что, собственно, он делает. Определенно, я не предполагаю, что сформулированные мною правила должны использоваться автоматически, однако именно так и происходит.

Моя старшая дочь (привет, Бетани! Я же говорил тебе, что ты попадешь в мою книгу, — не беспокойся, это не слишком обременительно) в течение семи лет пыталась научиться быстро перемножать числа. Как я, так и моя жена, когда были маленькими, научились этому за значительно более короткий срок. В чем дело? Оказывается каждый раз, когда перед Бетани вставала задача умножить, например, 6 на 9, она складывала число 6 девять раз (или число 9 шесть раз). Таким образом, можно сказать, что Бетани вообще не умела умножать числа так, как это делают другие люди, однако при этом она необычайно быстро складывала числа.

Я обратил внимание на один важный эффект, который, я надеюсь, смогут принять во внимание и другие. Если на основе постоянно повторяющихся действий формулируются правила, дальнейшее применение этих правил становится неосознанным и автоматическим. Естественно, ведь это проще, чем обдумывать все за и все против того или иного действия с самого начала. Благодаря этому повышается скорость работы, и если в дальнейшем вы сталкиваетесь с исключением или проблемой, которая не вписывается ни в какие правила, у вас появляется до-

полнительное время и энергия для того, чтобы в полной мере применить свои творческие способности.

Именно это произошло со мной, когда я писал книгу *Smalltalk Best Practice Patterns* (Лучшие паттерны Smalltalk). В какой-то момент я решил просто следовать правилам, описываемым в моей книге. В начале это несколько замедлило скорость моей работы, — мне требовалось дополнительное время, чтобы вспомнить то или иное правило, или написать новое правило. Однако по прошествии недели я заметил, что с моих пальцев почти мгновенно слетает код, над разработкой которого ранее мне приходилось некоторое время размышлять. Благодаря этому у меня появилось дополнительное время для анализа и важных размышлений о дизайне.

Существует еще одна связь между TDD и паттернами: TDD является методом реализации дизайна, основанного на паттернах. Предположим, что в определенном месте разрабатываемой системы мы хотим реализовать паттерн Strategy (Стратегия). Мы пишем тест для первого варианта и реализуем его, создав метод. После этого мы намеренно пишем тест для второго варианта, ожидая, что на стадии рефакторинга мы придем к паттерну Strategy (Стратегия). Мы с Робертом Мартином (Robert Martin) занимались исследованием подобного стиля TDD. Проблема состоит в том, что дизайн продолжает вас удивлять. Идеи, которые на первый взгляд кажутся вам вполне уместными, позже оказываются неправильными. Поэтому я не рекомендую целиком и полностью доверять своим предчувствиям относительно паттернов. Лучше думайте о том, что, по-вашему, должна делать система, позвольте дизайну оформиться так, как это необходимо.

Почему TDD работает?

Приготовьтесь покинуть галактику. Предположите на секунду, что TDD помогает командам разработчиков создавать хорошо спроектированные, удобные в сопровождении системы с чрезвычайно низким уровнем дефектов. (Я не утверждаю, что это происходит на каждом шагу, я просто хочу, чтобы вы немножко помечтали.) Как такое может происходить?

Отчасти этот эффект связан с уменьшением количества дефектов. Чем раньше вы найдете и устраните дефект, тем дешевле это вам обойдется. Иногда разница в затратах огромна (спросите у «Марс-лендера»¹). Снижение количества дефектов приводит к возникновению множества вторичных психологических и социальных эффектов. После того как я начал работать в стиле TDD, программирование стало для меня значительно менее нервным занятием. Когда я работаю в стиле TDD, мне не надо беспокоиться о множестве вещей. Вначале я могу заставить ра-

¹ Mars Lander — американский космический аппарат, был запущен в сторону Марса 3 января 1999 г. 3 декабря 1999 г. аппарат должен был осуществить посадку на Марс, однако в этот день связь с ним была потеряна, предположительно из-за ошибки в программном обеспечении. Стоимость миссии составила приблизительно 120 млн долларов, не считая стоимости ракеты-носителя и некоторого дополнительного оборудования. — *Примеч. перев.*

ботать только один тест, потом — все остальные. Уровень стресса существенно снизился. Взаимоотношения с партнерами по команде стали более позитивными. Разработанный мною код перестал быть причиной сбоев, люди стали в большей степени рассчитывать на него. У заказчиков тоже повысилось настроение. Теперь выпуск очередной версии системы означает новую функциональность, а не набор новых дефектов, которые добавляются к уже существующим.

Уменьшение количества дефектов. Имею ли я право утверждать, что подобное возможно? Есть ли у меня научное доказательство?

Нет. На текущий момент не проводилось никаких исследований, подтверждающих преимущества TDD по сравнению с альтернативными подходами в смысле качества, эффективности или удовольствия. Однако эпизодические подтверждения преимуществ TDD многочисленны, а вторичные эффекты очевидны. При использовании TDD у программистов действительно снижается стресс, в командах действительно повышается доверие, а заказчики действительно начинают смотреть на каждую новую версию продукта с энтузиазмом. Лично мне не приходилось сталкиваться с обратными эффектами. Однако ваши наблюдения могут оказаться иными, вы должны попробовать для того, чтобы оценить TDD самостоятельно.

Еще одним преимуществом методики TDD, объясняющим ее положительные эффекты, является сокращение времени, которое проходит между принятием проектного решения и проверкой результата его реализации. В рамках TDD это достаточно короткий промежуток времени — несколько секунд или минут. Вы принимаете решение, реализуете его в коде, запускаете тесты и анализируете полученный результат. В начале у вас возникает мысль — возможно, API должен выглядеть так, или, возможно, метафора должна быть такой, — затем вы создаете самый первый пример — тест, который воплощает вашу мысль в реальность. Вместо того чтобы сначала проектировать, а затем в течение нескольких недель или месяцев ожидать, окажется ли ваше решение правильным или нет, вы получаете результат уже через несколько секунд или минут.

Причудливый ответ на вопрос, «почему TDD работает?», основан на бредовом видении из области комплексных систем. Неподражаемый Флип (Phlip) пишет:

Следует использовать программистские практики, которые «притягивают» корректный код как предельную функцию, но не как абсолютную величину. Если вы пишете тесты для каждой присутствующей в системе функциональности и если вы добавляете в систему новые функции по одной и только после того, как сработают все тесты, вы создадите то, что математики обозначают термином «точка притяжения (аттрактор)». Точка притяжения — это точка в пространстве состояний, к которой сходятся все потоки. Со временем код с большей вероятностью изменяется в лучшую сторону, а не в худшую; точка притяжения приближается к корректности, как предельная функция.

Эта «корректность», которая устраивает всех программистов (за исключением, конечно же, тех, кто работает над медицинским или аэрокосмическим программным обеспечением). Я считаю, что важно быть знакомым с концепцией точки притяжения — ее не следует отвергать, ею не следует пренебрегать.

Что означает имя?

Название методики: Test-Driven Development — *разработка через тестирование*. Буквально можно перевести как «разработка, ведомая тестами» или «разработка исходя из тестов».

- *Development* (разработка) — старый поэтапный подход к разработке программного обеспечения обладает рядом недостатков, так как оценить результат проектного решения очень сложно, если решение и оценка результатов удалены друг от друга по времени. В названии TDD термин «разработка» означает сложную комбинацию анализа, логического проектирования, физического проектирования, реализации, тестирования, пересмотра, интеграции и выпуска.
- *Driven* (исходя из) — в свое время я называл TDD термином *test-first programming* (программирование «вначале тесты»). Однако антонимом слова *first* (вначале) является слово *last* (в конце). Огромное количество людей осуществляют тестирование уже после того, как они запрограммировали функциональный код. Этот подход считается вполне приемлемым. Существует любопытное правило именования, согласно которому противоположность придуманного вами имени должна быть, по крайней мере, отчасти неприятной или неудовлетворительной. (Термин «структурное программирование» звучит привлекательно, так как никто не хочет писать бесструктурный, то есть неорганизованный код.) Если в ходе разработки я исхожу не из тестов, то из чего? Из предположений? Домыслов? Спецификаций? (обратите внимание, что слово «спецификация» немножко похоже на слово «спекуляция»).
- *Test* (тест) — автоматическая процедура, позволяющая убедиться в работоспособности кода. Нажмите на кнопку, и он будет выполнен. Ирония TDD состоит в том, что это вовсе не методика тестирования. Это методика анализа, методика проектирования, фактически методика структурирования всей деятельности, связанной с разработкой программного кода.

Как методика TDD связана с практиками экстремального программирования?

Некоторые из людей, просматривавших рукопись данной книги, были обеспокоены тем, что книга целиком и полностью посвящена TDD, в результате читатели могут подумать, что остальными практиками XP (eXtreme Programming) можно пренебречь. Например, если вы работаете в стиле TDD, должны ли вы при этом работать в паре? Далее я привожу перечень соображений относительно того, каким образом остальные практики XP улучшают эффективность TDD, и наоборот, каким образом TDD повышает эффективность использования других практик XP:

- Программирование в паре. Тесты, разрабатываемые в рамках TDD, являются превосходным инструментом общения, когда вы программируете в паре. Зачастую, работая в паре партнеры не могут договориться — какую именно проблему они решают, несмотря на то что они работают с одним и тем же кодом. Это звучит бредово, однако подобное происходит постоянно, в особенности когда вы только осваиваете работу в паре. Именно этой проблемы удается избежать благодаря TDD. Существует и обратное влияние: когда вы работаете в паре, у вас есть помощник, который может взять на себя нагрузку в случае, если вы устали. Ритм TDD может исчерпать ваши силы, и тогда вы будете вынуждены программировать несмотря на то, что вы устали. Однако если вы работаете в паре, ваш партнер готов взять у вас клавиатуру и тем самым предоставить вам возможность немного расслабиться.
- Работа на свежую голову. XP рекомендует работать тогда, когда вы полны сил, и останавливать работу тогда, когда вы устали. Если вы не можете заставить следующий тест сработать или заставить работать те два теста одновременно, значит, настало время прерваться. Однажды мы с дядей Бобом Мартином (Bob Martin)¹ работали над алгоритмом разбиения линии и нам никак не удавалось заставить его работать. Несколько минут мы безуспешно бились над кодом, однако нам стало очевидно, что прогресса нет, поэтому мы просто остановили работу.
- Частая интеграция. Тесты — это великолепный ресурс, который позволяет вам выполнять интеграцию значительно чаще. Вы добились срабатывания очередного теста, удалили дублирование, значит, вы можете интегрировать код. Этот цикл может повторяться в течение 15–30 минут. Возможность частой интеграции позволяет более многочисленным командам разработчиков иметь дело с одной и той же базой исходного кода. Как сказал Билл Уэйк (Bill Wake): «проблема $n2$ не является проблемой, если n всегда равно 1».
- Простой дизайн. В рамках TDD вы пишете только тот код, который необходим для срабатывания тестов, вы удаляете из него любое дублирование, значит, вы автоматически получаете код, который идеально адаптирован к текущим требованиям и подготовлен к любым будущим пожеланиям. Общая доктрина требует, чтобы дизайна было достаточно для получения идеальной архитектуры для текущей системы. Эта доктрина также облегчает разработку тестов.
- Рефакторинг. Удаление дублирования — это основная цель рефакторинга. Тесты дают вам уверенность в том, что поведение системы не изменится даже в случае, если вы выполняете достаточно крупномасштабные рефакторинги. Чем выше ваша уверенность, тем более агрессивно вы выполняете

¹ Боб Мартин (Bob Martin) — известный деятель движения Agile Development, которого часто с уважением называют дядей. — *Примеч. перев.*

рефакторинг. Рефакторинг продлевает жизнь вашей системе. Благодаря рефакторингу вы упрощаете дальнейшую разработку тестов.

- Частые выпуски версий. Если тесты TDD действительно улучшают MTBF вашей системы (в этом вы можете убедиться сами), значит, вы можете чаще внедрять разрабатываемый код в реальные производственные условия и при этом не наносить ущерба вашим заказчикам. Гарет Ривс (Gareth Reeves) приводит аналогию с куплей-продажей ценных бумаг на бирже в течение дня. Если вы занимаетесь краткосрочной спекуляцией ценными бумагами, в конце торгового дня вы должны продать все имеющиеся у вас ценные бумаги, так как вы не хотите принимать риск, связанный с сохранением некоторых ценных бумаг до следующего торгового дня, — этот риск вам не подконтролен. Разрабатывая систему, вы хотите, чтобы вносимые вами изменения как можно быстрее были опробованы в реальных производственных условиях, так как не хотите тратить время на разработку кода, в полезности которого не уверены.

Нерешенные проблемы TDD

Дарач Эннис (Darach Ennis) бросил вызов поклонникам TDD, размышляющим о возможностях расширения области применения TDD. Он сказал:

Множество различных организаций сталкивается с многочисленными проблемами TDD, и эти проблемы никак не затронуты в книге. Возможно, эти проблемы вообще никак не решить в рамках TDD. Вот некоторые из них:

- *не существует способа автоматического тестирования GUI (например, Swing, CGI, JSP/Servlets/Struts);*
- *не существует способа автоматического тестирования распределенных объектов (например, RPC, Messaging, CORBA/EJB и JMS);*
- *TDD нельзя использовать для разработки схемы базы данных (например, JDBC);*
- *нет необходимости тестировать код, разработанный сторонними разработчиками, или код, генерируемый внешними инструментами автоматизации разработки;*
- *TDD нельзя использовать для разработки компилятора/интерпретатора языка программирования.*

Я не уверен в том, что он прав, однако я также не уверен в том, что он не прав. В любом случае это почва для размышлений о дальнейшем развитии TDD.

Послесловие

Мартин Фаулер (Martin Fowler)

Когда рассказываешь о разработке, основанной на тестировании, сложнее всего передать то психическое состояние, в котором ты находишься, когда работаешь в стиле TDD. Я помню, как в ходе проекта СЗ мы с Ральфом Битти (Ralph Beatie) работали над реализацией сложного набора условий выплаты. Ральф сформулировал набор соответствующих тестов, после чего мы приступили к реализации этих тестов одного за другим. Процесс был равномерным и неторопливым, из-за этого казалось, что мы работаем медленно. Однако, взглянув назад на сделанную работу, можно было понять, что, несмотря на кажущуюся неторопливость, мы работали очень даже быстро.

Несмотря на множество появившихся в последнее время мощных инструментов, программирование по-прежнему остается сложной работой. Я часто ощущаю себя в ситуации, когда мне кажется, что я жонглирую шариками, и мне приходится следить за несколькими шариками в воздухе в одно и то же время: малейшая потеря внимания, и все сыпется на пол. Методика TDD позволяет избавиться от этого ощущения.

Когда вы работаете в стиле TDD, в воздухе постоянно находится лишь один шарик. Вы можете сконцентрироваться на нем, а значит, хорошо справиться со своей работой. Когда я добавляю в программу новую функциональность, я не думаю о том, какой дизайн должен быть реализован в данной функции. Я просто пытаюсь добиться срабатывания тестов самым простым из доступных мне способов. Когда я переключаюсь в режим рефакторинга, я не беспокоюсь о добавлении в программу новых функций, я думаю только о правильном дизайне. На каждом из этих этапов я концентрируюсь на единственной задаче, благодаря этому мое внимание не распыляется.

Добавление новой функциональности при помощи тестов и рефакторинг — это две монологические разновидности программирования. Совсем недавно я открыл еще одну разновидность: копирование паттерна. Я занимался разработкой сценария на языке Ruby, выполняющего извлечение информации из базы данных. Я начал с создания класса, являющегося оболочкой таблицы базы данных, а затем сказал себе, что раз я только что закончил книгу о паттернах работы с базами данных, я должен использовать паттерн. Примеры программ в книге были написаны на Java, поэтому нужный мне код легко можно было перенести на Ruby. Когда я программировал, я не думал о решении проблемы, я думал лишь о том, каким образом лучше всего адаптировать паттерн для условий, в рамках которых я работал.

Копирование паттернов само по себе не является хорошим программированием, — я всегда подчеркиваю этот факт, когда говорю о паттернах. Любой паттерн — это полуфабрикат, — вы должны адаптировать его для условий своего проекта. Однако чтобы сделать это, лучше всего вначале, особо не задумываясь, скопировать паттерн, а затем, воспользовавшись смесью рефакторинга и TDD, выполнить адаптацию. В этом случае в процессе копирования паттерна вы также концентрируетесь только на одной вещи — на паттерне.

Сообщество XP интенсивно работает над тем, чтобы добавить в общую картину паттерны. Со всей очевидностью можно сказать, что сообщество XP любит паттерны. В конце концов, между множеством приверженцев XP и множеством приверженцев паттернов существует значительное пересечение: Уорд и Кент являются лидерами обоих направлений. Наверное, копирование паттерна — это третий монологический режим программирования наряду с разработкой в стиле «тесты вначале» и рефакторингом. Как и первые два режима, копирование паттерна — опасная штука, если ее использовать отдельно от двух других режимов. Все три вида программирования проявляют свою мощь только тогда, когда используются совместно друг с другом.

Если вы хотите сделать некоторый процесс эффективным, вы должны идентифицировать основные действия, из которых состоит процесс, а затем добиться того, чтобы в каждый момент времени внимание концентрировалось только на одном таком действии. Примером такого подхода является сборочная линия, в рамках которой каждый рабочий выполняет только одну из множества необходимых процедур. Внимание каждого рабочего сконцентрировано только на одном действии. Методика разработки, основанной на тестировании (TDD), подразумевает разделение процесса программирования на элементарные режимы, однако при этом она избавляет нас от монотонности, предлагая быстро переключаться между этими режимами. Комбинация монологических режимов и переключения между ними обеспечивает должную концентрацию внимания, снижает стресс и избавляет нас от монотонности сборочной линии.

Я признаю, что все эти мысли несколько сыроваты. Когда я пишу это, я по-прежнему не уверен в том, о чем я рассказываю. Я знаю, что буду обдумывать все эти идеи еще в течение нескольких, а может быть, и многих месяцев. Однако я полагаю, что эти идеи должны вам понравиться. Прежде всего, они стимулируют ваши размышления о более крупной картине, в которую вписывается разработка, основанная на тестировании. Мы еще не видим эту картину достаточно четко, однако, мне кажется, что она постепенно становится все яснее и яснее.

Приложение I

Диаграммы взаимовлияния

В данной книге можно встретить несколько примеров диаграмм взаимовлияния. Идея диаграмм взаимовлияния позаимствована из серии книг *Quality Software Management* Джеральда Вейнберга (Gerald Weinberg), точнее говоря, из книги 1: *Systems Thinking*¹. Цель диаграммы взаимовлияния — продемонстрировать, каким образом элементы системы влияют друг на друга.

Диаграмма взаимовлияния включает в себя элементы трех типов:

- Действие или деятельность² — обозначается словом или короткой фразой.
- Позитивное соединение — обозначается стрелкой, указывающей от одного действия к другому действию. Позитивное соединение обозначает, что усиление интенсивности исходного действия приводит к усилению интенсивности целевого действия, а снижение интенсивности исходного действия приводит к снижению интенсивности целевого действия.
- Негативное соединение — обозначается стрелкой между двумя действиями, поверх которой нарисован кружочек. Негативное соединение обозначает, что усиление интенсивности исходного действия ведет к снижению интенсивности целевого действия, и наоборот, снижение интенсивности исходного действия приводит к усилению интенсивности целевого действия.

Слишком много слов для очень простой концепции. На рис. П1.1–П1.3 приводятся несколько примеров диаграмм взаимовлияния.

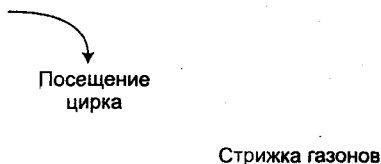


Рис. П1.1. Два действия, которые, по всей видимости, не влияют друг на друга

¹ Weinberg, Gerald. 1992. *Systems Thinking. Quality Software Management*. New York: Dorset House.

² Если исходить из примеров диаграмм взаимовлияния, приводимых здесь автором книги, то элемент этого типа может также называться «характеристика». — *Примеч. перев.*

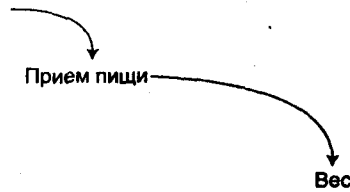


Рис. П1.2. Два действия, связанные позитивным соединением

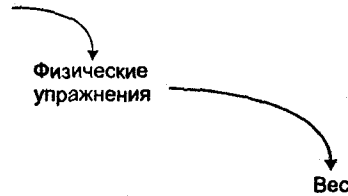


Рис. П1.3. Два действия, связанные негативным соединением

Чем больше я ем, тем больше я вешу. Чем меньше я ем, тем меньше я вешу. Конечно же, вес человеческого тела — это значительно более сложная система. Диаграммы взаимовлияния — это модели, которые помогают вам понять некоторые аспекты системы, однако они вовсе не предназначены для того, чтобы понимать и контролировать систему в полной мере.

Обратная связь

Влияние не распространяется только в одном направлении. Действие может быть связано обратной связью само с собой. Иначе говоря, в некоторых случаях изменение интенсивности действия влияет на само это действие. Иногда это влияние позитивно, а иногда — негативно. Пример подобной обратной связи продемонстрирован на рис. П1.4.

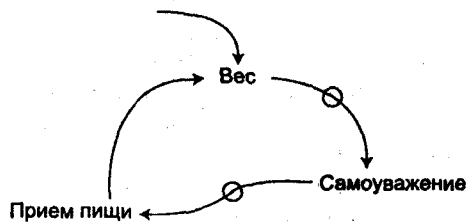


Рис. П1.4. Обратная связь

Существует два типа обратной связи — позитивная и негативная. Позитивная обратная связь приводит к тому, что интенсивность действия в системе постоянно увеличивается. Чтобы обнаружить позитивную обратную связь, достаточно посчитать количество негативных соединений в цикле. Если в цикле четное количество негативных соединений, значит, этот цикл является циклом позитивной обратной связи. На рис. П1.4 изображен цикл позитивной обратной связи: поправ

в этот цикл, вы продолжаете набирать вес до тех пор, пока в составе цикла не появится какое-либо дополнительное действие.

Негативная обратная связь снижает интенсивность действия. Если в цикле присутствует нечетное количество негативных связей, значит, цикл является циклом негативной обратной связи.

Вот три ключа хорошего системного дизайна:

- создание полезных циклов, в которых позитивная обратная связь приводит к увеличению интенсивности полезных действий;
- устранение вредных циклов, в которых позитивная обратная связь приводит к увеличению интенсивности бесполезных, вредных и деструктивных действий;
- создание циклов негативной обратной связи, которые предотвращают чрезмерное использование полезных действий.

Контроль над системой

Выбирая систему практик разработки программного обеспечения, вы должны добиться того, чтобы каждая практика способствовала применению других практик, благодаря этому вы сможете использовать каждую из практик в достаточном объеме даже в состоянии стресса. На рис. П1.5 показан пример системы практик, которая приводит к недостаточному тестированию.

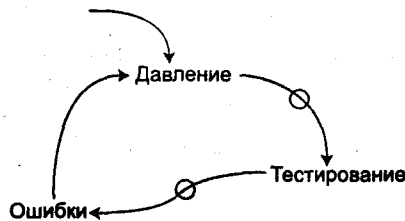


Рис. П1.5. Недостаток времени для тестирования приводит к общему недостатку времени

Когда время начинает поджимать, вы снижаете интенсивность тестирования, что приводит к увеличению количества ошибок, что, в свою очередь, приводит к еще большему недостатку времени. Со временем на сцене появляется некоторое внешнее действие (например, недостаток денег), которое заставляет вас завершить работу над проектом, несмотря ни на что.

Если вы имеете дело с системой, которая ведет себя не так, как вам того хотелось бы, у вас есть несколько вариантов исправить ситуацию:

- Сформируйте цикл позитивной обратной связи в обратном направлении. Если у вас цикл между тестами и уверенностью и тесты не срабатывают, снижая тем самым уверенность, тогда вы можете сделать больше срабатывающих тестов, повысив тем самым уверенность в вашей способности увеличить количество срабатывающих тестов.

- Сформируйте цикл негативной обратной связи, который позволит вам контролировать действие, интенсивность которого стала слишком большой.
- Создайте или разорвите соединения для того, чтобы устранить циклы, не являющиеся полезными.

Приложение II

Фибоначчи

В ответ на просьбу одного из людей, просматривавших мою рукопись, я включил в книгу описание разработки функции вычисления последовательности Фибоначчи в стиле TDD. Некоторые утверждают, что именно этот пример раскрыл им глаза на то, каким образом работает TDD. Однако этот пример очень короток, к тому же в нем не используются многие важные приемы, применяемые в рамках TDD. По этой причине его невозможно использовать в качестве замены примеров, рассмотренных ранее в данной книге. Если, ознакомившись с рассмотренными ранее примерами, вы до сих пор не можете понять, каким образом осуществляется разработка в стиле TDD, ознакомьтесь с данным материалом, возможно, он поможет вам прояснить ситуацию.

Первый тест показывает, что $\text{fib}(0) = 0$. Реализация возвращает константу.

```
public void testFibonacci() {
    assertEquals(0, fib(0));
}
int fib(int n) {
    return 0;
}
```

(Я использую класс `TestCase` как вместилище кода, так как мы разрабатываем всего одну функцию.)

Второй тест показывает, что $\text{fib}(1) = 1$.

```
public void testFibonacci() {
    assertEquals(0, fib(0));
    assertEquals(1, fib(1));
}
```

Я просто добавил еще один оператор `assert()` в тот же самый тестовый метод, так как не вижу особого смысла создавать новый метод с именем `testFibonacciOfOneIsOne`.

Чтобы заставить тест сработать, можно воспользоваться одним из нескольких методов. Я решаю использовать значение 0, как специальный случай:

```
int fib(int n) {
    if (n == 0) return 0;
    return 1;
}
```

Дублирование в тестирующем методе начинает действовать мне на нервы. По мере того как мы будем добавлять новые тесты, дублирование будет только усугубляться. Давайте попробуем выделить общую структуру операторов `assert()`, для этого добавим в тест таблицу входных и ожидаемых значений функции `fib()`:

```
public void testFibonacci() {
    int cases[][]= {{0,0},{1,1}};
    for (int i= 0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Теперь добавление нового теста требует всего шесть нажатий на клавиши и никаких дополнительных строк:

```
public void testFibonacci() {
    int cases[][]= {{0,0},{1,1},{2,1}};
    for (int i= 0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Как это ни удивительно, но новый тест срабатывает. Это происходит потому, что константа 1 также подходит и для входного значения 2. Переходим к следующему тесту:

```
public void testFibonacci() {
    int cases[][]= {{0,0},{1,1},{2,1},{3,2}};
    for (int i= 0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Ура! Наконец-то тест не сработал. Воспользуемся прежней стратегией (рассматриваем меньшие входные значения, как специальные случаи):

```
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return 2;
}
```

Теперь мы готовы обобщить код. Мы написали 2, однако на самом деле мы имели в виду $1 + 1$.

```
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return 1 + 1;
}
```

Первая единица в сумме — на самом деле `fib(n-1)`:

```
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return fib(n-1) + 1;
}
```

Вторая единица в сумме — на самом деле `fib(n-2)`:

```
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return fib(n-1) + fib(n-2);
}
```

Теперь окончательно чистим код. Та же самая структура должна работать для `fib(2)`, поэтому мы можем преобразовать второй условный оператор:

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Это и есть функция вычисления последовательности Фибоначчи, целиком и полностью разработанная в рамках методики TDD.

Алфавитный указатель

A

- Account, класс, 180
- Add Parameter, добавление параметра, 193
- add, метод, 119
- addend, 72
- aliasing, наложение имен, 171
- All Tests, все тесты, 166
- AllTests, класс, 166
- application-level test, 202
- assert, 132
- Assert First, вначале оператор assert, 132
- assert, оператор, 159
- assertEquals, 159
- assertion, 159
- ATDD, Application Test-Driven Development, 202
- augment, 72

B

- Bank, класс, 69
- Break, перерыв, 141
- Broken Test, сломанный тест, 151

C

- Child Test, дочерний тест, 145
- Clean Check-in, чистый выпускаемый код, 151
- clean code that works, 10
- Collecting Parameter
 - накапливающий параметр, 119
 - накопление в параметре, 182
- Command, команда, 170
- Composite, компоновщик, 118, 180
- copy-and-paste, 40
- Crossing to Safety, 42
- Crush Test Dummy, тестирование обработки ошибок, 149
- Currency, класс, 18
- currency, метод, 54
- cyclomatic complexity, цикломатическая сложность, 92

D

- Design Patterns, паттерны проектирования, 168
- Do over, начать сначала, 143

Dollar, класс, 17

- определение, 24
- отказ от использования, 50
- побочные эффекты, 30
- сравнение с Franc, 47
- удаление из программы, 63

E

- equals, метод, 33
 - обобщение, 42
 - сравнение разных валют, 47
- Evident Data, понятные данные, 135
- Exception Test, тест исключения, 166
- Explanation Test, объясняющий тест, 139
- Expression, класс, 67
- External Fixture, внешняя фикстура, 163
- Externalizable, интерфейс, 182
- Extract Interface, выделение интерфейса, 190
- Extract Method, выделение метода, 102, 188

F

- Factory Method
 - мультивалютный пример, 50
- Factory Method, фабричный метод, 178
- fail, метод, 166
- Fake It, подделка, 31, 153
- File, класс Java, 172
- Fixture, фикстура, 161
- framework, инфраструктура, 197
- Franc, класс
 - определение, 40
 - отказ от использования, 50
 - создание, 39
 - сравнение с Dollar, 47
 - удаление equals, 45
 - удаление из программы, 63

I

- Imposter, самозванец, 67, 179
- Inline Method, встраивание метода, 189
- Isolate Change, изоляция изменений, 185
- Isolated Test, изолированный тест, 128

J

java.io.Externalizable, интерфейс, 182
 java.io.File, класс, 172
 Jester, 94
 JHotDraw, 173
 JProbe, 94
 JUnit, 124, 160
 статистика использования, 91
 JXUnit, 160

L

LaxSecurity, класс, 173
 LifeWare, 201
 Log String, строка-журнал, 149

M

Method Object, метод в объект, 192
 Method Parameter to Constructor Parameter, 193
 MIDP, Mobile Information Device Profile, 139
 Migrate Data, миграция данных, 186
 Mock Object, поддельный объект, 146
 Money, класс
 абстрактный метод, 50
 определение, 43
 создание, 43
 фабричный метод, 50
 Move Method, перемещение метода, 191
 MTBF, Mean Time Between Failures, 199

N

None, ключевое слово, 99
 Null Object, ноль-объект, 172

O

Observer, наблюдатель
 вместе с журналом, 149
 Obvious Implementation, очевидная
 реализация, 31, 156
 One Step Test, тест одного шага, 136
 One to Many, от одного к многим, 157
 Open/Closed Principle, 198

P

Pair, класс, 78
 pass, ключевое слово, 99
 Pluggable Object, встраиваемый объект, 175
 Pluggable Selector, встраиваемый
 переключатель, 176
 Python, 97

R

Realistic Data, реалистичные данные, 134
 Reconcile Differences, согласование
 различий, 184

reduce, метод, 68
 Reflection API, 178
 Regression Test, регрессионный тест, 141
 Runnable, интерфейс, 170

S

scaffolding, 161
 Self Shunt, самошунтирование, 147
 self, ссылка, 122
 setUp, метод, 104, 161
 Shower Methodology, методология душа, 142
 SmallLint for Smalltalk, 90
 Starter Test, начальный тест, 137
 suite, метод, 166
 Sum, класс, 68
 определение, 72
 создание, 72
 summary, метод, 113
 SUnit, Smalltalk Unit, 160

T**TDD**

влияние среды разработки, 201
 длина
 цикла, 195
 шага, 195
 для кого предназначена, 204
 дополнительные тесты, 90
 завершение работы, 89
 имеющийся код, 203
 крупномасштабные системы, 201
 мотивация, 11
 навыки, 195
 начальные условия, 205
 паттерны проектирования, 205
 переход к использованию, 203
 проблемы, 210
 пятиэтапный процесс, 40
 расшифровка термина, 208
 риск, 37
 связь с XP, 208
 скорость работы, 56
 эффективность, 206
 ясность тестов, 85
 tearDown, метод, 108, 163
 Template Method, шаблонный метод, 173
 Test Data, тестовые данные, 134
 Test First, вначале тест, 131
 test infected, 13
 Test List, список тестов, 129
 Test Method, тестовый метод, 164
 Test, интерфейс, 124
 test, префикс, 164
 TestCase, класс, 101
 Test-Driven Development
 расшифровка термина, 208

testFailed, тест, 115
 test-first programming, 208
 TestResult, класс, 113
 testStarted, метод, 115
 TestSuite, класс, 118, 166, 187
 times, метод, 24

 обобщение, 49, 58
 toString, метод, 60
 Transaction, класс, 180
 Triangulate, триангуляция, 155
 Triangulation, триангуляция, 33

V-W

Value Object, объект-значение, 32, 170
 WasRun, класс, 99
 WyCash, 17

X

xP, eXtreme Programming
 связь с TDD, 208
 xUnit, 123
 отчет о результатах, 112
 паттерны, 159
 разработка, 97
 ретроспектива, 123

A-B

абстракция, 85
 база данных
 тестирование, 146
 банк, 68
 бумажник, 67
 метафора, 91

B

валюта, 53
 курс обмена, 68
 обмен, 76
 сложение разных валют, 81
 сравнение, 61
 стандартная, 67
 вектор, метафора, 90
 вначале тест, 131
 встраиваемый объект, 175
 встраиваемый переключатель, 176
 встраивание метода, 189
 выделение интерфейса, 190
 выделение метода, 102, 188, 220
 выражение, 67, 90

G

глобальная переменная, 183
 графический редактор, 175

D

данные
 миграция, 186
 для тестирования, 134
 денежный пример
 ретроспектива, 89
 дефект
 намеренное добавление, 94
 снижение количества, 206
 тестирование, 141
 диаграмма взаимовлияния, 213
 обратная связь, 214
 добавление параметра, 193
 доллар
 сложение, 66
 удаление дублирования, 71
 сложение с франком, 81
 сравнение с франком, 47
 дублирование, 26

Ж-З

журнал, 108, 149
 завершение работы, 89
 дополнительные тесты, 90
 зависимость кода, 26
 запах тестов, 196
 зеленая полоса, 23

I

изоляция изменений, 185
 изоляция тестов, 128
 изучение, 139
 имя класса
 сравнение, 60
 интеграция, 151
 интерфейс
 выделение, 190
 инфраструктура, 197
 исключение
 имитация, 150
 перехват, 112
 тестирование, 166

K

качество тестов, 93, 196
 класс
 соответствие между функциональными
 и тестовыми, 163
 код
 метрики, 92
 характеристики, 92
 код без тестов, 44, 60
 команда, 170
 компоновщик, 118, 180
 конструктор, 178

концентрация, 140
красная полоса, 24
критика кода, инструмент, 90
курс обмена валюты, 68
таблица, 77

М

мебель для TDD, 144
метафора, 90
метод
встраивание, 189
выделение, 188
перемещение, 191
преобразование в объект, 192
принадлежность к классу, 68
тестовый, 164
фабричный, 178
шаблонный, 173
метод в объект, 192
методология душа, 142
миграция данных, 186
мультивалютные деньги, 22

Н

набор тестов, 118, 166
накапливающий параметр, 119
наложение имен, 171
начальные условия, 205
начальный тест, 137
нуль-объект, 172

О

обработка ошибок
тестирование, 149
обратная связь, 214
объект
встраиваемый, 175
нуль-объект, 172
ограничение возможностей, 171
создание, 178
объект для теста, 161
объект-значение, 32, 170
объяснение, 139
объяснение при помощи тестов, 139
объясняющий тест, 139
одиночка, 183
от одного ко многим, 157
отказ от работы, 143
очевидная реализация, 31, 156

П

параметр
добавление, 77, 193
накопление, 182
параметр метода в параметр конструктора, 193

парное программирование, 209
паттерн, 168, 205

All Tests, 166
Another Test, 140
Assert First, 132
Break, 141
Broken Test, 151
Cheap Desk, Nice Chair, 144
Child Test, 145
Clean Check-in, 151
Collecting Parameter, 119, 182
Command, 170
Composite, 118, 180
Crush Test Dummy, 149
Do over, 143
Evident Data, 135
Exception Test, 166
Explanation Test, 139
External Fixture, 163
Factory Method, 178
Fake It, 31, 153
Fixture, 161
Imposter, 179
Isolated Test, 128
Learning Test, 139
Log String, 149
Mock Object, 146
Null Object, 172
Obvious Implementation, 31, 156
One Step Test, 136
One to Many, 157
Pluggable Object, 175
Pluggable Selector, 176
Realistic Data, 134
Regression Test, 141
Self Shunt, 147
Singleton, 183
Starter Test, 137
Template Method, 173
Test First, 131
Test List, 129
Test Method, 164
Triangulate, 155
Triangulation, 33
Value Object, 32, 170
проектирования, 168
паттерны проектирования
связь с TDD, 205
перемещение метода, 191
перерыв, 141
поддельная реализация, 31, 153
поддельный объект, 146
подклассы
уничтожение, 63
устранение, 53
покрытие тестами операторов кода, 94

порядок выполнения методов, 149
 принцип открытости/закрытости, 198
 простой дизайн, 209

Р

равенство
 тестирование, 32
 реалистичные данные, 134
 регрессионный тест, 141
 ресурсы
 освобождение, 163
 ретроспектива, 89
 ретроспектива xUnit, 123
 рефакторинг, 184, 209
 Add Parameter, 193
 Extract Interface, 190
 Extract Method, 102, 188, 220
 Inline Method, 189
 Isolate Change, 185
 Method Object, 192
 Method Parameter to Constructor Param, 193
 Migrate Data, 186
 Move Method, 191
 Reconcile Differences, 184
 автоматизация, 196
 отмена, 109

С

самозванец, 67, 179
 самошунтирование, 147
 сложение разных валют, 81
 сокет, 133
 список
 задач, 23, 129
 очистка, 66
 тезисов, 165
 тестов, 129
 среда разработки, 201

Т

тезис, 165
 тест, 126
 автоматический, 126
 взаимозависимость, 105
 вначале тест, 131
 выбор начального теста, 137
 выбор следующего теста, 136
 данные, 134
 для изучения, 139
 дочерний, 145
 запуск всех тестов, 166
 изоляция, 105, 128
 имя, 164
 качество, 93, 196
 количество, 94, 198

тест (*продолжение*)
 копирование, 39
 модуля, 202
 набор, 118
 наглядность, 36
 общие данные, 129
 одного шага, 136
 порядок реализации, 112
 последовательность реализации, 205
 разделение большого теста, 145
 регрессионный, 141
 самый первый, 98
 сломанный, 151
 список, 129
 удаление теста, 200
 уровня приложения, 202
 тестирование, 126
 баз данных, 146
 величина шага, 27
 имеющегося кода, 203
 исключения, 166
 объект тестирования, 196
 паттерны, 145
 чужого программного обеспечения, 139
 тестовый метод, 164
 тесты
 запах, 196
 треугольник
 задача о, 198
 триангуляция, 33, 155

У
 условный оператор
 альтернатива, 175

Ф

фабричный метод, 50, 178
 Фибоначчи, 217
 фикстура, 161
 внешняя, 163
 создание, 104

Франк

конвертация в доллар, 76
 сложение с долларом, 81
 создание класса, 39
 сравнение с долларом, 47
 умножение, 39

X-Э

хэширование, 78
 цикломатическая сложность, 92
 частая интеграция, 209
 частые выпуски версий, 210
 шаблонный метод, 173
 эволюция программы, 137